Automatically Proving Predicate Logic Theorems

INF 441 Project

[Draft of April 20, 2017]

Abstract

In this project you will design and implement an *automated theorem prover* (ATP) for predicate logic. A theorem prover is a program that is designed to search for proofs of mathematical statements that are written in a formal language that can be understood by a computer. The program uses simple rules of logical reasoning to determine if and how the statement is true. Several project directions are proposed: improving search performance, interfacing with standard input formats, and writing a proof exporter/printer to a formal language or to natural language (English, French, etc.).

1 Predicate logic: a primer

1.1 Syntax

To the greatest extent possible, we will be using a *formal language* for expressing mathematical statements that can be understood by a computer, analogous to a programming language for expressing algorithms. Specifically, our language will have two *sorts* of expressions: *terms* and *formulas*.

Terms A *term* is like a mathematical object or concept being described. Examples include: numbers, polynomials, vectors, etc. We will be very general in our outlook and consider all terms to be built out of the following primitive components:

- *variables*, which we will write as x, y, etc. A variable is a "dummy" that is often used to express entitites that are seen as *parameters* of a term. For instance, in the polynomial expression $a_0 + a_1\ x + a_2\ x^2$, the occurrences of x are variables.
- *constants*, which we will write as a, b, etc. A constant is not intended as a parameter of the expression but instead stands for some eternal unchanging concept. For example, in the above expression, the occurrence of 2 is a constant. Technically, we may choose to view a_0, a_1, and a_2 as constants in the expression if they are not expected to vary.
- *operators*, also known as *function symbols*, which are a special kind of constant that has arguments. In the expression above, the occurrences of $+$ are operators.

Mathematical expressions are often written using all kinds of notational conventions, but for the purposes of this project we will adopt a very simple grammar of terms, depicted as follows:

$$t ::= x \mid c \mid f(t_1, t_2, \ldots, t_n)$$

where x ranges over variables, c over constants, and f over operators. Hence, the polynomial expression above would be written as a term as follows:

```
plus(a0,plus(times(a1,x),times(a2,exp(x,2))))
```

where `plus`, `times`, and `exp` are operators; `a0`, `a1`, `a2`, and `2` are constants; and `x` is a variable.

Formula A *formula* is a mathematical assertion that can be assessed for its truth. Examples: "the sky is blue," "Lisa's parents are Homer and Marge," and "if all men are mortal and Socrates is a man then Socrates is mortal." Once again, we will use a formal language for formulas (depicted with capital letters A, B, etc.) that are built as follows:

$$\begin{aligned} A, B, \ldots ::= {}& p(t_1, t_2, \ldots, t_n) && \text{(predicates)} \\ \mid {}& A \wedge B \mid \top && \text{(conjunction and truth)} \\ \mid {}& A \vee B \mid \bot && \text{(disjunction and falsehood)} \\ \mid {}& A \Rightarrow B && \text{(implication)} \\ \mid {}& \forall x.A \mid \exists x.A && \text{(universal and existential quantification over terms)} \end{aligned}$$

. . . . [SELECTION] . . . .

$$\frac{\Gamma_1, A, \Gamma_2, [A] \vdash \Delta}{\Gamma_1, A, \Gamma_2 \vdash \Delta}\ \text{left-sel} \qquad \frac{\Gamma \vdash [C], \Delta_1, C, \Delta_2}{\Gamma \vdash \Delta_1, C, \Delta_2}\ \text{right-sel}$$

. . . . [STAGE 1] . . . .

$$\frac{}{\Gamma, [p(t_1, \ldots, t_n)] \vdash \Delta_1, p(t_1, \ldots, t_n), \Delta_2}\ \text{initL} \qquad \frac{\Gamma \vdash C, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma, [C \Rightarrow A] \vdash \Delta}\ {\Rightarrow}\text{L}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, [A \wedge B] \vdash \Delta}\ \wedge\text{L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, [\top] \vdash \Delta}\ \top\text{L} \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, [A \vee B] \vdash \Delta}\ \vee\text{L} \qquad \frac{}{\Gamma, [\bot] \vdash \Delta}\ \bot\text{L}$$

$$\frac{}{\Gamma_1, p(t_1, \ldots, t_n), \Gamma_2 \vdash [p(t_1, \ldots, t_n)], \Delta}\ \text{initR} \qquad \frac{\Gamma, A \vdash C, \Delta}{\Gamma \vdash [A \Rightarrow C], \Delta}\ {\Rightarrow}\text{R}$$

$$\frac{\Gamma \vdash C, \Delta \quad \Gamma \vdash D, \Delta}{\Gamma \vdash [C \wedge D], \Delta}\ \wedge\text{R} \qquad \frac{}{\Gamma \vdash [\top], \Delta}\ \top\text{R} \qquad \frac{\Gamma \vdash C, D, \Delta}{\Gamma \vdash [C \vee D], \Delta}\ \vee\text{R} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash [\bot], \Delta}\ \bot\text{R}$$

