Computational Natural Deduction

Seppo R. Keronen

A thesis submitted for
the degree of Doctor of Philosophy
of the Australian National University

Chapter 4

Expressive Power and Inference

Normal form natural deduction exhibits a simple correspondence between the expressive power of a language and the deductive machinery required for its implementation. A hierarchy of deduction systems properly contained in the deduction system for classical logic is explored incrementally. The important languages encountered along the way are identified. A short detour, to survey negation as failure and relevant deduction, concludes the chapter.

4.1 Containment and Conservative Extension

The normal form and atomic normal form formulations of deducibility for classical logic exhibit two important containment properties. The first of these is the simple correspondence between the expressive power of a language and the introduction and elimination rules required to solve deduction problems stated in that language.

Sublanguage Property: For any solution $\mathbf{\Sigma}$ of the deduction problem $\mathbf{\Phi}$:

- $\mathbf{\Sigma}$ contains instances of introduction rules only for those operators that appear as the primary operator of a goal subformula of $\mathbf{\Phi}$.
- $\mathbf{\Sigma}$ contains instances of elimination rules only for those operators that appear as the primary operator of an assertion subformula of $\mathbf{\Phi}$.

The second containment property extends the first, centering on the acceptance or rejection of the absurdity rule and excluded middle as acceptable principles of reasoning.

Sublogic Property: The rejection of the rule of excluded middle (MX) from the classical system yields a system for *intuitionistic logic* [Dummett 77]. The rejection of the absurdity rule (#X) from the intuitionistic system yields *minimal logic* [Johansson 36].

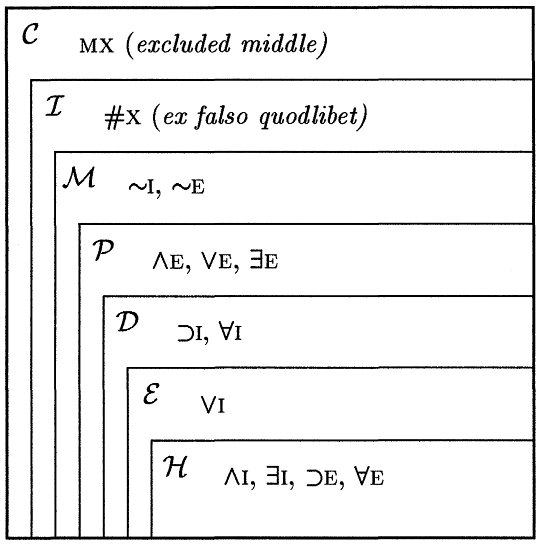

$\mathcal{C}$ MX (*excluded middle)*
$\mathcal{I}$ #X (*ex falso quodlibet)*
$\mathcal{M}$ ~I, ~E
$\mathcal{P}$ ∧E, ∨E, ∃E
$\mathcal{D}$ ⊃I, ∀I
$\mathcal{E}$ ∨I
$\mathcal{H}$ ∧I, ∃I, ⊃E, ∀E

Figure 4.1: key to languages and deduction systems

In this chapter, we consider the implementation of inference engines for a hierarchy of properly contained subsystems of $\mathcal{C}_\Sigma$, the system for atomic normal form solution in classical logic. The containment of systems and corresponding languages is illustrated in figure 4.1. This hierarchy consists of the languages:

$\mathcal{H}$: Horn language

$\mathcal{E}$: positive Edinburgh Prolog language

$\mathcal{D}$: positive definite language

$\mathcal{P}$: positive language

$\mathcal{M}$: minimal logic

$\mathcal{I}$: intuitionistic logic

$\mathcal{C}$: classical logic

The Horn language system is the simplest, requiring just four rules of inference. Each of the following systems is a conservative extension of the preceding one obtained by adding the inference rules indicated in figure 4.1.

4.2 The Horn Language

The deduction system $\mathcal{H}_\Sigma$, for ANF solutions for problems posed in the Horn language, is shown in figure 4.2. The Horn language occupies a special niche in resolution refutation proof theory [Kowalski 79^b]. This is also the case for the ANF formulation, which supports the reading of a Horn formula as a rule of inference, as discussed below.

introduction component

$$\begin{array}{c} {\scriptstyle \wedge\mathrm{I}}\;\dfrac{G \qquad H}{(G\wedge H)} \\ \Sigma_{\mathrm{I}} \end{array}$$

$$\begin{array}{c} {\scriptstyle \exists\mathrm{I}}\;\dfrac{G(X)}{(\exists x G(x))} \\ \Sigma_{\mathrm{I}} \end{array}$$

elimination component

$$ {\scriptstyle \supset\mathrm{E}}\;\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{G\supset F} \qquad \dfrac{\Sigma_{\mathrm{I}}}{G}}{F}$$

$$ {\scriptstyle \forall\mathrm{E}}\;\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\forall x F(x)}}{F(X)}$$

structural rule

$$\left(\begin{array}{c} \dfrac{\Sigma_{\mathrm{E}}}{B} \\ {\scriptstyle =\mathrm{CUT}}\;\overline{\overline{}} \\ \dfrac{(A)}{\Sigma_{\mathrm{A}}} \end{array} \right)\Theta \qquad \text{where: } A\Theta = B\Theta$$

x	–	variable
X	–	parameter
Θ	–	substitution
A, B	–	atomic formula
E, F	–	assertion formula
G, H	–	goal formula
Σ_{I}	–	introduction component
Σ_{E}	–	elimination component
Σ_{A}	–	atomic normal form solution

Figure 4.2: system $\mathcal{H}_{\Sigma}$ — atomic normal form solution for the Horn language

4.2.1 Horn Formulae and Their Extensions

A feature of the Horn language, and many of the subsequent languages, is that assertion and goal subformulae have distinct syntax. For these dual syntax languages we use the syntactic variables E and F to stand for assertion formulae, and G and H for goal formulae. The syntax of Horn axioms and queries is illustrated in figure 4.3. The large prefix universal (existential) quantifiers in this figure denote the universal (existential) closure of the prefixed formula — that is, the formulae are in prenex form. Notice that although this conventional notion of Horn formulae requires prenex quantification, the deduction system $\mathcal{H}_{\Sigma}$ does not.

The inferential extension of a Horn language axiom is illustrated in figure 4.3 (a), the inferential extension of a query in (d). For the purpose of AND/OR graph search we can prune away the input formula resulting in the simpler forms (b) and (e). For rule based inference, (b) and (e) may be represented as the *derived rules of inference* (c) and (f) respectively. The prime notation in the figure indicates that applications of the universal elimination (∀E) and existential introduction (∃I) rules have replaced the bound variables of the input formula by parameters.

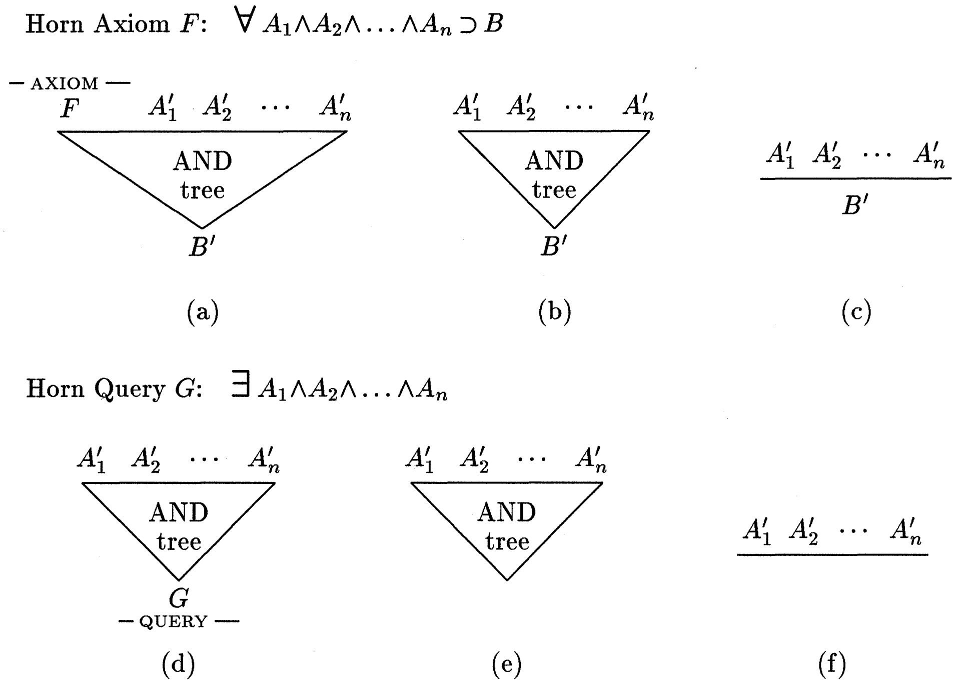

Horn Axiom F: $\forall A_1 \wedge A_2 \wedge \ldots \wedge A_n \supset B$
— AXIOM —
F A'_1 A'_2 $\cdots$ A'_n
AND tree
B'
(a)
A'_1 A'_2 $\cdots$ A'_n
AND tree
B'
(b)
$\dfrac{A'_1 \; A'_2 \; \cdots \; A'_n}{B'}$
(c)
Horn Query G: $\exists A_1 \wedge A_2 \wedge \ldots \wedge A_n$
A'_1 A'_2 $\cdots$ A'_n
AND tree
G
— QUERY —
(d)
A'_1 A'_2 $\cdots$ A'_n
AND tree
(e)
$\dfrac{A'_1 \; A'_2 \; \cdots \; A'_n}{}$
(f)

Figure 4.3: inferential extensions of Horn axioms and queries

The simple correspondence between Horn formulae and derived rules of inference, illustrated in figure 4.3, supports a proof theoretic view of a Horn problem. Read the input formulae, not as formulae, but as rules of inference or the clauses of an inductive definition of provable atomic formulae. This view is proposed and extended towards more expressive languages by [Hallnäs & Schroeder-Heister 90].

4.2.2 CUT and The Quantifier Rules ∀E and ∃I

It is time to consider in detail the role played by parameters and unification in the process of constructing solutions. The story begins here and is continued, when the other two quantifier rules ∀I and ∃E are adopted.

A solution is a composition, by application of the CUT rule, of renaming instances of search components. Applications of ∀E and ∃I replace the bound variables of input formulae by parameters, so that only quantifier free atoms appear as premisses and conclusions of components. The cut principle requires that its premiss and conclusion formulae be syntactically identical. The new, more procedural version of the principle is expressed by the clause:

Given the two components $\dfrac{\Sigma_1}{B}$ and $\begin{matrix}(A)\\ \Sigma_2\end{matrix}$ and a substitution Θ such that $(A\Theta = B\Theta)$

then $\left(\begin{matrix}\dfrac{\Sigma_1}{B}\\ =\text{CUT}\overline{\overline{}}\\ (A)\\ \Sigma_2\end{matrix}\right)\Theta$ is a component.

The $\forall$E and $\exists$I rules allow for any term whatever to replace a parameter. Given the two quantifier free atoms A and B we want to find a substitution (of terms for parameters) Θ such that $A\Theta$ and $B\Theta$ are the most general syntactically identical substitution instances of A and B. That is, Θ is the most general unifier (mgu) of A and B.

Most General Unifier (mgu): The mgu Θ of the two atoms A and B is a set of equality assertions:

$$\{X_1 = t_1,\ X_2 = t_2,\ \ldots,\ X_n = t_n\}$$

Θ satisfies the following constraints:

Unifier: $A\Theta$ and $B\Theta$ are syntactically identical.

Most General: Any common substitution instance of A and B is also a substitution instance of $A\Theta$ $(B\Theta)$.

Solved Form: Each X_i is a distinct parameter that occurs in either A or B. Each t_i is a term containing parameters that occur in either A or B but none that occur as an X_i.

The above constraints enable efficient composition of mgu's, a question considered in detail in chapter 5. The computation of an mgu given A and B has been extensively studied since the pioneering work of [Robinson 65], see for example [Lassez et al. 88].

4.3 The Positive Edinburgh Prolog Language

The logic programming language Prolog developed in the proof theoretic context of resolution refutation for the Horn language. Prolog implementors have, however, recognised the relative simplicity of the deductive machinery required for a richer goal syntax. Two syntactic extensions, negated and disjunctive goals were admitted by the classic Edinburgh dialect [Clocksin & Mellish 81]. The negation as failure (NAF) extension is discussed separately in section 4.9. The disjunctive goals extension is taken up here.

4.3.1 The Or Introduction Rules $\vee\mathrm{I}$

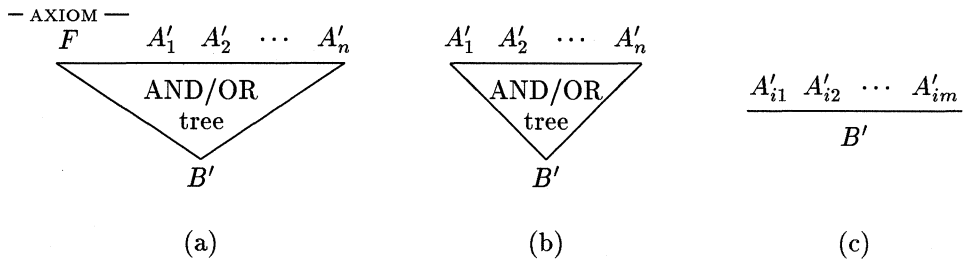
The sublanguage property states that just the introduction rules for the logical operators appearing in the goal syntax are required for a complete normal form deduction system. Thus to extend the deduction system $\mathcal{H}_\Sigma$ for disjunctive goals, we simply add the two or introduction rules of figure 4.4 to the existing rules.

$$-\vee\mathrm{I}\,\frac{G}{\underset{\Sigma_\mathrm{I}}{(G\vee H)}} \qquad\qquad -\vee\mathrm{I}\,\frac{H}{\underset{\Sigma_\mathrm{I}}{(G\vee H)}}$$

Figure 4.4: or introduction rules ($\vee$I)

The form of the inferential extensions for the extended syntax is illustrated in figure 4.5 (a) and (d). An inferential extension still consist of a single search component. The search component still has a single atomic conclusion. However, a search component may now contain OR branches, giving rise to multiple solution components. Each solution graph of the AND/OR graphs (b) or (e) is a derived rule of inference (c) or (f), featuring a subset of the atomic premisses.

Edinburgh Axiom F: $\forall\, A_1 \circ A_2 \circ \ldots \circ A_n \supset B$ (where: $\circ$ is $\wedge$ or $\vee$)



Edinburgh Query G: $\exists\, A_1 \circ A_2 \circ \ldots \circ A_n$ (where: $\circ$ is $\wedge$ or $\vee$)

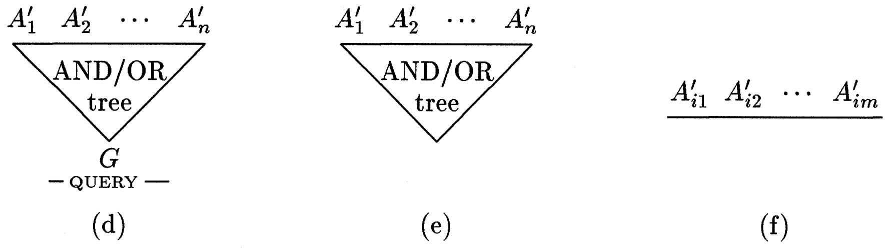


Figure 4.5: inferential extensions of Edinburgh axioms and queries

The procedural semantics of Prolog dictate that search component AND/OR trees be traversed left to right with backtracking to the most recent OR node on goal failure. For a more focused discussion on the relationship between logic programming and atomic normal form natural deduction see [Keronen 91].

4.4 The Positive Definite Language

In this section we consider the deductive machinery required for a full positive goal syntax. The universal quantifier introduction (∀I) and implication introduction (⊃I) rules are added to the Edinburgh system. This language is *definite* in the sense that disjunctive and existentially quantified assertion formulae are not admitted. The conclusion of a rule derived from a positive definite axiom is still an atomic formula.

4.4.1 The Quantifier Introduction Rule ∀I

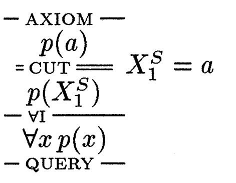
$$\frac{G(X^S)}{(\forall x G(x))}\,\forall\text{I} \qquad \Sigma_{\text{I}}$$

Figure 4.6: universal quantifier introduction rule (∀I)

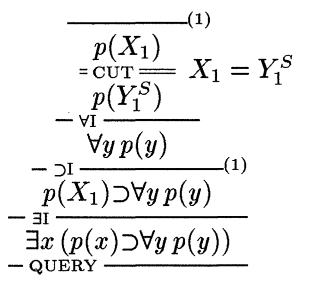
The universal quantifier introduction rule ∀I replaces the bound variable x in the goal $\forall x G(x)$ by a parameter X^S, resulting in the subgoal $G(X^S)$, see figure 4.6. The superscript S is used to distinguish the parameter generated by application of this rule as a *Skolem parameter.* Unlike the parameter generated by an application of ∃I, a renamed Skolem parameter X_i^S is subject to the following two constraints on its use:

Skolem Constraint: X_i^S is to appear literally in the solution. The mechanism to enforce this constraint is simply to treat the parameter as if it were a constant symbol, identical only to itself [Skolem 28]. That is, a Skolem parameter may only appear on the right hand side of any element $X_i = t_i$ of an mgu. As an example, the mgu in figure 4.7 (a) violates this constraint.

