CS 151, Spring 00
Logic 3

Margaret M. Fleck

February 9, 2000

Reading: Russell and Norvig, chapter 9 and sections 10.1-10.5.

1 Equality

The algorithms described above handle only FOPC sentences that do not use equality (=). We could handle equality by pretending that = is a random predicate and adding axioms for its properties:

- reflexive: $\forall x, x = x$
- symmetric: $\forall x, y, x = y \Rightarrow y = x$
- transitive: $\forall x, y, z, x = y \wedge y = z \Rightarrow x = z$
- substitution: $\forall x, y, P, x = y \wedge P(x) \Rightarrow P(y)$

Unfortunately, the axiom for substitution is second-order. To emulate it in first-order logic, we would need one axiom for each predicate and function used in our database. That's ugly and inefficient.

Instead, practical theorem provers supplement resolution with special inference rules to handle equality. A very general such rule is "paramodulation."

The simplest version of paramodulation is substitution. If we know $\alpha = \beta$ and also some literal $(\ldots \alpha \ldots)$ containing α, we can deduce $(\ldots \beta \ldots)$, where $(\ldots \beta \ldots)$ is the same expression with β substituted for some instance (not necessarily all instances) of α.

This can be generalized to include the possibility that the two sentences contain additional terms. Then paramodulation takes $(\alpha = \beta) \vee \gamma$ and $(\ldots \alpha \ldots) \vee \delta$ and produces the sentence $(\ldots \beta \ldots) \vee \gamma \vee \delta$.

For example, suppose we know that $\text{fluffy} = \text{culprit} \vee \text{asleep}(\text{fluffy})$ and $\text{spilled}(\text{culprit}, \text{milk}) \vee \text{leaky}(\text{pitcher})$. Paramodulation can then deduce and $\text{spilled}(\text{fluffy}, \text{milk}) \vee \text{asleep}(\text{fluffy}) \vee \text{leaky}(\text{pitcher})$.

In general, the inputs to paramodulation would be paramodulation takes $(\alpha = \beta) \vee \gamma$ and $(\ldots \alpha' \ldots) \vee \delta$, where α unifies with α' using the substitution list θ. The output is then $[(\ldots \beta \ldots) \vee \gamma \vee \delta]_\theta$. That is, substitute β for α', assemble the output sentence as in the simpler version of paramodulation, then apply the substitution θ to the output.

For example, suppose we have the sentences $P(g(x)) \vee Q(x)$ and $[g(B) = A] \vee M(t)$. $g(x)$ then unifies with $g(B)$ using the substitution $x = B$. So the general form of paramodulation would then produce the sentence $P(A) \vee Q(B) \vee M(t)$.

Or, suppose that we have the sentences $\text{dead}(\text{author} - \text{of}(x)) \vee \neg \text{before}(\text{date} - \text{of}(x), 1900)$ and $\text{author} - \text{of}(Hamlet) = \text{Shakespeare} \vee \text{author} - \text{of}(Lear) = \text{Bacon}$. We can then deduce $\text{dead}(\text{author} - \text{of}(Hamlet)) \vee \neg \text{before}(\text{date} - \text{of}(Hamlet), 1900$ $\text{author} - \text{of}(Lear) = \text{Bacon}$.

2 OTTER

OTTER is an extremely popular recent theorem prover. It was written by William McCune at Argonne National Labs and its name stands for "Organized Techniques for Theorem-proving and Effective Research." Details can be found at http://www.mcs.anl.gov/home/mccune/ar/.

OTTER represents the state of the art in implemented, usable theorem provers. Presumably there are some more recent ideas in research papers, but not incorporated into such a nice package. It is supposed to be efficient and its code portable.

OTTER is designed as an assistant rather than a fully autonomous reasoner. That is, it takes guidance from a human researcher and uses automatic theorem proving to spell out the details of the proof. It has some notable successes on reasonably constrained problems, including proving correctness of circuits, correctness of security systems, and new mathematical results in areas such as group theory and minimal axiom sets.

OTTER uses a resolution engine, with a number of strategies to speed up search. It uses a set of support strategy, with input from the user about which sentences to focus attention on. It uses user-defined weighting and filtering functions to decide which sentences are "better" and which are "not worth pursuing." For example, less complex sentences are usually better but the details are very domain dependent.

OTTER uses paramodulation to handle equality. It also has rules for simplifying algebraic equations, rather than having to handle them via general-purpose reasoning rules. And I believe OTTER includes a variety of other miscellaneous hacks to improve speed and performance.