. . . . [STAGE 2] . . . .

$$\frac{\Gamma, [t/x]A \vdash \Delta}{\Gamma, [\forall x.A] \vdash \Delta}\ \forall\text{L} \qquad \frac{c \notin \Gamma, \Delta \quad \Gamma, [c/x]A \vdash \Delta}{\Gamma, [\exists x.A] \vdash \Delta}\ \exists\text{L}$$

$$\frac{c \notin \Gamma, \Delta \quad \Gamma \vdash [c/x]A, \Delta}{\Gamma \vdash [\forall x.A], \Delta}\ \forall\text{R} \qquad \frac{\Gamma \vdash [t/x]C, \Delta}{\Gamma \vdash [\exists x.C], \Delta}\ \exists\text{R}$$

Figure 1: Inference rules of the sequent calculus

Predicates are the most primitive assertions about terms that cannot be further structurally decomposed into simpler formulas, These predicates are combined using *logical connectives* into compound formulas. Here is how the informal statements above might look in the formal language of formulas.

- "the sky is blue": `is_blue(sky)`. Here, `is_blue()` is a unary predicate and `sky` is a constant term.
- "Lisa's parents are Homer and Marge": we can use three constants `lisa`, `homer`, and `marge` to represent the entities and express theit relationship as:

  `is_parent(lisa,homer) ∧ is_parent(lisa,marge)`

  where we use a binary predicate `is_parent()` that may be interpreted as asserting that its second argument is a parent of the first argument. Note that the above formula does not rule out the possibility Lisa has other parents. We will additionally need to know that all parents of Lisa are either Homer or Marge. This means that we will need to conjoin the following to the above:

  `∀x. (is_parent(lisa,x) ⇒ (is(x,homer) ∨ is(x,marge)))`

  where we make use of an auxiliary binary predicate `is()` that is interpreted as asserting that its two arguments are the same.
- "If all men are mortal and Socrates is a man then Socrates is mortal": using obvious syntactic classes, this may be represented as:

  `(∀x. is_man(x) ⇒ is_mortal(x)) ∧ is_man(socrates) ⇒ is_mortal(socrates)`

1.2 Rules of reasoning

Some formulas are *theorems*, meaning that they can be shown to hold no matter what interpretation we give to constants, operators, and predicates. The third example above is a theorem: its truth doesn't depend on whether `is_mortal(`x`)` actually means that x is mortal. Other formulas are not theorems, which means that the formula is contradictory under *some* interpretation of the constants, operators, or predicates. For instance, `is_blue(sky)` is not a theorem if the interpretation we give to `is_blue(`x`)` is that x is a solid. A *proof system* is a system for determining if a given formula is a theorem by simply analyzing the *syntax* of the formula.

There have been many proposed proof systems for predicate logic. In this project we will use a system that is particularly suited to computerized search, known as the *sequent calculus*. The core concept in the sequent calculus is the notion of a *sequent*, which may be depicted in the following general form:

$$\underbrace{A_1, A_2, \ldots, A_m}_{\Gamma} \vdash \underbrace{C_1, C_2, \ldots, C_n}_{\Delta}$$

where each of the A_i and C_j are formulas. (We will use the notational convention of writing A to the left of $\vdash$ and C to the right, but all kinds of formulas are allowed on both sides.) That is to say, a sequent consists of two lists of formulas, a *left* list that we call the *assumptions* (and depict as Γ) and a *right* list that we call the *goals* (and depict as Δ). The way to read a sequent is that *all* of the assumptions must *entail* (i.e., logically lead to) *some* of the goals. While both assumption and goal lists can be empty in general, in this project we will only limit our attention to the case of a non-empty goal list.