Dependency Constraint: X_i^S may not appear in any assumption on which $G(X_i^S)$ depends. Assumptions may be present once any of the rules ⊃I, ∼I, ∃E or ∨E are admitted. In general terms, enforcing the dependency constraint requires that mgu elements of the form $X_i = Y_j^S$ be checked to determine that the ∀I rule responsible for Y_j^S occurs low enough in the solution, so that all assumptions in which X_i occurs have been discharged. As an example, the mgu in figure 4.7 (b) violates this constraint.



(a)



(b)

Figure 4.7: Skolem parameter constraint violations

4.4.2 The Implication Introduction Rule ⊃I

The deduction problem Δ ?– $(F \supset G)$ is reduced by the implication introduction rule, shown in figure 4.8, to the problem $\Delta \cup \{F\}$?– G. That is, the antecedent F is an *assumption* or temporary axiom that may be used for the purpose of deriving the conclusion G only.

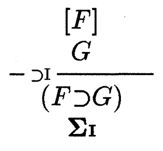


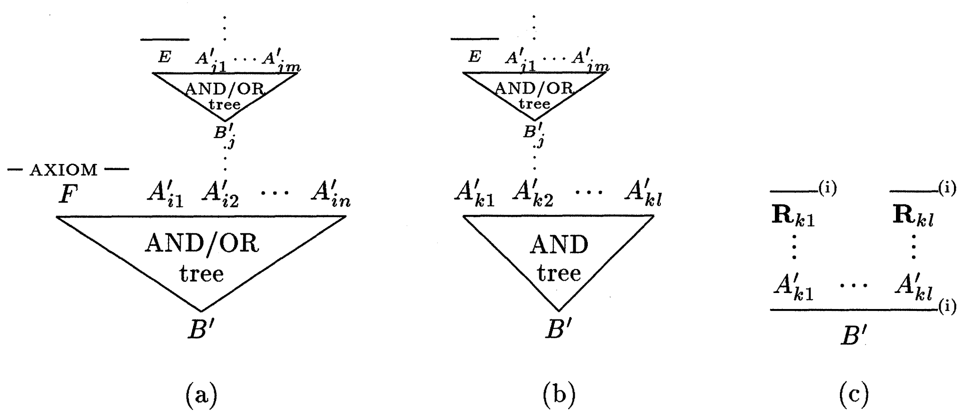
Figure 4.8: implication introduction rule (⊃I)

Inferential extension may now contain assumption search components arising from the antecedents of goal implications, as illustrated in figure 4.9 (a) and (d). For each inferential extension there is a set of derived rules of inference of the form (c) or (f). The intended reading of these rules is: For each premiss A_{kx} of the derived rule a set of derived rules $\mathbf{R}_{kx}$ is available as assumptions. This generalization of the notion of a rule of inference is explained in more detail in section 5.1.

Notice that the inferential extension of a formula still consists of search components with a single atomic conclusion. Hence the natural deduction formulation retains the definite character of the deduction problem. In contrast the resolution refutation proof theory is more severely affected. While any Edinburgh formula can be rewritten as a logically equivalent set of Horn clauses, once implications as goals are admitted we are outside Horn clause resolution. As an example, the axiom $(F \supset G) \supset H$ rewrites to the set of clauses $\{F \vee H, \sim G \vee H\}$. The multiple positive literals of the resulting clauses call for a full resolution refutation strategy [Chang & Lee 73].

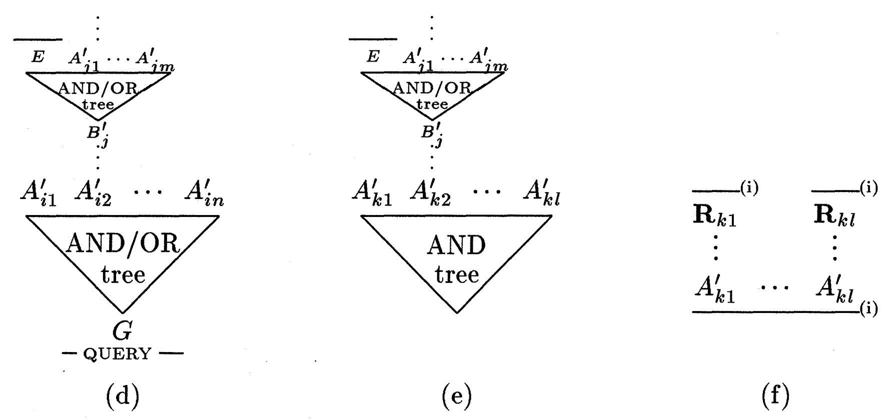
Though less severe than in the case of resolution refutation, there is still a computational price to be paid for the expressive power of implications as goals. The search process is complicated by the presence of assumptions. The set of search components

Positive Definite Axiom F: $\forall\, G \supset B$



E $A'_{i1} \cdots A'_{im}$
AND/OR tree
$B'_{.j}$
— AXIOM —
F A'_{i1} A'_{i2} $\cdots$ A'_{in}
AND/OR tree
B'
A'_{k1} A'_{k2} $\cdots$ A'_{kl}
AND tree
$\mathbf{R}_{k1}$ $\mathbf{R}_{kl}$
(i)
B'
(a)
(b)
(c)

Positive Definite Query G: any formula constructed using operators $\exists$, $\forall$, $\wedge$, $\vee$ and $\supset$



E $A'_{i1} \cdots A'_{im}$
AND/OR tree
$B'_{.j}$
A'_{i1} A'_{i2} $\cdots$ A'_{in}
G
— QUERY —
A'_{k1} A'_{k2} $\cdots$ A'_{kl}
AND tree
$\mathbf{R}_{k1}$ $\mathbf{R}_{kl}$
(i)
(d)
(e)
(f)

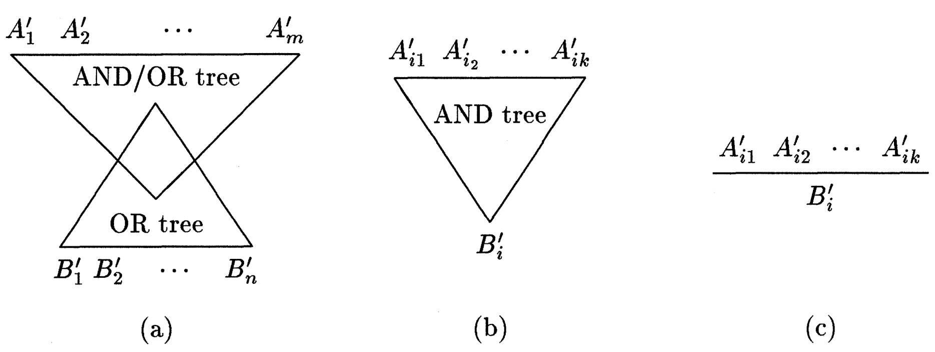
Figure 4.9: inferential extensions of positive definite axioms and queries

available for constructing the choice point for a given goal atom now depends on its *subgoal context.* Recall that this context is determined by the path from the query to the goal atom in question. This raises the following challenges for inference engine implementations:

- A choice point cannot be completely constructed until the path to the query is known. A simple approach to this problem is to employ backward chaining search in the **compose** phase of the inference engine.

- Efficient logic programming engines construct choice points, as far as possible at compile time. In the presence of implications as goals, such a mechanism needs to be extended to incorporate the lookup of search components from a tree structured database at run time.

- The various search components making up an inferential extension may share common parameters, as well as containing parameters to be renamed for each



A'_1 A'_2 $\cdots$ A'_m
AND/OR tree
OR tree
B'_1 B'_2 $\cdots$ B'_n
(a)
A'_{i1} A'_{i_2} $\cdots$ A'_{ik}
AND tree
B'_i
(b)
A'_{i1} A'_{i2} $\cdots$ A'_{ik}
B'_i
(c)

Figure 4.11: components and derived rule with $\wedge$E

4.5.2 The Existential Elimination Rule $\exists$E

The existential elimination rule $\exists$E, shown in figure 4.12, reduces an assertion $\exists x\, F(x)$ to the quantifier free assertion $F(X^S)$. Like the $\forall$I rule, this rule generates a Skolem parameter, subject to both the Skolem and dependency constraints.

$$\dfrac{\dfrac{\Sigma\text{E}}{\exists x F(x)}}{F(X^S)}\,{\scriptstyle \exists\text{E}}$$

Figure 4.12: existential quantifier elimination rule ($\exists$E)

As we moved from the orthodox formulation of natural deduction proof (chapter 2) to the more computational notion of a solution (chapter 3), we adopted new notation for existential elimination. Figure 4.13 (a) illustrates the orthodox notation for an application of existential elimination, and (b) our computational notation for the same. The transformation from the form (a) to the form (b) can always be performed, provided the existential elimination discharges its assumption. That is, the new notation does not permit vacuous applications of the rule. The new notation is also more convenient in connection with the AND/OR graph search paradigm.

Figure 4.14 displays an example solution using the orthodox notation (a) and the computational notation (b). As a disadvantage of the new notation, the assumption does not stand out as well here as it does in the orthodox notation. Figure 4.15 illustrates the need to carefully discharge assumptions and to check the dependency constraint to avoid unsound inference.

The discharge of the assumption is a simple deterministic operation. To ensure completeness one must discharge the assumption as high up in the solution graph as possible. A simple implementation may traverse down the solution, applying substitutions, until a formula occurrence that does not contain the Skolem parameter in

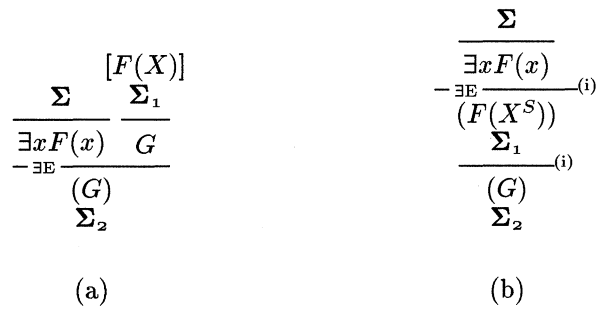


Figure 4.13: notation for existential elimination

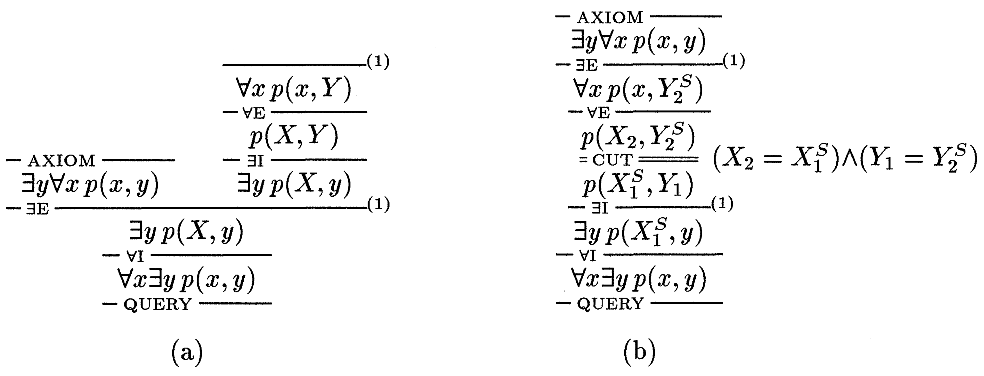


Figure 4.14: existential elimination representation example

question is encountered. A more efficient implementation would associate discharge requirements and capabilities with parameter occurrences in solution components to avoid the need for traversing the solution graph.

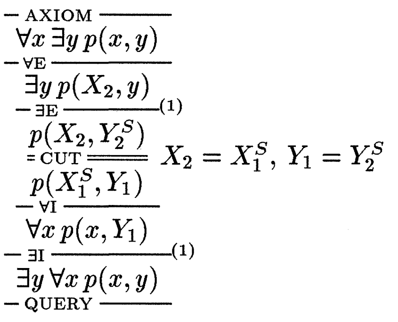


Figure 4.15: dependency constraint violation

4.5.3 The Or Elimination Rule $\vee$E

The or elimination rule $\vee$E of figure 4.16 may be read as: The assertion $E \vee F$ gives rise to two *possible worlds*, one contains E, the other F. More generally, n binary disjunctions give rise to 2^n possible worlds. Any goal formula G is to be demonstrated for all worlds. In the presence of disjunctive assertions, a solution for query G consists of a set of *case arguments.* Each case argument establishes G for a subset of worlds.

$$
{\scriptstyle -\vee\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{E\vee F}}{E \qquad F}
$$

Figure 4.16: or elimination rule ($\vee$E)

For the same reasons as in the case of the $\exists$E rule, we employ an alternative notation for the computational notion of or elimination. The transformation between applications of the orthodox natural deduction rule and our computational rule is illustrated in figure 4.17. Note that natural deduction (even in normal form) permits vacuous applications of the $\vee$E rule, but that such application cannot be expressed in the new notation. For a different perspective on multiple conclusion rules of inference see [Shoesmith & Smiley 78].

$$
{\scriptstyle -\vee\mathrm{E}}\dfrac{\dfrac{\Sigma}{E\vee F} \quad \dfrac{\begin{matrix}[E]\\ \Sigma_1\end{matrix}}{G} \quad \dfrac{\begin{matrix}[F]\\ \Sigma_2\end{matrix}}{G}}{\begin{matrix}(G)\\ \Sigma_3\end{matrix}}
\qquad\qquad
{\scriptstyle -\vee\mathrm{E}}\dfrac{\dfrac{\Sigma}{E\vee F}}{\begin{matrix}(E) & (F)\\ \dfrac{\Sigma_1}{(G)}{\scriptstyle (\mathrm{i})} & \dfrac{\Sigma_2}{(G)}{\scriptstyle (\mathrm{i})}\\ \Sigma_3 & \Sigma_3\end{matrix}}{\scriptstyle (\mathrm{i})}
$$

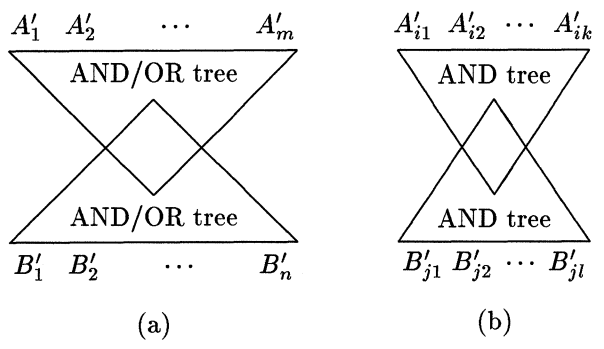
(a) (b)

Figure 4.17: notation for or elimination

In the presence of the $\vee$E rule, search components contain AND related atomic conclusions. We have now reached the most general form for search components, illustrated in figure 4.18.

