Notes on Proof and Proof Search

DCS 440, AI – Fall 1999

Matthew Stone

We are investigating logic so as to better understand the overall conceptual background underlying the majority of AI research. In this picture, we want to design and build a computational artifact—a program—to carry out some task in the real world; we view the program as an *agent* acting in its environment. Logical results tell us how the agent can have a *representation*, viewed a sentence in a formal language, which makes a claim about a real-world relationship—this is the province of semantics—and tell us how the agent can *use* that representation to make correct decisions about the world—this is the province of proof theory.

These notes come in two parts. The first part, represented by sections 1 through 5, provides documentation for the results presented in class about semantics, the structure of proofs and the use of proofs to represent arguments, for example in our New Brunswick route planning problem.

The second part, which consists just of section 6, considers the algorithms and data structures you need to build proofs effectively. This section provides a logical perspective on the operations Prolog does when proving a query, and it also carefully introduces an alternative strategy for executing knowledge bases bottom up. Like the first part, the second part provides mathematical detail about material that is important to AI and is only covered very informally in the text. The second part also includes some formal details that I can present only in broad strokes in class, but that allow you to be very precise about why the logical results about proofs that we studied apply very directly to the AI systems that have been written to carry out representation and reasoning problems for agents. In particular, even though Prolog does some very funny computations involving rule selection, unification and backtracking search, this is exactly what is required to *use* Prolog clauses to derive correct consequences of a description.

1 Background

The representations we're investigating are a simple form of *knowledge base* (KB); which are constructed in terms of *terms* and *predicates*.

Terms take two forms. There are *constants*, conventionally written with symbols beginning with lower case letters, which the designer of a KB intends to correspond to specific real-world objects. Then there are *variables*, conventionally written with symbols beginning with upper case letters (or underscores), which the designer of a KB intends to *range over* the real-world objects that are relevant to the task at hand.

Predicates, like constants conventionally written with symbols beginning with lower case letters, are names the designer gives to real-world relationships. Predicates have *arities* (a mathematician's pun generalizing unary, binary, ternary, etc.)

that says how many objects stand in the corresponding relationship in the world.

Together the set of constants and predicates that are relevant to the designer's task determine the *language* of the knowledge base for that task.

The claims that can figure in a knowledge base are called *definite clauses*. Terms and predicates can be assembled into two kinds of clause. An *atomic clause* is an expression of the form

$$p(t_1, \ldots, t_m)$$

where p is an m-ary predicate and each t_i is a term. It is understood as making the claim about the world that the objects designated by those terms stand in the relationship represented by the predicate.

A *rule* is an expression of the form

$$h \leftarrow b_1 \wedge \ldots b_m$$

where h is an atomic clause (called the *head* of the rule) and each b_i is another atomic clause. (The conjunction $b_1 \wedge \ldots b_m$ is called the *body* of the rule.) A rule is understood as making the claim about the world that in any cases where each clause in the body of the rule makes a true claim about the world, the head of the rule also makes a true claim about the world.

Clauses are called *ground* when they contain no variables.

A *knowledge base* is simply a set of definite clauses. Informally, it collectively claims that the world matches each of the clauses in it.

Compare page 30, *CI*

2 Semantics

An *interpretation* formalizes the process of making claims about the world that we saw in the informal motivation of knowledge bases. An *interpretation* is a triple $\langle D, \phi, \pi \rangle$ where:

- D is a set of objects, called the *universe* or *domain* of the interpretation. (Domain is overloaded; I prefer to use it in its informal sense, meaning a coherent set of real-world tasks that can be solved using a coherent body of knowledge.)
- ϕ is a map associating each constant c in the language with an element $\phi(c)$ of D.
- π is a map associating each n-ary predicate p in the language with a function from $D^n \leftarrow \top, \bot$.

To describe general claims, we also introduce *assignments* to temporarily link each variable to an object; formally an assignment is a map ρ associating an element $\rho(X)$ of D with each variable X. Now we can introduce the idea of the denotation of

a term t on an assignment ρ—written $\delta(t,\rho)$, which formalizes the real-world object that a term picks out in a particular case (as required in a rule, say).

$$\delta(t,\rho) = \begin{cases} \phi(t) \text{ if } t \text{ is a constant} \\ \rho(t) \text{ if } t \text{ is a variable} \end{cases}$$

Given an interpretation, we can combine the denotation of terms with the functional relationship associated with predicates to say when a clause is *true at an assignment* (in that interpretation).

- $p(t_1,\ldots,t_m)$ is true at ρ just in case

$$\pi(p)\langle\delta(t_1,\rho),\ldots,\delta(t_m,\rho)\rangle = \top$$

- $h \leftarrow b_1 \wedge \ldots b_m$ is true at ρ just in case either h is true at ρ or some b_i is not true at ρ.

A clause is *true* (in an interpretation) if and only if it's true at every assignment (in that interpretation). A knowledge base is true (in an interpretation) if and only if every clause in it is true (in that interpretation).

A knowledge base K entails f (written $K \models f$) if and only if f is trun in every interpretation in which K is true. We will focus on the case where f is a ground clause.

Compare page 34–36, *CI*

3 Proof

A *proof* is a *data structure* that describes why a knowledge base entails some fact in a concrete form that a computer program can construct or check.

We'll think of a proof is a *tree* of *judgments*. A *judgment* takes the form

$$K \longrightarrow f$$

(read f follows from K.) K is a knowledge base and f is a ground clause.

There are two basic computational operations—*algorithms*—that go into constructing proofs. The first operation is applying a *substitution*. Informally, this algorithm is a computational simulation of the way assignments allow variables to range over all elements of the universe of an interpretation.

A *substitution* σ is a finite set of the form

$$\{V_1/t_1,\ldots,V_n/t_n\}$$

Here V_i is a variable and t_i is a term; V_i/t_i is called a *binding*. (For some technical conditions on substitutions, see page 52, *CI*; an important one here is that no variable occurs in two bindings V/t_1 and V/t_2.) The *application* of σ to an expression e (like

a term or a clause or even a proof, as we'll see) consists of e with each occurrence of a variable V_i replaced by an occurrence of the corresponding t_i. It is written $e\sigma$; the expression $e\sigma$ is called an *instance* of e; the process of going from e to σ is called instantiation.

This first operation allows us to say what the simplest proofs are, the leaves. Leaves take the form

$$K, e \longrightarrow e\sigma$$

In other words, at leaves, you argue that a ground clause $e\sigma$ follows from the because the ground clause explicitly names one of the cases where a more general clause e applies.

The second operation is matching identical formulas, which provides an algorithm for combining smaller proofs together to make larger proofs. To match proofs, you only have to look at the judgment at the root of the proof: the knowledge base and the ground clause that follows from it. So when you describe matching proofs, you write a proof this way

$$\begin{array}{c} P \\ K \longrightarrow f \end{array}$$

P names the *whole* proof; $K \longrightarrow f$ names the *judgment* at the root of the proof.