The reasoning calculus that uses these sequents will be depicted using a standard notational convention known as an *inference rule*.. You have already seen inference rules in this course for type-checking. All inference rules have the form:

$$\frac{\text{premise}_1 \quad \cdots \quad \text{premise}_n}{\text{conclusion}}\ \text{rule-name}$$

where each of the premises and the conclusion are sequents, and the rule has a name indicated on the side. A sequent is *derivable* if it is the conclusion of an inference rule where all the premises (if any) are also derivable. Note that the conclusion of a premise-less rule is always derivable. The way to read an inference rule is from conclusion to premises: *in order to prove the conclusion, it suffices to prove the premises*. At the start of proof search, the overall conclusion is the theorem you want to prove, which you put in a list of *obligations*. Then, in a loop you select (and remove) an obligation, select an inference rule that may have proved the obligation, and then add its corresponding premises as new obligations. If you get stuck, you go back to an earlier choice you made of inference rules and pick a different one, which is often called *backtracking*.

To help to make the search aspects of the calculus clear, we will use two slight variations of our notion "sequent":

left-selected (or just *left*) sequents	$\Gamma, [A] \vdash \Delta$
right-selected (or just *right*) sequents	$\Gamma \vdash [C], \Delta$

We will then say that the only way to prove a sequent is to first prove a selected form using one of these two rules:

$$\frac{\Gamma_1, A, \Gamma_2, [A] \vdash \Delta}{\Gamma_1, A, \Gamma_2 \vdash \Delta}\ \text{left-sel} \qquad \frac{\Gamma \vdash [C], \Delta_1, C, \Delta_2}{\Gamma \vdash \Delta_1, C, \Delta_2}\ \text{right-sel}$$

The formula that is selected and copied into the brackets is called the *principal formula*. Thus, to prove a sequent, it suffices to enumerate all the possible principal formulas and try to prove each corresponding selected sequent form.

The full list of inference rules for selected sequents is shown in figure 1. For a left sequent, we apply a left rule whose name ends in L; for a right selected formula we use a right rule whose name ends in R. The proof system is divided into two *stages* for this project. In stage 1, discussed in section 4.1, we ignore terms and concentrate on the *propositional* core of the system, which is sometimes called *propositional logic*. In stage 2, we will incorporate terms; this stage is discussed in section 4.2.

2 Implementing the *kernel* by leveraging the ML type system: the LCF approach

The first task in the implementation is to have a trustworthy implementation of the logical reasoning system. If it is done correctly with judicious use of type abstraction, then no matter what the rest of the system does the type system of OCaml will guarantee that only correct theorems are ever derived. This will make our theorem prover *correct by construction*. This use of the ML type system goes back to the very origin of ML to build the LCF theorem prover at Edinburgh in the mid 1980s. Many standard features such as type-inference and the ML module system came out of this project and have had a huge impact on generations of programming languages and on software engineering in general.

Start by defining a module called `Syntax` whose module type might have the following form:

```
module type SYNTAX = sig
  type term = private
    | Variable of string
    | Constant of string
    | Operator of string * term list

  type formula = private
    | Predicate of string * term list
    | And of formula * formula | True
    | Or of formula * formula | False
    | Implies of formula * formula
    | Forall of string * formula
    | Exists of string * formula
end
```

The code you write in this module will need to be bug-free, so it is important that you keep it to a minimum and go over it carefully. The use of `private` above guarantees that no terms or formulas can be constructed except through the functions provided in this module. You may then want to implement *smart constructors* in the module to build terms and formulas. For instance, suppose you want to make sure that operators and predicates are always applied to the correct number of arguments. You could enforce this by exposing functions:

```
  val operator : string -> term list -> term
  val predicate : string -> term list -> term
```

whose implementations would make sure that the right number of arguments have been provided. If you are feeling adventurous, you may also implement a light type system for terms and formulas and make sure that only well-typed terms and formulas can be built.