The word "AND", used above to describe the relationship between conclusion atoms, is not totally satisfactory. It is true that a complete set of case arguments corresponds to a solution graph of the search space when just one disjunctive assertion occurs in the solution. There are two ways in which this model fails to reflect the application of disjunction elimination more generally:

- Any one case argument may use only one of the disjuncts from any one solution component instance. The example in figure 4.19 illustrates an unsound solution, resulting from a failure to observe this condition.



$$\frac{A'_{i1}\ A'_{i2}\ \cdots\ A'_{ik}}{B'_{j1}\ B'_{j2}\ \cdots\ B'_{jl}}$$

(c)

Figure 4.18: components and derived rules with ∨E

$\{\ a{\vee}b\ \}$?– $a{\wedge}b$

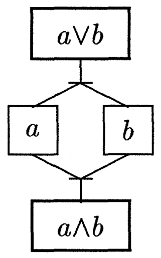


Figure 4.19: failure to separate cases

- A solution must contain case arguments to *cover* all worlds generated by disjunctive assertions. Figure 4.20 illustrates a failure on this count.

$$\left\{\begin{array}{l} a{\vee}b \\ c{\vee}d \\ (a{\wedge}c){\supset}f \\ (b{\wedge}d){\supset}f \end{array}\right\} \text{ ?– } f$$

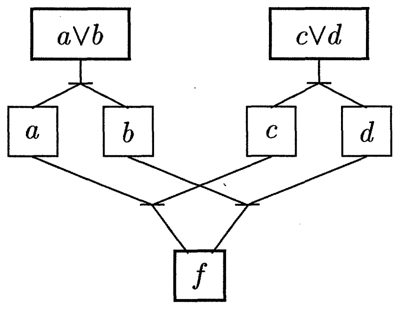


Figure 4.20: failure to cover all cases

The set of worlds to be covered by case arguments is generated as the cartesian product of sets of disjunctive conclusions. For the example of figure 4.21 there are the four worlds:

	a	b
c	w1	w2
d	w3	w4

There are three case arguments in figure 4.21. The leftmost case argument establishes w1 as inconsistent. The middle case argument concludes f for the world w2. The rightmost case argument concludes f for the two worlds w3 and w4. In contrast, the unsound case argument in figure 4.19 used more than one disjunct from an axis of such a diagram, while the set of case arguments in figure 4.20 failed to cover the two worlds $w2$ and $w3$.

$$\left\{\begin{array}{l} a\vee b \\ c\vee d \\ \sim(a\wedge c) \\ (b\wedge c)\supset f \\ d\supset f \end{array}\right\} \;?\!-\;\; f$$

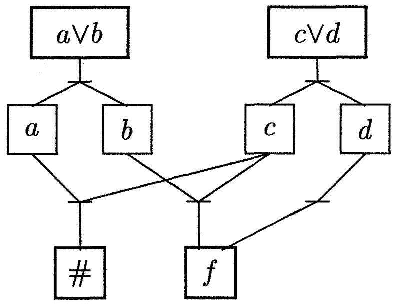


Figure 4.21: or elimination example

The above discussion suggests that we recognise the supervision of case arguments as a separate subtask for the inference engine. This supervisory level of the inference engine sets up the case argument context, being a set of single conclusion derived rules, and calls for a case argument search in that context.

4.6 Minimal Logic

The addition of the introduction and elimination rules for the negation operator ($\sim$) to the positive system results in a system for minimal logic [Johansson 36]. The elimination rule for negation constitutes a simple definition of the notion of *contradiction.* The subsequent use that is made of contradiction in deriving new conclusions is more controversial. The introduction and elimination rules for negation highlight the inadequacy of pure forward or backward chaining search strategies for **compose**.

$$\begin{array}{c} [G] \\ \# \\ \hline {\scriptstyle -\sim\mathrm{I}} \quad (\sim G) \\ \Sigma_{\mathrm{I}} \end{array} \qquad\qquad \begin{array}{cc} \dfrac{\Sigma_{\mathrm{E}}}{\sim F} & \dfrac{\Sigma_{\mathrm{I}}}{F} \\ \hline \multicolumn{2}{c}{{\scriptstyle -\sim\mathrm{E}}\quad \#} \end{array}$$

(a) (b)

Figure 4.22: negation introduction ($\sim$I) and elimination ($\sim$E) rules

4.6.1 The Negation Rules $\sim$I and $\sim$E

The elimination rule for negation, shown in figure 4.22 (b), detects a contradiction (#), given that both a formula and its negation have been established. The corresponding introduction rule, shown in (a), can be viewed as an amalgam of two principles:

Reductio ad Absurdum: A familiar method of argument to establish that a negation formula $\sim G$ holds is by demonstrating that a contradiction can be derived from the assumption G together with other current assumptions and axioms.

Absurdity Principle: Adherence to the semantics of classical logic demand that any formula whatsoever be derivable from a contradiction.

The reductio ad absurdum principle provides us with two points, the assumption G and the conclusion #, around which to construct a solution. Let us distinguish this as a new kind of deduction problem.

Relevant Deduction Problem: Given a set of *required axioms* Γ and a set of *ordinary axioms* Δ, is there a proof of G in the system S? In symbols:

$$\Gamma : \Delta \ \underset{\mathrm{S}}{?\!-}\ G$$

Any solution to the deduction problem

$$\Gamma \cup \Delta \ \underset{\mathrm{S}}{?\!-}\ G$$

that features every member of Γ as a premiss is a solution for the corresponding relevant deduction problem.

Neither the pure backward nor forward chaining search strategy makes full use of the constraints on premisses and conclusion. Notice that this point can also be made for the implication introduction rule. The absurdity principle is discussed in the next section.

4.7 Intuitionistic Logic

Intuitionistic deducibility requires an implementation of the absurdity principle for any goal formula, not just the negated ones. A first reading of the absurdity rule, shown in figure 4.23, might then be as a kind of introduction rule to be applied for all goal formula occurences.

$$\begin{array}{c} {}_{-\#\mathrm{x}}\dfrac{\#}{(G)} \\ \Sigma_{\mathrm{I}} \end{array}$$

Figure 4.23: absurdity rule (#x)

Given that the task is to find solutions to a deduction problem $\Delta \ ?\!- \ G$, application of the absurdity rule can, however, be reduced to the following cases:

- Check the consistency of the original problem theory Δ. In the case that an inconsistency is found all queries receive an affirmative answer, until the theory is repaired.
- Given the consistency of Δ, a contradiction may still be derivable in some subgoal or case argument contexts. For each additional assumption we call for a search for contradiction derivable using the assumption in question. Notice that this is another example of a relevant deduction problem.

Many (most) theorem prover implementations do not perform the first of the above checks, preferring to assume the consistency of Δ. An example of this is the set of support strategy for resolution refutation systems [Chang & Lee 73].

4.8 Classical Logic

A system for classical logic results if any one of the constructs shown in figure 4.24 is added to the intuitionistic system. These constructs are:

$$\begin{array}{cccc} \dfrac{}{F \vee {\sim}F} & \dfrac{\begin{array}{c}[{\sim}G]\\ \#\end{array}}{G} & \dfrac{{\sim}{\sim}G}{G} & \dfrac{\begin{array}{cc}[F] & [{\sim}F]\\ G & G\end{array}}{G} \\ \\ \text{(a)} & \text{(b)} & \text{(c)} & \text{(d)} \end{array}$$

Figure 4.24: excluded middle

(a): Axiom schema for *excluded middle,*

(c): rule of *classical reductio,*

(d): rule of *double negation.*

(b): rule of *dilemma,*

In the presence of any of these alternatives the subformula property is not strictly observed. To see this, consider the deduction problem

$$\{\}\ ?-\ (a \supset b) \vee (b \supset a)$$

There is no intuitionistic solution for this problem. The classical solution therefore must include at least one application of the excluded middle principle, and therefore a negated formula occurrence. No negated subformulae, however, occur in the statement of the problem. One of the possible classical solutions is shown in figure 4.26.

rules for moving in negations

$$\begin{array}{c}{\scriptstyle\sim\wedge\mathrm{I}}\dfrac{\sim G}{(\sim(G\wedge H))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad\begin{array}{c}{\scriptstyle\sim\wedge\mathrm{I}}\dfrac{\sim H}{(\sim(G\wedge H))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\wedge\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(E\wedge F)}}{\sim E\quad\sim F}$$

$$\begin{array}{c}{\scriptstyle\sim\vee\mathrm{I}}\dfrac{\sim G\quad\sim H}{(\sim(G\vee H))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\vee\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(E\vee F)}}{\sim E}\qquad{\scriptstyle\sim\vee\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(E\vee F)}}{\sim F}$$

$$\begin{array}{c}{\scriptstyle\sim\supset\mathrm{I}}\dfrac{G\quad\sim H}{(\sim(G\supset H))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\supset\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(E\supset F)}}{E}\qquad{\scriptstyle\sim\supset\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(E\supset F)}}{\sim F}$$

$$\begin{array}{c}{\scriptstyle\sim\forall\mathrm{I}}\dfrac{\exists x(\sim G(x))}{(\sim(\forall xG(x)))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\forall\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(\forall xF(x))}}{\exists x(\sim F(x))}$$

$$\begin{array}{c}{\scriptstyle\sim\exists\mathrm{I}}\dfrac{\forall x(\sim G(x))}{(\sim(\exists xG(x)))}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\exists\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim(\exists xF(x))}}{\forall x(\sim F(x))}$$

$$\begin{array}{c}{\scriptstyle\sim\sim\mathrm{I}}\dfrac{G}{(\sim\sim G)}\\ \Sigma_{\mathrm{I}}\end{array}\qquad{\scriptstyle\sim\sim\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim\sim F}}{F}$$

rules for negative literals

$$\begin{array}{c}(A)\\ \Sigma_{\mathrm{A}}\\ \hline \#\\ \hline\hline (\sim A)\end{array}{\scriptstyle=\sim\mathrm{I}=}\qquad\qquad{\scriptstyle\sim\mathrm{E}}\dfrac{\dfrac{\Sigma_{\mathrm{E}}}{\sim B}\quad\dfrac{\Sigma_{\mathrm{I}}}{B}}{\#}$$

$$\left(\begin{array}{cc}\hline\hline \multicolumn{2}{l}{\scriptstyle\mathrm{MX}}\\ (A) & (\sim B)\\ \Sigma_{\mathrm{A}_1} & \Sigma_{\mathrm{A}_2}\\ \hline G & G\end{array}\right)\Theta\quad\begin{array}{l}\text{where:}\\ A\Theta=B\Theta\end{array}$$

x	–	variable	G,H	–	goal formula
Θ	–	substitution	Σ_{I}	–	introduction component
A,B	–	atomic formula	Σ_{E}	–	elimination component
E,F	–	assertion formula	Σ_{A}	–	(partial) solution

Figure 4.25: extended rules for negation

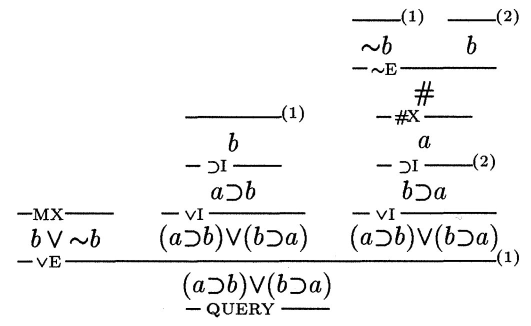


Figure 4.26: excluded middle example

The direct computational interpretation of any of the rules (b)–(d), as a kind of introduction rule for any formula whatever, suffers from serious combinatorial problems. Also, a literal implementation of alternative (a), that is the presence of all formulae of the form $F\vee{\sim}F$ as axioms, is not possible.

An incomplete implementation relying on recognising occurrences of complementary subgoal literals is suggested by consideration of the RGR rule of [Nilsson 80]. Well known equivalences of classical logic, enable any formula to be rewritten so that the negation operator applies only to atoms. This operation of *moving in negations* is part of the process for translating formulae into clausal form for resolution. These equivalences may be included in our system as rules of inference, as shown in figure 4.25. As a result of this extension the absurdity reasoning called for by negation introduction is confined to atoms, as indicated by the special ~I rule in the figure. The special MX rule may be applied whenever two complementary, unifiable atomic goals arise in distinct case arguments.

4.9 Alternative Languages

As seen above, classical and even intuitionistic natural deduction proof theories suffer from severe combinatorial problems. We place the blame for this on the following two features of these systems:

- The complexity of the search for contradictions.
- The complexity of applying excluded middle.

The Prolog family of logic programming languages avoids these problems. Negated goals but not assertions are admitted, removing contradictions altogether. Also, these languages are commonly not expressive enough to require application of excluded middle. For example the deduction problem $\{\}$?– $(a\supset b)\vee(b\supset a)$, discussed in the preceding section, cannot be expressed. Many proposed extensions of these languages, for example [Gabbay & Reyle 84], are based on intuitionistic logic, again avoiding the need for excluded middle.

Prolog has adopted the *negation as failure (NAF)* rule of inference [Clark 78] as the proof theoretic device for negated goals. The NAF rule of inference fits neatly, as a negation introduction rule, into a natural deduction framework. According to this rule, the deduction problem Δ ?– $\sim G$ receives an affirmative answer given that there is a failure demonstration, denoted by Σ_{F} in figure 4.27, for the problem Δ ?– G.

$$\begin{array}{rc} & \Sigma_{\mathrm{F}} \\ \cline{2-2} & G \\ {\scriptstyle =\mathrm{NAF}} & \overline{\overline{\quad(\sim G)\quad}} \end{array}$$

Figure 4.27: negation as failure rule (NAF)

An alternative to the Prolog approach is to still admit negated assertions, but to limit the search for contradictions. This can be done by rejecting the absurdity principle. The rejection involves removing the absurdity rule and requiring that other rules discharge assumptions. Conceived on the basis of philosophical objections to classical logic, the so called relevant (relevance) logics follow this scheme. The systems of [Anderson & Belnap 75] reject the absurdity rule, but retain excluded middle. The intuitionistic relevant logic of [Tennant 87] rejects both principles.

Chapter 5

Inference Engines

The search for solutions is viewed within two paradigms:

- deduction employing derived rules of inference
- AND/OR graph search

The derived rules of inference paradigm provides a simple conceptual model of the inference engine. AND/OR graph search, on the other hand, exposes issues relevant to efficient search. The impressive implementation technology of the logic programming language Prolog is examined. The extension of this technology to more expressive languages and full AND/OR parallel evaluation is considered.

5.1 AND/OR Graphs and Derived Rules of Inference

We began chapter 3 with a brief examination of search strategies in the context of the natural deduction rules of inference. The fact that this set of rules is systematic and fixed (for a given logic) enabled us to eliminate them, and replace each axiom and query by its inferential extension. In chapter 4 we saw that it is possible to think of the AND/OR graph, that is an inferential extension, as representing a set of derived rules of inference. Consequently, we now have two further perspectives on the search task:

Deduction employing derived rules of inference: Subgraphs of inferential extensions correspond to derived rules of inference. Hence, search can be viewed in the context of reasoning within a deduction system consisting of a set of such derived rules.

AND/OR graph search: The search space for solutions consists of AND/OR graph fragments (renamed search components) connected by CUT rule instances. A solution to the deduction problem at hand is a solution subgraph of the AND/OR search space.

A *derived rule of inference* is a generalisation of the notion of inference rule, as presented in chapter 2. The natural deduction rules are all instances of the schema shown in figure 5.1 (a). Such a rule leads from a set of premiss formulae $\{F_1 \dots F_n\}$ to a single formula G as conclusion. The rule may also discharge assumption formulae $E_1 \dots E_n$. In contrast, a derived rule, figure 5.1 (b), leads from a set of premiss formulae $\{A_1 \dots A_n\}$ to a set of conclusion formulae $\{B_1 \dots B_m\}$. The set of conclusions being read disjunctively. Further, the assumptions that may be discharged are not formulae but sets of derived rules $\mathbf{R}_1 \dots \mathbf{R}_n$.

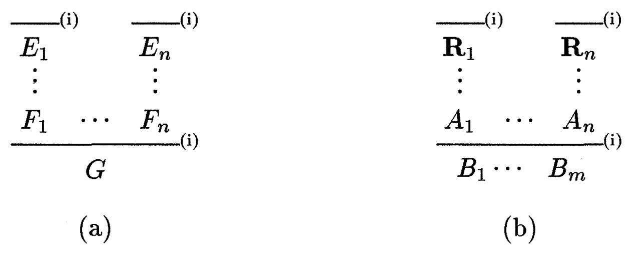


Figure 5.1: inference rule schemas

The set of derived rules produced by **extend** depends on the particular deduction problem statement at hand. Further, as the search proceeds, the set of available rules varies depending on subgoal and case argument context. The best we can do, a priori, is to distinguish the three subclasses of derived rules illustrated in figure 5.2. Rules drawn from each of these subclasses occupies a distinct niche in a complete solution:

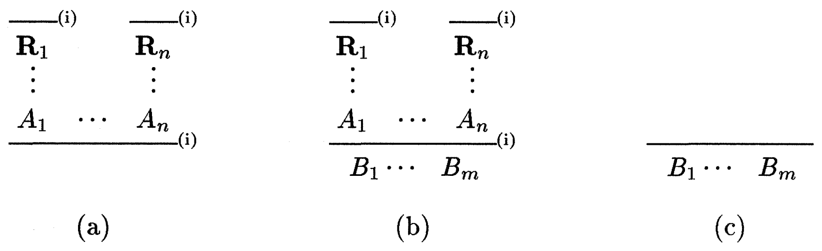


Figure 5.2: subclasses of derived rules

(a): A *query rule* has an empty conclusion, and is derived from the query formula. All paths in a solution are terminated at the bottom by a query rule instance.

(b): A *proper rule* is derived from an axiom or query that contains implications or negations as assertion subformulae.

(c): A *fact rule* has an empty set of premisses, and may be derived from an axiom or query. All paths of a solution are terminated at the top by a fact rule instance.

Deduction systems, such as the above, that include extended forms of inference rules have received some attention recently. [Shoesmith & Smiley 78] investigate the extension to rules with multiple conclusions. [Schroeder-Heister 84] argues that rules that discharge other rules as assumptions are a "natural extension of natural deduction".