Suppose you have $m+1$ proofs that match together as follows:

$$\begin{array}{cccc} P_0 & P_1 & & P_m \\ K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m & K \longrightarrow b_1 & \cdots & K \longrightarrow b_m \end{array}$$

Then you can combine these proofs together into a single larger proof written like this:

$$\frac{\begin{array}{cccc} P_0 & P_1 & & P_m \\ K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m & K \longrightarrow b_1 & \cdots & K \longrightarrow b_m \end{array}}{K \longrightarrow h}$$

When we have a proof

$$\begin{array}{c} P \\ K \longrightarrow f \end{array}$$

we say f is *provable* from K, or $K \vdash f$.

4 Proof, semantics and computation

Proofs are defined completely syntactically, without any reference to how the symbols are interpreted. This is essential to the role that proofs play in the AI picture of an agent in its environment. The agent only has access to its representations; and all it can do with them is carry out syntactic manipulations on them. So this is what it has to use in making its decisions.

So the picture we have needs to be completed by *relating* the *proofs* that an agent can construct with the *information* that we understand the agent to have, given the

meaning of its knowledge base. This relationship takes the form of two mathematical results, *soundness* and *completeness*.

4.1 Soundness

Soundness states that if $K \vdash f$ then $K \models f$. We prove this in two steps corresponding to the two kinds of operations that go into constructing proofs: applying substitutions and combining proofs together. Each steps some insights into why it is possible for a syntactic algorithm to correctly simulate the semantic process of interpretation.

Step 1. In step 1, we prove that, in any interpretation I, if a clause e is true then any ground clause $e\sigma$ is true. To show that $e\sigma$ is true, we must show that $e\sigma$ is true at any assignment ρ. To do that, we will consider an arbitrary such ρ; we will construct a *new* assignment, ρ_σ and apply the assumption that e is true at ρ_σ. We choose ρ_σ in such a way that the *semantic* assignment corresponds to the *syntactic* substitution; this makes the truth e true at ρ_σ *the same thing* as the truth of $e\sigma$ at ρ.

Here's how it works. Write σ out in full as

$$\{V_1/t_1, \ldots, V_n/t_n\}$$

Now, we can explicitly consider the effect of σ on the interpretation of terms. Suppose we want to find the denotation at ρ of a term t that occurs in e, which we have rewritten using σ to an occurence of $t\sigma$ in $e\sigma$. Recall that this is given by $\delta(t\sigma, \rho)$:

$$\delta(t\sigma, \rho) = \begin{cases} \phi(t\sigma) \text{ if } t\sigma \text{ is a constant} \\ \rho(t\sigma) \text{ if } t\sigma \text{ is a variable} \end{cases}$$

Now, let's flesh out this out using what we know about the substitution. If t is a constant, $t\sigma$ is just t. Substitutions only affect variables. If t is a variable V_i on which σ is defined, then $t\sigma$ is the corresponding t_i. As it happens, the t we are considering fall into one of these cases; but otherwise, we would have t unaffected by σ too; it doesn't hurt to write this case down too.

$$\delta(t\sigma, \rho) = \begin{cases} \phi(t) \text{ if } t \text{ is a constant} \\ \phi(t_i) \text{ if } V_i/t_i \text{ is in } \sigma \\ \rho(t) \text{ if } t \text{ is a variable with no binding in } \sigma \end{cases}$$

Now, we can set up ρ_σ so that $\delta(t, \rho_\sigma) = \delta(t\sigma, \rho)$—and echo the substitution semantically. Explicitly, ρ_σ is:

$$\rho_\sigma(V) = \begin{cases} \phi(t_i) \text{ if } V/t_i \text{ is in } \sigma \\ \rho(V) \text{ if } V \text{ is a variable with no binding in } \sigma \end{cases}$$

In each of the cases for t—constant, variable with a binding in σ, variable with no binding in σ—we see that $\delta(t, \rho_\sigma)$ by this definition does coincide with $\delta(t\sigma, \rho)$.

We now use ρ_σ. Suppose e is an atomic clause:

$$p(t_1, \ldots, t_m)$$

Then $e\sigma$ is

$$p(t_1\sigma, \ldots, t_m\sigma)$$

$e\sigma$ is true at ρ just in case

$$\begin{aligned} & \pi(p)\langle\delta(t_1\sigma, \rho), \ldots, \delta(t_m\sigma, \rho)\rangle = \top \\ = \quad & \pi(p)\langle\delta(t_1, \rho_\sigma), \ldots, \delta(t_m, \rho_\sigma)\rangle = \top \end{aligned}$$

that is, just in case, e is true at ρ_σ.

Suppose e is a rule

$$h \leftarrow b_1 \wedge \ldots b_m$$

Then $e\sigma$ is

$$h\sigma \leftarrow b_1\sigma \wedge \ldots b_m\sigma$$

$e\sigma$ is true at ρ just in case

$h\sigma$ is true at ρ or some $b_i\sigma$ is false at ρ.

By the argument we just gave for atomic formulas, this is equivalent to

h is true at ρ_σ or some b_i is false at ρ_σ.

And this is equivalent to the condition that e is true at ρ_σ. ■

This argument contains the nugget of an *induction* on the *structure* of formulas; this kind of arguments argues that any representation has a property by showing first that the simplest representations have this property and then that any way of building a larger representation representation out of smaller ones preserves this property. The complexity of a clause is limited: you have either atoms or rules. Given a representational system that had more ways to make claims about the world this kind of argument would be even more important. For example, we can use the same kind of induction to describe *proofs* which can already grow substantially, even in the simple language we have. This is step 2 of the soundness argument.

Step 2. We prove that if $K \vdash f$ then $K \models f$. First, we consider leaf proofs

$$K, e \longrightarrow e\sigma$$

Suppose K, e is true in some interpretation. Then in particular e is true in that interpretation. By what we proved in step 1, then, $e\sigma$ is also true in that interpretation. Since this is true of any interpretation, we have in other words

$$K, e \models e\sigma$$

Now, we consider a proof formed recursively:

$$\frac{\begin{array}{c}P_0\\ K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m\end{array} \quad \begin{array}{c}P_1\\ K \longrightarrow b_1\end{array} \quad \cdots \quad \begin{array}{c}P_m\\ K \longrightarrow b_m\end{array}}{K \longrightarrow h}$$

We suppose that whenever there is a proof smaller than this one whose root judgment is $K \longrightarrow f$, then $K \models f$. We use this to show that $K \longrightarrow h$ here.

Explicitly, we consider each of the judgments

$$\begin{array}{c} K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m \\ K \longrightarrow b_1 \\ \vdots \\ K \longrightarrow b_m \end{array}$$

Each is the root of a smaller proof. Therefore if we consider any interpretation where K is true, $h \leftarrow b_1 \wedge \ldots \wedge b_m, b_1, \ldots b_n$ are all true. Consider an assignment ρ; applying the clause for interpreting a rule, we see that either h is true at ρ or some b_i is false at ρ. But no b_i can be false at ρ; this means that h must be true at ρ. Since ρ was arbitrary, h is true in that interpretation: $K \models h$.