3 Horn clauses

Horn clauses are a special class of CNF sentences, which contain only one positive literal. For example, bright(x) $\vee$ $\neg$working(bulb(x)) $\vee$ $\neg$on(switch(x)) is a Horn clauses. If written as an implication, a Horn clauses has only positive literals and only one literal on the righthand side. For example: working(bulb(x)) $\wedge$ on(switch(x)) $\Rightarrow$ bright(x).

Some sentences cannot be written as Horn clauses, e.g. palmate(leaves(x)) $\Rightarrow$ maple(x) $\vee$ sweetgum(x). So Horn clause representations can't be used on all problems. But many real problems can be written using only Horn clauses.

Horn clauses offer faster inference. The conclusion of a non-Horn clause can be an OR, which is hard to use in subsequent reasoning because you have to use proof by cases. The conclusion of a Horn clause is only a single literal, easy to use in deriving further facts.

The very fast search strategy of unit resolution is not complete for random sets of facts. However, it is complete if all sentences are Horn clauses.

Finally, reasoning for non-Horn sentences must be done by working backwards from the goal. It's hopeless to work forwards from the input facts, because there are so many possibilities. For Horn clauses, working forwards from the input facts can sometimes work, when combined wtih appropriate other constraints.

4 Prolog

Prolog is a hybrid of a theorem prover and a general-purpose programming language. As a theorem prover, it is neither sound nor complete. But it's fast and works on a useful range of problems. As a programming language, it does some control automatically and only lets the user manipulate a limited range of control features. Whether this is good or bad depends on whether its automatic control matches the needs of your problem. Prolog is fast for a high-level programming language (especially when compiled) but slow compared to languages such as C.

Prolog simplifies inference by assuming that all inputs are Horn clauses. Moreover, it only allows equality statements ($\alpha = \beta$) if one side of the equation is a variable. On the other hand, Prolog includes special-purpose arithmetic handling rules, I/O commands, and other standard programming features.

Prolog uses a depth-first search, with special hacks to speed up processing. This means that Prolog can get stuck in infinite loops but search is very fast. Literals are consistently searched from left to right, giving the user some control over search order. Finally, the occurs-check step is removed from unification. This makes unification much faster but can allow unsound deductions (rare in practice).

Prolog has been varied and extended in a variety of ways. Constraint logic programming (CLP) uses special inference rules to reason about constraints on numerical variables. The Prolog Technology Theorem Prover (PTTP) makes Prolog into a sound, complete theorem prover by restoring occurs-check, replacing DFS with iterative deepening, and adding hacks to handle non-Horn clauses and negative literals without sacrificing too much speed. Finally, other variations on Prolog add special handling for type predicates.

5 Production systems

A final variation on theorem provers are production systems. These systems keep a database of facts (the knowledge base) and a set of pattern-action rules. They use forward-chaining: reasoning forwards from the known facts rather than backwards from some goal.

The pattern to each rule is a conjunction of literals. The actions include adding facts to the knowledge base, deleting facts from the knowledge base, and perhaps actions of other random types (print to terminal, open valves in a plant, evacuate building).

```
pattern                          action

cat(x) ^ chasing (Robert,x)      ADD afraid(x)

due(cs131,today)                 PANIC
   ^ due(clinic-report,today)
   ^ due(cs140,today)
```

The production system repeatedly executes the following sequence of actions:

- find all rules matching facts in knowledge base
- choose best rule or rules

- execute their actions
- (optional) add information to knowledge base from outside sources, e.g. consequences of actions

Two well-known production systems are OPS-5 and SOAR. They have been used in a variety of applications, involving management of large sets of rules for real domains.

Searching the knowledge base for matching rules can be slow. To improve performance, rules whose patterns start the same way are merged so that the common tests are done only once. This produces a structure called a "rete" network. Since the contents of the knowledge base change only slightly from cycle to cycle, the network remembers the matches from the last cycle and only updates parts of the network that are related to the changes in the database.

Some production systems execute all rules whose patterns match. Other systems choose only the "best" rules to execute. Heuristics for deciding which rules to run include:

- don't do the same thing twice
- prefer matches related to recent changes to the knowledge base
- prefer more specific patterns
- prefer certain types of actions

```
 pattern                        action

bird(x)                        ADD flies(x)

bird(x) ^ huge(x)              ADD not(flies(x))    <== more specific rule


 pattern                           action

burning(x) ^ fat(x) ^ in(pan,x)   COVER pan           <== emergency response
                                                          has higher priority
not(brown(meat))                  STIR contents(pan)
```