The account of implementation techniques in this chapter relies on both the derived rules and AND/OR graph search paradigms. The AND/OR graph view is strong for many issues in search and representation. The derived rule view comes into its own when we wish to present a simple user view of the inference engine, see chapter 6.

5.2 Search as a Constraint Satisfaction Problem

We take the definition of *ANF solution graph* of section 3.4.2 as a starting point for the exploration of implementation issues. That definition consisted of the following five constraints on the form of a solution graph:

- Query Relevance
- Axiom Relevance
- Resolved Choice
- Substitution Consistency
- Loop Freeness

As a first step towards computational realization we read this definition as a constraint satisfaction problem. The remainder of this chapter deals with the problem of applying these constraints constructively to the task of finding solutions.

Traditionally inference engines apply either forward chaining (axiom relevance) or backward chaining (query relevance) elaborating a single partial solution at a time (resolved choice) while maintaining full substitution consistency for the current partial solution. The loop freeness constraint is often ignored. The Prolog inference engine, to be described a little later, conforms to these conventions, and provides us with a good point of reference. The following subsections examine the five constraints in more detail.

5.2.1 Axiom and Query Relevance

Much of the discussion in this chapter will assume a backward chaining search strategy. The factors that speak in favour of this approach are briefly these:

- As illustrated in section 3.2, the query relevance constraint is built into backward chaining strategies. Only those subgraphs of the search space containing the query are explored.

- In cases where the search space is too large to permit exhaustive, uninformed search, the generic backward chaining scheme can be specialised to incorporate control knowledge. Backward chaining is conceptualised as simple goal reduction with choice of subgoal and rule. This point is expanded in section 6.2.

- Backward chaining simplifies some implementation issues. For example the construction of complete choice points is possible, as the path to the query is always known. Implementation techniques developed in the logic programming context may be applied.

Backward chaining does not apply the axiom relevance constraint to limit search. This problem is painfully obvious in the case of relevant deduction problems when the conclusion is a contradiction, as is the case with reductio ad absurdum. Haridi [Haridi 81] suggests that forward chaining should be adopted for these kinds of subproblems. Note, however, that a forward chaining strategy does not make effective use of query relevance. What is called for is a search regime that is sensitive to all available constraints. A first step in this direction would seem to be to apply the relatively expensive substitution consistency constraint incrementally. For some suggestions in this direction see the work [Sickel 76] on clause interconnection graphs.

5.2.2 Resolved Choice

A spectrum of search strategies from depth first to breadth first is characterized by the number of partial solutions maintained at any one time. Common practice is to simplify implementation and maintain resolved choice by choosing the extreme depth first end of the spectrum. The full breadth first strategy, at the other extreme, is often impractical on combinatorial grounds. In section 5.5.3 we consider the implementation of strategies in the middle ground, enabling the concurrent exploration of a number of promising partial solutions. The following paragraphs set the scene in the context of backward chaining search strategies.

The resolved choice constraint requires the selection of a single CUT instance from each choice point. The term "backward chaining" refers to an abstract search process, leaving unspecified the order in which either choice points or CUT rule instances within them are to be selected. In the preceding sentence we have distinguished two kinds of choice:

Subgoal Choice: Select a subset of the subgoals for expansion from the current set of partial solutions. In other words, given a set of partial solutions, select a set of choice points. For sequential, one solution at a time implementations both of these sets are singletons.

Rule Choice: Given a subgoal, select the derived rules to be applied from the set of candidate rules. In other words, given a choice point, select a set of CUT rule instances.

The above conceptual model of the choices faced by a search algorithm is commonly found in accounts of the Prolog language. Prolog relies on the programmer to exploit his understanding of a fixed choice algorithm to control the search process. In section 6.2, encouraged by the success of this approach, we explore an alternative control paradigm based on the same conceptual model.

5.2.3 Substitution Consistency

A simple extension of the *composition of substitutions* operation of [van Vaalen 75] replaces a set of mgu's in solved form (see section 4.2.2) $\{\Theta_1, \Theta_2, \ldots, \Theta_m\}$ with a single mgu in solved form Θ, such that for any term t

$$((t\Theta_1)\Theta_2)\ldots\Theta_m \quad = \quad t\Theta$$

Composition of MGUs: Given a set of mgu's $\{\Theta_1, \Theta_2, \ldots, \Theta_m\}$ in solved form the following algorithm finds their composition if one exists and otherwise halts with fail status.

step 1: Let Θ be $\Theta_1 \cup \Theta_2 \cup \ldots \cup \Theta_m$. That is Θ is a set of equality assertions $\{X_1 = t_1,\ X_2 = t_2,\ \ldots,\ X_n = t_n\}$.

In order to reduce Θ into solved form repeat step 2 until no longer applicable.

step 2: Choose any pair of equality assertions $(X_i = t_i)$ and $(X_j = t_j)$, such that the two parameters X_i and X_j are identical. If the two terms t_i and t_j unify then replace the two assertions in Θ by the set of assertions that is the mgu of t_i and t_j, otherwise halt with failure status.

Return Θ as the answer.

Notice that the above operation is associative, implying that there is no restriction on the order in which unifiers from a solution graph are composed. The operation is also incremental, in the sense that applications of step 2 of the above algorithm may be postponed. These freedoms are often not exploited by implementors to improve performance. Composition of unifiers is typically the most expensive operation of an inference engine implementation.

As discussed in chapter 4, the quantifier rules ∀I and ∃E impose two further constraints on substitutions:

Skolem Constraint: This condition can be maintained by restricting Skolem parameters to appear only on the right hand side of mgu equality assertions.

Dependency Constraint: To maintain this condition it is necessary to check that any ∀I rule occurrences do not rely on undischarged assumptions. A very simple implementation can check the well formedness of a complete candidate solution.

5.2.4 Loop Freeness

The normal form for natural deduction proofs constrains the form of subproof for major premisses of elimination rules only. This leaves the door open for paths through minor premisses containing multiple occurrences of the same subproblem. In the case of a "no assumptions" language like Prolog a subgoal that is identical or subsumes a subgoal lower down on the path to the query signals the presence of a loop. In the case that assumptions are present, a sufficient additional condition for a loop is that the upper subproblem not have more premisses available to it than does the lower one.

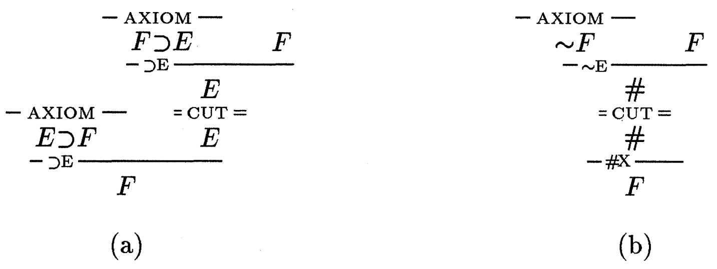


Figure 5.3: proof loops

The potential for loops exists when occurrences of the elimination rules with minor premisses, that is ⊃E and ∼E are present. Figure 5.3 illustrates the simplest loop elements. Notice that in the case of ∼E, the absurdity rule #X is also involved. The first step in minimizing the cost of loop detection is to perform a loop check only when multiple instances of components containing applications of these rules are present.

5.3 Implementation Technology

The purpose of this section is to establish a point of reference for inference engine implementation technology. Programs for solving deduction problems have been developed in a number of distinct settings:

- Resolution refutation theorem provers.
- Question answering systems for deductive databases.
- Logic programming language implementations.
- Expert systems inference engines.
- Verification of the correctness of programs and hardware designs.

Despite the various demands of the intended area of application, many current systems are based on a set of common elements:

- Backward chaining search is used to maintain query relevance. Some systems incorporate a carefully limited forward chaining preprocessor.
- A single partial solution is explored at any one time, with backtracking on failure or depth bound. An important exception is the processing of ground atoms by database operations.
- Substitution consistency is maintained by a unification algorithm together with clever representation schemes for terms instantiated by unification. Substitution consistency is fully maintained for the single current partial solution.

Conforming to the above model, the logic programming community has largely focused its attention on the development of implementation techniques for the Horn language. First, recall that Horn clauses are mapped into Horn rules by **extend**. Second, recall that case arguments are derivations consisting of instances of Horn rules only. These two relationships suggest that we adopt this work as a point of reference. The remaining sections of this chapter explore implementation issues from this perspective.

5.3.1 The Prolog Inference Engine

The following paragraphs point out the key elements of current logic programming systems implementations. We focus on the structure sharing implementation technology developed for the Prolog language. This section deals with the the pure Horn language only. This discussion is used as a starting point for a subsequent investigation of implementation techniques for more expressive languages. For more comprehensive descriptions of current techniques see for example [Aït-Kaci 90], [Bruynooghe 82] and [Campbell 84].

Prolog exploits the reading of sets of formulae as programming language procedure definitions. This *procedural semantics* of Prolog determines that the AND/OR search space be explored sequentially in left to right and depth first order. The stack based representation of computation state, developed for procedural programming language implementations, is used.

As an illustration, the partial solution of figure 5.4 (a) is represented by the data structures in (b). These data structures can be divided into a static and dynamic component:

Structure Store: The static structure of the Horn rules is kept here.

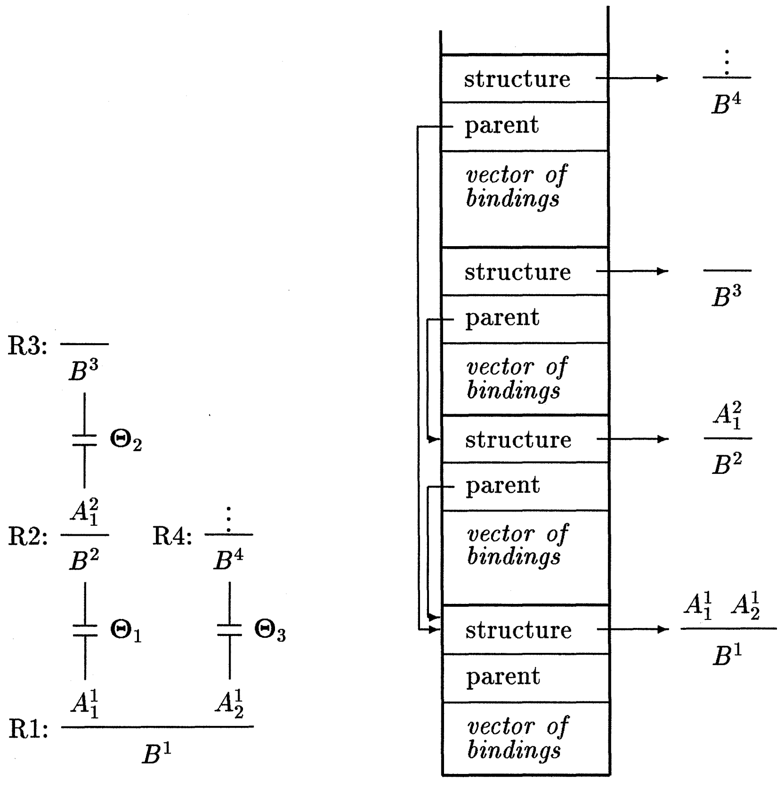
Stack: The structure and substitutions for the current (partial) solution are maintained as a sequence of stack frames.

The current (partial) solution tree is mapped onto the stack in chronological order. There is one stack frame for each occurrence of a Horn rule instance in the solution being represented. A stack frame consists of a pointer to the Horn rule structure, a pointer to a "parent" stack frame, a vector of bindings, as well as other information left out of the figure for clarity. The vector of bindings contains one entry for each distinct parameter occurring in the rule. The result of applying this vector as a substitution to the rule structure is the desired rule instance.

Each binding, an equality assertion of the form $X_i = t$, is represented by a binding vector entry. The renaming i is implicit in the context of the binding as part of a stack frame. The name X is associated with the vector offset. The term t is explicitly represented as a pair of pointers, one pointing to the term structure, the other to the stack frame where bindings for parameters occurring in the term structure are to be found.

The fact that only a single partial solution is represented at any one time implies that just a single set of bindings, being the composition of unifiers for the current partial solution, is required. The composition of unifiers is represented by the entire set of non-null bindings on the stack. In the case of the example of figure 5.4, the composition of the three unifiers Θ_1, Θ_2 and Θ_3 is represented by four binding vectors.

The above representation of the composition of unifiers has more structure than our textual representation as a set of equality assertions. The binding pointers form a graph consisting of a set of connected components. Each connected component corresponds to an equivalence class of terms. A pair of equivalence classes is unified by connecting



(a) (b)

Figure 5.4: stack data structure for single solution

two components. That is, the Prolog unifier applies the well known UNION-FIND algorithm (see for example [Aho, Hopcroft & Ullman 74] for a description of union-find). For future reference note that the unifier is free to apply the path compression optimization of UNION-FIND.

Figure 5.4 is a simplification. The structure of the current partial solution is represented, while the information required for conducting the search is left out. This information consists of choice points and a trail. Two kinds of choice points are recorded on the stack:

Rule Choice: This corresponds to the CUT choice point of our AND/OR graph model. Just a single pointer is needed to step through the sequence of rules.

Subgoal Choice: The antecedent of a Horn clause is a flat sequence of atoms. A single pointer is again sufficient to maintain the state of the left to right traversal of this sequence.

The trail is a chronological record of binding operations, consulted when undoing bindings on backtracking. The trail also fits in with the stack discipline.

Apart from the simplicity of the above scheme, the compilation of unification and choice points, together with clever indexing schemes, contribute to the efficiency of Prolog implementations.

5.4 Extended Logic Programming

A logic programming language consists of two sublanguages, as recommended by the slogan

$$\text{Algorithm} \quad = \quad \text{Logic} \; + \; \text{Control}$$

of [Kowalski 79^a]. We shall refer to these two component languages as the *problem language* and the *control language.* In this section we consider the prospect of extending the expressive power of these two languages as well as the issue of exploiting parallel hardware for solving deduction problems.

Let us first look at current proposals for extending the Prolog inference engine to deal with larger subsets of the full first order language as the problem language. It is instructive to consider goal and assertion syntax separately:

Goal Syntax: [Gabbay & Reyle 84] and [Bollen 88] have demonstrated implementations incorporating the implication introduction rule. Implementations of the full positive goal syntax, based on the transformations of [Lloyd & Topor 84], also exist [Thom & Zobel 88]. Even negated goals, implemented by the negation as failure mechanism, may be regarded as intuitionistic negation with respect to a completed program [Clark 78], [Shepherdson 88].

Assertion Syntax: Recall that the only operators admitted by the Prolog assertion syntax are $\forall$ and $\supset$. The author knows of no implementations that extend the inference engine to deal directly with enriched assertion syntax. Several meta interpreters have been proposed for asserted disjunctions and negations, see for example [Smith & Loveland 88].

In terms of the language hierarchy of chapter 4, direct implementations have not reached beyond the positive definite language $\mathcal{P}$. We suggest that the reason why logic programming systems do not offer the expressive power of the full assertion syntax are at least threefold:

Procedural Semantics: It is not clear what the procedural semantics should be once inferential extensions have more than a single atomic conclusion. Also, the procedural semantics of Prolog dictate a statically ordered backward chaining search, which is very inefficient in the case of relevant deduction subproblems.

Negation As Failure: Both disjunctive and existentially quantified assertions tend to "block" negation as failure results. The semantics of a language incorporating these constructs is not clear.

Proof Theory: No resolution refutation proof theories are known for languages intermediate between the Horn language and full first order classical logic. The natural deduction formulation now informs us what proofs in these languages look like.

Concerning the expressive power of the control language: The procedural semantics of Prolog determine a backward chaining, depth first, single solution search strategy. The knowledgeable programmer escapes these restrictions using meta programming techniques. That is, the control language is expressively inadequate for many application areas. Again, the procedural semantics blocks extension of expressive power.

Recently, the exploitation of parallel computing hardware has become a major focus for logic programming research, see for instance [Gregory 87], [Kacsuk 90] and [Wise 86]. The procedural semantics of Prolog enable the efficient implementation of the language on sequential machines. While some parallelism is available within this model, a fuller exploitation of parallelism cannot tolerate a sequential execution model.

Perhaps it is obvious by now that we propose an approach to logic programming, that does not rely on the procedural reading of formulae. Our aim is to unblock the development of more expressive languages and implementations that can exploit full AND/OR parallelism. The prospect of an efficient inference engine based on the AND/OR graph paradigm is explored in the next section. Top level language design issues are taken up in the next chapter.

5.5 Implementation Techniques for Extended Languages

The following subsections present refinements of the Prolog data structure to support implementation of more expressive and parallel languages. The AND/OR graph representation for solutions and search spaces suggests data structures for implementation of extended languages.