We now conclude that whenever we construct a proof with root judgment $K \longrightarrow f$, then $K \models f$: it's true of simple proofs and it's true of any proof we build of simpler proofs that have this property. So $K \vdash f$ implies $K \models f$. ■

4.2 Completeness

In general, *completeness* states that when $K \models f$, $K \vdash f$. It says that any true statement can be derived. Another way to understand it is by taking the contrapositive: $K \not\vdash f$ implies $K \not\models f$. If there is no proof of a fact from a knowledge base, then there is some model of the knowledge base where the fact does not hold. Computationally, that means that in a systematic attempt to build a proof for some conjecture that fails, there is all the information you need to construct a model where the conjecture is false.

We will prove case of completeness where there is a particularly clear relationship between an algorithmic attempt to build a proof and a special kind of model. This is the case where f is a ground atomic clause and the model you look at is the canonical or *minimal* model where as few ground atomic clauses are true as possible.

Step 1. We construct a *minimal model* for a knowledge base K. The model consists of a universe, an interpretation for the constants, and an interpretation for the predicates in the knowledge base. Now, the model will give "a robot's solipsistic view of the world", so it will just have placeholders for all of the objects named in the knowledge base. The robot, in its syntactic confines, needs placeholders because it does not (and cannot) have any idea what its designer intended its constants to des-

ignate. To start, the constants themselves make good placeholders, so we set

$$\begin{array}{rcl} D_M & = & \{c : c \text{ is a constant in the knowledge base }\} \\ & \textit{or} & \{0\} \text{ if no constants appear in the knowledge base} \end{array}$$

Now the interpretation of a constant is just the placeholder we're using for the constant's value, which boils down to this simple defintion of ϕ_M:

$$\phi_M(c) = c$$

Now what about the predicates? The robot knows, by building proofs, that it should include a placeholder for some real-world relationships, but it doesn't know about anything else. So, in the minimal model, we set $\pi_M(p)$ to be the function f_p defined by:

$$f_p\langle t_1, \ldots, t_n\rangle = \begin{cases} \top \text{ if } K \vdash p(t_1, \ldots, t_n) \\ \bot \text{ otherwise} \end{cases}$$

So our minimal model $I_M = \langle D_M, \phi_M, \pi_M \rangle$.

I have been talking like the minimal model of the knowledge base provides an interpretation where the knowledge base is true. That is correct, but we have to prove it. To show every f in K is true in I_M, we derive a contradiction from the assumption that some f in K is false in I_M. This contradiction relies on using the simulation between assignments and substitutions we saw above, only in the reverse direction.

The contradiction also relies on the proposition that if there is a proof P consisting of the leaf $K, e \longrightarrow e\sigma$ then $e\sigma$ is true in I_M. This is not just a special case of soundness because we don't yet know that I_M *is* a model of K. So we prove this proposition directly; it is not hard. First, suppose $e\sigma$ is a ground atomic clause $p(t_1, \ldots, t_n)$; then $p\sigma$ is true in I_M just in case there is a proof of $p(t_1, \ldots, t_n)$; of course P is that proof.

Otherwise, suppose $e\sigma$ is a ground rule $h\sigma \leftarrow b_1\sigma \wedge \ldots \wedge b_m\sigma$. Consider an assignment ρ; there are two cases. Suppose some $b_i\sigma$ is false at ρ; then $h\sigma \leftarrow b_1\sigma \wedge \ldots \wedge b_m\sigma$ is true at ρ. Otherwise, suppose each $b_i\sigma$ is true at ρ; since each $b_i\sigma$ is a true ground atomic clause at ρ, there must be a proof of $K \longrightarrow b_i\sigma$ in each case. We can combine these proofs with the proof P to obtain a proof whose root is the judgment $K, e \longrightarrow h\sigma$. So $h\sigma$ must be true at ρ, and thus the rule $h\sigma \leftarrow b_1\sigma \wedge \ldots \wedge b_m\sigma$ is also true at ρ.

Now we can return to the main line of the argument. Call the putative false clause in our knowledge base e; there must be some assignment ρ such that e is false at ρ. This assignment takes each variable V_i that occurs in e to some element in D, $\rho(V_i)$—in other words, to some term t_i. Since only finitely many variables occur in e, we can define a substitution σ by

$$\{V_i/t_i : V_i \text{ occurs in } e \text{ and } \rho(V_i) = t_i\}$$

Now, using the notation of soundness, step 1, $\rho = \rho_\sigma$ and e is true at ρ_σ just in case $e\sigma$ is true at ρ. But $e\sigma$ is a ground formula. So there is a proof $K, e \longrightarrow e\sigma$. And so,

by the proposition, $e\sigma$ is true in the interpretation. This is a contradiction, so there can be no false clause in our knowledge base; I_M is a model of K. ■

Step 2. We can now argue immediately that, given a ground atomic clause f, when $K \models f$, $K \vdash f$. For consider I_M. If $K \models f$ then f is true in I_M (at any assignment ρ). But by the construction of truth in I_M, that means that there is a proof of f! ■

While we have proved a special case of completeness, completeness is true in general for the kind of language we have here; however, it has to be proved in a different way. To start, you need an additional rule for building proofs:

$$\frac{\begin{array}{c} P \\ K, b_1, \ldots, b_m \longrightarrow h \end{array}}{K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m}$$

What's more, you cannot construct a single countermodel to any conjecture; you need to construct a distinct model for each formula f for which there is no proof. We'll leave such arguments for a hard-core logic course.

5 Using proofs

We can now connect the logical work we did in Sections 1 through 4 with the picture of an agent acting in its environment that we want to understand more precisely. The agent's *representation* of the world is its knowledge base. With the semantics for terms, clauses and knowledge bases that we considered in Section 2, we can say what claim that knowledge base makes about the world. Put another way, we can characterize what the world must be like if that knowledge base is true.

The agent can *use* its knowledge base by constructing proofs. Suppose the agent needs to know whether a specific fact f is true; if the agent can construct a proof of f from its knowledge base, it can con conclude that if the world is the way the knowledge base says it is, then f must actually be true in the world. On the other hand, if the agent tries all possible ways to construct a proof of f and is unable to do it, then it can conclude that the knowledge base could be true and still f could be false. This is the content of the results about models and proofs presented in Section 4.

This view of the use of proofs gives rise to the idea of *queries* and *answers*. We will define a query as an expression of the form:

$$?K \longrightarrow f$$

In this expression, f is an atomic clause, possibly containing variables; the ? notation indicates that this query represents a request to prove instances of f from K. We can also use the notation $?f$ for queries when the knowledge base K is understood in context (that's what the book does).

