# Tutorial Guide to Oyster Proof Development System

Jane Hesketh

November 25, 1988

This is not meant to be a complete introduction to Oyster, just something to get you started. You will also want to look at Christian Horn's report on the Oyster system, but that's more of a reference manual.

Oyster is a rational reconstruction of the Cornell Nuprl system. It is written in Prolog, and run at the Prolog prompt level, so it is controlled by using Prolog predicates as commands. Proof tactics can be built as Prolog programs, incorporating Oyster commands, which are, of course, just Prolog predicates.

## 1 Introduction

It's probably best to start by doing a proof, and look at what the steps are and what it all means[1].

The following proof is of the propositional theorem

$$a \& b \rightarrow a \vee b$$

To start Oyster, just type

```
    oyster
```

as a unix command. At this point you have started up the program, and it is displaying the usual Prolog prompt.

But you can't do anything yet, because Oyster has no theorem to work on. You have to create a theorem, by typing:

[1] I shall try to stick to Christian's font conventions, to make things as clear as possible: The logic itself is documented in sans serif font, using smaller fonts to document utility predicates, which are publicly available, but of no general interest, and greek letters, *italics*, or special mathematical symbols to hide sensitive predicates from the unauthorised user. Examples are given in `teletype` like fonts

```
create_thm(firstproof,user).
```

The first argument is the name you're going to use for this theorem within Oyster. The second is the source of the theorem text, in this case the keyboard, so now type:

```
[] >> a:u(1)-> b:u(1)-> ((a#b) -> (a\b)).
```

Don't forget the full-stop, you're in Prolog, remember.

What does this mean? Well, it's a Prolog term (the full stop just signifies the end of the term as usual), with various infix connectives defined as Prolog operators:

- the `>>` corresponds to $\vdash$ or the sequent arrow, so it is separating the (empty) list of hypotheses `[]` from the conclusion, which at this stage contains all the information.
- The conclusion, `a:u(1)-> b:u(1)-> ((a#b) -> (a\b))` has `a:u(1)->` as its topmost bit. This means *for all a in u(1)*. So it states a type for *a*, that it is in universe *u(1)*. This is necessary because all objects used in the theorem and proof must be well-formed, and so must have a type. *u(1)* contains all primitive propositions.
- `a:u(1)->` is dominating `b:u(1)-> ((a#b) -> (a\b))`, the topmost bit of which, `b:u(1)->` is very similar, and just says *for all b in u(1)*.
- And then inside them comes `((a#b) -> (a\b))`, which is the $a\&b \rightarrow a \vee b$ you've been expecting. `->` corresponds to $\rightarrow$ or $\supset$, `#` to & or $\wedge$, and `\` to $\vee$.

Now tell it that this is the theorem you want to work with:

```
select(firstproof).
```

and have a look at it:

```
display.
```

the `display` command displays the current node of the proof tree. At the moment, you only have the root, because you haven't done anything.

The first line contains the name of the proof (`firstproof`), the position in the tree of the node you're looking at (here `[]` since this is the root), and the status of the proof subtree below this node (incomplete since you haven't done anything yet).

The next line is the expression to be proved.

The bottom line indicates that no instruction or inference rule has yet been applied at this node. That's what you have to do next.

What you have to do now is develop the proof tree, by telling Oyster what rule of inference to use at each node. Some of these rules will result in new nodes (things to prove) being attached to the tree. Some will complete branches. You have to traverse the tree in order to do this, because you can only issue an instruction for the node you are at. The proof is complete when there is nothing left to prove, i.e. all the branches end in completed proofs. This will become clearer as you go along.

First however, you should disable some of Oyster's automatic tactic application, or it will do a lot of the proof for you, and it will be hard to see what's going on. Each time you tell Oyster to apply an inference rule, it also uses whatever its autotactic is. To stop this we'll set the autotactic just to be the identity tactic, so that it does not affect the expression[2]. Type:

```
apply(autotactic(idtac)).
```

and then `display` the node again. Various things have happened.

1. A child node has appeared (the bottom two lines). This is because the instruction to set the autotactic is attached to the current node, and what remains to be proved (everything here) is assigned to a subgoal represented by the child. Notice the child has an identity (`[1]`) and a status of its own.

2. The current node's status has become partial, since it now has some nodes below it.

3. The current node's instruction slot has been filled in according to what you told it to do. Although you have only set the autotactic at this node, it will apply all the way down the tree from here until you reset it.

To continue with the proof, you must move down to the child. If you accidentally type in another instruction here, you will overwrite the autotactic setting instruction you have just issued. Type:

```
down.
```

[2]It is a good idea to do this in all your proofs to start with. Eventually you will want the autotactic, but you can still have the option of seeing what a rule would do without the autotactic by applying it *pure* – e.g. by typing `pure(intro)` for *intro* without accompanying autotactic

and display the node[3] Things look pretty much as before. The main changes are in the top line which now reflects that you have moved through the tree (it shows your path `[1]`) and declares the setting of the autotactic. But since this is a new node, it has no associated instruction. Now we can get down to some real work.

What we have is a complex expression of which we need to access the components in order to arrive at subexpressions which are immediately true. The expression we want to operate on is `(a#b)->(a\b)` but that is dominated by two universal quantifiers `a:u(1)->` and `b:u(1)->`. They can be stripped off and turned into hypotheses by using the *intro* rule twice. Type:

```
        intro,display.
```

to make the first universal quantifier a hypothesis, and display the result of having done this. You'll see how Oyster has filled in the result of your instruction, and generated two new things for you to prove. The first is the partially dismantled expression, with its new hypothesis, which postulates some typical element/proposition *a* in *u(1)* The second is a well-formedness goal, that this universe *u(1)* is legitimate.