5.5.1 More Expressive Sequential Languages

We saw in chapter 4 that the increased expressive power of a language is reflected as an increase in the complexity of the inferential extensions of input formulae. The preceding discussion of Prolog implementation techniques assumed the Horn language, and consequently dealt with the corresponding simple Horn rule form of inferential extensions only. The question we address in this section is this: Can we extend the stack based scheme, with each stack frame representing a derived rule of inference, for the more expressive languages?

The generalisation of the implementation to the Edinburgh language is very simple. The only point of change concernes the subgoal choice pointer. For a Horn rule this pointer traverses left to right through a sequence of atoms. For the Edinburgh language this is generalised to a left to right, depth first enumeration of the solution trees of an AND/OR tree with atoms as leaves.

The next step up in complexity of inferential extensions is the presence of assumption search components. Recall that assumption components are generated by implications and logical negations as goals (the $\supset$I and $\sim$I rules of inference), present in the positive definite language. The set of inferential extensions available for the construction of a solution to a subproblem depends now on the context in which the subproblem occurs.

The necessary extension of data structures is illustrated in figure 5.5. A context pointer is allocated in each stack frame. A context being a set of assumption search components, each sharing some of their bindings with the stack frame that created the assumption. The representation of an assumption component in the structure store includes a list of common parameters the component shares with other components of the inferential extension. This information is needed to initialize binding vectors for the component's stack frame. Contexts are stored in a tree structured data base.

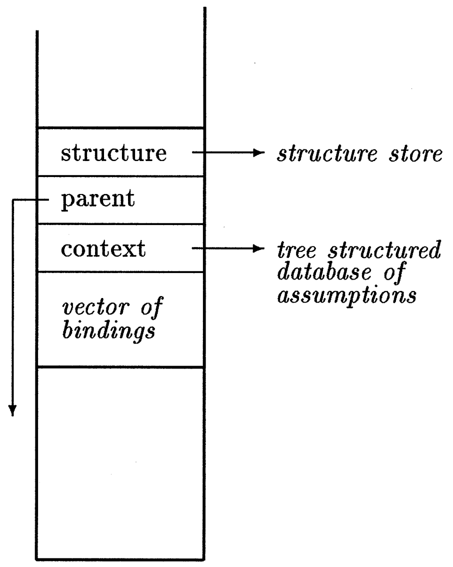


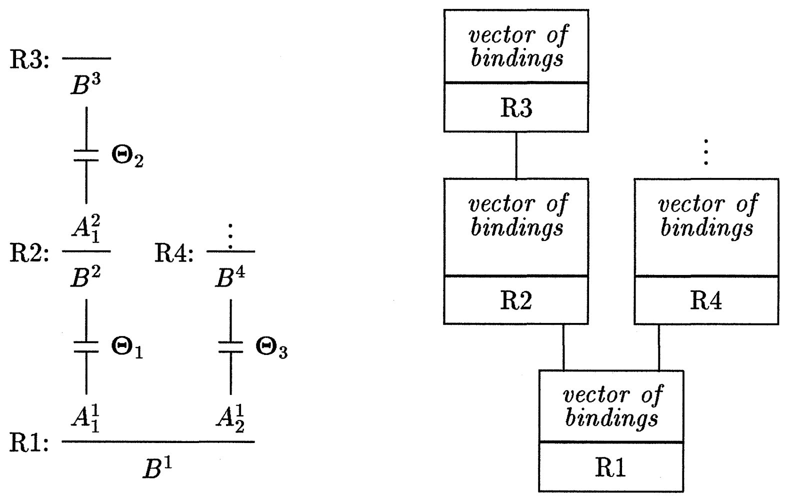
Figure 5.5: extended stack data structure

The most general form of search component contains an AND/OR tree of conclusions, as well as premisses. A stack based inference engine can still be used to construct the case arguments from which a solution is built. A case argument supervisor issues a sequence of calls to the inference engine, each resulting in either a case argument being returned or in failure.

The data structure illustrated in figure 5.5 is adequate for representing a case argument. The context mechanism, however, needs to be extended to provide a wider range of services. A non-empty initial context may be supplied to the inference engine to direct the search for a case argument. As well as a context of assumptions, a case argument is associated with an "anti-context" of assumptions. The anti-context is the set of assumptions not available in the world of the case argument.

5.5.2 Single Solution AND Parallelism

While appropriate for sequential implementations, a stack based representation cannot be maintained when a set of asynchronous processes co-operate to build up a solution. If we abandon the stack discipline, we arrive at the data structure displayed in figure 5.6. This data structure is a graph, maintained in a heap store, perhaps distributed across a number of processors. This AND graph data structure is also appropriate for implementations designed to avoid unnecessary recomputation of subgoals on backtracking.



(a) (b)

Figure 5.6: AND data structure for single solution

Notice that this data structure still represents just a single (partial) solution, and therefore can only support *single solution AND parallelism.* It is common in the literature to distinquish two forms of single solution AND parallelism:

Restricted AND Parallelism: This form occurs when the partial solution graph is extended concurrently at a number of subgoals that do not share variables.

Stream AND Parallelism: This form occurs when concurrent subgoals share variables.

The distinction shows up in our AND graph model in two ways: Firstly, concurrent access to shared bindings must be controlled to maintain the integrity of the composition of unifiers operation. Secondly, a bindings dependency analysis is required to determine the consequences of a failure on concurrent subgoals. For a more detailed discussion of AND parallelism see [Gregory 87].

5.5.3 Multiple Solutions AND/OR Parallelism

A further refinement of the data structure is required to represent multiple solutions. Let us suppose that the premisses A_1^1 and A_2^1, of the example in the preceding section, also unify with $R5$ and $R6$, as shown in figure 5.7 (a). Depending on the success of the composition of unifiers operation, there are from zero to four well formed partial solutions here. The data structure, shown in (b), represents the four candidate solutions.

The reader may have noticed already that, unlike in the simple motivational presentation of chapter 1, bindings are not associated with unifiers but with derived rule occurrences. The advantage of the current scheme is that the binding for any specified parameter is readily located at a fixed vector offset.

When multiple partial solutions are represented, a number of binding vectors may be associated with a single derived rule occurrence. That is, a rule occurrence in this data structure may stand for a number of distinct substitution instances of the rule.

The data structure of figure 5.7 stands for a set of candidate solutions. We need a second level representation to pick out the well formed (substitution consistent and loop free) solutions from among these candidates. The representation we propose here is a refinement of the ATMS labelling scheme introduced in chapter 1. The graph structure of a solution is represented explicitly by a label, while the composition of substitutions is not, as explained below.

The graph structure of a solution is uniquely determined by its set of binding vectors. Further, only the ambiguity of multiple binding vectors for the one subgoal needs to be resolved. For the example of figure 5.7, the structures of the candidate solutions are picked out by the labels shown in figure 5.8. Only labels corresponding to the well formed (partial) solutions are to be kept. Any label for a (partial) solution containing an inconsistent set of bindings or a loop (a *nogood*) is removed. Search effort should not be wasted on those portions of the AND/OR data structure that do not appear in any label. Many implementations would garbage collect such structures.

The set of non-null bindings associated with a solution can no longer be maintained

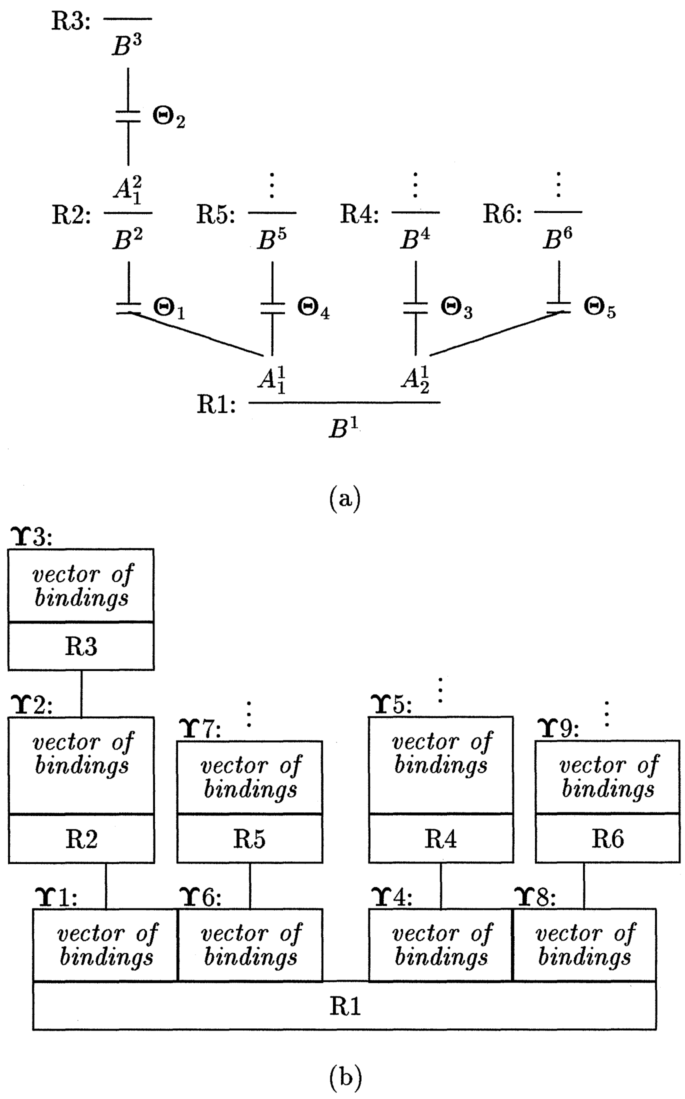


Figure 5.7: AND/OR data structure for multiple solutions

$$\{\Upsilon 1,\ \Upsilon 4\}$$
$$\{\Upsilon 1,\ \Upsilon 8\}$$
$$\{\Upsilon 6,\ \Upsilon 4\}$$
$$\{\Upsilon 6,\ \Upsilon 8\}$$

Figure 5.8: labels

as the composition of substitutions in solved form, since a number of bindings may exist for the one parameter. Also, the path compression optimization cannot always be applied. Two options for implementing the composition of substitutions are:

- Call on the unifier to determine the composed binding for a parameter dynamically. In this case, the degree to which the set of bindings approximates solved form is critical to performance.

- Maintain an explicit representation of the composition with the label. This may well be feasible when restricted to a critical subset of parameters.

The above description of AND/OR parallel evaluation omits discussing mechanisms to support backtracking search. The reason for this omission is that the author's experimental implementation work has focussed on the non-backtracking language described in the introduction. It has also assumed that only Horn rules are present. A comprehensive description of feasible implementation techniques for more expressive parallel languages has to wait on further experimental work.

Chapter 6

Exploiting the Representation

A natural deduction solution can be readily understood as an argument leading from a set of axioms, by way of simple principles of deduction, to the query. Atomic normal form extends this explanative power of natural deduction. The very detailed steps of reasoning are replaced by derived rules of inference, each justified by a particular input formula. This perspicuous representation can be exploited as follows:

- As a graphic display, it may be used for purposes of explanation, testing and debugging.
- Reflected as a theory accessible to introspection, it may be used for purposes of control and to meet other practical demands placed on the reasoner.

6.1 Visualization

Having presented a mechanical reasoner with a deduction problem $\Delta \;?\!-\; G$ and a finite amount of time for computation, we expect to receive as the answer a set (possibly empty) of proofs, together with an indication of whether this set contains all the proofs there are. Each of the proofs is to be a solution for the given deduction problem, that is they are proofs of $\Gamma \vdash G$ (where: $\Gamma \subseteq \Delta$).

In this setting, we can think of a proof as *explaining* which subset of the axioms, and by what methods of reasoning, lead to the conclusion G. The following two subsections treat explanation in this sense only. The aim is to display a solution in such a way that it can readily be grasped as an explanation. The third subsection extends this treatment to the display of partial solutions, for the purposes of testing and debugging. The aim being to observe the progress being made in covering the search space.

6.1.1 Flat Explanation

Given that some primitive principles of reasoning, their representation as rules of inference and the instantiation and composition of these rules are understood and accepted, a proof in any formal system may claim to explain its conclusion. In addition, the natural deduction rules claim to represent principles actually used when the most detailed account of an argument is presented by a human mathematician. Why don't we just present the user with the ANF natural deduction proof as explanation?

A weakness of the ANF form of natural deduction is illustrated in figure 6.1. The ANF solution for the deduction problem $\{a \wedge b\}$?− $a \wedge b$ is shown in (a), whereas the very simple solution in (b) is clearly better as an explanation. The atomization transformations (Lemma 3 in Chapter 2) can be applied in reverse to remove such unnecessary elimination–introduction pairs for any of the connectives.

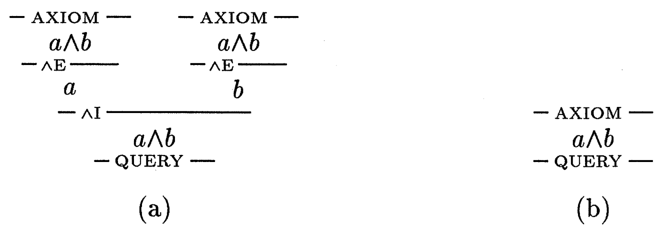


Figure 6.1: atomization example

The person reading the explanation is likely to be familiar with many sound rules of inference, which need to be derived when using the natural deduction rules. For example, the commutativity result, established in figure 6.2, may be displayed as in (b). Such transformations for the commutative and associative operators $\wedge$ and $\vee$ can significantly simplify the presentation of a solution.

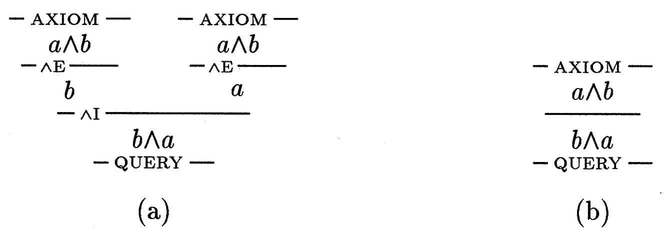


Figure 6.2: commutativity example

In chapter 1 resolution proofs were criticized on the grounds that refutations are not as perspicuous as direct proofs. Yet, the negation introduction rule (reductio ad absurdum) calls on a kind of refutation for the proof of a negated goal. Some applications of reductio may be removed by transformations. For example, the solution in figure 6.3 (a) may be simplified into the form shown in (b), being a single application of modus tollens. More generally however, the reductio rule remains.

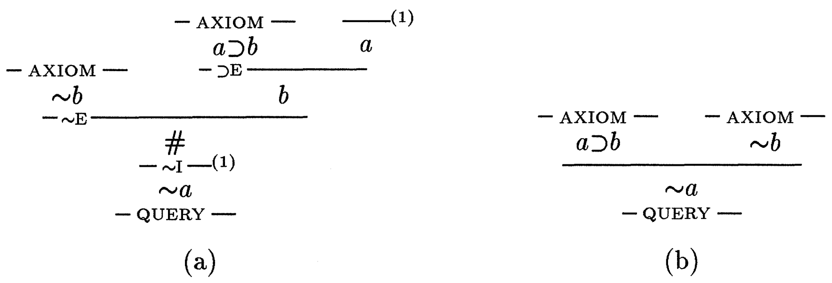


Figure 6.3: reductio example

In the presence of disjunctive assertions, a solution consists of a set of case arguments for the query. For such solutions, the case arguments may be presented separately. Recall, from chapter 4, that some of the cases may lead to absurdity, requiring a terminal application of the absurdity rule. The example of figure 6.4 (a) illustrates this complication. As in this example, some applications of or elimination may be presented as disjunctive syllogism, as shown in (b).

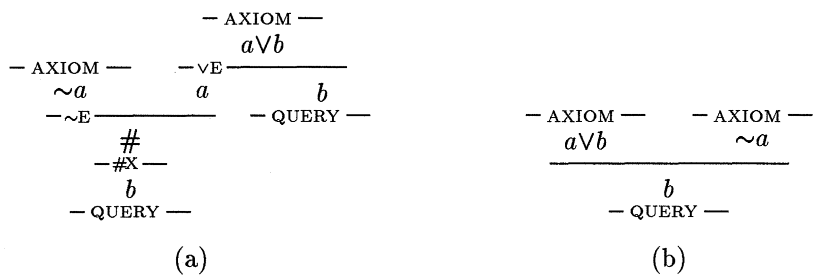


Figure 6.4: proof by cases example

Even after the above simplifications, natural deduction solutions for all but the most trivial problems are too large and detailed to have much more than a curiosity value. In the next subsection we exploit the notions of solution fragment and derived rule of inference to improve the situation.

6.1.2 Structured Explanation

It is possible to partition any given ANF solution into a set of fragments, each justified by a particular axiom or query. Alternatively, the solution can be seen as being composed of applications of derived rules of inference, again justified by a particular input formula. For these reasons, the ANF scheme can also be characterized as *input form natural deduction.* We now claim that this feature of the ANF schme extends the explanative power of natural deduction.