An *answer* gives a possible response to a query; an answer is either positive or negative. A positive answer consists a substitution σ together with a proof of $K \longrightarrow f\sigma$. A negative answer is just the symbol *no*; it indicates that the knowledge base is compatible with all instances of f being false.

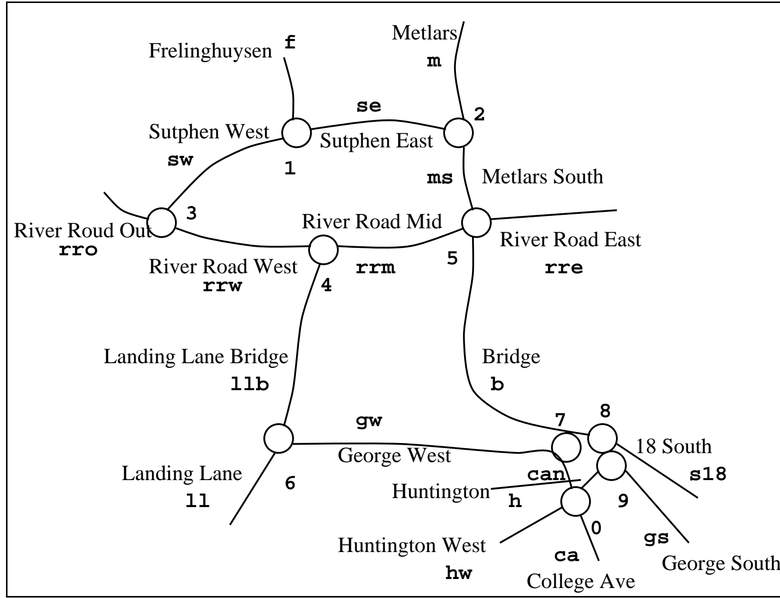

Figure 1: A schmatic map of some of New Brunswick and Piscataway

Posing queries and deriving answers gives a way for an agent to make decisions. Suppose an agent has a control program in which each possible action is associated with a query that says what the world must be like for that action to be an appropriate one. To choose its next action, the agent evaluates all of the queries, and picks one with a positive answer.

Example. We can see how the Antarctic meteorite robot could make decisions in this way. If it is considering a particular visible object $o1$, it should go pick up $o1$ and investigate it further if $o1$ turns out to be interesting; otherwise it should move on to consider the next visible object. We can cast this as a rule associating queries (and answers) with choices the robot will make:

If there is a positive answer to $?K \longrightarrow \mathit{interesting}(o1)$, do $\mathit{pick\text{-}up}(o1)$.

If there is a negative answer to $?K \longrightarrow \mathit{interesting}(o1)$, do $\mathit{try\text{-}next}()$.

■

Answers are useful for more than just making your next decision, however. In planning, a proof can provide a useful data structure for keeping track of a series of actions to perform and the expected results of those actions. In communication, a proof can keep track of a collection of facts and the relations between them; it suggests how the facts can be organized into a comprehensible explanation.

Example. Suppose we have represented the road network of Figure 1 using a predicate *meets* to name a relation that holds between I, R and D when following road segment R along direction D leads into intersection I. If we assume that a walker is

free to walk along the side of any road segment in either direction, then we can define clauses that describe when someone *can walk* from a start road segment S to a finish road segment F: $cw(S,F)$:

$$\begin{aligned}&cw(F,F)\\ &cw(S,F) \leftarrow meets(I,S,D1) \wedge meets(I,M,D2) \wedge cw(M,F)\end{aligned}$$

Here is an informal explanation of these clauses. You can get from where you are to anywhere else on that segment immediately. You can get from where you are to anywhere else by going to the nearest intersection, walking on to another road that enters the intersection, and continuing your walk.

Say the knowledge base K includes the clauses defining Figure 1, together with the clauses for *cw* given above. We can ask whether you can go from Johnson Park in Piscataway to Buccleuch Park in New Brunswick by posing the query

$$?K \longrightarrow cw(rrw, gw)$$

(This assumes you exit Johnson Park on the west segment of River Road and enter Buccleuch Park on the west segment of George Street.)

Each answer to this query encodes a *way to get* from Johnson Park to Buccleuch Park. In fact, this is the easiest way to see how each answer is constructed. To write answers down compactly, let's use the notation $step(s,m) - by(i,d1,d2,f)$ to abbreviate three leaf proofs. The first proof establishes an instance of the second clause for *cw*, using a substitution σ defined by

$$\sigma = \{S/s, F/f, I/i, D1/d1, M/m, D2/d2\}$$

This proof is

$$K \longrightarrow cw(s,f) \leftarrow meets(i,s,d1) \wedge meets(i,m,d2) \wedge cw(m,f)$$

(To see that this is a leaf, say $K = K', c$ where c is the second clause for *cw*: then this proof is $K', c \longrightarrow c\sigma$.) The second and third proofs are also leaves that access appropriate facts about the interconnections among road segments in Figure 1.

$$\begin{aligned}K &\longrightarrow meets(i,s,d1)\\ K &\longrightarrow meets(i,m,d2)\end{aligned}$$

Here is an answer to the query encoded using this notation.

$$\dfrac{\begin{array}{l}step(rrw, llb)-\\ by(i4, east, north, gw)\end{array} \qquad \dfrac{\begin{array}{l}step(llb, gw)-\\ by(i6, south, west, gw)\end{array} \qquad K \longrightarrow cw(gw, gw)}{K \longrightarrow cw(llb, gw)}}{K \longrightarrow cw(rrw, gw)}$$

The notation brings out the *way* that this answer says to go from River Road to George Street. In the first *step* subproofs, you follow River Road to the Landing Lane Bridge; as the corresponding *by* term indicates, that means going east on River Road and reaching *i4*. In the second *step* subproofs, you follow Landing Lane Bridge to George Street; as the *by* term indicates here, that means going south on Landing Lane Bridge and reaching *i6*. With that, the proof finishes at the observation that you're at your destination.

This example answer, and the informal discussion of it, should suggest how *proofs* provide data structures that an agent can build to create a plan; an agent can draw on to follow a plan; or to communicate a plan to a person or another artificial agent. A data structure like the answer above contains a lot of technical structure, but in a nutshell it makes a specific argument about *why* the query is true. That *argument* can be understood and used, just like the information that the query is true can be understood and used.

6 Building proofs

So far, we have studied a particularly simple representation of proofs. Each step in a proof applies a specific clause from the knowledge base to a specific case; the case is decided in advance by using a specific substitution to obtain the ground clause that the proof uses from the more general clause that's available explicitly in the knowledge base.

This representation provides the best way to think about *why* proofs work. It allows soundness and completeness to be proved particularly simply. This representation also provides the best way to think about how to *use* proofs. This representation writes out explicitly the facts used in the proof—the facts that you need to check to follow a plan, or the facts that you need to communicate to explain an argument.