Let's digress a moment and try and get a picture of this. Essentially what's happening is a backwards proof. Consider a Gentzen Sequent Calculus proof tree:

$$\dfrac{\dfrac{\dfrac{A \vdash A}{A\&B \vdash A}\,\mathit{Antecedent\&introduction}}{A\&B \vdash A \vee B}\,\mathit{Succedent} \vee \mathit{introduction}}{\vdash A\&B \to A \vee B}\,\mathit{Succedent} \to \mathit{introduction}$$

The Oyster proof starts with the expression to be proved at the root of its proof tree, and constructs the tree back towards the leaves. In GSC terms this would be starting from the bottom and going to the top. (Confusingly Oyster represents its proof trees the opposite way round from GSC, i.e. down towards simpler conclusion expressions instead of up). So telling Oyster to *intro* means it must apply an intro rule **backwards, i.e. in reverse**. The effect of this is not to introduce, but to **remove** the topmost connective, since the tree is being

[3]Since this is Prolog, you could give both instructions at once: `down,display`.

developed backwards. Oyster extends the tree to produce child node(s) which when manipulated (forwards) according to the inference rule you have given as an instruction, would yield the expression you started with. You keep developing the tree backwards until all the leaves are satisfactorily accounted for. It isn't usually necessary to specify which intro rule you're using, because the context usually determines which one it must be. All intro rules in Oyster operate on the conclusion expression. There are other rules for manipulating hypotheses. End of digression.

Back to the proof, which looks like this:

```
firstproof : [1] partial autotactic(idtac)
>> a:u(1)->b:u(1)->(a#b)->a\b
by intro

  [1] incomplete
  1. a:u(1)
  >> b:u(1)->(a#b)->a\b

  [2] incomplete
  >> u(1)in u(2)
```

Subgoal 1 is a partly opened version of what we started out with. The outmost universal quantifier has been removed from the succedent expression, and a corresponding hypothesis about the variable quantified over has been added as a hypothesis.

Subgoal 2 is necessary to establish the well-formedness of the type of a in the new hypothesis in Subgoal 1. This subgoal is not very interesting, and if you hadn't disabled the autotactic, it would have been completed for you. Universe well-formedness is specially recognised by the system, and it generates the next higher level of universe for `u(1)` to be in.

Move down the tree again to this first subgoal, and display. `down` will automatically move you to the first subgoal here.[4]

Now use the `intro` command again to strip off the next universal quantifier. `display` what you've done. Go down to the first subgoal, which is the interesting one, and display it. The conclusion is now a simple expression corresponding to $(a\&b) \rightarrow (a \vee b)$. Another `intro` splits the conclusion apart and makes the

[4]If you had wanted to go to the second one, you'd need to use `down(2)`, `down/1` being the predicate that takes you down to the *i_th* subgoal. Other useful tree-traversing commands are `next`, `up` and `top` to get you to the next open subgoal, the one above in the tree, and the top of the tree respectively. `pos(Path)` gets you to position Path, where Path is a list of numbers.

left hand argument of -> a hypothesis. Notice the way it does this, creating a new object v0 of *type* a#b. This necessitates the other subgoal to prove the well-formedness of this type. Display what you've done, and go down to the first subgoal and display it.

This is what you should see:

```
firstproof: [1,1,1,1] incomplete autotactic(idtac)
1. a:u(1)
2. b:u(1)
3. v0:a#b
>> a\b
by _
```

What are you expecting to do here? Split the conclusion expression again? Try `intro` again. It doesn't work. Why?

The answer is to do with this being a proof in a constructive logic. We can't prove $a \vee b$ without saying which of $a$ or $b$ we're proving. This is an example of where we will do an `intro`, but we will have to give it some arguments and be more explicit. To find out what your options are, type

```
    apply(X).
```

and for each instantiation of `X` you are offered, type `;` until there are no more. When you have been typing `intro` this has been taken by Oyster as shorthand for `apply(intro)` where `apply/1` is the predicate which applies a rule of inference to the goal. Using prolog with the command argument uninstantiated lets it tell you what commands you could use – without performing them. It won't tell you all the commands, so don't rely on it too much, but it's useful for the common ones, and their format if you can't remember it.

So looking to see the obvious rules you could apply will suggest 3 options:

1. Split the $\vee$ and prove its left-hand disjunct;

2. Split the $\vee$ and prove its right-hand disjunct;

3. Split the & in the list of hypotheses, so that each conjunct is directly available.

Clearly, you'll have to split the disjunct, so do that. It doesn't matter whether you pick the left or right disjunct, since the problem is symmetrical. Say you take the right–hand one, type:

```
intro(at(1),right).
```

and display the result. It has set up two new subgoals, one to prove b, and one to prove that the rest of the `a\b` expression was well–formed. Move down to the next goal and display it.

It corresponds to $a\&b \rightarrow b$ so you must split the & to get access to the $b$ in the hypothesis which matches the $b$ in the conclusion. Use `apply(X)` again to see how to do this. This time only one rule applies, an elimination rule. This acts on a hypothesis, and (remember we're operating backwards) introduces some new hypotheses which express parts of the selected hypothesis. So if you think of it working forwards it would actually eliminate a few hypotheses, hypotheses 4, 5 and 6 are eliminated in favour of hypothesis 3.

Go down and display again. You're home and dry. hypothesis 5 is $b$ and the conclusion is $b$. This is a valid leaf. Type `intro,display.` and this branch should be completed.

If you want, you can now go back and tidy up the various well-formedness subgoals you left before. You could have done them at the time, but it would have broken up the flow of the proof.

You may find it interesting to repeat this proof with the default autotactic (repeat intro) in place.

## 2 Making Life Easy For Yourself

### 2.1 Shorthand

As I have already mentionned, some commands are shortened forms of others, where an extra argument can be worked out automatically. This concerns the cases where new entities are being created, and given names – i.e. all arguments `new[ ... ]` . Also where the universe level is being declared with an `at(_)` argument – it doesn't always get this right, since it usually defaults to u(1).