The structure of ANF solutions, as a composition of derived rules of inference or so-

$$\left\{\begin{array}{l}
alpinist(tony)\\
alpinist(mike)\\
alpinist(john)\\
likes(tony, rain)\\
likes(tony, snow)\\
\forall u\, alpinist(u) \supset (skier(u) \vee climber(u))\\
\forall v\, climber(v) \supset {\sim}likes(v, rain)\\
\forall w\, skier(w) \supset likes(w, snow)\\
\forall x\, likes(tony, x) \supset {\sim}likes(mike, x)\\
\forall y\, {\sim}likes(tony, y) \supset likes(mike, y)
\end{array}\right\}$$

Figure 6.5: Δ_{alps} — example problem theory

lution components, is not exploited by the flat explanations of the preceding subsection. The idea is to suppress the display of the detailed internal structure of these components. The mapping of formulae into derived rules of inference for the Horn language is a very simple one. Many current systems exploit this mapping implicitly for their explanation facilities. What are the issues raised by the more expressive languages?

Consider the problem theory Δ_{alps}, for the so called alpinist puzzle[1], of figure 6.5. As examples of structured explanations, figure 6.6 (a) and (b) offer solutions to the two deduction problems

$$\Delta_{alps} \; ?\!- \; \exists z\, {\sim}skier(z)$$

and

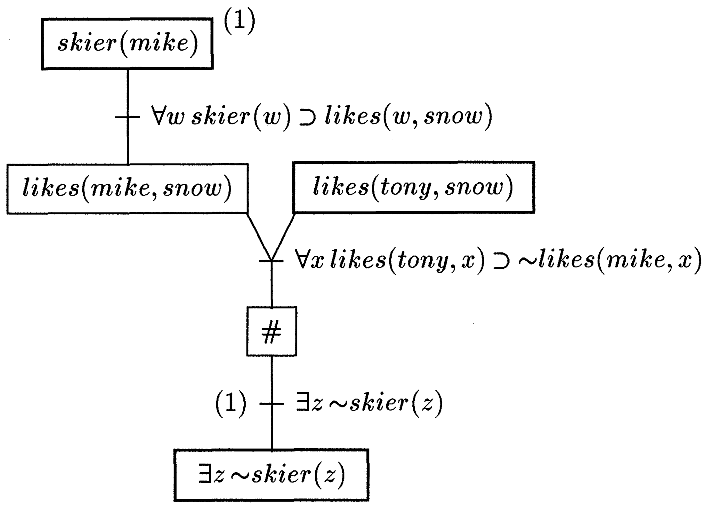
$$\Delta_{alps} \; ?\!- \; \exists z\, climber(z)$$

respectively. Each derived rule instance is displayed here as an inference stroke annotated with the input formula that justifies it. In the event that the derived rule involves assumptions (contains applications of $\supset$I or $\sim$I), both the inference stroke and the assumption are annotated with a unique number. In (a) the formula *skier(mike)* is such an assumption. The solution in (b) consists of two case arguments.

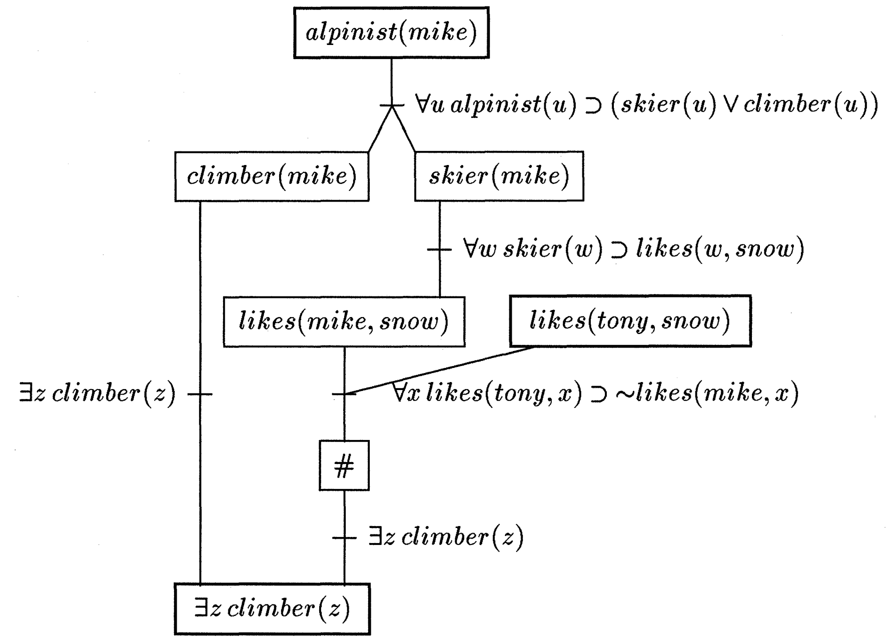
For existentially quantified queries it is often important that we be able to extract from the solution the so called *answer substitution.* For natural deduction solutions the answer substitution is simply a set of pairs, each pair $\langle x, t\rangle$, extracted from an occurrence of the existential introduction rule:

$$\frac{G(t)}{\exists x G(x)}\,{\scriptstyle \exists\mathrm{I}}$$

[1]This puzzle appeared in the comp.lang.prolog group of the internet news distribution.



(a)



(b)

Figure 6.6: structured explanations

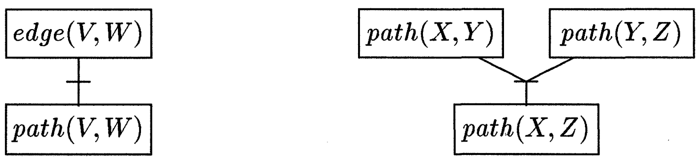
As illustrated by the examples in figure 6.6 the substitution is not always easily spotted in the structured display. For both examples the answer substitution is just $\{\langle z, \text{mike}\rangle\}$. Where the solution consists of case arguments, the answer substitution may differ between arguments. The answer substitution may even be absent, as in (b), where a case argument terminates in an application of the absurdity rule.

6.1.3 Testing and Debugging

In the event that an unexpected solution, caused by an erroneous axiomatization of the problem, is found, an explanation display can reveal the error. Where the solution is not found within acceptable time or fails to be found altogether, displays of the search space and partial solutions can be useful. The following discussion is limited to these issues only. For a thorough treatment of testing and debugging, in the logic programming context, see [Shapiro 83].

$$\left\{ \begin{array}{l} \forall v\,\forall w\, edge(v,w) \supset path(v,w) \\ \forall x\,\forall y\,\forall z\, path(x,y) \wedge path(y,z) \supset path(x,z) \end{array} \right\}$$

(a)



(b) (c)

Figure 6.7: Δ_{path} — path axioms

We will use the axiomatization Δ_{path}, shown in figure 6.7 (a), to illustrate the discussion in the remainder of this chapter. The *path*/2 predicate is intended to be interpreted as path in a directed graph. The example problems may also be read at the meta level — think of the directed graph as representing a search space for solutions. The two rules of inference derived from the axioms are shown in (b) and (c). In this section we diagnose a number of problems in applying these derived rules to solve problems. In section 6.2 we express meta knowledge needed to apply the rules more intelligently.

The search space generated by the two derived rules is illustrated by the *connection graph*[2] *display* of figure 6.8. Circuits in this figure represent recursive application of rules. Solutions are obtained by creating fresh renaming instances of nodes, "unrolling" such circuits. For the example problem the warning is clear — Rules need to be applied carefully to avoid wasted computation on *path*/2 subgoals, due to violation of the loop freeness constraint.

The normal form for natural deduction solutions does not prevent the construction

[2] Although the connection graph paradigm was developed by [Kowalski 75] for the resolution refutation proof theory, many of the ideas are equally applicable here. We go no further in this direction than to point out that this kind of display can be very useful in analysing the computational characteristics of a set of rules.

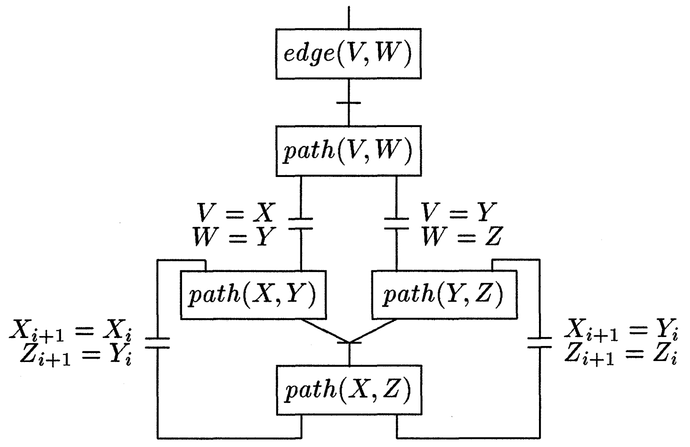


Figure 6.8: connection graph display for Δ_{path}

of multiple solutions for the one query relying on identical premisses. A set of axioms Δ_{chain}, and its intended model, for use in conjunction with Δ_{path} are shown in figure 6.9. Given the deduction problem

$$\Delta_{path} \cup \Delta_{chain} \text{ ?- } path(a, d)$$

two solutions, as shown in figure 6.10 (a) and (b), are possible. Once we notice that these two solutions rely on the same set of axioms, we are likely to be disappointed by this state of affairs. We will suggest remedies in section 6.2.

$$\left\{ \begin{array}{l} edge(a, b) \\ edge(b, c) \\ edge(c, d) \end{array} \right\}$$

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[r] & d \end{tikzcd}$$

Figure 6.9: Δ_{chain} — simple directed graph

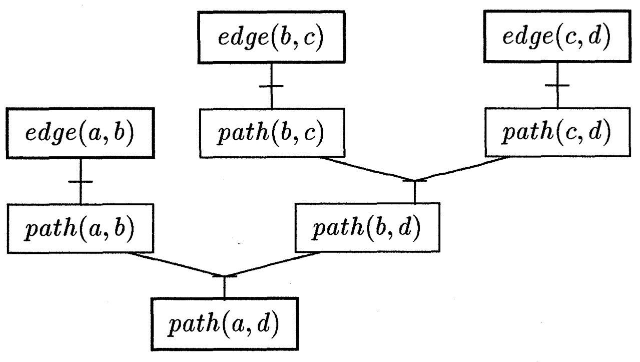
Even when the inference engine enforces the loop freeness constraint, as best it can, solutions may fail to appear when expected. Two kinds of failure are commonly distinguished:

Finite Failure: An inference engine can, at least in principle, complete the search in finite time without finding any solutions.

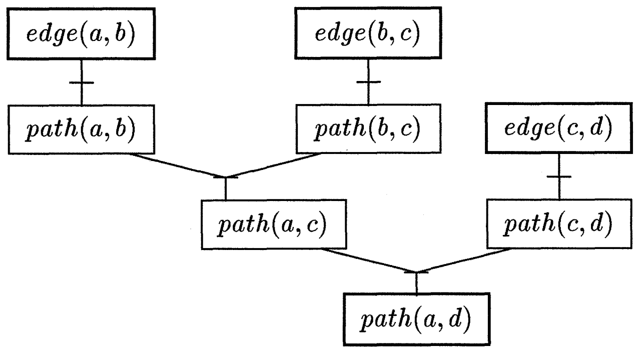
Non Termination: The search does not terminate within any finite interval of time.

From a pragmatic point of view we distinguish two kinds of finite failure:

Constructed Non-demonstrability: The inference engine completes the search within acceptable time without finding any solutions.



(a)



(b)

Figure 6.10: $\Delta_{path} \cup \Delta_{chain}$?– $path(a, d)$

Inefficient Search: Solutions are not returned within acceptable time, although in principle the search completes in finite time.

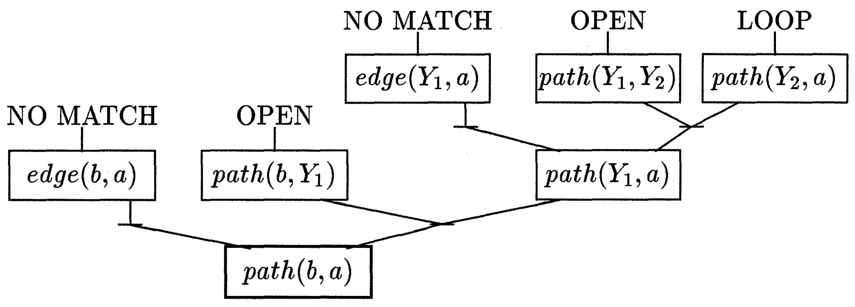


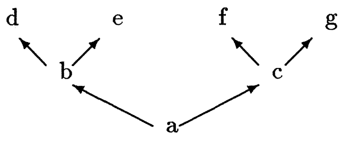
Figure 6.11: $\Delta_{path} \cup \Delta_{chain}$?– $path(b, a)$

As an example of constructed non-demonstrability consider the annotated fragment of search space, shown in figure 6.11, for the query $path(b, a)$, given the set of axioms $\Delta_{path} \cup \Delta_{chain}$. There are three partial solutions here, each of which incorporates, as a leaf, a goal atom A that is either:

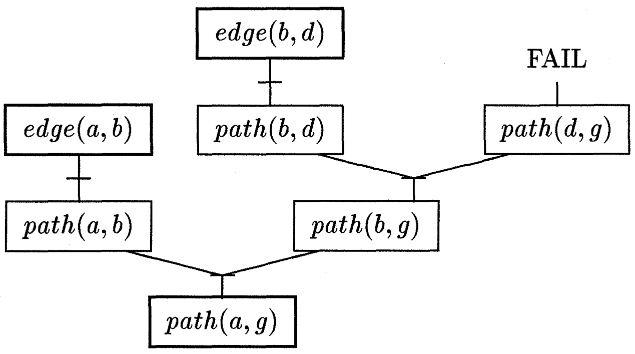
No Match: No derived rule of inference has a conclusion that matches A.

Loop: A subsumes another goal that occurs on the path from A to the query.

Notice again that care is required in selecting which goal to expand. Expansion of either of the two "OPEN" goals in the figure is wasted effort.



(a)



(b)

Figure 6.12: inefficient search for tree path problem

Both solutions and finite failure demonstrations represent the final state of a computation. In the presence of either inefficient search or non-termination we need to understand the progress of the computation in covering the search space. Consider the path problem for the tree form directed graph shown in figure 6.12 (a). The partial solution shown in (b) displays an intermediate state for a search that traverses the tree left to right bottom up. Clearly a more efficient regime for this problem would traverse down the tree, and if possible in parallel starting from the two end points.

6.2 Introspection

As well as the nominated purposes, the discussion in the first half of this chapter was intended to support the claim that search spaces, solutions and perhaps even the process of search can be readily conceptualized and understood. In the remainder we argue that such conceptualization and understanding can be harnessed to solve many of the problems that arise in practical applications of computational logic.

6.2.1 An Extended Introspective Architecture

Chapter 1 introduced the idea of introspection, and its application to the task of controlling the selection of subgoals on behalf of the object level inference engine. The application was described as a two level architecture, being:

Object Level: An axiomatization of the object problem domain used by an object level inference engine to construct solutions in response to deduction problems.

Meta Level: An axiomatization of the choice of subgoal problem used by a meta level inference engine to choose a subgoal in response to a query from the object level.

Figure 6.13 illustrates an extension of the introspective arhitecture, of chapter 1. This extended architecture is designed to allow control of performance critical activities by meta language assertions, as well as enabling the exploitation of parallel hardware. The current state of the search is maintained as an AND/OR graph in a *blackboard* [Waterman & Hayes-Roth 78] memory, accessible to inspection and change by a number of agents. A parallel implementation will need to support concurrent access to the blackboard.