Unfortunately, this representation does not provide the best way to think about *building* proofs. Intuitively, this is because you don't really know the specific case that you want to apply a clause at when you select a clause; you only have a "rough idea". Such a "rough idea" of how to apply a clause can actually be represented explicitly. This allows us to be precise about the algorithms that are best used to build proofs.

6.1 Constraints and Unification

To do so, we introduce a *lifted* representation for proofs. By contrast with the lifted representation, the representation introduced in Section 3 is called a *ground* representation for proofs.

The lifted representation is based on *equation constraints* and *substitutions*. A *constraint* is the general term in Artificial Intelligence for a representation that you use to record partial information about values for variables. An *equation* is an expression of the form $e = f$ that says two expressions must be equal. An equation is a *kind* of constraint. Some equations tell you exactly what the value of a vari-

able should be, like $X = a$, but other equations, like $X = Y$, give partial information. $X = Y$ doesn't tell you what values X and Y have; it tells you that whatever those values are, they have to be the same.

A substitution *satisfies* an equation $e = f$ exactly when $e\sigma$ is identical to $f\sigma$. A substitution satisfies a *set* of equations E exactly when it satisfies every equation in the set. A substitution σ that satisfies an equation or set of equations E is also called a *unifier* for E.

We can use equations to build data structures called *proof skeletons* that do not commit to use particular ground instances of clauses. These proof skeletons leave open the choice of instantiations in proofs, but otherwise record how the proof is constructed and what constraints instances of clauses in the proof should satisfy. A proof skeleton, together with a substitution that satisfies its equations, will give enough information to construct a ground proof, and vice versa.

Formally, a proof skeleton takes the form $\langle P; E \rangle$. P is a tree of judgments of the form $K \longrightarrow f$ where K is a knowledge base and f is a clause; obviously these judgments are inspired by the judgments that proofs are made out of, except these f contain variables in addition to the constants required in ground proofs. E is a list of equations which must be solved to obtain a proof from the skeleton.

Again, we spell out the cases for proof skeletons in terms of leaves and internal nodes. Leaves look like this:

$$\langle K, e \longrightarrow f; e\theta = f \rangle$$

Normally at leaves, we instantiate a clause e from the knowledge base to a particular ground case using a substitution σ. In skeletons, we do this instantiation in two steps (or more!).

First, we don't want to choose a particular value for $e\sigma$ at this stage; we simply leave open the possibility that e can be instantiated in a range of ways. So we apply a substitution θ to e, which may not be ground, to leave open the choice of terms that will appear in $e\sigma$. (In fact, typically we will assume that θ is a substitution that simply *renames* or translates the variables that occur in e to new variables that do not occur anywhere lower down in the proof skeleton we are building.)

Second, we want to anticipate how this leaf may fit into the broader proof we are building. To do that, we frame the rule in terms of a clause f that we assume the leaf provides an instance of. Often, f will be a query and we will want the leaf to provide an anser for f. To ensure that the leaf is a proof of an instance of f, we add the equation $f = e\theta$. When we eventually obtain our proof from our skeleton, we will apply some substitution ρ to this leaf. The equation we add here will ensure that this leaf accesses an instance of the knowledge base—$e\sigma$ or $e\theta\rho$—that provides an instance $f\rho$ of the query we need the leaf to address.

Internal nodes are relatively easy by comparison. Suppose you have $m+1$ proof

skeletons that match together as follows:

$$\langle \overset{P_0}{K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m} ; E_0\rangle \quad \langle \overset{P_1}{K \longrightarrow b_1} ; E_1\rangle \quad \ldots \quad \langle \overset{P_m}{K \longrightarrow b_m} ; E_m\rangle$$

Then you can combine them together into a larger proof skeleton. You build the new tree by using an analogue of the rule for constructing a larger ground proof tree from smaller proof trees; you build the new list of equations by appending together all the lists of equations from the smaller proof skeletons. Explicitly, what you get is this:

$$\langle \frac{\overset{P_0}{K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m} \quad \overset{P_1}{K \longrightarrow b_1} \quad \ldots \quad \overset{P_m}{K \longrightarrow b_m}}{K \longrightarrow h} ; E_0, E_1, \ldots E_m\rangle$$

A *lifted proof* is defined as a proof skeleton $\langle P, E\rangle$ together with a substitution σ that satisfies E and associates each variable with a ground term. We will shortly define algorithms to translate between ground and lifted proofs; this will show that the lifted proof system is sound and complete the same way the ground proof system is.

Example. But first, we illustrate proof skeletons and lifted proofs by returning to the map example of Section 5.

Start by considering what a proof skeleton looks like for the query:

$$?K \longrightarrow cw(gw, gw)$$

By accessing the clause $cw(F, F)$ and applying a substitution θ that renames F to a new variable $F1$ to it, we get the following proof skeleton:

$$\langle K \longrightarrow cw(gw, gw); cw(F1, F1) = cw(gw, gw)\rangle$$

Of course, the substitution $\sigma = \{F1/gw\}$ shows that this is not just a proof skeleton but a lifted proof.

More interesting things happen with a longer proof skeleton. Information about the values of variables accumulates gradually during the construction of the proof. Let's get a sense of this by considering the query $?K \longrightarrow cw(llb, gw)$.

Suppose we decide that the proof will begin by reasoning from the recursive clause for cw; suppose we use θ_0 as the placeholder substitution in applying this clause:

$$\theta_0 = \{S/S_0, F/F_0, I/I_0, D1/D1_0, M/M_0, D2/D2_0\}$$

Write $h \leftarrow b$ to abbreviate the result of applying θ_0 to this clause, which is:

$$cw(S_0, F_0) \leftarrow meets(I_0, S_0, D1_0) \wedge meets(I_0, M_0, D2_0) \wedge cw(M_0, F_0)$$

Meanwhile, from the query, we know that to apply this clause it will also have to take this form:

$$cw(llb, gw) \leftarrow b$$

We can put this all together into a proof skeleton:

$$\langle K \longrightarrow cw(llb, gw) \leftarrow b; (h \leftarrow b) = (cw(llb, gw) \leftarrow b)\rangle$$

The equations here are just an elaborate way of requiring that $S_0 = llb$ and $F_0 = gw$.

Thus, when we decide that our proof will begin by reasoning from the recursive *cw* clause, we are really fixing three things. First, the overall proof skeleton for $cw(llb, gw)$ will begin with an internal node, showing that the query will be derived indirectly by some number of steps. Second, the leftmost subtree will be derived from the proof skeleton above; that settles the other queries we must answer to complete the proof. Finally, we adopt the equation constraint that $S_0 = llb$ and $F_0 = gw$ as we continue to build the proof.

What is important about this representation is not what we *have* decided, but what we haven't. So far, the equations place no constraints on any of the variables that appear only in the body of the *cw* clause. So as far as we know, I_0 could be anything, and $D1_0$ could be anything, for example.