In general, most commands are short for `apply( ** )`, where `**` is the command you actually type in. Not all have this shorthand though, and sometimes it is necessary to type in the full form.

### 2.2 Loading and Saving from Files

To save the current state of a proof, type:

```
save_thm(firstproof,'first.proof').
```

The first argument is the internal name of the theorem you used to `select`. The second argument is any filename you want. `load_thm` has the same arguments and loads the saved file back in and gives it a name.

For either of these, the theorem can be partial.

Another way of getting a theorem in from a file is `create_thm`. You've already used this for reading in from the keyboard, but the second argument may be a filename instead of "user".

After getting a theorem in, you must select it.

## 2.3 Getting A Printed Copy Of What You've Done

```
snapshot('myproof.snap')
```

Will produce a readable version of the proof tree *below the current node* in a file called myproof.snap – so remember to go up to the top if you want the whole thing. Then you can print it out if you want.

## 2.4 Interface

You can customize a lot of the commands to shorter names or to be a macro for something like applying a rule, moving down and displaying the new subgoal. An example set of Prolog predicates to do this is in the Oyster distribution directory /usr/local/lib/oyster in utilities/interface.pl

You can also run Oyster through emacs, there is a file in the Oyster distribution directory, utilities/pl.emacs This makes life much easier. It gives you:

- bracket matching while typing to your Prolog system,
- line editing (just use normal Emacs commands)
- looking back at previous bits of the interaction (just use normal Emacs paging/scrolling commands)
- re-using old command lines (just position the cursor anywhere on an old command line, and hit return. That command will then be copied to the bottom line of the buffer, you can edit it if you want, and send it to the Prolog interpreter.

Both of these are courtesy of Frank van Harmelen

- Using induction. If you have a hypothesis of the form `x:pnat` the instruction `elim(x)` will produce subgoals corresponding to the base and step cases appropriate to the natural numbers. If you check this using `apply(X)` you will see the full form of the command, which allows you to specify the names of the new entities created. If you don't specify them, the system generates names if your command was unambiguous. You will see the origin of the new hypotheses tacked on to the front of them.
- The compute instruction. This is what lets you manipulate functions according to their definitions. If you have a conclusion containing an expression you wish to fold or unfold in terms of its definition, use the `compute` instruction. This takes a single argument – the current conclusion with any subterms to be (un)folded replaced by `[[fold(plus)]]` , `[[fold(X)]]` or `[[unfold]]` as required.

  So if your current conclusion is:

  ```
  plus(0,plus(y,z))=plus(plus(0,y),z)in pnat
  ```

  and you say:

  ```
  compute([[unfold]]=plus([[unfold]],z)in pnat).
  ```

  The plusses will be replaced by p_ind's:

  ```
  p_ind(0,plus(y,z),[~,v,s(v)])=plus(p_ind(0,y,[~,v,s(v)]),z)in
  pnat
  ```

  This may not look a lot nicer at first, but some of this is the raw form that Oyster works in. If you now tell it to `simplify` (no arguments), it can do so on the p_ind terms. The other subgoals below this can be solved by intro, except one more compute to turn `plus` into `p_ind` .
- The *subst* instruction so that you can use the induction hypothesis' equality. This instruction is another fiddly one. Suppose after the odd unfolding and folding compute, you've reached this:

  ```
  1. plus(x,y)<==>p_ind(x,y,[~,v,s(v)])
  2. x:pnat
  3. y:pnat
  4. z:pnat
  5. x~>v0:pnat
  6. x~>v1:plus(v0,plus(y,z))=plus(plus(v0,y),z)in pnat
  ```

```
>> s(plus(v0,plus(y,z)))=plus(s(plus(v0,y)),z)in pnat
by _
```

What you really want to do is use the induction hypothesis to substitute for some of the conclusion. You do this by specifying the substitution you wish to apply to the conclusion, and Oyster leaves you with the obligation of justifying your substitution:

```
        subst(over(p,s(p)=_ in pnat),
               (plus(v0,plus(y,z))=plus(plus(v0,y),z)in pnat))
```

(This is only split over 2 lines to make it easier to read). The second argument is the substitution you want to make, RHS for LHS. The first argument is where to make the substitution – a new name *p* is introduced and used to indicate the subterm (of the whole conclusion) within which substitution is to take place.

This gives you three subgoals:

(a) `>> plus(v0,plus(y,z))=plus(plus(v0,y),z)in pnat`
Justify your substitution. In this case it is one of the hypotheses and the instruction `hyp(v1)` tells it that.

(b) `>> s(plus(plus(v0,y),z))=plus(s(plus(v0,y)),z)in pnat`
This is the conclusion substituted as you requested. A bit of folding unfolding and simplifying should sort this out.

(c)
```
       7. p:pnat
  >> s(p)=plus(s(plus(v0,y)),z)in pnat in u(1)
```

A well-formedness goal. Some intros and unfolds clear this up.

## 4 Reserved Words

Don't use any of the following as variable or function names: atom, void, pnat, int, axiom, nil, ext, >>, <==>, +, -, *, /, mod, &, ::, of, list, u, lambda, rec, spread, decide, p_ind, ind, list_ind, rec_ind, term_of. Most of them you probably wouldn't pick anyway, to avoid confusion, but it's easy to use u accidentally. I did so while trying out proofs for this and spent quite a while being baffled by completely unhelpful syntax error messages.

## 5 Synthesis Proofs

Are much the same as any other proof except that they involve proving the existence of a function to do whatever it is you want synthesised. The existence proof ends up building the function for you. At some point, you will be required to provide an actual object which demonstrates the existence, and this may be quite hard. Usually the object you introduce will incorporate the functionality you want. Much of the rest of the proof will be about verifying the object you provided.