Next, you should implement the rules of inference in a module called `Kernel`, say, that has this type:

```
module type KERNEL = sig

  type sequent = {
    left : formula list ;
    right : formula list ;
  }

  type theorem
  exception Invalid

  val conclusion : theorem -> sequent

  val sel_left  : theorem -> pos:int -> principal:formula -> theorem
  val sel_right : theorem -> pos:int -> principal:formula -> theorem

  val init_left  : sequent -> principal:formula -> theorem
  val init_right : sequent -> principal:formula -> theorem

  val and_left  : theorem -> theorem * formula
  val true_left : theorem -> theorem * formula

  (* and so on ... *)

end
```

The type `theorem` is left abstract, so that the only way to construct them is by using the functions in the module, each of which corresponds to a rule of the sequent calculus. The implementation of the `theorem` type need not be anything special; it can just be:

```
type theorem = sequent
```

The functions in the `Kernel` module closely mirror the inference rules in figure 1. To illustrate, take the ∧L rule:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, [A \wedge B] \vdash \Delta}\ \wedge\mathrm{L}$$

To create the corresponding theorem, we would take `th : theorem` that corresponds to $\Gamma, A, B \vdash \Delta$, and return a pair `(th', f)` such that `th'` corresponds to $\Gamma \vdash \Delta$ and `f` corresponds to $A \wedge B$. All code that uses the `Kernel` implementation will therefore only be able to construct `theorems` that correspond to derivable sequents in the calculus.

3 Proof search: iterative deepening

The algorithm used to search for proofs is pretty simple: iterative deepening with a depth bound, which is a parameter that the user should be able to set. The main search function has this signature:

```
val search : conclusion:Kernel.sequent -> bound:int -> Kernel.theorem
```

It does the following:

1. If `bound < 0`, then fail; otherwise:
2. Apply the sel-left and sel-right rules to compute the possible principal formulas on the left and the right.
3. For each left or right principal formula, compute what the possibilities are for left and right rules on them. For each possibility, recursively call search with that computed premise sequent and a bound of `bound - 1`.
4. For whichever recursive call succeeds, apply the corresponding `Kernel` rule to create the corresponding `theorem`.

You may find it useful to write the `search` method with a *pair* of continuations, one each for success and failure. That is to say, write a function with type:

```
val search_aux : conclusion:Kernel.sequent -> bound:int ->
  succeed:(Kernel.theorem -> 'a) ->
  fail:(unit -> 'a) ->
  'a
```

Then, whenever you have more than one possibility at search, continue with the first possibility and use the failure continuation for the remaining possibilities. Finally, the "main loop" of the prover is to repeatedly call `search` with increasing values of the `bound` parameter.

4 Stages of development

Instead of trying the full language of formulas at once, you may find it useful to do the project in *stages*.

4.1 Stage 1: propositional logic

The most basic core of the search algorithm is already exposed with no terms or predicates with arguments This fragment is often called *propositional* or *Boolean logic*. Here are some test problems that you can try to prove with your prover.

1. $\bot \Rightarrow p$, the famous classical law of *ex falso quodlibet* that from falsehood anything/everything follows.
2. $p \vee (p \Rightarrow \bot)$, another famous law known as *excluded middle*. A generalization of this is $p \vee (p \Rightarrow q)$.
3. $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$, also known as *Peirce's Law*.
4. $((p \Rightarrow \bot) \Rightarrow \bot) \Rightarrow p$, known as the principle of *elimination of double-negation*.
5. $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$, which is one of Hilbert's axioms of propositional logic and is also related to the *S combinator* from combinatory logic.
6. $((p \wedge q) \Rightarrow \bot) \Rightarrow ((p \Rightarrow \bot) \vee (q \Rightarrow \bot))$, which is one of *De Morgan's Laws*.
7. $((p_1 \wedge q_1) \vee (p_2 \wedge q_2)) \Rightarrow ((p_1 \vee p_2) \wedge (q_1 \vee q_2))$, known as the *medial rule*.

You should see that your prover terminates for all theorems of propositional logic. (It is OK for it to loop on non-theorems.)

4.2 Stage 2: dealing with terms

When adding terms to the mix, there are a pair of complications. First in the $\exists$R and $\forall$L rule, one has to guess the term t that is used to instantiate the quantifier, called the *witness* During proof search it would be very expensive to start enumerating and testing all possible terms. One may try to *guess* the witness by looking at other terms that occur in the sequent, but this is not guaranteed to work.

One method that is known to work is to replace the term with a *meta-variable* (written X, Y, etc.), which stands as a place-holder for a term. The selection of the precise term is delayed to the *init* rule, where now we are no longer merely checking that the assumption and goal formula are identical but that they are *unifiable*, which is to say that there is a way to instantiate the meta-variables in the two formulas in such a way that the terms become equal. There is a general algorithm for this unification that guarantees maximum generality: computing *most general unifiers* (mgu). You have already seen mgus in the type inference algorithm in class.