That's important when we go to construct the next subproof, where we pose the query $?K \longrightarrow meets(I_0, S_0, D1_0)$. We can use the fact that $S_0 = llb$ to access clauses from the knowledge base that describe where segment *llb* goes. But then the clause we choose will tell us the values for I_0 and $D1_0$.

In particular, suppose that at this stage we access the clause:

$$meets(i6, llb, south)$$

Then we build the proof skeleton

$$\langle K \longrightarrow meets(I_0, S_0, D1_0); meets(i6, llb, south) = meets(I_0, S_0, D1_0)\rangle$$

So we learn that $I_0 = i6$ and $D1_0 = south$.

Similarly, at the next stage we pose the query $?K \longrightarrow meets(I_0, M_0, D2_0)$, so we are looking for another *meets* clause that describes intersection *i*6. We build the proof skeleton

$$\langle K \longrightarrow meets(I_0, M_0, D2_0); meets(i6, gw, west) = meets(I_0, M_0, D2_0)\rangle$$

and learn that $M_0 = gw$ and $D2_0 = west$. We complete the proof with the query $?K \longrightarrow cw(M_0, F_0)$. But we already know that $M_0 = gw$ and $F_0 = gw$. Only know do we know the particular instantation of the recursive *cw* clause that we have settled on to build the proof.

To finish up, we rebuild the first proof skeleton given above, and splice the trees and equations together into an overall proof skeleton. ■

To conclude this section, we show that every lifted proof corresponds to a ground proof, and vice versa. These arguments give another straightforward illustration of reasoning by induction on the structure of proofs.

Soundness. Suppose we are given a lifted proof consisting of a proof skeleton $\langle P, E \rangle$ and a substitution σ that satisfies E. We can convert it into a ground proof P'. Together with the soundness theorem for the ground proof system that we proved in Section 4, this shows that the lifted proof system is also sound.

First we consider the case where the tree P is a leaf. Then the proof skeleton actually takes the specific form:

$$\langle K, e \longrightarrow f; e\theta = f \rangle$$

We know because we have a lifted proof that $(e\theta)\sigma$ is identical to $f\sigma$. I claim that

$$K, e \longrightarrow f\sigma$$

is a ground proof. We have to show that $f\sigma$ is an instance of e. To establish this, we will define a new σ' in terms of σ and θ such that $e\sigma'$ is $f\sigma$. Recall θ and σ are represented as sets of bindings V_i/t_i. So we have:

$$\sigma' = \{V_i/(t_i\sigma) : V_i/t_i \in \theta\} \cup \{V_i/t_i : V_i/t_i \in \sigma \text{ and there is no } V_i/t_j \in \theta\}$$

For any e, θ and σ, and any σ' defined from them this way, $(e\theta)\sigma$ and $e\sigma'$ are identical. One way to prove this is by induction on the structure of expressions. We start with terms. If e is a constant c, then $(e\theta)\sigma$ and $e\sigma'$ agree: they are both c. If e is a variable V_j that has a binding V_j/t_j in θ, then $e\theta$ is t_j and $(e\theta)\sigma$ is $t_j\sigma$. This is also $e\sigma'$. Otherwise, e must be a variable V_j that has no binding in θ. So $e\theta$ is V_j and $(e\theta)\sigma$ is $V_j\sigma$. If there is a binding V_j/t_j in σ, then, $(e\theta)\sigma$ is t_j; but $e\sigma'$ is also t_j, because with no binding for V_j in θ we have defined σ' to include V_j/t_j too. If there is no binding for V_j in σ either, then both $(e\theta)\sigma$ and $e\sigma'$ are simply V_j. Now that we have established what we need for terms, we can continue the induction. Assuming the claim is true for smaller expressions, it extends to larger expressions. We simply apply the alternative substitutions to obtain identical subexpressions and then recombine the subexpressions into identical overall expressions.

So far, then, we have shown that in the case that P is a leaf the following proposition is true: if there is a lifted proof

$$\langle \overset{P}{K \longrightarrow f}, E \rangle, \sigma$$

then there is a ground proof of the form

$$\overset{P'}{K \longrightarrow f\sigma}$$

Suppose this proposition is true of all proofs of height smaller than h, and consider a lifted proof of height h; it takes the form

$$\langle \frac{\overset{P_0}{K \longrightarrow f \leftarrow b_1 \ldots b_m} \quad \overset{P_1}{K \longrightarrow b_1} \;\ldots\; \overset{P_m}{K \longrightarrow b_m}}{K \longrightarrow f}, E \rangle, \sigma$$

We know by the construction of the tree in the skeleton here that there must be skeletons

$$\langle P_0, E_0 \rangle \ldots \langle P_m, E_m \rangle$$

Each P_i has smaller height than h; each E_i consists of equations that are recorded in E, so σ satisfies each E_i. Therefore we have lifted smaller proofs that we can apply our hypothesis to. With the proofs $P_0', P_1', \ldots, P_m'$ that we obtain, we can construct the overall ground proof P' that we need like this:

$$\frac{\overset{P_0'}{K \longrightarrow f\sigma \leftarrow b_1\sigma \ldots b_m\sigma} \quad \overset{P_1'}{K \longrightarrow b_1\sigma} \;\ldots\; \overset{P_m'}{K \longrightarrow b_m\sigma}}{K \longrightarrow f\sigma}$$

By the induction principle, then, the proposition is true for all lifted proofs. This completes the argument that lifted proofs are sound. ■

Completeness. Suppose we are given a ground proof P'. We can convert it into a lifted proof P. Together with the completeness theorem for the ground proof system that we proved in Section 4, this shows that the lifted proof system is also complete.

Again, we start by considering the case where the ground proof P' is a leaf, so it has the specific form:

$$K, e \longrightarrow e\sigma$$

We introduce a substitution θ with a binding that associates each variable V_i that occurs in e with a totally new variable N_i. Then we create another substitution σ_0 defined by

$$\sigma_0 = \{N_i/t_i : V_i/t_i \in \sigma \text{ and } V_i \text{ occurs in } e\}$$

Now we exhibit a corresponding lifted proof:

$$\langle K, e \longrightarrow e\sigma; e\theta = e\sigma \rangle, \sigma_0$$

In general now, we assume that from a ground proof

$$\overset{P}{K \longrightarrow f}$$

we can construct some lifted proof

$$\langle\ \overset{P}{K \longrightarrow f}\ ; E \rangle, \sigma$$

where, of course, σ satisfies E, but where also σ is defined exclusively for new variables that we have introduced only in constructing the lifted proof. (That means the variables that appear in the equations E are just these new ones.) Suppose this is true

of ground proofs of height smaller than h, and consider a ground proof P of height h; it takes the form:

$$\frac{\overset{P_0}{K \longrightarrow f \leftarrow b_1 \ldots b_m} \quad \overset{P_1}{K \longrightarrow b_1} \ \ldots \ \overset{P_m}{K \longrightarrow b_m}}{K \longrightarrow f}$$

Again, each P_i is a smaller proof, so we can apply the induction hypothesis to obtain lifted proofs of the form

$$\langle \ \overset{P_i}{K \longrightarrow f_i} \ ; E_i \rangle, \sigma_i$$

Now suppose we consider the set of bindings $\sigma = \bigcup_i \sigma_i$. We know that σ so defined is in fact a substitution, because each σ_i is defined only for the new variables introduced in lifting subproof P_i; these sets are disjoint. We also know that σ satisfies each E_i, because σ and σ_i agree on all the variables that appear in E_i. What's more, we know that σ is defined exclusively for the new variables that we introduced in constructing the overall lifted proof corresponding to P. That gives us a lifted proof

$$\langle \ \overset{P}{K \longrightarrow f} \ ; E_0, E_1, \ldots, E_m \rangle, \sigma$$

where σ has the needed property. By the induction principle, the result holds for all ground proofs. This rounds out the argument that lifted proofs are complete. ■

6.2 Building proofs bottom up

This *lifted* representations of proofs finally allows us to make precise a variety of procedures for building proofs. We start with a *bottom up* proof procedure. This procedure is named because it starts with concrete facts and gradually builds up larger proofs that establish higher-level, more abstract conclusions.

In broad outline, the bottom up procedure works by filling repository of proofs; the repository stores a representative proof for each provable formula that has been found. The repository is filled using a queue that holds proofs that have been discovered but whose consequences have not yet been considered. The bottom up procedure repeatedly takes a proof off the queue and lets it react with other proofs in the repository to produce new proofs. Since more consequences may follow in turn from the new proofs, the new proofs are added to the queue. Finally, the dequeued proof is added to the repository.

Pages 47 and 54 of *CI* describes a brute-force bottom up proof procedure for the language we are studying. (Brute force means that the procedure doesn't adopt any representations or algorithms that are designed to compute the results in an efficient way; it uses simple representations at the cost of possibly doing more work.) This procedure instantiates each clause in all possible ways. Then it repeatedly looks for

a ground rule whose body has a proof but whose head does not yet; when it finds such a rule, the procedure adds a proof for the head.

One can sometimes do better in implementing this strategy. First, instead of applying rules all at once, you can apply rules in stages, proving one formula in the body at a time. To do this, you need to add elements to the queue and the repository that store a clause together with proofs for some of the atoms in its body. When you find a proof for the next atom in its body, you can combine the two elements together to get another stage in the application. If it's the final stage, you can collapse the result into a proof of the head of the rule.

In addition to this trick for handling rules, you can also use some specialized procedures for handling variables. First, you can use *most general unifiers* to solve the equations associated with proofs. Recall that a unifier for a set of equations is a subsititution that satisfies all the equations. One substitution σ is at least as *general* as another σ' if there is a third subsitution θ such that $(e\sigma)\theta$ is always the same as $e\sigma'$. It turns out that the simple equations you solve in ordinary first-order logic always have one unifier that is at least as general as any other. We can assume we have a function that computes such unifier for such a set of equations—$mgu(E)$. You can use $mgu(E)$ to help compute $mgu(E,E')$. Since the substitution $mgu(E,E')$ satisfies E, and $mgu(E)$ is a most general unifier for E, there is a θ such that $mgu(E)\theta = mgu(E,E')$.

You can also use the generality of substitutions to determine which proofs provide new results. Suppose you have proved a fact f with substitution σ. You may already have a fact g with substitution σ' where there is a θ such that $(g\sigma')\theta$ is identical to $f\sigma$. Then any formula you could derive using f and σ, you could already derive using g and σ'. In this case, we say that $g\sigma'$ *subsumes* $f\sigma$. We don't need to add the proof for f and σ to the queue.

We can now describe the bottom up proof procedure more precisely. The elements we consider are of two kinds. For atomic clauses, we have lifted proofs, where the substitution we store is as general as possible:

$$\begin{array}{l} Atom \\ \langle P,E\rangle, \\ mgu(E) \end{array}$$

To compare two such structures using subsumption, suppose A_1 has a proof that ends $K \longrightarrow f$ and a unifier σ and A_2 has a proof that ends $K \longrightarrow g$ and a unifier σ'; if $f\sigma$ subsumes $g\sigma'$ then A_1 subsumes A_2.

To store partial information about the use of a rule, we keep records of the form

$$\begin{array}{l} \mathit{Rule} \\ h \leftarrow b_1 \dots b_m (\in K), \\ \left\{ \langle\ K \xrightarrow{P_1} b'_1\ ; E_1 \rangle \dots \langle\ K \xrightarrow{P_i} b'_i\ ; E_i \rangle \right., \\ \sigma = mgu(E_1, b_1 = b'_1, \dots, E_i, b_i = b'_i) \end{array}$$

To compare two such structures using subsumption, suppose R_1 involves a clause $h_1 \leftarrow B_1$ and a unifier σ and R_2 involves a clause $h_2 \leftarrow B_2$ and a unifier σ'; if $(h_1 \leftarrow B_1)\sigma$ subsumes $(h_2 \leftarrow B_2)\sigma'$ then A_1 subsumes A_2.

Now we give an extended explanation of how we combine two such structures. Under appropriate conditions, *reacting Atom* with *Rule* or *Rule* with *Atom* gives a new result N; we explain both what the conditions are and what N looks like below. First, though, observe that *reacting Atom* with *Atom* or *Rule* with *Rule* never results in a new result.

We'll assume the entry *Rule* takes the form above; suppose also that the entry *Atom* takes the form

$$\begin{array}{l} \mathit{Atom} \\ \langle\ K \xrightarrow{P_{i+1}} b_{i+1}\ ; E_{i+1} \rangle, \\ \theta = mgu(E_{i+1}) \end{array}$$

Also suppose none of the variables of *Atom* and *Rule* overlap. As in the proof of completeness, we can take $\sigma \cup \theta$ to give

$$\sigma \cup \theta = mgu(E_1, b_1 = b'_1, \dots, E_i, b_i = b'_i, E_{i+1})$$

We can use this to construct

$$\sigma' = mgu(E_1, b_1 = b'_1, \dots, E_i, b_i = b'_i, E_{i+1}, b_{i+1} = b'_{i+1})$$

if these equations have any solution.