Here is an example of this crucial introduction step in a synthesis of *append*:

```
app : [1] complete autotactic(idtac)
>> append:(pnat list->pnat list->pnat list)#
    l1:(pnat list)->
      (append of nil of l1=l1 in pnat list)#
      h:pnat->l2:pnat list->
         append of (h::l1) of l2 = h::append of l1 of l2 in pnat list
by intro(lambda(l1,lambda(l2,list_ind(l1,l2,[h,t,v,h::v]))))

  [1] complete
  >> (lambda(l1,lambda(l2,list_ind(l1,l2,[h,t,v,h::v])))
     in (pnat list->pnat list->pnat list))

  [2] complete
  >> l1:(pnat list)->
       ((lambda(l1,lambda(l2,list_ind(l1,l2,[h,t,v,h::v])))of nil of l1)
            = l1 in (pnat list))#
        h:pnat->l2:pnat list->
            (((lambda(l1,lambda(l2,list_ind(l1,l2,[h,t,v,h::v])))
                                                     of (h::l1)) of l2)
               =h::(lambda(l1,lambda(l2,list_ind(l1,l2,[h,t,v,h::v])
                                                 of l1 of l2 in pnat list

  [3] complete
  1. append:pnat list->pnat list->pnat list
  >> ((l1:(pnat list)->
        (append of nil of l1=l1 in pnat list)#
        h:pnat->l2:pnat list->
          append of (h::l1) of l2=h::append of l1 of l2 in pnat list)
     in u(1))
```

The original goal here may be paraphrased as:

∃ append (of type pnat list -> pnat list -> pnat list) such that
  ∀ l1 (of type pnat list)

```
      append(nil)(l1) = l1 in pnat list
  and ∀ h (of type pnat)
          ∀ l2 (of type pnat list)
              append(h::l1)(l2) = h::append(l1)(l2) in pnat list
```

New things here:

- *h:pnat# expression* for ∃ h (of type pnat) such that expression
- *::* is the list constructor like . or cons.
- *of* is function application. *of* being an operator between two entities, functor and argument, makes multiple argument functions awkward. There are two ways round this. One is to write a function of the first argument and apply that to the next argument etc. e.g. *(append of a) of b*. This corresponds to `append:(A->B)->C` .

  The other is *append of (a&b)*, which corresponds to `append:(A#B)->C` . You may find this more difficult to handle.
- *list_ind* like p_ind but for lists. Its three arguments are:
  1. recursion variable
  2. value of function when recursion variable is the empty list
  3. a four-tuple consisting of variables representing the head of the recursion variable, its tail, the value of the function applied to the tail of the recursion variable, and an expression built from these which defines the value of the function.

The intro rule introduces a witness for the existence, a function defined lambda-calculus style. The three subgoals it generates are:

1. Prove the introduced thing has the right type for whatever it's witnessing
2. Verify that the expression which is the scope of the ∃ is true for the introduced expression. I.e. the introduced expression genuinely demonstrates existence.
3. Check the well-formedness of the expression which is the scope of the ∃

You might like to try this as an exercise.

## 6 Extract Term

As you probably know, Oyster embodies a constructive logic. That is why it can be used for synthesising programs corresponding to proofs. For each rule of inference you can use, there is a constructor which links together the constructions corresponding to the arguments (subgoals) of the major connective the rule involves. A trivial example:

```
t : [1] complete autotactic(idtac)
>> 0=0 in int#1=1 in int
by intro

  [1] complete
  >> 0=0 in int

  [2] complete
  >> 1=1 in int


yes
| ?- extract.
axiom&axiom
```

Where the two axioms correspond to the subgoals `0=0 in int` and `1=1 in int` . An *elim* rule generating the step and base cases of an induction corresponds to a recursive loop and its termination.

Thinking about this recursively, it is top down functional programming. The leaves of the proof are well-formedness checks or things which are axiomatically true (0=0 in int).

When you use the *extract* command at a node, you will only see the extract term (as the construction is called) for the proof subtree below that node.

The *extract* command is useful in synthesis proofs, when the extract term includes the function you're synthesising. It can be run as a program. To do this, use *extract* with a variable argument, which will be instantiated to the extract term. You may have to decompose this a bit, to pull out the functional part and drop the verification part.

This is the extract term for the node of the append synthesis shown above:

```
(lambda(l1,lambda(l2,list_ind(l1,l2,[h,~,v,h::v])))
  & lambda(~,axiom&lambda(~,lambda(~,axiom))))
```

The bit before the & is the computationally interesting part. So if you were at the node shown in the synthesis section, and had completed the proof, you could use extract and eval like this:

```
| ?- extract(Fn&Rest),eval(Fn of (1::2::nil) of (0::2::nil),Ans).
extract(Fn&Rest),eval(Fn of (1::2::nil) of (0::2::nil),Ans).

Fn = lambda(l1,lambda(l2,list_ind(l1,l2,[h,~,v,h::v]))),
Rest = lambda(~,axiom&lambda(~,lambda(~,axiom))),
Ans = 1::2::0::2::nil
```

*eval* has two arguments, the term to be evaluated, and the result.

## 7 Tactics

You are really running a tactic every time you instruct Oyster to do something with the proof. But so far all your instructions have been simple, primitive ones. What you want to do is write macros of these instructions. That's what we more usually call a tactic.

Tactics are bits of Prolog code, containing Oyster commands and tacticals. The standard autotactic is an example of a tactic. It is just

```
repeat intro
```

`repeat` is a tactical. The tacticals are special words derived from the original ML version of Oyster for specifying how the combination of the instructions is to take place. This is on top of the usual Prolog flow of control. There are only a few of them:

- *repeat* T tries to apply T on the given problem and recursively repeats it on all the subproblems generated by T.
- R *or* S applies R, and if it fails tries S.
- R *then* S
- *complete* T succeeds only if T applies and generates no new subgoals.
- *try* U always succeeds. If U applies, it is performed, if it doesn't, the current goal is left unchanged.

Look at the section of the manual on The Inference Engine, particularly the subsection on The $\models$ Predicate for more details.