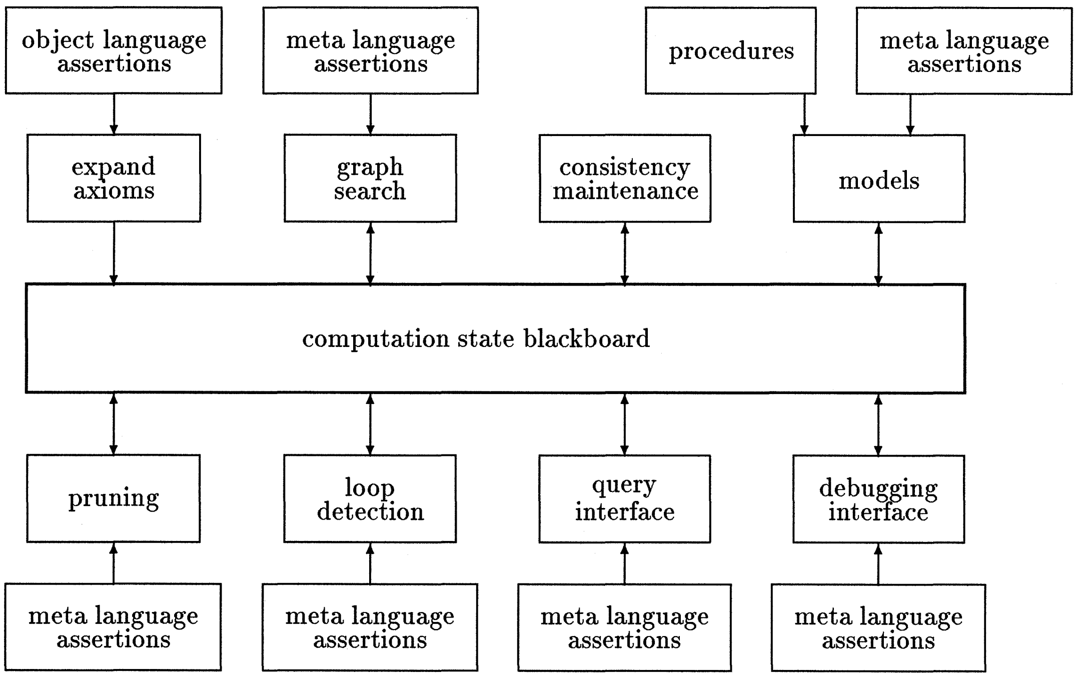


Figure 6.13: extended introspective architecture

For the subsequent discussion of this model, we once again rely on Prolog as a point of reference. A Prolog program is more than just a set of assertions about the problem domain. The assertions are organized as a set of procedures, each procedure consisting of a sequence of statements. A Prolog statement is more than just a logical formula. A statement is an expression constructed recursively as an operator applied to a sequence of expressions. Some of the operators may be read as logical connectives, others have only a procedural reading. Some primitive expressions may be read as atomic formulae of either the object or meta language with the remainder again having just a procedural reading.

The introspective architecture attempts to build on the successes of Prolog, while addressing its shortcomings. In broad terms, the issues are these:

Language: In Prolog knowledge about the problem domain is expressed in logic, while the knowledge that directs subgoal and rule selection, search space pruning, input/output etc. is not. Also, the failure to separate knowledge about the various domains can make it difficult to understand, modify and reuse programs.

We suggest that knowledge about each distinct domain be regarded as a distinct theory. The problem domain theory being axiomatized in the object language, the multiple other theories in the meta language.

Search: The range of available search strategies in Prolog is limited. The statically determined search strategy is not sensitive to the instantiation of parameters and other runtime context. Recent implementations feature a range of "meta predicates" in an effort to overcome this limitation. Also, search is restricted to a single solution at a time with chronological backtracking on failure.

In principle, the introspective architecture suffers from none of these limitations. The identification of practical alternatives is, however, a challenging problem. As a starting point for such an investigation, we can retain backward chaining and adopt the Prolog "meta predicates" as part of the meta language.

Parallelism: The sequential procedural semantics of a Prolog program locks away parallelism. Many attempts have, however, been mounted in an effort to identify useful non-sequential operational readings.

In contrast the introspective architecture of figure 6.13 suggests a parallel implementation, based on a set of co-operating processes.

Figure 6.13 represents just one intermediate point in a range of possible architectures for introspective computation. There is no a priori assignment of functionality between the object and meta levels. At the one extreme, every action is encoded as meta level assertions. At the other extreme, every action is performed by a monolithic object level inference engine. The former extreme gives total control of every action to the meta level assertions, while the latter provides none. While very efficient implementation techniques are known for object level engines, expressive power at the meta level is bought at a relatively high cost in computation speed. Experience with theorem proving and logic programming systems suggests a compromise, where the following issues are addressed by meta language assertions:

- Ordering the Search
- Detecting Loops
- Pruning the Search Space
- Negation As Failure
- Allocating Computational Resources
- Exploiting Models
- Specifying Communication

These issues are discussed in a little more detail in the remaining sections. We focus on meta language assertions of the form:

$$\textbf{condition} \supset \textbf{action}$$

Recall from chapter 1, that the **condition** is tested by introspecting the current computation state, while the **action** reflects down, specifying a computation to be performed. In the examples that follow, a **condition** is expressed in terms of a *goal/2* predicate, which picks out the atomic goal formula nodes in the current computation state. Some of the **condition** and **action** predicates are borrowed from the Prolog language, the remainder being proposed new constructs.

6.2.2 Ordering the Search

For any real machine the speed of computation is limited, the size of partial solutions is bounded by available storage and the number of concurrent operations is bounded by the number of available processors. These limitations imply that the order in which the search space is explored is often crucial to performance. This order may be specified declaratively by assertions in the meta language.

Often we have only limited knowledge (perhaps none) to bring to bear on the problem of deciding which, of a number of available expansions of the AND/OR graph, to pursue next. For a computation state for which the user supplied theory is mute, the choice may be determined by a default theory. If it turns out that the defaults lead to difficulties, the user supplied theory may be incrementally strengthened. We can go further and recognise a number of useful knowledge sources:

Catchall: A simple, uniform search strategy enables one to reliably predict the effects of overriding assertions. Prolog's left to right, depth first choice order is an example of such a catchall theory for a sequential implementation.

Static: Search advice computed from the static structure of the problem (analysis of connection graph for instance), can reduce the amount of overriding user supplied knowledge required for acceptable performance.

Dynamic: Search decisions may depend on an analysis of the dynamic behaviour of the system. Such "learned" strategies may further reduce the amount of user intervention.

User: The user may be in possession of knowledge about the intended interpretation of the problem axioms and the range of queries likely to be encountered. This knowledge may be put to use as search advice.

The kind of default theory determines, to a large extent, the kind of overriding assertions needed. For example, a depth first strategy is easily trapped by infinite branches, while space can quickly become a problem for a breadth first strategy.

As an illustration of the formulation of search advice, consider the path axioms Δ_{path} of figure 6.7. For this problem breadth first search is a reasonable catchall theory. A static analysis of the problem can reveal that the *edge*/2 relation is defined entirely by a set of atomic axioms. We may thus regard *edge*/2 goals as relatively tractable, and specify that they be selected whenever they occur. The syntax for this piece of advice might be:

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, edge(n_1, n_2)) \supset select(g)$$

If we know that Δ_{path} is to be applied to the kind of tree shown in figure 6.12 (a), we may specify preferential selection of *path*/2 goals that have an instantiated second argument.

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge ground(n_2) \supset select(g)$$

Many current logic programming languages provide constructs to suppress the selection of a goal atom that contains non-ground terms. The preferential selection of goals, illustrated above, is not available in any of the languages known to the author.

6.2.3 Detecting Loops

Advice about the conditions under which loops may occur and the frequency of checks may be specified by meta language assertions. The knowledge sources for a loop detection strategy may be diverse:

Catchall: A simple strategy is to check for a loop in the event that a predetermined depth bound is exceeded, or even, as a last resort, when memory space is exhausted.

Static: An analysis of circuits and the associated substitutions in the connection graph can identify potential loops.

User: The user may wish to override decisions derived from the above sources.

For the example Δ_{path} axioms, *path*/2 goals that do not have both arguments ground can recur as part of a loop. We might make use of this knowledge by specifying that loop checking be performed for such goals. The assertion might look like this:

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge (var(n_1) \vee var(n_2)) \supset loopcheck(g)$$

We propose that controlled loop detection, as illustrated above, be incorporated into logic programming systems.

6.2.4 Pruning the Search Space

Large portions of search space can often be removed by careful application of knowledge about the problem axiomatization and the current state of search. In Prolog such pruning is effected by use of the `!` (cut) and `once` constructs. We suggest that pruning be specified by assertions in the meta language.

Where a computationally expensive subproblem occurs more than once, an opportunity exists to reduce the size of the search space by sharing results. Whenever a new subproblem arises, two knowledge sources may be consulted:

Introspect: In the event that two identical subproblems are concurrently represented, results may be fully shared. Partial sharing may be possible when one of the subproblems subsumes the other.

Memorize: In the normal course of events, subproblems fail due to the *no match* or *loop* conditions, as was illustrated in figure 6.11. The failure of a subgoal implies the failure of any partial solution that incorporates that subgoal. In terms of the AND/OR graph paradigm, the failure propagates to siblings and the parent node at AND nodes and to the parent node at exhausted OR nodes.

> The space taken up by these data structures is normally reclaimed, making them inaccessible to introspection. The retention of crucial failure results may be specified by meta language assertions.

The checking of every new subgoal against these knowledge sources is likely to be infeasible. The user may identify subgoals to be checked by declarations in the meta language.

An opportunity for pruning the search space exists when duplicate solutions, like the ones illustrated in figure 6.10, occur. This situation is commonly referred to as *don't care nondeterminism* in the logic programming literature. For our path example, we might phrase the request for a single solution like this:

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge ground(n_1) \wedge ground(n_2) \supset once(g)$$

More generally, we can provide constructs for the arbitrary pruning of choice points. For the example path problem we may confine the search to proceed as a sequential left-to-right edge following search thus:

$$\forall g_1\, \forall g_2\, \forall n_1\, \forall n_2\, \forall n_3\, goal(g_1, path(n_1, n_2)) \wedge goal(g_2, path(n_2, n_3)) \supset chop(g_1, path/2)$$

The *chop/2* construct here is a generalization of the Prolog `!` (cut). In this case any element of goal g_1's choice point that would reduce the goal to a further *path/2* subgoal is removed.

Ideally, each meta language assertion that specifies pruning of the search space can be read as a theorem about proof search for the intended problem domain. Failure on this point results in the loss of solutions.

6.2.5 Negation As Failure

Our knowledge about a problem axiomatization Δ may include the fact that it is *complete for* a particular predicate p/n. That is,

$$\Delta \vdash p(a_1, \ldots, a_n)$$

if and only if $p(a_1, \ldots, a_n)$ is true in the intended domain, and

$$\Delta \not\vdash p(a_1, \ldots, a_n)$$

if and only if $\sim p(a_1, \ldots, a_n)$ is true. In this case the negation as failure (NAF) rule of inference

$$\frac{\Delta \not\vdash p(a_1, \ldots, a_n)}{\Delta \vdash \sim p(a_1, \ldots, a_n)}$$

is sound. Recall that in section 4.9 we suggested that the deduction system for the object language could be extended to include an inductive definition of the notion of a failure demonstration. We now propose that negation as failure reasoning be applied whenever it is sound, and reductio reasoning otherwise to answer negative goals.

The knowledge that the axiomatization of the example *path*/2 predicate is complete might be expressed in the meta language like this:

$$\forall n_1\, \forall n_2\, complete(path(n_1, n_2))$$

Within the first order language the knowledge of completeness of predicate p/n may be expressed by the syntactic transformation of completing the axiomatization for p/n. Clark [Clark 78] introduced this transformation for the Horn language. Although this work generalizes easily to the positive definite language, the extension to disjunctive assertions is more problematic. As an example of the completion transformation see the axiomatization Δ_{comp} of figure 6.14, being the result of completing $\Delta_{path} \cup \Delta_{chain}$. Arguably the completed axiomatization is less readable and modular than the original.

$$\left\{ \begin{array}{l} \forall v\, \forall w\, edge(v,w) \equiv ((v=a) \wedge (w=b)) \vee ((v=b) \wedge (w=c)) \vee ((v=c) \wedge (w=d)) \\ \forall x\, \forall y\, \forall z\, path(x,z) \equiv edge(x,z) \vee (path(x,y) \wedge path(y,z)) \end{array} \right\}$$

Figure 6.14: Δ_{comp}: completion of $\Delta_{path} \cup \Delta_{chain}$

A solution for the query $\sim path(b,a)$ for the completed axiomatization is shown in figure 6.15. The form of the solution, a set of case arguments embedded in an application of reductio ad absurdum, is the expected response for negated queries from completed axiomatizations. Recall that a backward chaining search strategy is not well suited to the task of finding such solutions.

Comparing the reductio solution with the finite failure demonstration of figure 6.11 we note that: The failure demonstration is simpler and therefore likely to be more readily understood as an explanation. Further, the failure demonstration appears as a subgraph of the reductio solution. We conjecture that this is the case generally, and that a procedure for translating failure demonstrations into reductio solutions is feasible.

6.2.6 Allocating Computational Resources

Meta language assertions may address the problem of allocating limited computational resources:

Time: The user may wish to impose a time limit on a computation, or perhaps specify a time dependent search strategy.

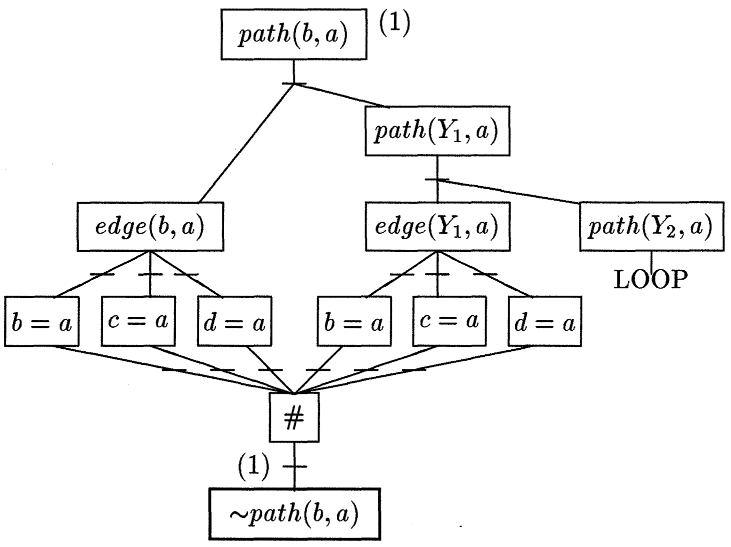


Figure 6.15: Δ_{comp} ?– $\sim path(b, a)$

Space: Once memory is exhausted, rollback may be specified for the less promising partial solutions. For distributed memory implementations, advice for memory allocation may be given.

Processors: Concurrent AND/OR search can exhibit genuine superlinear speedup. In practice worthwhile computational tasks need to be identified and allocated to the various processors with care.

As an example consider Δ_{path} with an arbitrary directed graph. We may wish to specify concurrent search by two processes, working in from the two endpoints of any given goal path.

$$\begin{aligned}
&\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge ground(n_1) \wedge ground(n_2) \supset \\
&\quad \exists p_1\, \exists p_2\, process(g, p_1) \wedge strategy(p_1, LeftToRight) \wedge \\
&\qquad process(g, p_2) \wedge strategy(p_2, RightToLeft)
\end{aligned}$$

The two search strategies *LeftToRight* and *RightToLeft* are simple variants of the edge following search illustrated in section 6.2.4 above. Once either of the processes reaches a decision, the other may be terminated on the grounds of duplication. This may be achieved by using the *once/1* construct, also discussed in section 6.2.4.

6.2.7 Exploiting Models

Models for a problem domain can be used to speed up computation in two ways:

Counterexamples: In his pioneering work, [Gelernter 59] used diagrams as counterexamples for proposed theorems of geometry. This idea generalizes to models for any domain. Such testing against models may be specified in the meta language.

Procedural Attachment: Efficient algorithms are known for many problems. As an example, many arithmetic functions are commonly provided for directly in machine hardware. Such procedural attachment may be specified declaratively.

For the example path problem, it may be the case that even carefully controlled deduction does not yield acceptable performance. As a last resort, we can write a procedure, call it *PathFinder*, to decide these goals. We then specify a procedural attachment in the meta language.

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge ground(n_1) \wedge ground(n_2) \supset attach(g, PathFinder)$$

6.2.8 Specifying Communication

The relationship between the computation state and any interaction with the system's environment may be specified by assertions in a meta language. The facilities that may be provided include:

Read/Write: Prolog programmers have found it useful to embed various input and output requests in their programs. We can specify that a given input/output **action** take place once the computation state satisfies a given **condition**.

Debugging: The idea of declarative debugging can be realized in the introspective framework. A debugging **action** is specified to occur in response to the given **condition** being met by the current computation state.

Carelessness in pruning the search space may result in unexpected failures. In the case of our example path problem we can attempt to diagnose the problem thus:

$$\forall g\, \forall n_1\, \forall n_2\, goal(g, path(n_1, n_2)) \wedge ground(n_1) \wedge ground(n_2) \wedge failed(g) \supset display(g)$$

The *display/1* construct will generate a failure demonstration display, such as the one illustrated in figure 6.11.

Chapter 7

Conclusion

The formalization of the notion of a logically sound argument, as a natural deduction proof, offers the prospect of a computer program capable of constructing such arguments in response to queries. We have presented a constructive definition for a new subclass of natural deduction proofs, called atomic normal form (ANF) proofs. We have argued that this is the right framework for mechanical reasoning on both proof theoretic and computational grounds.

7.1 Proof Theory

ANF is a well motivated normal form for natural deduction. In chapter 2, we demonstrate that ANF proofs form a deductively complete subclass of the normal form proofs of [Prawitz 65]. In subsequent sections we propose that both these normal forms be strengthened as follows[1]:

No Vacuous Applications of Inference Rules: Every occurrence of the existential elimination, or elimination and negation introduction rules must discharge assumptions. See sections 4.5.2, 4.5.3 and 4.6.1.

Absurdity Rule: The absurdity rule may only occur as the terminal rule application for:

- the entire deduction,
- a case argument (minor premiss of disjunction elimination),
- premiss of implication introduction

See section 4.7.

[1]These remarks address the intuitionistic and classical systems. Some modifications are required for minimal logic and other subsystems.

Discharging Assumptions: Assumption discharge is to occur as early in the proof as is permitted by the discharge constraints. See sections 4.4.1 and 4.5.2.

Loop Free: The proof must be loop free. See section 5.2.4.

These additional constraints do not affect what is deducible in the intuitionistic or classical systems. Further, any proof that does not observe these constraints, violates the claim:

> "A deduction in normal form proceeds from the assumptions of the deduction to the conclusion in a direct and rather perspicuous way without detours" — [Prawitz 65] p 8.

We therefore submit that there is a need for a *strong normal form* for natural deduction, and that these constraints be incorporated.

In section 3.4.1 we propose that, for the purpose of deduction, an assertion or query formula be represented by its inferential extension. Further, each inferential extension may be read as a set of derived rules of inference. These derived rules take on an interesting form that incorporates the extensions of [Shoesmith & Smiley 78] and [Schroeder-Heister 84], as described in section 5.1.

The notion of constructed non-demonstrability, introduced in section 6.1.3, is an important contribution of logic programming research to proof theory. In section 6.2.5 we conjecture that failure demonstrations can be translated to reductio proofs.

7.2 Languages

A wide range of languages and logics are available as natural deduction systems. In chapter 4, we present a spectrum of subsystems of the classical first order calculus. For these systems the ANF formulation exhibits a simple tradeoff between the expressive power of the language in which a problem is expressed and the deductive machinery required to solve that problem. This analysis offers simple, natural deduction based accounts for many current logic programming languages. It also reveals the deductive machinery required for the implementation of more expressive logic programming languages.

In chapter 4 we also raise possible objections to the application of classical principles of reasoning in automated theorem proving:

Excluded Middle: The rejection of excluded middle distinguishes the intuitionistic from the classical reasoner.

Absurdity Rule: Rejection of the absurdity rule is required for coherent reasoning in the presence of contradictions.

The application of these principles can be computationally extremely expensive. This point is implicitly acknowledged by the many mechanical reasoners that fail to implement them. Much more research on computationally tractable logics in this neighbourhood is required.

7.3 Computation

ANF inference engines make use of well known computational techniques. We introduce the computation, as AND/OR graph search, in section 3.3.3. An alternative view of the computation, as deduction employing derived rules of inference, is presented in section 5.1. The fundamental operation of the ANF inference engine is the unification of two atomic formulae, see sections 4.2.2 and 5.2.3.

Chapter 5 investigates the application and extension of logic programming implementation technology for ANF inference engines. The application of truth maintenance techniques is developed in sections 1.8 and 5.5.3.

The exploitation of parallel computing hardware for logic programming is an area of much current research. The AND/OR graph search model is related to the popular AND/OR process model of [Conery 83]. Implementation data structures, based on the AND/OR graph search model are analysed in section 5.5.3. The introspective architecture, described in section 6.2, is designed to support parallel evaluation.

Our investigation is confined to the classic forward and backward chaining search strategies. As pointed out in section 5.2, such strategies do not constitute the best possible use of all the available search constraints. Relevant deduction problems, are particularly poorly served by these strategies. Work is needed to identify more appropriate search strategies for these problems.

The representation of arguments as natural deduction proofs provides a good foundation for research on efficiency gain by emulating human reasoning abilities. A characteristic of human reasoning in a particular domain is the incremental accumulation of reasoning expertise for problems in that domain. Two related aspects of this expertise are:

Lemmas: derived rules of inference, carefully selected for their expected utility in solving problems.

Analogy: the recognition of a class of problems which may be solved by the instantiation of a common proof schema.

7.4 Visualization

The visualization of proofs, failure demonstrations and search spaces is considered in section 6.1. A natural deduction proof can be understood as an argument that leads from a set of premisses, by way of simple rules of inference, to the conclusion of interest. ANF extends this explanative power of natural deduction. The argument may be presented in terms of derived rules of inference, each justified by a particular input formula.

In section 6.2 we propose that control and other pragmatics be formulated as deduction problems at the meta level. An advantage of this approach is that the work on visualization can be carried over to explanation, testing and debugging of these meta level functions also.

Visualization of the process of search is discussed only very briefly. Much more work is needed to identify useful schemes here.

7.5 Introspection

We present an introspection based architecture for ANF inference engines, see sections 1.7 and 6.2. The architecture is aimed to exploit both parallel computing hardware and the perspicuous natural deduction representation of reasoning to overcome the combinatorial and other practical problems faced by computational logic applications. The model represents an extension of the schema [Kowalski 79^a]

$$\text{Algorithm} \quad = \quad \text{Logic} + \text{Control}$$

The new schema looks something like this

$$\begin{aligned}\text{Algorithm} \quad = \quad & \text{Deduction Problem} + \\ & \text{Search Control} + \\ & \text{Resource Allocation} + \\ & \text{Computational Models} + \\ & \text{Input/Output}\end{aligned}$$

Each of the components on the right hand side represents a distinct logical theory. The deduction problem consists of a set of axioms and a query formula expressed in an object language. The remaining theories are expressed as sets of axioms in a meta language.

Our experiments have been confined to a Horn meta language and the simple conceptualization of the computation state as a frontier of atomic goals. The conceptualization and some of the meta predicates were borrowed from current logic programming languages. Even within this restricted framework we were able to identify several useful new constructs.

The efficient implementation of meta level inference is crucial for achieving acceptable performance for implementations of the architecture. The introspective model is based on the notoriously expensive operations of pattern matching, associative recall and logical deduction. More work is needed to determine the practicality of this approach.

Bibliography

[Aho, Hopcroft & Ullman 74]

Alfred V. Aho, John E. Hopcroft and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[Aït-Kaci 90]

Hassan Aït-Kaci. *The WAM: A (Real) Tutorial.* Report No. 5, Digital, Paris Research Laboratory, 1990.

[Anderson & Belnap 75]

Alan R. Anderson and Nuel D. Belnap. *Entailment: the logic of relevance and necessity.* Princeton University Press, 1975.

[Apt & vanEmden 82]

Krzysztof R. Apt and M. H. van Emden. *Contributions to the Theory of Logic Programming.* Journal of the Association for Computing Machinery, 29(3) 841–862, 1982.

[Bollen 88]

Andrew W. Bollen. *Conditional Logic Programming.* PhD Thesis, Department of Computer Science, Australian National University, 1988.

[Bowen & Kowalski 82]

Kenneth A. Bowen and Robert A. Kowalski. 'Amalgamating Language and Metalanguage in Logic Programming'. in K. L. Clark and S-Å. Tärnlund (eds). *Logic Programming.* Academic Press, 1982.

[Boyer & Moore 72]

R. S. Boyer and J. S. Moore. 'The Sharing of Structure in Theorem-proving Programs'. in B. Meltzer and D. Michie (eds). *Machine Intelligence Vol 7.* Edinburgh University Press, 1972.

[Bruynooghe 82]

Maurice Bruynooghe. 'The Memory Management of Prolog Implementations'.

in K. L. Clark and S-Å. Tärnlund (eds). *Logic Programming.* Academic Press, 1982.

[Campbell 84]
J. A. Campbell (ed). *Implementations of Prolog.* Ellis Horwood, 1984.

[Chang & Lee 73]
C. L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[Clark 78]
Keith L. Clark. 'Negation as Failure'. in H. Gallaire and J. Minker (eds). *Logic and Data Bases.* Plenum Press, 1978.

[Clocksin & Mellish 81]
W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Conery 83]
John S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD Thesis, Department of Information and Computer Science, University of California, Irvine, 1983.

[Curry 77]
Haskell B. Curry. *Foundations of Mathematical Logic.* Dover, 1977.

[Davis76]
Randall Davis. 'Applications of Meta Level Knowledge in the Construction, Maintenance and Use of Large Knowledge Bases'. in R. Davis and D. Lenat. *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill, 1980.

[de Kleer 86]
Johan de Kleer. *An Assumption-based TMS.* Artificial Intelligence 28 127-162, 1986.

[Doyle 79]
Jon Doyle. 'A Truth Maintenance System'. in B. L. Webber and N. J. Nilsson (eds). *Readings in Artificial Intelligence.* Tioga, 1981.

[Dummett 77]
Michael Dummett. *Elemwnts of Intuitionism.* Oxford: Clarendon Press, 1977.

[Dyckhoff 91]
Roy Dyckhoff. *Theorem Proving in Intuitionistic Logic.* St. Andrews University (draft of paper), 1991.

[Fuller & Abramsky 88]
David A. Fuller and Samson Abramsky. *Mixed Computation of Prolog Programs.* New Generation Computing 6 119-141, 1988.

[Gabbay & Reyle 84]
D. M. Gabbay and U. Reyle. *N-Prolog: An Extension of Prolog with Hypothetical Implications.* Journal of Logic Programming 4 319-355, 1984.

[Gallaire & Lasserre 82]
Herve Gallaire and Claudine Lasserre. 'Metalevel Control for Logic Programs'. in K. L. Clark and S-Å. Tärnlund (eds). *Logic Programming.* Academic Press, 1982.

[Gelernter 59]
Herbert Gelernter. 'Realization of a Geometry Theorem-proving Machine'. in *Proceedings of the International Conference on Information Processing.* UNESCO, 1959.

[Genesereth & Nilsson 87]
Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[Gentzen 35]
Gerhard Gentzen. *Untersuchungen über das logische Schliessen.* Mathematische Zeitschrift 39 176-210, 405-431, 1935. English translation in M. E. Szabo (ed). *The Collected Papers of Gerhard Gentzen.* North Holland, 1969.

[Gregory 87]
Steve Gregory. *Parallel Logic Programming in Parlog.* Addison Wesley, 1987.

[Gödel 31]
Kurt Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme.* Monatshefte für Mathematik und Physik 38 173-198, 1931. English translation in J. van Heijenoort (ed). *From Frege to Gödel: a sourcebook in mathematical logic, 1879–1931.* Harvard University Press, 1967

[Hallnäs & Schroeder-Heister 90]
Lars Hallnäs and Peter Schroeder-Heister. *A Proof Theoretic Approach to Logic Programming 1.* Journal of Logic and Computation, 1(3) 1990.

[Haridi 81]

Seif Haridi. *Logic Programming Based on a Natural Deduction System.* Technical Report TRITA-CS-8104, Dept. of Telecommunication Systems – Computer Systems, The Royal Institute of Technology, Stockholm, Sweden.

[Hayes 73]

Patrick J. Hayes. *Computation and Deduction.* Mathematical Foundations of Computer Science – 2nd Symposium. Czechoslovakian Academy of Sciences, 1973.

[Hogger 84]

Christopher J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[Johansson 36]

Ingebrigt Johansson. *Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus.* Compositio mathematica 4 119-136, 1936.

[Kacsuk 90]

Peter Kacsuk. *Execution Models of Prolog for Parallel Computers.* MIT Press, 1990.

[Keronen 91]

Seppo Keronen. *Natural Deduction Proof Theory for Logic Programming.* Technical Report, Department of Computer Science, Australian National University, 1991.

[Kowalski 75]

Robert Kowalski. *A Proof Procedure Using Connection Graphs.* Journal of the Association for Computing Machinery, 22(4) 572–595, 1975.

[Kowalski 79^a]

Robert Kowalski. *Algorithm = Logic + Control.* Communications of the ACM 22(7) 424-436, 1979.

[Kowalski 79^b]

Robert Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[Lassez et al. 88]

J-L. Lassez, M. J. Maher and K. Marriott. 'Unification Revisited'. in Jack Minker (ed). *Foundations of Deductive Databases and Logic Programming,* Morgan Kaufmann, 1988.

[Lloyd 84]

John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Lloyd & Topor 84]

John W. Lloyd and Rodney W. Topor. *Making Prolog More Expressive.* Journal of Logic Programming 3 225-240, 1984.

[Maes & Nardi 88]

Pattie Maes and Daniele Nardi (eds). *Meta-Level Architectures and Reflection.* Elsevier, 1988.

[McCarthy & Hayes 69]

John McCarthy and Patrick J. Hayes 'Some Philosophical Problems from the Standpoint of Artificial Intelligence'. in B. Meltzer and D. Michie (eds). *Machine Intelligence 4.* Edinburgh University Press, 1969.

[Minker 88]

Jack Minker (ed). *Foundations of Deductive Databases and Logic Programming,* Morgan Kaufmann, 1988.

[Nakamura 84]

K. Nakamura. 'Associative Evaluation of Logic Programs'. in J.A. Campbell (ed). *Implementations of Prolog.* Ellis Horwood, 1984.

[Nilsson 80]

Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[Paulson 87]

Lawrence C. Paulson. *Logic and Computation (Interactive Proof with Cambridge LCF).* Cambridge University Press, 1987.

[Pereira 84]

L. M. Pereira. 'Logic Control with Logic'. in J.A. Campbell (ed). *Implementations of Prolog.* Ellis Horwood, 1984.

[Prawitz 65]

Dag Prawitz. *Natural Deduction (A Proof-Theoretical Study).* Almqvist & Wiksell, 1965.

[Robinson 65]

J. Alan Robinson. *A Machine-Oriented Logic Based On the Resolution Principle.* JACM, 12(1) 23–41, 1965.

[Schroeder-Heister 84]

Peter Schroeder-Heister. *A Natural Extension of Natural Deduction.* The Journal of Symbolic Logic, 49(4) 1284–1300, 1984.

[Sergot 83]

M.J. Sergot. 'A Query-the-User Facility for Logic Programming'. in P. Degano and E. Sandewall (eds). *Integrated Interactive Computer Systems.* North-Holland, 1983.

[Shepherdson 88]

John C. Shepherdson. 'Negation in Logic Programming'. in Deductive Databases and Logic Programming, Morgan Kaufmann, 1988.

[Shapiro 83]

Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Shoesmith & Smiley 78]

D. J. Shoesmith and T. J. Smiley. *Multiple-conclusion Logic.* Cambridge University Press, 1978.

[Sickel 76]

Sharon Sickel. *A Search Technique for Clause Interconnectivity Graphs.* IEEE Transactions on Computers C-25(8) 823–835, 1976.

[Skolem 28]

Thoralf Skolem. 'On Mathematical Logic'. in J. van Heijenoort (ed). *From Frege to Gödel: A Source Book in Mathematical Logic.* Harvard University Press, 1967.

[Smith 86]

Brian C. Smith. 'Varieties of Self-Reference'. in J. Y. Halpern (ed). *Reasoning About Knowledge.* Morgan Kaufmann, 1986.

[Smith 89]

David E. Smith. *Controlling Backward Inference.* Artificial Intelligence 39 145-208, 1989.

[Smith & Loveland 88]

Bruce T. Smith and Donald W. Loveland. 'A Simple near-Horn Prolog Interpreter'. R. A. Kowalski and K. A. Bowen (eds). *Logic Programming, Proceedings of the Fifth International Conference and Symposium.* MIT Pree, 1988.

[Tennant 78]

Neil W. Tennant. *Natural Logic.* Edinburgh University Press, 1978.

[Tennant 87]

Neil W. Tennant. *Anti-Realism and Logic.* Oxford University Press, 1987.

[Thom & Zobel 88]

James A. Thom and Justin Zobel (eds). *NU–Prolog Reference Manual (version 1.3).* University of Melbourne, 1988.

[van Harmelen 88]

Frank van Harmelen. 'A Classification of Meta-Level Architectures'. in J. W. Lloyd (ed). *Proceedings of the Workshop on Meta-Programming in Logic Programming.* University of Bristol, 1988.

[van Vaalen 75]

J. van Vaalen. *An extension of unification to substitutions with an application to automatic theorem proving.* in IJCAI-4, 1975.

[Waterman & Hayes-Roth 78]

D. A. Waterman and Frederick Hayes-Roth (eds). *Pattern-directed Inference Systems.* Academic Press, 1978.

[Weybrauch 80]

R. Weybrauch. *Prolegomena to a Theory of Mechanized Formal Reasoning.* Artificial Intelligence 13 133-170, 1980.

[Whitehead & Russell 1910-1913]

A. N. Whitehead and B. Russell. *Principia Mathematica* (3 volumes). Cambridge University Press, 1910-1913.

[Wise 86]

Michael J. Wise. *Prolog Multiprocessors.* Prentice-Hall, 1986.