Once we have the new unifier, there are two possible ways to store the result. If $i+1 = m$ then we have completed the application of the rule; we can assemble a proof tree P' of

$$\frac{K \longrightarrow h \leftarrow b'_1 \dots b'_m \quad K \xrightarrow{P_1} b'_1 \ \dots\ K \xrightarrow{P_m} b'_m}{K \longrightarrow h}$$

we can use P' to construct N as:

$$\begin{array}{l} \mathit{Atom} \\ \langle P'; E_1, \dots E_m, (h \leftarrow b_1 \dots b_m) = (h \leftarrow b'_1 \dots b'_m) \rangle, \\ \sigma' \end{array}$$

The bottom-up procedure processes queue Q against repository D to yield final result R if

- Q is empty and D is R; or
- - We dequeue X from Q, leaving new queue Q'.
  - Let S be the set of elements C such that *reacting* F with X for F in D gives C.
  - Let S' hold a fresh copy of each element C of S such that no F in Q or D subsumes C.
  - We enqueue each element of S' in Q', giving next queue Q''.
  - We add X to D giving new repository D''.
  - The bottom-up procedure processes Q'' against D'' to yield final result R.

Figure 2: Bottom-up proof procedure with variables

Otherwise we still have a partial application of the rule; we construct N as:

$$\begin{array}{l} \textit{Rule} \\ h \leftarrow b_1 \ldots b_m (\in K), \\ \left\{ \langle\, K \xrightarrow{P_1} b'_1\ ; E_1 \rangle \ldots \langle\, K \xrightarrow{P_i} b'_i\ ; E_i \rangle, \langle\, K \xrightarrow{P_{i+1}} b'_{i+1}\ ; E_{i+1} \rangle\ , \right. \\ \sigma' \end{array}$$

These definitions allow us to define the bottom up proof procedure of Figure 2.

6.3 Building proofs top down

A top down procedure builds proofs nondeterministically. The procedure maintains an incomplete lifted proof of the query. At each step, the procedure extends the proof by guessing a way to extend it. (The book uses the elegant term *choose* for this kind of guess; see page 50.)

Incomplete proofs are also called *tableaux*. Incomplete proofs differ from complete proofs in that some of the leaves in incomplete proofs can be unproved queries.

You can probably imagine how such a data structure could be defined, by analogy with complete proofs. For completeness, here is such a definition. A *tableau judgment* either has the usual form

$$K \longrightarrow e$$

where e is a clause or the form of a query

$$?K \longrightarrow g$$

where g is an atomic clause. (Because we can assume that rules are accessed directly from the knowledge base, that the only unproved clauses in a tableau are atomic.)

A *tableau skeleton* consists of a pair $\langle P; E \rangle$. P is a tree of tableau judgments, and E is a list of equations. Leaves either take the form of proof skeletons:

$$\langle K, e \longrightarrow f; e\theta = f \rangle$$

Or they take the form of queries:

$$\langle ?K \longrightarrow g; \emptyset \rangle$$

Internal nodes in tableau skeletons are defined by the same rule as internal nodes in proof skeletons. You have some tableau skeletons of this form—where all of the skeletons for the bodies may involve queries or usual judgments:

$$\langle \begin{array}{c} P_0 \\ K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m \end{array} ; E_0 \rangle \quad \langle \begin{array}{c} P_1 \\ (?)K \longrightarrow b_1 \end{array} ; E_1 \rangle \ \ldots \ \langle \begin{array}{c} P_m \\ (?)K \longrightarrow b_m \end{array} ; E_m \rangle$$

Then you can combine them together into a larger tableau skeleton. Again, you build the new tree by using an analogue of the rule for constructing a larger ground proof tree from smaller proof trees; you build the new list of equations by appending together all the lists of equations from the smaller proof skeletons. Explicitly, what you get is this:

$$\langle \frac{\begin{array}{c} P_0 \\ K \longrightarrow h \leftarrow b_1 \wedge \ldots \wedge b_m \end{array} \quad \begin{array}{c} P_1 \\ (?)K \longrightarrow b_1 \end{array} \quad \ldots \quad \begin{array}{c} P_m \\ (?)K \longrightarrow b_m \end{array}}{K \longrightarrow h} ; E_0, E_1, \ldots E_m \rangle$$

A *lifted tableau* is defined as a proof skeleton $\langle P, E \rangle$ together with a substitution σ that satisfies E and associates each variable with a ground term.

A tableau skeleton without any queries in its tree of judgments is a proof skeleton. And a lifted tableau without any queries in its associated tree of judgments is a lifted proof.

Subject to the equation constraints, you can obtain one tableau from another by *extending* it. Suppose you have a tableau skeleton T of

$$\langle P; E \rangle$$

where P contains a leaf $?K \longrightarrow g$. And suppose you have another tableau skeleton T' of

$$\langle \ P' K \longrightarrow g \ ; E' \rangle$$

Modify P by replacing the leaf $?K \longrightarrow g$ in P by the new subtree P'. Call the result P''. Then you extend T by T' by creating the following tableau skeleton:

$$\langle P; E, E' \rangle$$

If you have a substitution σ that satisfies E, you may be able to start from σ' to obtain a substitution that satisfies E and E'. Then the extended skeleton will in fact be an extended tableau.

The top down procedure uses two kinds of extensions in particular. First, you can extend at a leaf

$$?K \longrightarrow f$$

by using a tableau skeleton that matches f against a fact in the knowledge base:

$$\langle K, e \longrightarrow f; e\theta = f \rangle$$

Alternatively, you can extend that same leaf by using a tableau that matches f against the head of a rule in the knowledge base and sets up goals corresponding to each of the atomic clauses in the body of the rule. Such a tableau looks like

$$\langle \frac{K, e \longrightarrow f \leftarrow b_1 \wedge \ldots \wedge b_m \qquad ?K \longrightarrow b_1 \qquad \ldots \qquad ?K \longrightarrow b_m}{K \longrightarrow f}; \\ e\theta = (f \leftarrow b_1 \wedge \ldots \wedge b_m), E_1, \ldots E_m \rangle$$

The top down proof procedure works simply by starting from a tableau skeleton and an associated unifier for its equations:

$$\langle ?K \longrightarrow g; \emptyset \rangle, \emptyset$$

Then it iterates nondeterministically. When it has a complete proof with no query nodes, it returns the result. Otherwise, it chooses a way to extend the tableau skeleton using one of these two patterns, and computes a corresponding new *mgu*. If none exists, it abandons the attempt as a failure.

To compare the definitions in the text on pages 57 and following, think of a *generalized answer clause*

$$yes(t_1, \ldots, t_n) \leftarrow a_1 \wedge \ldots a_m$$

as an *abbreviation* of a tableau skeleton together with a substitution satisfying its equations:

$$\langle P; E \rangle, \sigma$$

For each a_i there is a corresponding query leaf $?K \longrightarrow g_i$ in P such that $g_i\sigma = a_i$ (and vice versa). You can check that the steps outlined in Figure 2.6 of the text describe operations that correspond exactly to what is needed extend a tableau and obtain the new abbreviation for it. The abbreviation can be nice if you want to think about just

what the top down procedure has left to do to prove a goal. However, it is also useful to think about the proof that the procedure is actually building. Then you have to continue to remember the work that you have done even after you finish proving a subgoal, rather than erasing it the way subgoals are erased in resolving generalized answer clauses.