There are various Oyster commands (which are Prolog predicates) which fetch parts of the proof tree for you:

- *status(S)* instantiates $S$ to the status of the current goal, partial, complete or incomplete.

- *goal(G)* instantiates $G$ to the current goal, which is just a Prolog term, and so can be broken apart in the usual way.

- *hypothesis(H)* instantiates $H$ to a hypothesis. So, you can use it to generate hypotheses one by one, or to find one that matches some pattern if you partially instantiate $H$.

- *hyp_list(X)* instantiates $X$ to the list of hypotheses of the current node.

See the section on selector predicates in the manual for more details and a full list.

And of course there are all the rules and instructions you have already used, next, up, down, intro, etc.

As an exercise, you might like to try writing a tactic which would notice things like the contradiction between hypotheses 6 and 8 here, and deal with them for you:

```
n : [1,1,1,1,1,1,2,1] incomplete
1. a:u(1)
2. p:a->u(1)
3. v0:x:a#p of x->void
4. v1:x:a->p of x
5. v0~>v2:a
6. v0~>v3:p of v2->void
7. v0~>v0=v2&v3 in (x:a#p of x->void)
8. v1 of v2~>v4:p of v2
>> void
by _
```

You could try one which worked a bit harder and saw the contradiction when it was still inside the quantifiers and forced it out.

End of week 5.

Practical on the use of Logic to make Theorem Proving more Manageable.

This is a summary of some of the results on Herbrand's theorem, and some questions, looking like this about its implications. Also a programming task. The examples are mostly taken from Chang and Lee's book, look there if you want more information.

A Herbrand Universe of a formula is guaranteed to make a formula (un)satisfiable if and only if every other universe would too. Remember that to generate the Herbrand Universe of a formula, you create all the terms from the legitimate combinations of constants and function symbols in the formula. (If there is no constant symbol, you just have to use a token one).

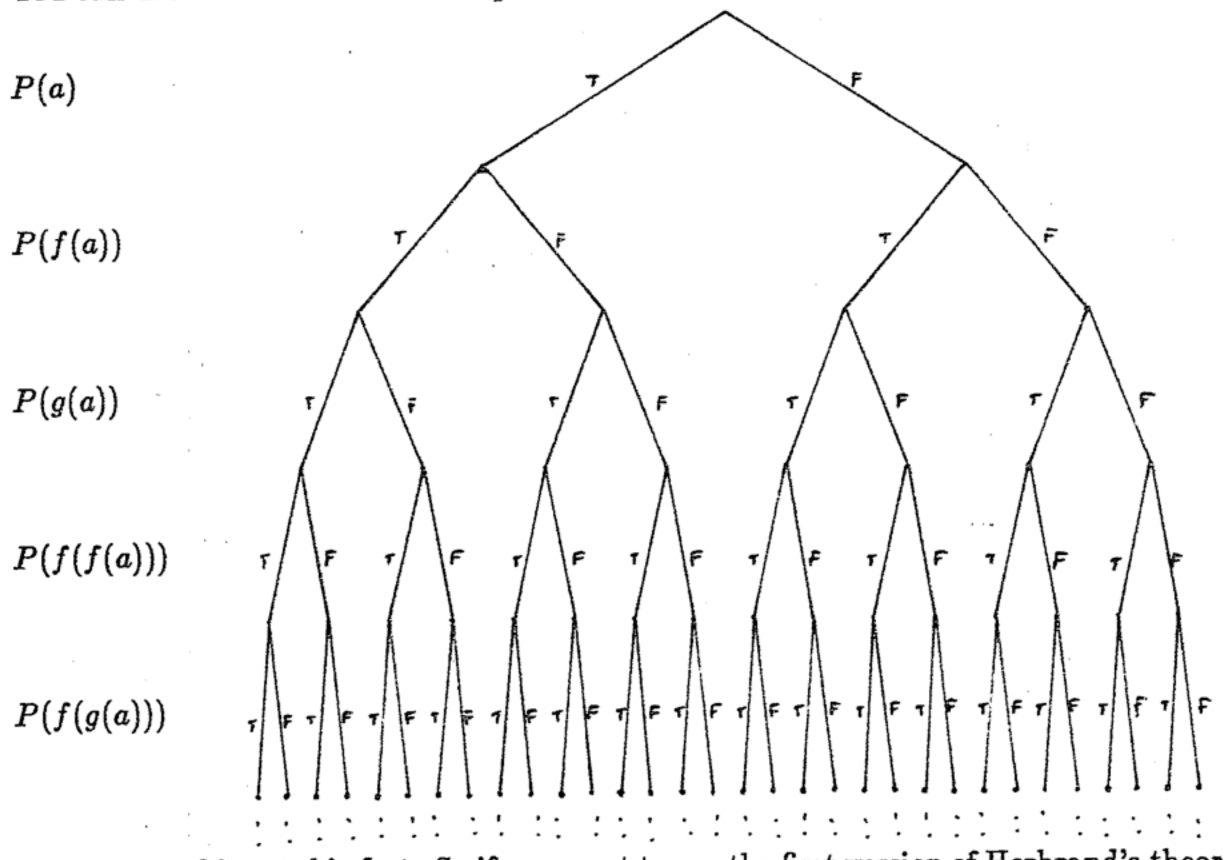
To do this systematically, you can think of the Universe growing by starting with constants, adding all possible applications of functions to them, adding all possible applications of functions to everything you've got now, and so on. Since this is a set, any duplications ~~vanish~~. Assuming everything is conveniently in clausal form, if you start with $P(x, f(y)) \vee \neg P(g(x), y)$ and use $a$ as your token constant, There are a sequence of subsets of the Herbrand Universe:

$P(a) \wedge (\neg P(x) \vee P(f(x)))$ $P(x) \wedge P(y)$.

| | |
|---|---|
| $H_0$ | $a$ |
| $H_1$ | $a, f(a), g(a)$ |
| $H_2$ | $a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a))$ |
| $H_3$ | $a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), f(f(f(a))), \ldots$ |

You can use these to define the expansion of the semantic tree:

Method (1)



You can see this gets big fast. So if you want to use the first version of Herbrand's theorem I gave:

*A conjunction of clauses is unsatisfiable if and only if corresponding to every complete semantic tree of S, there is a finite closed semantic tree.*

true in no interpretation
false in all interpretations $\Big]$ ⇒ ∃ prove false.