The second complication is with the $\forall$R and $\exists$L rules; these rules extend the domain of discourse by inventing fresh (but arbitrary) constants that do not occur in the conclusion sequent to instantiate the quantifiers. However, this has an interaction with meta-variables, which is illustrated with the sequent $\forall x.\exists y.p(x,y) \vdash \exists y.\forall x.p(x,y)$. This sequent should be unprovable, since it doesn't hold for many predicates; for instance, imagine if the predicate $p(x,y)$ meant that $x = y$. There is an apparent "proof" that can be built for this sequent as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(\mathsf{X},c) \vdash p(d,c)}}{p(\mathsf{X},c) \vdash \forall x.p(x,c)}\,*}{p(\mathsf{X},c) \vdash \exists y.\forall x.p(x,y)}}{\exists y.p(\mathsf{X},y) \vdash \exists y.\forall x.p(x,y)}}{\forall x.\exists y.p(x,y) \vdash \exists y.\forall x.p(x,y)}$$

This proof can apparently be finished by instantiating X with d. But this would make the instance of $\forall$R marked with a $*$ invalid! To fix this, meta-variables must track what constants are allowed to occur in possible instances.

Here are a few test problems for your first-order prover.

1. $(\exists x.\forall y.p(x,y)) \Rightarrow (\forall y.\exists x.p(x,y))$, the converse of the above implication that *is* true.
2. $(\forall x.((p(x) \Rightarrow \bot) \Rightarrow \bot)) \Rightarrow (((\forall x.p(x)) \Rightarrow \bot) \Rightarrow \bot)$, known as the principle of *double-negation shift*.
3. $\exists x.(p(x) \Rightarrow \forall y.p(y))$, which is Smullyan's *Drinker Paradox*. Imagine if $p(x)$ stands for "x is drunk"; then, the formula says that there is someone such that if they are drunk then everyone is drunk. (A veritable soul of the party!)
4. You can translate many famous "logic puzzles" into predicate logic problems. For instance, consider the classic puzzle of a farmer with a head of cabbage, a goat, and a wolf, trying to cross a river on a boat that allows him to take at most one thing at a time, in such a way that no animal is left alone with its prey. Define constants `a` and `b` for the banks, and a 4-ary predicate `pos` that defines the positions of the four entities in the order farmer, wolf, goat, cabbage. Imagine a list Γ consisting of the following formulas:

$$\left.\begin{array}{ll}\texttt{pos(a,b,a,b)} \Rightarrow \texttt{pos(b,b,a,b)}, & \texttt{pos(b,b,a,b)} \Rightarrow \texttt{pos(a,b,a,b)},\\ \texttt{pos(b,a,b,a)} \Rightarrow \texttt{pos(a,a,b,a)}, & \texttt{pos(a,a,b,a)} \Rightarrow \texttt{pos(b,a,b,a)},\end{array}\right\}\text{farmer goes alone}$$

$$\left.\begin{array}{ll}\texttt{pos(a,a,a,b)} \Rightarrow \texttt{pos(b,b,a,b)}, & \texttt{pos(b,b,a,b)} \Rightarrow \texttt{pos(a,a,a,b)},\\ \texttt{pos(b,b,b,a)} \Rightarrow \texttt{pos(a,a,b,a)}, & \texttt{pos(a,a,b,a)} \Rightarrow \texttt{pos(b,b,b,a)},\end{array}\right\}\text{farmer takes wolf}$$

$$\left.\begin{array}{ll}\texttt{pos(a,b,a,a)} \Rightarrow \texttt{pos(b,b,a,b)}, & \texttt{pos(b,b,a,b)} \Rightarrow \texttt{pos(a,b,a,a)},\\ \texttt{pos(b,a,b,b)} \Rightarrow \texttt{pos(a,a,b,a)}, & \texttt{pos(a,a,b,a)} \Rightarrow \texttt{pos(b,a,b,b)},\end{array}\right\}\text{farmer takes cabbage}$$

$$\left.\begin{array}{l}\forall x.\ \forall y.\ \Big(\texttt{pos(a,}x\texttt{,a,}y\texttt{)} \Rightarrow \texttt{pos(b,}x\texttt{,b,}y\texttt{)}\Big),\\ \forall x.\ \forall y.\ \Big(\texttt{pos(b,}x\texttt{,b,}y\texttt{)} \Rightarrow \texttt{pos(a,}x\texttt{,a,}y\texttt{)}\Big).\end{array}\right\}\text{farmer takes goat}$$