↳ no (∃x) quantification.
deliberately closed by using universal quantifiers

(Where closed means that every branch ends with a failure node, N. This means that the branch's (sub)interpretation makes a ground instance of the formula false, but none of the subinterpretations along the branch before N did.)

Since you have to allow for each instance being true *and* false, trying to check all branches can be cut at failure nodes would be hard work.

Here is a formula with 4 3-place functions and no constants.

**How big is the semantic tree up to $H_3$?** ~~86 nodes~~. $2^{70^3}$. forth.

$$\neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(x,v,w) \vee P(u,z,w)$$
$$\wedge \neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(u,z,w) \vee P(x,v,w)$$
$$\wedge P(g(x,y),x,y)$$
$$\wedge P(x,h(x,y),y)$$
$$\wedge P(x,y,f(x,y))$$
$$\wedge \neg P(k(x),x,k(x))$$

method 1

● method 2 ?

So how about the second version of Herbrand's theorem? With respect to Herbrand Universes:

*A conjunction of clauses is unsatisfiable if and only if there is a finite unsatisfiable conjunction of ground instances of its clauses.*

Well, this tells us that we can start building a maximal possible conjunction and see if we can detect that one that is unsatisfiable, regardless of the assignment of truth values. If we find satisfiable ones on the way, we continue, because, we might find a later, larger conjunction which is unsatisfiable.

This is the basis of Gilmore's procedure. For some formula $F$, with sets $H_0$,$H_1$, ... approaching the Herbrand Universe, let $F_i$ be the conjunction of all instances of $F$ with all elements of $H_i$ substituted for variables in all possible ways. So you check successively for the unsatisfiability of $F_0$, $F_1$, etc. At each stage, $F_i$ is a propositional formula.

**How much does this gain you in terms of the size of the object you have to search?**

● Take $P(a) \wedge (\neg P(x) \vee Q(f(x))) \wedge \neg Q(f(a))$

$H_0$ is $a$

$$F_0 \text{ is } P(a) \wedge \overline{(\neg P(a) \vee Q(f(a)))} \wedge \neg Q(f(a))$$
$$= \underline{(P(a) \wedge \neg P(a) \underset{F}{\wedge} \neg Q(f(a)))} \vee \underline{(P(a) \wedge Q(f(a)) \overset{F}{\wedge} \neg Q(f(a)))}$$
$$= \square \vee \square$$
$$= \square$$

looks for contradiction.

✓ ≈ Gilmore

method 4.

So having created a conjunction, Gilmore uses the distributivity rule to turn it into disjunctive normal form, and looks for a <u>contradiction</u> in each disjunct. This is not very efficient when things get bigger.

3°

method 2 - Davis and Putnam were working on this at the same time, and came up with something better. Here is (a version of) their procedure:

1. Again, create a conjunction of ground instances of the formula, $F$.

   (a) Delete all clauses that are tautologies, i.e. they contain $Q$ and $\neg Q$ for some $Q$.

(b) If any of the clauses in $F$ is a unit ground clause, $P$, i.e. $P$ is a ground literal[1] #
then:

i. If all of $F$'s clause contain $P$, $F$ is satisfiable, stop, try another conjunction of clauses. [illegible] $H_{i+1}$.

$P(a)$ - (claus) $\neg P(a)$ ii. If $\neg P$ is a unit clause, it is replaced by $\square$, so, $F$ is unsatisfiable, stop.

iii. Otherwise, delete all the clauses which contain $P$, and all occurrences of $\neg P$ from the rest of the clauses. ?

(c) There is a literal $L$ in $F$, and no occurrence of $\neg L$, delete all clauses containing $L$.

(d) If $F$ can be put into the form

$$(A_1 \vee L) \wedge ... \wedge (A_m \vee L) \wedge (B_1 \vee \neg L) \wedge ... \wedge (B_{\underline{n}} \vee \neg L) \wedge R$$

where $A_i$ and $B_i$ and $R$ are free of $L$ and $\neg L$, then for

$$F_1 = A_1 \wedge ... \wedge A_m \wedge R$$

and

$$F_2 = B_1 \wedge ... \wedge B_m \wedge R$$

$F$ is unsatisfiable if and only if both $F_1$ and $F_2$ are, i.e. $F_1 \vee F_2$ is.

(e) Repeat these procedures as long as possible.

2. If you haven't proved unsatisfiability, use the next $H_i$ to create another conjunction, and repeat.

Examples:

$(P \vee Q \vee \neg R) \wedge (P \vee \neg Q) \wedge \neg P \wedge R \wedge U$
1. $(Q \vee \neg R) \wedge (\neg Q) \wedge R \wedge U$ — rule (b) on $\neg P$
$\neg R \wedge R \wedge U$ — rule (b) on $\neg Q$
$\square \wedge U$ — rule (b) on $\neg Q$ $\neg R$ ?
which is unsatisfiable, since it contains the empty clause ( the OR of no literals).

$(P \vee Q) \wedge \neg Q \wedge (\neg P \vee Q \vee \neg R)$
$P \wedge (\neg P \vee \neg R)$ — rule (b) on $\neg Q$
$\neg R$ — rule (b) on $P$
which is satisfiable, since using rule (b) on $\neg R$ , we get the AND of no literals. b(i)

$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (Q \vee \neg R) \wedge (\neg Q \vee \neg R)$ explain? $\neg R \wedge \neg R$ by repeated (b)?
$(\neg Q \wedge (Q \vee \neg R) \wedge (\neg Q \vee \neg R))$
$\vee (Q \wedge (Q \vee \neg R) \wedge (\neg Q \vee \neg R))$ — rule (d) on $P$
$\neg R \vee \neg R$ — rule (b) on $Q$ and $\neg Q$
which is satisfiable by using rule (b) on $\neg R$ b(iii)

$(P \vee Q) \wedge (P \vee \neg Q) \wedge (R \vee Q) \wedge (R \vee \neg Q)$
$(R \vee Q) \wedge (R \vee \neg Q)$ — rule (c) on $P$
which is satisfiable using rule (c) on $R$

---

#1, [1] an atom or the negation of an atom

Write a Prolog program to do one of these two procedures (both if you're keen), and try it out. Here are some more examples to try it out on

1. $(\neg P \vee Q) \wedge \neg Q \wedge P$ ✓
2. $(P \vee Q) \wedge (R \vee Q) \wedge \neg R \wedge \neg Q$ ✓
3. $(P \vee Q) \wedge (\neg P \vee Q) \wedge (\neg R \vee \neg Q) \wedge (R \vee \neg Q)$ ⊗ ↑ Method 2.

✓ 4. $P(x) \wedge (\neg P(x) \vee Q(x,a)) \wedge \neg Q(y,a)$
✓ 5. $P(x,a,g(x,b)) \wedge \neg P(f(y),z,\underline{g(f(a),b)})$ fixed? Method 1.
✓ 6. $P(x) \wedge ((\underline{Q(x,f(x))} \vee \neg P(x)) \wedge \neg Q(g(y),z)$

They are all unsatisfiable, if I didn't make any mistakes.

**Look at some of these problems from the point of view of resolution, and see how big you think the search space would be**

Do not spend an unreasonable amount of time on all this. It's only meant to be two tutorials worth. Just see how much you can do.

assume in standard form & skolemised.

```
From jane@ed.aiva.etive Fri Dec  2 20:07:40 1988
Received: from etive.aiva.ed.ac.uk (rannoch) by et.aiva.ed.ac.uk; Fri, 2 Dec 88 20:07:24
Date: Fri, 2 Dec 88 20:09:34 GMT
Message-Id: <23717.8812022009@etive.aiva.ed.ac.uk>
To: aicam@ed.aiva.et, aigl@ed.aiva.et, aijpd@ed.aiva.et, ailed@ed.aiva.et,
        aimg@ed.aiva.et, ainjs@ed.aiva.et, airah@ed.aiva.et, cam@ed.ecsvax,
        gl@ed.ecsvax, jpd@ed.ecsvax, led@ed.ecsvax, mg@ed.ecsvax,
        njs@ed.ecsvax, rah@ed.ecsvax
Subject: oysterrules
Status: R
```

Here are most of the rules you need to start with. There are more, to do with sequencing in new formulae, but you shouldn't need any of that for the early part of the practical if at all. I'll be back in tomorrow to add some more to this, but just now, I'm tired and I'm going home, but I thought I'd get some of it to you.

Jane

The commands are often short for something more explicit which Oyster works out from the context, but sometimes you have to specify. Usually what is inferred is the universe level or the names of new objects. Occasionally it is the part of the term it is to work on.

## 1. Operating on the conclusion - intro rules:

Intro rules apply to the conclusion's dominating connective. This may be one of:

a. A#B (A and B)

You will get two subgoals, one is A the other is B.

```
d : [1] partial autotactic(idtac)
>> 0=0 in pnat#s(0)=s(0) in pnat
by intro

  [1] incomplete
  >> 0=0 in pnat

  [2] incomplete
  >> s(0)=s(0) in pnat
```

b. A\B (A or B)

There is a choice here, you must specify whether you wish Oyster to let you prove A or B. You say intro(left) for A or intro(right) for B. Either way, you will get the extra subgoal of proving that the other one is well-formed:

```
b : [1,1,1,1] partial autotactic(idtac)
1. a:u(1)
2. b:u(1)
3. v0:a#b
>> a\b
by intro(left)

  [1] incomplete
  >> a

  [2] incomplete
  >> b in u(1)
```

The full form of the rule lets you specify the universe level for the well-formedness goal: intro(at(1),left).

c. A->B (A implies B)

```
b : [1,1,1] partial autotactic(idtac)
1. a:u(1)
2. b:u(1)
>> (a#b)->a\b
by intro

  [1] incomplete
  3. v0:a#b
  >> a\b

  [2] incomplete
  >> (a#b) in u(1)
```

This has creates two subgoals, one to prove a\b assuming a#b, the other to check that a#b is well-formed.

d. A:B->C (for all A of type B, C is true)

```
b : [1] partial autotactic(idtac)
>> a:u(1)->b:u(1)->(a#b)->a\b
by intro

  [1] incomplete
  1. a:u(1)
  >> b:u(1)->(a#b)->a\b

  [2] incomplete
  >> u(1) in u(2)
```

The first subgoal is the quantified term with the quantification removed to the hypothesis list. The second is a check on the well-formedness of the type you are asserting the quantification over. I.e. that this u(1) that a is a member of is a valid type, it inhabits a type universe. You may specify the universe level yourself by saying intro(at(i)), where i is the universe level you want.

e. A:B#C (there exists A of type B such that C)

You can't just say intro here. You have to tell it what the A is which exists such that C is true.

```
e : [1] partial autotactic(idtac)
>> x:pnat#x=0 in pnat
by intro(0)

  [1] incomplete
  >> 0 in pnat

  [2] incomplete
  >> 0=0 in pnat

  [3] incomplete
  1. x:pnat
  >> x=0 in pnat in u(1)
```