Then, the problem is solvable if you can derive this sequent:

$$\Gamma, \texttt{pos(a,a,a,a)} \vdash \texttt{pos(b,b,b,b)}.$$

5 Ideas for extensions

5.1 Reading TPTP format problems

You can find many problems in the *Thousands of Problems for Theorem Provers* (TPTP) library:

`http://www.cs.miami.edu/~tptp/`

This library has a certain easy to parse input format. You can use it to load problems in the FOF class, which falls in the language of predicate logic. Each problem file contains a *challenge rating*, which is a floating point number between 0.0 (trivial) and 1.0 (next to impossible for automated theorem provers). You may want to use a parser generator for OCaml such as `ocamlyacc` or Menhir.

5.2 Improving search performance

You may have noticed that without any further tweaks your prover is pretty slow on most problems. One way to speed it up is to perform the sel-left and sel-right rules less often. For instance, you can replace the $\Rightarrow$R rule with two rules:

$$\frac{(C \text{ is a} \Rightarrow) \quad \Gamma, A \vdash [C], \Delta}{\Gamma \vdash [A \Rightarrow C], \Delta} \qquad \frac{(C \text{ is not a} \Rightarrow) \quad \Gamma, A \vdash C, \Delta}{\Gamma \vdash [A \Rightarrow C], \Delta}$$

The parenthesized premise is not a sequent *per se* but is a *side condition*, which is to say it is a condition that must be satisfied in order for the rule to be usable in search. The effect of this change is to make sure that whenever you have a chain of implications selected on the right, you can get rid of the whole chain at once. Another optimization is to modify the sel rules to sometimes delete the selected formula instead of merely copying it:

$$\frac{(A \text{ is deletable}) \quad \Gamma_1, \Gamma_2, [A] \vdash \Delta}{\Gamma_1, A, \Gamma_2 \vdash \Delta} \qquad \frac{(C \text{ is deletable}) \quad \Gamma \vdash [C], \Delta_1, \Delta_2}{\Gamma \vdash \Delta_1, C, \Delta_2}$$

The benefit is that deletable formulas are only allowed to be principal at most once on any proof attempt, which avoids useless repetition. If the formula is not deletable, of course, we have to fall back to the existing sel rules. Which formulas are deletable and on which sides? Some hints: both $\wedge$ and $\vee$ are deletable on both sides, $\Rightarrow$ is *not* deletable on the left, and $\exists$ is *not* deletable on the right. Try to figure it out for all the other formulas on both sides.

One final optimization is to change the data structure of the sides of the sequents from *lists* to *sets* of formulas. For instance, we can write the $\vee$L rule like so:

$$\frac{\Gamma \cup \{A\} \vdash \Delta \quad \Gamma \cup \{B\} \vdash \Delta}{\Gamma, [A \vee B] \vdash \Delta}$$

With this change, you will never have multiple copies of the same formula on each side of the sequent, which also helps to reduce useless repetition.

5.3 Presenting proofs

Sometimes you may want to see the proofs; for instance, in the logic puzzle example above, you may want to see the solution that is encoded in the proof that was found. Without making *any* changes to the search method, you can replace the kernel with a proof-recording kernel that can be used to recover the precise derivation of the conclusion that was found. One way to do this is to enrich your `KERNEL` implementation with something like this:

```
module type PROOF_KERNEL = sig
  include KERNEL

  type proof =
    | Sel_left of sequent * int * theorem
    | Sel_right of sequent * int * theorem

    | And_left of theorem
    | And_right of theorem * theorem

  (* and so on for every rule of the calculus *)

  (* all theorems must now be able to produce the proof *)
  val proof : theorem -> proof
end
```

You can then present the proof to the user in a number of ways: either simply pretty-print the OCaml value of type `proof`, or turn it into a LaTeX source file that shows the proof tree built in terms of the rules of figure 1. If you are feeling adventurous, a harder alternative is to consider how you would present such a proof in ordinary natural language. For this you would have to consider all the inference rules in figure 1 and figure out how you would explain them in words. You would probably also need to extend your notion of the sides of the sequent from lists of formulas to maps from formula *labels* to formulas, since humans find it much harder to reason positionally in lists than to use names. (Imagine how unreadable most programs would be if all variables were named `x1`, `x2`, ...)