You have three subgoals here: one that the thing you have introduced is of the correct type, one that the original goal is true with this object you've introduced, and the third that the quantified term is well-formed. You can adjust the universe level here, if you want by using intro(0,at(i)) where i is the level you want. In this case 1 is fine.

f. There is a hypothesis exactly matching the conclusion:

```
    b : [1,1,1,1,1,1] complete autotactic(idtac)
    1. a:u(1)
    2. b:u(1)
    3. v0:a#b
    4. v0~>v1:a
    5. v0~>v2:b
    6. v0~>v0=v1&v2 in (a#b)
    >> a
    by intro
```

2. Operating on the hypotheses - elim rules:
-------------------------------------------

The general form of these is elim(X) where X is the thing you want the elim performed on.

a. v0:A#B (A and B)

```
    b : [1,1,1,1,1] partial autotactic(idtac)
    1. a:u(1)
    2. b:u(1)
    3. v0:a#b
    >> a
    by elim(v0)

      [1] incomplete
      4. v0~>v1:a
      5. v0~>v2:b
      6. v0~>v0=v1&v2 in (a#b)
      >> a
```

The subgoal is to prove the same thing as before with some extra hypotheses: a, b and a link of them together. All these statements of the form A:B can be read equivalently as

```
A is a member of B
A is a witness for B
A is a proof of B
A is a program for B
```

So this is literally that v1 is a proof of a, v2 is a proof of b and one which links them together, saying that v0, the proof of a#b is v1&v2.

b. A\B (A or B)

You get two subgoals, each of proving the original goal assuming one of the two things in the disjunct:

```
    a : [1,1,1,1] partial autotactic(idtac)
    1. a:u(1)
    2. v0:a\a->void
    3. v1:(a->void)->void
    >> a
    by elim(v0)

      [1] incomplete
      4. v0~>v2:a
      5. v0~>v0=inl(v2) in (a\a->void)
      >> a

      [2] incomplete
      4. v0~>v3:a->void
      5. v0~>v0=inr(v3) in (a\a->void)
      >> a
```

Again, the extra hypothesis is the information to let it reconstruct the v0 witness/proof from v2 and v3.

c. A->B (A implies B)

An elim on a hypothesis of this form will give you two subgoals, one to prove A, and one to prove the original goal assuming B.

```
    a : [1,1,1,1,2] partial autotactic(idtac)
    1. a:u(1)
    2. v0:a\a->void
    3. v1:(a->void)->void
    4. v0~>v3:a->void
    5. v0~>v0=inr(v3) in (a\a->void)
    >> a
    by elim(v1)

      [1] incomplete
      >> a->void

      [2] incomplete
      6. v1~>v2:void
      >> a
```

If you already have A as a hypothesis, you can specify this:

```
    a : [1,1,1,1,2] partial autotactic(idtac)
    1. a:u(1)
    2. v0:a\a->void
    3. v1:(a->void)->void
    4. v0~>v3:a->void
    5. v0~>v0=inr(v3) in (a\a->void)
    >> a
    by elim(v1,on(v3))

      [1] incomplete
      >> v3 in (a->void)

      [2] incomplete
      6. v1 of v3~>v2:void
      7. v1 of v3~>v4:v2=v1 of v3 in void
      >> a
```

Now you still have to prove the original goal using the succedent of the hypothesis, but instead of proving the antecedent, you only have to show its witness is well-formed.

d. A:B->C (for all A of type B, C is true)

Eliminating a 'for all" is effectively saying which of the set it is quantified over you to use it for, and you must choose one:

```
    f : [1,1] partial autotactic(idtac)
    1. v0:x:pnat->y:pnat#y=s(x) in pnat
    >> z:pnat#z=s(s(0)) in pnat
    by elim(v0,on(s(0)))
    [1] incomplete
    >> s(0) in pnat

    [2] incomplete
    2. v0 of s(0)~>v1:y:pnat#y=s(s(0)) in pnat
    >> z:pnat#z=s(s(0)) in pnat
```

So here, the elimination lets you take the general statement that every number has a successor, generate the particular case that s(0) has a successor as an extra hypothesis, and proceed. There is a side goal that this thing you introduced is of the correct type.

e. A:B#C (there exists A of type B such that C)

Eliminating a "there exists" lets it generate an object corresponding to this existence:

```
   f : [1,1,2] partial autotactic(idtac)
   1. v0:x:pnat->y:pnat#y=s(x) in pnat
   2. v0 of s(0)~>v1:y:pnat#y=s(s(0)) in pnat
   >> z:pnat#z=s(s(0)) in pnat
   by elim(v1)

     [1] incomplete
     3. v1~>v2:pnat
     4. v1~>v3:v2=s(s(0)) in pnat
     5. v1~>v1=v2&v3 in (y:pnat#y=s(s(0)) in pnat)
     >> z:pnat#z=s(s(0)) in pnat
```

This is a more artificial example than most, in that an intro at this point would have matched up hypothesis 2 and the conclusion anyway.

f. There is a hypothesis "void". You can prove anything from this:

```
   c : [1,1] complete autotactic(idtac)
   1. v0:void
   >> a:u(1)->a:u(1)->void
   by elim(v0)
```

g. A:B

If B is a type for which there is a recognised induction, elim(A) generates an induction for you. The subgoals are the base and step cases.

## 3. Operating on the conclusion - well-formedness goals:

These are of the form:

```
   X in Y
```

where Y is a type. E.g.

```
   u(1) in u(2)
```

these can almost invariably be solved with a "repeat intro". Most types you will come across are in u(1) - pnat, int, pnat list, pnat->pnat, void, a#b where both a and b are in u(1) etc.

If you find yourself trying to prove

```
   u(1) in u(1)
```

you have a problem. This is not true, u(1) is in u(2). In general u(i) is in u(i+j) for all j > 0. Sometimes Oyster will have guessed the universe level correctly for you. If not, you have to back up to the goal where you generated this and use universe level 2, like this: