λ*Clam* v.4.0.1: User / Developer's Manual

James Brotherston & Louise A. Dennis

October 11, 2002

Contents

Chapter 1

Introduction

This manual describes the λ*Clam* proof planner.

Proof planning, first conceived by Bundy [4] attempts to capture common structure in proofs through the development of *proof plans*, which are proofs represented at some level of abstraction — usually above the level of inference rules, but not necessarily so[1]. The proof plan is typically generated using AI-style planning techniques and represented as a tree. The planning operators used by the proof planner are known as *methods*; these are somewhat akin to tactics in a traditional theorem prover, except that they are generally intended to encapsulate chunks of 'human' mathematical reasoning rather than individual inference rules. For these reasons, methods are often referred to as "partial tactic specifications". Methods can be either atomic or compound. An atomic method is defined by its pre- and postconditions which partially describe (in some meta-language) the proof state before and after the application of the method; a method will only be applied if its preconditions are satisfied, and will enforce its postconditions on successful application. A compound method is built up from atomic methods using combinators called *methodicals* (which are analogous to LCF-style tacticals), allowing the user to build up more complex proof strategies and heuristics.

In the original proof planning paradigm, each method has a tactic attached to it which when executed produces the corresponding part of the object level proof. The use of pre- and postconditions allows methods to be (potentially) more expressive than tactics alone, so that a method can sensibly be thought of as "guarantee + explanation", with the tactic component providing the guarantee component and an explanation provided by the high-level pre- and postconditions. However, λ*Clam* currently has no theorem prover attached to it, and hence no tactics attached to its methods, leaving considerations of soundness squarely in the hands of the method writer[2]. Another important feature of proof planning is the use of *critics* — planning operators which respond to various types of planning failure by attempting to 'patch' a proof attempt, for example by generalising the conjecture to be proved. All of these ideas will be explored in further detail later in this manual.

λ*Clam* is the latest in a series of proof planners developed by the Mathematical Reasoning Group at the University of Edinburgh. Unlike its predecessors, the CLAM series, it supports proof planning over higher-order domains, since the underlying implementation language Teyjus (an implementation of λProlog[3]) incorporates

[1]If not, then the proof plan is in fact a *bona fide* proof.

[2]The OMEGA proof planner [3] allows the tactics justifying nodes to be further proof plans hence the process of executing the plan can be performed iteratively giving proofs at lower and lower levels of abstraction until it grounds out in the object-logic (a natural deduction style sequent calculus).

[3]http://www.cse.psu.edu/~dale/lProlog/

many useful higher-order features such as HO unification, automatic β-reduction and explicit language constructs for existential and universal quantification over terms. The built-in methods of λ*Clam* are specifically tailored to induction proofs, and the system has successfully been used to plan proofs in, amongst other domains, ordinal arithmetic [9], nonstandard analysis [16], and synthesis of logic programs [15].

Further technical details can be found in the Implementation Taskforce note series. Background motivation can be found in [18]. Some of this information, and other resources, can be found on the λ*Clam* web page at:

`http://dream.dai.ed.ac.uk/systems/lambda-clam/`.

Both this document and the λ*Clam* system are under development. Throughout the text you will find comments and suggestions for improvements and additions indicated by *italics*.

Chapter 2

Installing and Running λ*Clam*

2.0.1 Installing λ*Clam*

Chapter 3

λ*Clam* Tutorial

3.1 Introduction

This document is intended to act as a tutorial to get you started in the use of λ*Clam*. This tutorial presupposes some prior knowledge of proof planning, the rippling proof method, the λ-calculus and Prolog (though not λProlog in which λ*Clam* is implemented).

A maths tutor was once heard to say that his task was leading the students through a marsh along the safe path, shoving them into every sticky pool as they went past then hauling them out again. This tutorial has been written with this philosophy somewhat in mind. As a result it frequently instructs you to do things that don't work — the idea being to accustom you to the sort of errors and problems you may expect to encounter from time to time.

3.2 Getting Started

Check that the environment variables `LCLAM_HOME` and `TJPATH` have been set properly. `LCLAM_HOME` should point to the top level directory of your λ*Clam* distribution. `TJPATH` should point to each of the subdirectories of `$LCLAM_HOME/src`. You should also check that the location of the `tjsim` executable is on your `$PATH` and on `$TEYJUS_HOME`. On the whole we advise that you add this to your `.bashrc` file to save you setting them every time you open a new window. If you haven't set environment variables before you may wish to get some help with this.

We also suggest that you interact with the build procedures in an emacs window; for interacting with the program itself, we recommend the use of the supplied ProofGeneral interface which runs within XEmacs. Both the Teyjus compiler and λ*Clam* itself generate a lot of output which emacs will allow you to scroll through. To get a normal command line prompt from within emacs type M-x shell. To start the program you type:

```
$LCLAM_HOME/bin/lclam
```

or alternatively open a new .lcm file in XEmacs to start ProofGeneral and λ*Clam* automatically.

3.3 A First Proof

λ*Clam* comes with a number of built-in conjectures you can try to prove. We will now look at the proof of one of these.

Type: `query_top_goal X assp.`

`assp` stands for the associativity of plus, i.e. the theorem

$$\forall xyz.((x+y)+z) = (x+(y+z))$$

Something like the following should now appear on the screen.

```
[prompt] lclam
lclam:
query_top_goal X assp.
arithmetic
forall <lc-0-2>:nat
 forall <lc-0-3>:nat
  forall <lc-0-4>:nat
   (plus (plus (<lc-0-4>, <lc-0-3>), <lc-0-2>) =
    plus (<lc-0-4>, plus (<lc-0-3>, <lc-0-2>)))
lclam:
```

This gives you the name of the theory in which the goal is found (i.e. `arithmetic`) and the goal itself is pretty printed for you. Note that object-level variables in Teyjus are currently pretty-printed in the form ¡lc-x-y¿.

Type: `pds_plan (induction_top normal_ind) assp.`

This calls the standard PDS planner with the normal induction strategy on the `assp` theorem. A lot of output will be generated by the planner, ending with

```
backtracking over
ind_strat
ind_strat
failed
backtracking over
induction_top normal_ind
induction_top normal_ind
failed
  Failed.

lclam:
```

This means *λClam* has failed to find a plan. Don't worry about this at the moment but scroll back through the attempt to get an idea of what is happening. You will notice that it records all attempts at applying methods:

```
Attempting...
Method application:  induction_top normal_ind
succeeded
```

and

```
Attempting...
allFi
failed
```

After each successful method attempt *λClam* prints out the new goal that it is working on. It also prints out the address in the final proof plan tree that the goal appears at. The first of these is:

```
Method application:  induction_top normal_ind
succeeded

GOAL:
>>> forall <lc-0-2>:nat
     forall <lc-0-3>:nat
      forall <lc-0-4>:nat
       (plus (plus (<lc-0-4>, <lc-0-3>), <lc-0-2>) =
        plus (<lc-0-4>, plus (<lc-0-3>, <lc-0-2>)))
ADDRESS: nil
```

This is the first successful method application. The `nil` means it appears at the top of the proof tree. In fact this goal is identical to the initial goal. Why is this?

λClam maintains a distinction between *atomic* and *compound* methods. A compound method, such as `(induction_top normal_ind)`, does not actually alter the goal at all. It expands into a set of submethods that are to be applied in a particular order (more on this later). So in this case the proof tree has not been altered at all. All that has happened is that a compound method has been expanded.

Scroll down a bit and you will find

```
Attempting...
Method application:  all_i
succeeded

GOAL:
Don't know how to display goal:
allGoal nat (Y\ seqGoal (otype_of Y nat :: nil >>> app forall (tuple (nat :: abs (y\ app forall
ADDRESS: 1 :: nil

Attempting...
Moving allGoal to meta-level
succeeded

GOAL:
<lc-0-2>:nat
>>> forall <lc-0-3>:nat
     forall <lc-0-4>:nat
      (plus (plus (<lc-0-4>, <lc-0-3>), <lc-0-2>) =
       plus (<lc-0-4>, plus (<lc-0-3>, <lc-0-2>)))
ADDRESS: 1 :: 1 :: nil
```

`all_i` is an atomic method. For the logically minded this method corresponds to the All Introduction or generalisation inference rule i.e. from $\mathcal{A}$ deduce $\forall x_i.\mathcal{A}$, where A is any well-formed formula and x_i is any variable [12]. Since proof planning works by backwards proof (i.e. starting with the theorem and moving back towards the axioms by applying methods) the effect is that the universal quantifier appears to disappear. The formal proof represented by executing the proof plan will move forwards from the axioms to the theorem applying inference rules. This is an example where the link between the method and the inference rule is particularly clear. In many cases a method will represent a whole sequence of inference rules (in this case the method is supposed to refer to an LCF-style *tactic* which is a program for constructing sequences of inference rules). At this point the goal is quantified (the `allGoal` bit with which the pretty print can't cope – hence the "Don't know how to display goal" message) this quantification is then moved to the plan level (called meta-level above) where it is hidden from the user. This is a technical detail which forall x, y, z. ((x + y) + z) = (x + (y + z))

forall c

forall y, z. ((c + y) + z) = (c + (y + z))

Figure 3.1: A Proof Tree

it is usually safe to ignore. Once the quantification is moved the planner displays the new goal.

So the new goal is $c : nat \vdash \forall y, z.((c + y) + z) = c + (y + z)$ where c is an arbitrary constant. λ*Clam* universally quantifies c but does so at the meta-level (i.e. outside the logic). The proof tree at this point looks something like figure 3.1. So the address of the new node in the tree is $[1, 1]$. NB. The address of a proof node is read from right to left, (i.e. $[3, 2]$ is the 3rd child of the 2nd child of the root node).

So what went wrong with the proof? Well we would expect the proof to be an inductive one. The atomic method that applies induction is `induction_meth`. This was the last atomic method attempted and it failed. λ*Clam*'s source code contains a number of rewrite rules and induction schemes only some of which are ever wanted at a particular time. In fact, potentially the collection could become inconsistent if people are experimenting with different logics etc. Therefore you always have to tell λ*Clam* which induction scheme, rewrite rules and wave rules you want to use at any one time. When developing heuristics it can often be very helpful to restrict yourself to a very small number of such rules and schemes to avoid confusion before widening your view to a full theory. This is not the case here. We can add all the induction schemes etc. in the arithmetic theory and use them to plan `assp`. To do this type:

```
lclam:
add_theory_to_induction_scheme_list arithmetic.
Done
lclam:
add_theory_to_sym_eval_list arithmetic.
Done
lclam:
set_wave_rule_to_sym_eval.
Done
lclam:
```

(`set_wave_rule_to_sym_eval` sets the wave rules used by the system to be the same as those rewrite rules used for symbolic evaluation). We also will want one other rewrite rule (namely $X = X \leftarrow:\rightarrow true$) this is called `idty` and is generally useful in nearly all proofs. So we will add this to the symbolic evaluation (i.e. general rewriting) list and to the wave rule list.

```
lclam:
add_to_sym_eval_list idty.
Done
lclam:
set_wave_rule_to_sym_eval.
```

```
Done
lclam:
```

Now the command `pds_plan (induction_top normal_ind) assp` should end with

```
GOAL:
>>> trueP
ADDRESS: 1 :: 1 :: 1 :: 1 :: 1 :: 2 :: 1 :: nil

Attempting...
Method application:  trivial
succeeded

GOAL:
trueGoal!
ADDRESS: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 2 :: 1 :: nil

Attempting...
branch closed!

reached empty agenda

Plan found (not displayed) ...
Plan Succeeded
lclam:
```

Look back through the proof and try to see how it proceeded. Paying particular attention to the method names that it attempts, whether they succeed or fail and the goals they produce if they succeed. Notice also that λ*Clam* sometimes "backtracks" over methods. This is prolog-style backtracking and means λ*Clam* is attempting to apply a different method at that point. Do not worry too much about exactly what methods do, but try to get an idea of the sequence of goals that were proved.

3.3.1 Stepping through the Proof

As well as playing the entire proof through automatically you can step through the proof one method at a time.

Type `step_by_step on`

Restart the proof with `pds_plan (induction_top normal_ind) assp`. This will restart the proof. Almost immediately this will stop after the application of `induction_top normal_ind` with the prompt `Continue Planning?`. Notice you also get some extra output:

```
c_and_node (seqGoal (nil >>> app forall (tuple (nat :: abs (z\ app
forall (tuple (nat :: abs (y\ app forall (tuple (nat :: abs (x\ app eq
(tuple (app plus (tuple (app plus (tuple (x :: y :: nil)) :: z ::
nil)) :: app plus (tuple (x :: app plus (tuple (y :: z :: nil)) ::
nil)) :: nil))) :: nil))) :: nil))) :: nil)))) nil nomethodyet nil
```

This is the current state of the plan expressed in an abbreviated form of λ*Clam*'s internal format. `c_and_node` is a proof plan node and you see the goal at that node `seqGoal ...`, the address of the node (`nil`), the method (`nomethodyet`) and the children of the node `nil`.

Type `continue.`

Now you get to the point where the `taut` compound method was applied. This is the tautology checking method that calls `all_i` above. If you look at the trace from your last attempt you will see that ultimately λ*Clam* backtracks out of `taut`.

Type `backtrack.`

This forces the planner to try a backtrack now rather than attempt all the submethods of tautology.

Now the planner will try the `sym_eval` compound method. Again the previous plan tells you that it will ultimately attempt `ind_strat normal_ind` as the first successful method. Instead of backtracking until this is attempted try:

Type `try (ind_strat normal_ind)`

This immediately attempts the `ind_strat normal_ind` method. Continue and the `induction_meth` method will be applied. This will produce a tree containing a child node of the root node (you can also see the root node at the top – notice that the method slot has now been instantiated). If you look carefully at the child node you will see it contains the `allGoal` constructor for a quantified goal and the symbol `**` which means there are two subgoals – one for the step case and one for the base case. As its next step the planner will split these this conjoint subgoal into two separate subgoals.

Type `continue.` **again**

This time the split child nodes are displayed.

Normally the plan would now proceed with planning the step case. But you can switch to the base case if that is what you are more interested in. You can (just about!) determine the number of the node you wish to switch to – it appears just before the `nomethodyet` place holder. So the base case is node `(2::1::nil)`

Type `plan_node (2::1::nil).`

You can carry on experimenting with backtracking, attempting specific methods and directing which goal to concentrate upon. When you are fed up you can type `abandon.` to give up the attempt. In an idiosyncratic fashion λ*Clam* will report success at this point *something should really be done about this!!*.

3.4 Entering and Planning Your Own Theorem

It is likely that once you are using λ*Clam* that you will want to enter your own conjectures for proof. You can do this by entering new information (typed in the internal syntax) at the command line and the commands for this are listed in chapter 4.

However in general you should create a personal theory file for your work. Exit λ*Clam* by typing `halt.` and set up a directory for your own λ*Clam* work. Add this directory to your `TJPATH`.

3.4.1 A Theorem: RevQrev

The object of this section is to step you through the planning of the "RevQrev" theorem. RevQrev compares two different functions for reversing a list and shows that they do the same thing.

rev works according to the definition:

$$
\begin{aligned}
rev(nil) &= nil && (3.1)\\
rev(X :: Y) &= rev(Y) <> X :: nil && (3.2)
\end{aligned}
$$

where $<>$ appends two lists together and $::$ conses an element on to a list (i.e. places the element first in a new list followed by the elements of the old list in order). Intuitively *rev*'s recursive step strips the first element off the top of the list and then glues it onto the end of the reversed list using append.

qrev works according to the definition:

$$
\begin{aligned}
qrev(nil, L) &= L && (3.3)\\
qrev(H :: T, L) &= qrev(T, H :: L) && (3.4)
\end{aligned}
$$

qrev maintains two lists and gradually strips the first element off one and conses it to the other.

The object here is to show that

$$\forall l_1, l_2 . rev(l_1) <> l_2 = qrev(l_1, l_2)$$

To prove this goal *λClam* has to know the equations for *rev* and *qrev* and have the goal in its list of queries. To do this you will need to write a theory file for *λClam*.

3.4.2 The Anatomy of a Theory File

To plan a conjecture in *λClam* you need to create a theory file. There are many elements that can make up a theory file including arbitrary λProlog predicates. Examples of theory files can be found in the `theories/` subdirectory of the *λClam* distribution. In particular this section uses the arithmetic theory as an example which can be found in the files `arithmetic.mod` and `arithmetic.sig`. There is more detail on creating theories in chapter 6.

λClam Syntax

In writing a theory file you are going to need to use *λClam*'s internal syntax. This is based on Higher Order Abstract Syntax or λ-tree syntax. A term can be a constant, a variable, an function application, a λ-abstraction or a tuple of terms.

$$t ::= \mathtt{A} \mid \mathtt{a} \mid (\mathtt{app}\ t\ t) \mid (\mathtt{abs}\ \mathtt{x}\backslash\ t) \mid \mathtt{tuple}[t, \ldots, t] \qquad (3.5)$$

where `A` is a λProlog variable and `a` is a λProlog constant. In this syntax $s(X)$ for some variable X becomes `(app s X)`. $X + Y$ becomes `(app plus (tuple [X, Y]))`. $\lambda x.s(x)$ becomes `(abs x\ (app s x))`.

The λProlog types for the contructors are:

```
type abs        (osyn -> osyn) -> osyn.
type app        osyn -> osyn -> osyn.
type tuple      (list osyn) -> osyn.

type arrow      osyn -> osyn -> osyn.
type tuple_type (list osyn) -> osyn.

type otype_of   osyn -> osyn -> osyn.
```

Basic terms (objects of type `osyn`) can be Prolog variables or constants (e.g. `P` or `plus`). The constructors `abs`, `app` and `tuple` are then used to build up more complex terms. `app F X` applies the function `F` to the argument X. If there is more than one argument then `tuple` is used, so, for instance `app plus (tuple [1, 2])` represent the term, $1+2$. `abs` is used to represent λ-abstraction. This uses λProlog's internal λ-abstraction mechanism. So the function $\lambda x.x$ would be `abs X\ X` in *λClam*'s internal syntax.

`otype_of` is used to assign types to terms and `arrow` and `tuple_type` are used to construct function types.

Mostly the format for terms should be obvious. By convention quantifiers take a tuple of arguments the first being the type of the quantified variable and the second being a λ-abstraction. So the formula $\forall x : nat.x = x$ becomes `(app forall (tuple [nat, (abs x\ (app eq (tuple [x, x]))))]))`

Goals

A very simple theory file might contain only a top goal. This starts with the keyword `top_goal` followed by the name of the theory, a list of hypotheses and then the statement of the theorem. So a file conjecturing only the associativity of plus, `assp`, would contain

```
top_goal mytheory assp []
    (app forall (tuple [nat,
     (abs z\ (app forall (tuple [nat,
      (abs y\ (app forall (tuple [nat,
       (abs x\ (app eq (tuple [
        (app plus (tuple [(app plus (tuple [x, y])),  z])),
        (app plus (tuple [x, (app plus (tuple [y, z]))]))])))]))))]))))])).
```

Take a good look at the syntax for the goal and make sure you understand how it relates to the theorem.

Definitions and Lemmas

Another major component of a theory file is a list of definitions and lemmas which you think will be needed by the plan. Mostly these are used to "rewrite" subexpressions of some goal and are often referred to as rewrite rules as a result. You should give the type of any new function that you introduce as well as the equations that govern its behaviour. The type is signalled by the `has_otype` keyword (*at the moment type information is only used by the generalisation method – λClam does not enforce type correctness in the syntax*). `has_otype` takes three arguments: the name of the theory, the name of the new constant and the constant's type.

So for instance, the arithmetic theory file contains the following type declarations:

```
has_otype arithmetic zero nat.
has_otype arithmetic s (nat arrow nat).
has_otype arithmetic plus ((tuple_type [nat, nat]) arrow nat).
has_otype arithmetic otimes ((tuple_type [nat, nat]) arrow nat).
```

These can be viewed as the assertions that $0 : nat$ (0 is of type natural number), $s : nat \to nat$ (s is a function from natural numbers to natural numbers), $+ : (nat, nat) \to nat$ (plus is a function from a pair of natural numbers to natural numbers) etc. Notice that `arrow` is used for function types (i.e. applications and λ-abstractions) and there is a `tuple_type` for tuples.

Definitions are started by the `definition` keyword. Definitions are equations that define how some function or constant work. So the definition of $+$ in this context is:

$$0 + Y = Y \tag{3.6}$$
$$s(X) + Y = s(X + Y) \tag{3.7}$$

A definition takes five arguments: the name of the theory, a name of the definition equation, any conditions on the rule, the left and right hand sides of the equality. e.g. the definitional rule, (3.6 above): $0+Y = Y$ in the arithmetic theory is written as

```
definition arithmetic plus1 trueP (app plus (tuple [zero, Y])) Y.
```

The theory is called `arithmetic`, the rule is called `plus1`, there are no conditions so this is simply true (`trueP`) the LHS is $Y + 0$ and the RHS is Y. You can see this definition by looking in `arithmetic.mod` in the theories directory of the λ*Clam* distribution.

Lemmas can be started by the keywords `lemma` or `axiom`. These take an additional argument to `definition` which indicates the direction in which they may be used. If the lemma is $X = Y$ then the keyword is `equiv` indicating that it can be used to rewrite rule in either direction.

Signature files and the pre and post-amble of your theory file

Theory files should be called `mytheory.mod` (where mytheory is the name of your theory). They should begin with the declaration

```
module mytheory.
```

They should then state the name of a file they wish to accumulate. `objlists` will give you induction over natural numbers and lists and so is a good general file to accumulate. You will also need to accumulate the file specifying the rule of the logic you intend to use, in this case this should be `constructive_logic`. It is unwise to accumulate more than one file plus the logic file (for technical λProlog reasons see chapter 10) so it is a good idea to impose a linear structure. If you have more than one theory file you wish to use then the first should accumulate objlists and the second should accumulate the first. Accumulation commands should immediately follow the module command. NB. Standard Prolog conventions still hold, i.e. names beginning with capitals are treated as variables.

```
accumulate objlists.
accumulate constructive_logic.
```

You will also need a signature file `mytheory.sig`. This should start

```
sig mytheory.

accum_sig objlists.
accum_sig constructive_logic.

type mytheory theory.
```

The signature file declares the types of any constants you've introduced. So the signature file for arithmetic contains

```
type    nat     osyn.

type    zero    osyn.
type    s       osyn.
type    plus    osyn.
type    otimes  osyn.

type    plus1   rewrite_rule.
type    plus2   rewrite_rule.

type assp       query.
```

As you can see the type of all the syntax constants you've introduced is `osyn`. The type of definitions, lemmas and axioms is `rewrite_rule` and the type for goals is `query`.

Both module and signature files should end with the keyword `end` (this is the only clause/statement that should *not* conclude with a full stop).

Create a theory file for the RevQrev conjecture

To prove the conjecture λ*Clam* will need the definitions of *rev* and *qrev* shown in equations (3.1), (3.2), (3.3) and (3.4). As it happens *rev* is already present in λ*Clam* (called `orev` in the objlists theory) as is append (`oapp`) so only *qrev* needs to be newly defined in a separated file (see figure 3.2 for some of the syntax you will need for built-in functions and constructors). The list type is written `(olist X)`.

Mathematical Notation	λ*Clam* Syntax
nil	`onil`
$H :: T$	`(app ocons (tuple [H, T]))`
$A <> B$	`(app oapp (tuple [A, B]))`
$rev(A)$	`(app orev A)`

Figure 3.2: λ*Clam* Syntax needed for RevQrev: Quick Crib

When you have finished creating the file you will need to compile it. Check that `TJPATH` points to all the src subdirectories in the λ*Clam* distribution and to the directory in which you are working. Type

```
tjcc mytheory.lp
```

We suggest you perform compilation within emacs since the Teyjus compiler produces a lot of output you may wish to scroll through. All compiler errors are flagged with one of the keywords `Error` or `Warning` so it is probably easiest just to search the buffer for these (type `C-r` in XEmacs to search backwards). Watch out for syntax errors (called parse errors) caused by missing or extra brackets and commas in the tuple lists.

Type

```
[prompt] $LCLAM_HOME/bin/lclam mytheory
```

This should start up λ*Clam* using your theory.

You can find example files for RevQrev in §3.10.

Plan your conjecture.

You do this just as you planned `assp.` in the first part of the tutorial (except with your conjecture name replacing `assp`). Make sure you have added all your rewrite rules and the definitions to the wave rule and sym_eval lists. Also add the definitions for `oapp` and `orev`, `idty` and cancellation of cons cells, `cons_functional`. You may find the commands `set_sym_eval_list` etc. useful for this:

```
lclam:
set_induction_scheme_list [list_struct].
Done
lclam:
set_sym_eval_list [qrev1, qrev2, orev1, orev2, oapp1, oapp2, idty, cons_functional].
Done
lclam:
set_wave_rule_to_sym_eval.
Done
```

It should become obvious that there is problem. Sooner or later you will get bored of watching the plan go past, give it a while though so you get a good output to analyse. You will have to kill it by typing control-C, or is you are using ProofGeneral, just click on the Stop button on the PG toolbar. Scroll back up and try and see if there is something missing.

The plan is diverging – it keeps performing induction again and again. There are a number of ways you might patch this – the simplest is to add a lemma.

The planner can't effectively handle the expression

$$qrev(L2, L3 :: nil) <> L4$$

which appears in the first step case after weak fertilisation (rewriting with a hypothesis).

You need the lemma:

$$qrev(A, B) <> C = qrev(A, B <> C)$$

You could just add this as another rule to the list of equations in your library file (calling it `revqrevlemma` perhaps) but it is always possible that some ad hoc lemma like this is false so it is good practice to try and plan it first.

Add this as a conjecture to your theory file and recompile

A sample file is shown in §3.11. Run this new file, set the various lists as above and plan it.

Once it is planned correctly add this lemma to the sym_eval and wave_rule lists to plan `orevqrev` again.

Once again you can find this library file in §3.11.

Once it has been successfully planned you can convert the goal to a lemma for future backward reasoning by typing `qed bwd revqrevlemma`. The new rewrite rule can then be added to one of the rule lists by typing:

```
add_to_sym_eval_list [(user_rewrite ``revqrevlemma_bwd'')].
```

RevQrev can also be proved with the lemma, the associativity of append (`ass_oapp`), which is one of the rewrite rules in the objlists theory. In fact if all the rewrites in that theory were switched on then the proof would have gone through straight away.

The theorem can also be proved using the built-in critics. More (but not much) on this in § 3.6.

3.5 Writing Your Own Methods

The last major aspects to using *λClam* are writing and experimenting with your own methods and critics. This tutorial doesn't cover critic construction since this is rather an advanced topic. To demonstrate method construction, consider a simple theorem about the predicate *divides* defined as:

$$divides\, x\, y = \exists z.(y = (x * z))$$

divides is a function from two natural numbers to true or false (i.e. a predicate). The conjecture we will try to plan is:

$$\forall x.(divides\, 0\, x) \Leftrightarrow (x = 0)$$

To plan this you will need to defined *divides* (as above) and biimplication ($\leftrightarrow$):

$$A \leftrightarrow B \stackrel{\text{def}}{=} (A \rightarrow B) \wedge (B \rightarrow A)$$

Construct a theory file for this conjecture and try planning this using the usual `induction_top normal_ind`. You will need to add your definitions and the rules `plus1, times1` and `idty` to the symbolic evaluation list before you attempt planning.

When planning *λClam* will probably loop again, so stop it and have a look through the output. Your plan should be going wrong at the goal

$$l2 : nat \vdash (\exists l3.(l2 = zero) \rightarrow (l2 = zero))$$

The quantifier is redundant (because $l3$ doesn't appear anywhere else in the formula) but it is preventing the hypothesis being used to prove the conclusion. What we need is some way to discard the redundant quantifier.

If you look through a standard text book containing rules for manipulating equations in the predicate calculus you will probably find the following rule.

$$\frac{\Gamma \vdash \forall x.(P \rightarrow Q)}{\Gamma \vdash (\exists x.P) \rightarrow Q} x \text{ does not appear free in } Q$$

This inference rule could be transformed into a method (so used backwards, basically) and used *before* `imp_i` moves the antecedent of the implication onto the hypothesis list. So the sequence of proof would look like.

$$\vdash (\exists V_0.U_1 = 0) \rightarrow U_1 = 0$$

$$\vdash \forall V_0.(U_1 = 0 \rightarrow U_1 = 0) \text{ using the new method}$$

$$\vdash U_1 = 0 \rightarrow U_1 = 0 \text{ using } \texttt{all\_i}$$

$$U_1 = 0 \vdash U_1 = 0 \text{ using } \texttt{imp\_i}$$

Axiom

We need to write a new method to correspond to this inference rule.

3.5.1 The Anatomy of a Method

An atomic method in *λClam* consists of six slots: Name, Input, Preconditions, Effects, Output and Tactic. These appear in library files as seven arguments to the predicate `atomic`. They are written as fragments of λProlog code. The method for `all_i` appears in the constructive_logic theory file as:

```
atomic constructive_logic all_i
          (seqGoal (H >>> (app forall (tuple [Otype, (abs A)]))))
          true
          true
          (allGoal Otype Y\ (seqGoal (((otype_of Y Otype)::H) >>> (A Y))))
          notacticyet.
```

1. `constructive_logic` is the theory name.

2. `all_i` is the method name.

3. The input is a sequent goal of the form $H \vdash \forall x.A$. This is written in λ*Clam*'s internal higher order syntax. See § 3.4.2 for details.

4. There are no preconditions.

5. There are no effects.

6. `all_i` removes a universally quantified variable, replacing it with an arbitrary constant in the object-level. In λ*Clam* arbitrary constants at the object-level are still universally quantified at the meta-level. `allGoal` is the meta-level universal quantifier. Using `allGoal`, the code above introduces a variable `Y` which will appear as an arbitrary constant within the object-level sequent goal (which is displayed to the user). It supplies `Y` to `A` to replace the universally bound variable. So the resulting goal is the sequent $H \vdash A[x/Y]$

7. `notacticyet` is the tactic that justifies the node in the plan created by this method. There are no tactics implemented in λ*Clam* so `notacticyet` is invariably the appropriate place-holder here.

3.5.2 Compound Methods and the Waterfall

λ*Clam* has a *waterfall* of methods which determines the order in which methods are attempted. The only method λ*Clam* explicitly calls is the top method supplied by the user. This is usually `(induction_top with_critics)` or `(induction_top normal_ind)`. The first of these calls λ*Clam* with the critics mechanism switched on. In this tutorial we have been using `normal_ind` (normal induction) which doesn't use critics for the sake of simplicity. The waterfall is determined by the composition of the top method and the composition of the methods called by it. All other methods are invoked by *compound methods* using *methodicals* (which glue methods together). These are documented in chapter 5. Compound methods (those that use methodicals) are defined using `compound` which has five arguments, the theory name, the name of the method and then a method built up from methodicals, the address at which it was called (usually unused), and lastly preconditions (usually just `true`). For example, the ind_strat method is defined as follows:

```
compound induction (ind_strat IndType)
      ( then_meths (induction_meth IndType Scheme Subst)
                   (pair_meth (map_meth (step_case IndType)) (map_meth id_meth)) )
        _
        true.
```

This applies induction and then applies `id_meth` (do nothing) to the base case(s) and `step_case` to the step case(s). Note how it passes the "sort" of induction being attempted (e.g. `with_critics` or `normal_ind`) on to the step case method.

Once you have defined a method you need to determine the appropriate place in the waterfall for it to be applied. This will mean defining your own top method and possibly other compound methods. Most often you will use the predicate `orelse_meth` to do this. `orelse_meth methodA methodB` tries `methodA` and if that fails then tries `methodB`. It is used extensively in common top methods like `induction_top_meth` (below) which defines the list of methods to be tried and the order they are to be tried in.

```
compound induction (induction_top IndType) (complete_meth
                (repeat_meth
                (orelse_meth trivial
                (orelse_meth allFi
                (orelse_meth taut
                (orelse_meth sym_eval
                (orelse_meth existential
                (orelse_meth (repeat_meth generalise)
                         (ind_strat IndType)
        ))))))))
        _
        true.
```

It is important to decide where in the waterfall you want your method to appear (e.g. before or after induction?).

Decide where in the waterfall you need your new method

Look at the original proof attempt and identify the goal to which you want your method to apply. It is important that your new method is attempted in the waterfall *before* the method that is being called on the goal in the failed attempt. To see the waterfall you can look at the method descriptions in chapter 7. Be aware that the method you are interested in may not be called by `top_meth` but by one of the compound methods in `top_meth` such as `taut` or `ind_strat`.

A good rule-of-thumb is to place your new method in the waterfall directly before the old method that was used in the failed proof attempt. This analysis should lead you to place your new method before `imp_i` in the `taut` method.

Define a method for the new inference rule and a new `taut` method and top method to call it

The best way to program new methods is generally by analogy to similar methods like `all_i` and `imp_i`. (look in the constructive logic theory file)

Fortunately, you do not need to worry about the side condition on the inference rule (x not free in Q) because the implementation of the syntax in λ*Clam* takes care of this for you.

Try planning the conjecture once more

To call your top method you will need to use

```
lclam:
pds_plan my_top_meth divides_zero.
```

where `my_top_meth` is the name of your top method.

A sample file is shown in §3.12.

3.6 Critics

On the whole we would advise using `induction_top with_critics` as a top method not `induction_top normal_ind` since this allows the use of critics to patch rippling proofs, allowing more to go through. The critics code is rather complicated and so it is not really possible to provide a tutorial on its usage here. To get an idea about how critics work in λ*Clam* it is best to look in the `wave` theory file and `wave_critics` theory file for examples and read the discussion in chapter 5.

3.7 Automating Command Line interactions

You are likely to get tired of continually having to type certain sets of instructions into the command line. The best way to take care of this is to use the ProofGeneral interface which allows you to store proof scripts in a .lcm file.

See also chapter 6 for a discussion of the `benchmark_plan` predicate which helps with the automation of benchmarking tasks.

3.8 Conclusion

This tutorial has attempted to overview the main tasks involved in using and experimenting with the λ*Clam* proof planning system. It has not discussed in any detail the methods and methodicals available in λ*Clam*, nor has it provided any guidance in the use of λProlog – an understanding of which is required for all but the most basic attempts at method creation. Hopefully it has given a rough framework for more detailed investigation and pointed you in the direction of the examples and manuals needed for this.

3.9 Further Reading

At some point when using λ*Clam* you might well need to get to grips with λProlog. There are not many good manuals available on λProlog at time of writing. The implementation we use is Teyjus and its web page with documentation can be found at `http://teyjus.cs.umn.edu/`. A web page dedicated to the language itself, with some course notes etc. can be found at `http://www.cse.psu.edu/ dale/lProlog/`.

3.10 RevQrev Library File

3.10.1 Signature File

```
sig ex1.

accum_sig objlists.
accum_sig constructive_logic.

type ex1 theory.

type qrev osyn.

type qrev1 rewrite_rule.
type qrev2 rewrite_rule.

type orevqrev query.
```

```
end
```

3.10.2 Module File

```
module ex1.

accumulate objlists.
accumulate constructive_logic.

has_otype ex1 qrev ((tuple_type [(olist X), (olist X)]) arrow (olist X)).

definition ex1 qrev1
        trueP
        (app qrev (tuple [onil, L]))
        L.
definition ex1 qrev2
        trueP
        (app qrev (tuple [(app ocons (tuple [H, T])), L]))
        (app qrev (tuple [T, (app ocons (tuple [H, L]))])).

top_goal ex1 orevqrev []
        (app forall (tuple [(olist nat), (abs x\
                (app forall (tuple [(olist nat), (abs y\
                        (app eq (tuple [
        (app oapp (tuple [(app orev x), y])),
        (app qrev (tuple [x, y]))]))))])))])).
end
```

3.11 RevQrevLemma Library File

3.11.1 Signature File

```
sig ex2.

accum_sig objlists.
accum_sig constructive_logic.

type ex2 theory.

type qrev osyn.

type qrev1 rewrite_rule.
type qrev2 rewrite_rule.
type oapp_lemma rewrite_rule.

type orevqrev query.
type revqrevlemma query.

end
```

3.11.2 Module File

```
module ex2.
```

```
accumulate objlists.
accumulate constructive_logic.

has_otype ex2 qrev ((tuple_type [(olist X), (olist X)]) arrow (olist X)).

definition ex2 qrev1
        trueP
        (app qrev (tuple [onil, L]))
        L.
definition ex2 qrev2
        trueP
        (app qrev (tuple [(app ocons (tuple [H, T])), L]))
        (app qrev (tuple [T, (app ocons (tuple [H, L]))])).

lemma ex2 oapp_lemma equiv
        trueP
        (app oapp (tuple [(app qrev (tuple [L2, L3])), L4]))
        (app qrev (tuple [L2, (app oapp (tuple [L3, L4]))])).

top_goal ex2 orevqrev []
        (app forall (tuple [(olist nat), (abs x\
                (app forall (tuple [(olist nat), (abs y\
                        (app eq (tuple [
        (app oapp (tuple [(app orev x), y])),
        (app qrev (tuple [x, y]))]))))])))])).

top_goal ex2 revqrevlemma []
        (app forall (tuple [(olist nat), (abs x\
                (app forall (tuple [(olist nat), (abs y\
                        (app forall (tuple [(olist nat), (abs z\
        (app eq (tuple [
        (app oapp (tuple [(app qrev (tuple [x, y])), z])),
        (app qrev (tuple [x, (app oapp (tuple [y, z]))]))]))))])))])))])
```

3.12 Divides Library File

3.12.1 Signature File

```
sig ex3.

accum_sig arithmetic.
accum_sig constructive_logic.

type ex3 theory.

type divides osyn.
type iff osyn.

type divides1 rewrite_rule.

type divides_zero query.
```

```
type ex3_top_meth meth.
type ex3_taut meth.
type all_imp_left meth.

end
```

3.12.2 Module File

```
module ex3.

accumulate arithmetic.
accumualte constructive_logic.

has_otype ex3 divides ((tuple_type [nat, nat]) arrow nat).
has_otype ex3 iff ((tuple_type [bool, bool]) arrow bool).

definition ex3 iff1
        trueP
        (app iff (tuple [A, B]))
        (app and (tuple [(app imp (tuple [A, B])),
                         (app imp (tuple [B, A]))])).

definition ex3 divides1
        trueP
        (app divides (tuple [A, B]))
        (app exists (tuple [nat, (abs c\
                (app eq (tuple [B, (app otimes (tuple [A, C]))]))))])).

top_goal ex3 divides_zero []
        (app forall (tuple [nat, (abs x\
                (app iff (tuple [
                        (app divides (tuple [zero, x])),
                        (app eq (tuple [x, zero]))])))])).

compound ex3 ex3_top_meth (complete_meth
                (repeat_meth
                (orelse_meth trivial
                (orelse_meth allFi
                (orelse_meth ex3_taut
                (orelse_meth sym_eval
                (orelse_meth existential
                (orelse_meth (repeat_meth generalise)
                         (ind_strat normal_ind)
        ))))))))
        _
        true.

compound ex3 ex3_taut
      (complete_meth
        (repeat_meth
          (orelse_meth trivial
           (orelse_meth false_e
           (orelse_meth all_imp_left
          (orelse_meth neg_i
```

```
            (orelse_meth neg_e
            (orelse_meth imp_i
            (orelse_meth all_i
            (orelse_meth and_i
            (orelse_meth and_e
            (orelse_meth or_e
            (orelse_meth imp_e1
            (orelse_meth imp_e2
            (orelse_meth imp_e3
            (orelse_meth imp_e4
            (orelse_meth or_il or_ir)))))))))))))))))
        true.

atomic ex3 all_imp_left
        (seqGoal (H >>> (app imp
                        (tuple [(app exists (tuple [QType, (abs P)])), Q]))))
        true
        true
        (seqGoal (H >>> (app forall (tuple [QType,
                        (abs X\ (app imp (tuple [(P X), Q]))))]))))
        notacticyet.


end
```

Chapter 4

Using λ*Clam*

4.1 Introduction

Proof planning in λ*Clam* basically works as follows. A goal is presented to the system, expressed using terms which may have types (the terms are not supposed to be tied down to any particular logic although there is an implicit assumption throughout λ*Clam* that we are working within a sequent calculus).

A plan is formed to find a proof of this goal at some level of abstraction (usually above the level of basic inference rules, but not necessarily so). This plan is formed using methods placed within a waterfall. The waterfall determines a general strategy for the process of the proof and imposes a hierarchy upon the methods. This means that only certain methods are considered as viable ways to extend the plan at certain points in a proof. There are two sorts of method, compound and atomic. Compound methods determine the structure of the waterfall. When a compound method is being attempted only its *submethods* can be considered and the compound may even impose a specific order in which the submethods should be attempted. In this way, for example, the `step_case` method for induction ensures that rippling is not attempted unless appropriate meta-level annotations have been put into place. Atomic methods have pre-conditions and effects (or post-conditions). If all of an atomic method's pre-conditions are satisfied then it is applied and the effects are used to determine the new goal on which planning should be attempted.

If we want to come more into line with the OMEGA description of proof planning then we should also consider compound methods as atomic methods at a higher level of abstraction. The submethods of the compound method would then be the tactic that justified that node. There are obviously some practical problems with this (e.g. we often need to execute the submethods to determine the output of the compound methods).

We might also want to consider a return to the original proof planning spec. where post-conditions had a larger role than simply the calculation of output. Do we have any good examples where it would have been useful to have some extra information expressed (other than as annotations on the goal) where an explicit post-condition would have been useful?

In theory the plan constructed by the methods and the planning mechanism is a tree each node of which is justified by a tactic. This plan is a proof at some level of abstraction. The proof can be expanded at a lower level by executing the tactics at the nodes and so on until the plan is a tree whose nodes are justified by axioms and inference rules in some logic. In practice λ*Clam* produces just one plan and there is no way at present to execute the nodes and get a proof at a lower level of abstraction.

4.2 Running *λClam*

1. Add the following paths to `TJPATH`:

   `$LCLAM_HOME/src/util`
   `$LCLAM_HOME/src/theories`
   `$LCLAM_HOME/src/planner`
   `$LCLAM_HOME/src/compiler`
   `$LCLAM_HOME/src/theories/tutorial`
   `$LCLAM_HOME/src/io` `$LCLAM_HOME/src/syntax` `$LCLAM_HOME/src/core_theories` `$LCLAM_HOME/src/`

2. Type `$LCLAM_HOME/bin/lclam`

We advise you run *λClam* from within emacs (and preferably using the supplied ProofGeneral interface) since *λClam* generates a lot of output. A sample small proof is shown below.

```
[lad@bartok lambda-clam-teyjus]$ $LCLAM_HOME/bin/lclam
lclam:
pds_plan taut taut1.
planning
taut1
GOAL:
>>> (obj -> obj)
ADDRESS: nil

Attempting...
Method application:  taut
succeeded

GOAL:
>>> (obj -> obj)
ADDRESS: nil

Attempting...
trivial
failed
Attempting...
false_e
failed
Attempting...
neg_i
failed
Attempting...
neg_e
failed
Attempting...
Method application:  imp_i
succeeded

GOAL:
obj
>>> obj
ADDRESS: 1 :: nil

Attempting...
Method application:  trivial
succeeded
```

```
GOAL:
trueGoal!
ADDRESS: 1 :: 1 :: nil

Attempting...
branch closed!

reached empty agenda

Plan found (not displayed) ...
Plan Succeeded
lclam:
```

4.3 λ*Clam* Operation

Below are the basic command line options for running λ*Clam*.

4.3.1 Miscellaneous

`quit`

`type quit o.`

Synopsis

Quits out of λ*Clam*.

Description

This will return you to the prolog prompt.

See Also

`halt`

`halt`

`type halt o.`

Synopsis

Quits out of λ*Clam* and λProlog.

See Also

`quit`

`reset_lclam`

`type reset_lclam o.`

Synopsis

Restores the default settings of λ*Clam*. The reset command clears the induction schemes list, the sym_eval list, the general list and the wave rule list.

`step_by_step`

`type step_by_step switch -> o.`

Synopsis

Switches step by step planning on and off.

Description

The switch type can have two values `on` or `off`.

Examples

`step_by_step on.`
This will switch λ*Clam* into step by step mode.
`step_by_step off.`
This will switch λ*Clam* into automatic mode.

See Also

`continue`.
`abandon`.
`backtrack`.
`plan_node`.
`try`.

4.3.2 Output control

`interaction_mode`

`type interaction_mode interaction -> o.`

Synopsis

Changes the output options (e.g. switches on or off pretty printing).

Description

Currently interaction type can have two values `command` (print to command line) or `command_pretty` (pretty print to command line). Mostly you will want the mode set to `command_pretty` which is the default. Occasionally this suffers memory problems in long proofs in which case the `command` option can be used to enable the plan to go through.

There is also an `xbmode` intended for use with a graphic interface. This has not been fully implemented yet.

Examples

`interaction_mode command.`
This will switch off the pretty printer, simply printing the raw λ*Clam* syntax to the command line.

`plan_printing`

`type plan_printing switch -> o.`

Synopsis

Switches printing of plans on and off.

Description

The switch type can have two values, `on` or `off`.

Examples

`plan_printing on.`
This will switch λ*Clam* into a mode where it prints out the proof plans once tthey are found.
`plan_printing off.`
This will switch λ*Clam* into a mode where it doesn't print plans. This is the default.

---

silent_mode

`type silent_mode o.`

Synopsis

Switches λ*Clam* into silent output mode. Silent mode suppresses output from failed method applications and backtracking attempts.

---

super_silent_mode

`type super_silent_mode o.`

Synopsis

Switches λ*Clam* into super-silent output mode. Super-silent mode works like silent mode but also suppresses goal printing, which can be useful when dealing with very large goals.

---

verbose_mode

`type verbose_mode o.`

Synopsis

Switches λ*Clam* into verbose output mode, which is the default. In verbose mode all goal, method and backtracking information is displayed.

4.3.3 Planning

---

plan_this

```
type plan_this (plan_state -> (action -> plan_state -> plan_state ->
o) -> plan_state -> o) -> meth -> query -> (action -> plan_state -> plan_state
-> o) -> o.
```

Synopsis

Starts a plan.

Description

This is the top level call for starting planning. It has four arguments. The first argument is the planner being used. There are currently two planners built into the system, `pds_planner` and `pds_iter_planner`. Both are partial data structure planners which can be run in normal and interactive modes; the former is a basic depth-first planner whereas the latter performs an iterative deepening search. It is also possible to build in a naive depth-first planner `df_plan` is a depth-first planner, which can not be run interactively. The second argument is the top method the third the goal query and the last is the predicate which will control composition of agendas and hence the search strategy. Only one search strategy is currently implemented `depth_first_plan`.

When the planner is executing it returns some basic information by default. This is the name of the method the planner is currently attempting to apply and the name of the method it has succeeded in applying.

The planner generates quite a lot of output . When it attempt to extend the plan at a given node the planner prints the goal at that node and the address in the plan of the node. It then records the names of each method it applies at that node and whether they succeeded or failed. It also reports when it backtracks over a method application (see the section 4.3.2 for information on suppressing some of this output).

When a critic is going to be invoked the planner also prints out the instantiations of all method preconditions and records whether they succeeded or failed. This can be used for debugging new methods.

In general it will be easier to use one of the wrapper functions such as `pds_plan` or `pds_iter_plan`.

Examples

`plan_this pds_planner taut taut1 depth_first_plan.`
This will apply the tautology method, `taut`, to the goal `taut1`.
`plan_this pds_planner (then_meth (then_meth sym_eval trivial) triv_meth) eqzero depth_first_plan.`
This applies the method `(then_meth (then_meth sym_eval trivial) triv_meth)`, (rewrite exhaustively and then look for the goal in the hypotheses), to the query `eqzero`.

See Also

`pds_plan`.

---

pds_plan

`type pds_plan meth -> query -> o`

Synopsis

Starts a plan.

Description

This is a wrapper for `plan_this` which applies the PDS planner using depth-first search. It has two arguments. The first argument is the top method and the second is the goal query.

When the planner is executing it returns some basic information by default. This is the name of the method the planner is currently attempting to apply and the name of the method it has succeeded in applying.

The planner generates quite a lot of output . When it attempt to extend the plan at a given node the planner prints the goal at that node and the address in the plan of the node. It then records the names of each method it applies at that node and whether they succeeded or failed. It also reports when it backtracks over a method application (see the section 4.3.2 for information on suppressing some of this output).

When a critic is going to be invoked the planner also prints out the instantiations of all method preconditions and records whether they succeeded or failed. This can be used for debugging new methods.

Examples

`pds_plan (induction_top with_critics) assp.`
This will apply the induction top method with critics to the associativity of plus, `assp`.
`pds_plan (then_meth (then_meth sym_eval trivial) triv_meth) eqzero.`
This applies the method `(then_meth (then_meth sym_eval trivial) triv_meth)`, (rewrite exhaustively and then look for the goal in the hypotheses), to the query `eqzero`.

See Also

`plan_this`.

---

pds_iter_plan

`type pds_iter_plan meth -> query -> o`

Synopsis

Starts a plan using the PDS iterative deepening planner.

Description

`pds_iter_plan` works exactly like `pds_plan` but uses the iterative deepening planner instead of the standard depth-first planner. The default configuration uses an initial depth limit of 3 and a depth increment of 2 after each pass. At the moment these variables are hard-coded in but they may be included as command line arguments in the future if there is user demand for it. The depth of the current plan node is also displayed in the planning output after the node address.

Call the iterative deepening planner with the command `pds_iter_plan [method] [goal]`. If a spypoint is set at the top method (`set_spypoint my_top_meth`) this will have the effect of asking the user whether to continue at each new iteration.

See Also

`pds_plan`.
`plan_this`.
`set_spypoint`.

---

claudio_plan

`type claudio_plan meth -> query -> plan_state -> o`

Synopsis

Starts a plan using the Claudio PDS planner.

Description

`claudio_plan` plans the goal using the same mechanism as `pds_plan` but, unlike `pds_plan`, is able to return the plan as a metavariable.
Call the Claudio planner with the command `claudio_plan [method] [goal] X`, which will bind X to the final plan state.

See Also

`pds_plan`.
`plan_this`.
`claudio_iter_plan`.

claudio_iter_plan

`type claudio_iter_plan meth -> query -> plan_state -> o`

Synopsis

Starts a plan using the Claudio iterative deepening planner.

Description

`claudio_iter_plan` works exactly like `claudio_plan` but uses the iterative deepening planner instead of the standard depth-first planner. The default configuration uses an initial depth limit of 3 and a depth increment of 2 after each pass. At the moment these variables are hard-coded in but they may be included as command line arguments in the future if there is user demand for it. The depth of the current plan node is also displayed in the planning output after the node address.

Call the Claudio iterative deepening planner with the command `claudio_iter_plan [method] [goal] X`, which will bind X to the plan state. If a spypoint is set at the top method (`set_spypoint my_top_meth`) this will have the effect of asking the user whether to continue at each new iteration.

See Also

`pds_plan`.
`plan_this`.
`claudio_plan`.

qed

`type qed reasontype -> query -> o`

Synopsis

Generates a rewrite rule from a query for use in further planning attempts.

Description

The `qed` command converts a goal to a rewrite rule which can then be used in future proofs. The command syntax is: `qed {fwd,bwd} [goal].` The first argument is used to indicate whether the lemma is to be used for forward or backward reasoning, and affects the polarity of the rewrite rule generated. The second argument is the goal to be used in generating the rewrite rule. The name of the rewrite rule generated by `qed fwd mygoal` is `user_rewrite(‘‘mygoal_fwd’’)` (and similarly for `qed bwd`).

Unlike the Isabelle tt qed say, λ-Clam's `qed` does *not* require a goal to have been proven before a rewrite can be generated from it. This is in order to allow users to perform 'rapid prototyping' of large proofs, deferring the proofs of sub-lemmas until later in the development.

The rewrite rule is generated from the goal according to the following scheme. Goals in λ*Clam* consist of a list of hypotheses and a conclusion, in which variables are explicitly quantified at the object level. A rewrite rule has four main components: a *polarity* indicating the direction of the rule, a slot for any side condition on the rule, and two slots for the left and right hand sides of the rule. Variables which can be instantiated when the rule is applied are represented by λProlog metavariables. The side condition on the generated rewrite rule then is just the conjunction of all the hypotheses in the goal; typically there are no hypotheses, in which case the condition is trivial (`trueP`). All outer universal quantifiers are removed from the conclusion; the quantified variables become logical λProlog variables. The direction of the rule is specified by the user (forward reasoning generates left-to-right rewrites; backward reasoning generates right-to-left) and is used in combination with the main connective of the conclusion in determining the left and right hand sides of the rule:

Goal	Direction	LHS	RHS
$A \to B$	$\Rightarrow$	A	B
$A \to B$	$\Leftarrow$	B	A
$A \leftrightarrow B$	$\Rightarrow$	A	B
$A \leftrightarrow B$	$\Leftarrow$	B	A
$A = B$	$\Rightarrow$	A	B
$A = B$	$\Leftarrow$	B	A
A	Any	A	trueP

Examples

`qed bwd mygoal.`
This will generate a backwards reasoning lemma `user_rewrite (‘‘mygoal_bwd’’)` from the goal `mygoal`.

---

set_spypoint

`type set_spypoint meth -> o`

Synopsis

Sets a spypoint.

Description

When a spypoint is set for a particular method, subsequent planning attempts will be interrupted immediately after successful applications of that method. Once interrupted any of the commands normally available in step-by-step mode can be attempted (see section 4.3.3).

Examples

`set_spypoint sym_eval.`
This will flag the method `sym_eval` as a spypoint. In subsequent planning attempts, planning will be interrupted immediately after successful applications of `sym_eval`.

`remove_spypoint`

`type remove_spypoint meth -> o`

Synopsis

Removes a spypoint.

Examples

`remove_spypoint sym_eval.` This will remove the spypoint for the method `sym_eval`.

4.3.4 Commands in Step by Step Mode

If you are planning with step by step mode switched on then after every plan step you will be shown the newly introduced nodes (in fact a cut down version of them) and given a prompt

```
Continue Planning?:
```

There are several options at this point.

`continue`

`type continue o.`

Synopsis

Tells the planner to continue planning the next step.

`abandon`

`type abandon o.`

Synopsis

Tells the planner to abandon this proof plan attempt entirely.

`backtrack`

`type backtrack o.`

Synopsis

Tells the planner to backtrack.

Description

`backtrack` causes *λClam* to backtrack over the last method application. If the method could succeed in more than one way you will get these alternative successes before another method is attempted.

plan_node

`type plan_node (list int) -> o.`

Synopsis

Switches planning to the named node.

Description

`plan_node` tells the planner to plan next the node with the given address. It will only allow you to switch to an nodes on the current agenda (i.e. one's to which no method has been applied) and will return an error message if the address is not that of an node on the agenda.

try

`type try meth -> o.`

Synopsis

Tells the planner to attempt the given method before it attempts the method it would normally apply at this point.

4.3.5 Theory Information Database Commands

There are a number of sorts of *theory information* which are used in planning but are not a specific part of the planning process. Lots of this information is compiled into *λClam* as part of its internal theory mechanism but more information can be added by the user.

To use theory information properly a user will, at present, need to understand *λClam*'s internal syntax. This is shown in figure 4.1. The internal syntax is based

```
type abs   (osyn -> osyn) -> osyn.
type app   osyn -> osyn -> osyn.
type tuple (list osyn) -> osyn.

type arrow      osyn -> osyn -> osyn.
type tuple_type (list osyn) -> osyn.

type otype_of osyn -> osyn -> osyn.
```

Figure 4.1: Constructors for *λClam*'s internal syntax

on Higher Order Abstract Syntax (HOAS). Basic terms (objects of type `osyn`) can be Prolog variables or constants (e.g. `P` or `plus`). The constructors `abs`, `app` and `tuple` are then used to build up more complex terms. `app F X` applies the function `F` to the argument X. If there is more than one argument then `tuple` is used, so, for instance `app plus (tuple [1, 2])` represent the term, $1 + 2$. `abs` is used to represent λ-abstraction. This uses λProlog's internal λ-abstraction mechanism. So the function $\lambda x.x$ would be `abs X\ X` in λ*Clam*'s internal syntax.

`otype_of` is used to assign types to terms and `arrow` and `tuple_type` are used to construct function types.

All theory information is indexed by the name of the theory to which it belongs (e.g. arithmetic). Users may specify their own theories as `user_theory ‘‘string’’`.

Syntax

Users may specify their own syntax constants using `user_object ‘‘string’’`.

add_osyn

`type add_osyn theory -> osyn -> osyn -> o.`

Synopsis

Adds a new syntactic constant into λ*Clam*.

Description

`add_osyn` takes three arguments: the name of the theory that contains the constant, the constant name and the type of the constant (expressed in higher order abstract syntax).

This adds the constant with its type and theory information to the database.

Example

```
add_osyn (user_theory ‘‘mytheory’’) (user_object ‘‘myplus’’) ((tuple_type
[nat, nat]) -> nat).
```

This introduces a new constant `user_object ‘‘myplus’’`, which is of type $myplus : (nat, nat) \rightarrow nat$ - (i.e. it takes a pair of natural numbers and returns a natural number. This is placed in the user defined theory `user_theory ‘‘mytheory’’`.

query_osyn

`type query_osyn theory -> osyn -> osyn -> o.`

Synopsis

`query_osyn` allows the user to query the database for information about a syntax constant.

Description

Note that the arguments can be left as Prolog variables if desired. You can not (at present) backtrack over this so it is impossible to use it (for instance) to find all the constants in a given theory.

Example

Suppose, for instance, given you've added the piece of syntax in the example for `add_osyn` at the command line you could then query it. A session transcript follows:

```
ordie[src] tjsim lclam.
Welcome to Teyjus

Copyright (C) 1999 Gopalan Nadathur
Teyjus comes with ABSOLUTELY NO WARRANTY
This is free software, and you are welcome to redistribute it
under certain conditions.  Please view the accompanying file
"COPYING" for more information.
[lclam] ?- lclam.
lclam:
add_osyn (user_theory "mytheory") (user_object "myplus")
                           ((tuple_type [nat, nat]) arrow nat).
Done
lclam:
query_osyn T (user_object "myplus") Type.
user_theory "mytheory"
myplus
tuple_type (nat :: nat :: nil) arrow nat
lclam:
```

Goals

Users may specify their own goals using `user_query ‘‘string’’`.

---

add_query

`type add_query theory -> query -> (list osyn) -> osyn -> o.`

Synopsis

Adds a new goal into λ*Clam*.

Description

It takes four arguments the name of the Theory that contains the goal, the goal name, a list of hypotheses (terms in λ*Clam*'s internal syntax) and the conclusion (a term in λ*Clam*'s internal syntax)

This adds the goal to the database. Proof plan attempts may then be made on that goal. The constructor `user_query` can be used to define new goal names.

---

query_top_goal

`type query_top_goal theory -> query -> o.`

Synopsis

`query_top_goal` allows the user to query the database for information about a goal.

Description

Note that the arguments can be left as Prolog variables if desired. You can not (at present) backtrack over this so it is impossible to use it (for instance) to find all the goals in a given theory.

Rewrite Rules

Users may specify their own rewrite rule names using `user_rewrite ‘‘string’’`.

add_definition

`type add_definition theory -> rewrite_rule -> osyn -> osyn -> osyn -> o.`

Synopsis

Adds a definition to the theory database.

add_axiom

`type add_axiom theory -> rewrite_rule -> direction -> osyn -> osyn -> osyn -> o.`

Synopsis

Adds and axiom to the theory database.

Description

`direction` should be one of `ltor`, `rtol` or `equiv`. Depending on whether the rule represents an implication which holds from *left to right*, *right to left* or is an equality.

add_lemma

`type add_lemma theory -> rewrite_rule -> direction -> osyn -> osyn -> osyn -> o.`

Synopsis

Adds a lemma to the theory database.

query_rewrite_rule

`type query_rewrite_rule theory -> rewrite_rule -> o.`

Synopsis

Returns information about the rewrite rule.

Methods

Users may specify their own method names using `user_method ‘‘string’’`.

add_atomic

`type add_atomic theory -> meth -> goal -> o -> o -> goal -> tactic -> o.`

Synopsis

Adds an atomic method to the theory database.

add_compound

`type add_compound theory -> meth -> meth -> o -> o.`

Synopsis

Adds a compound method to the theory database. NB. compound methods now have precondition slots.

---

query_method

`type query_method theory -> meth -> o.`

Synopsis

Returns information about the method.

Induction Schemes

Users may specify their own scheme names using `user_scheme ‘‘string’’`.

---

add_scheme

`type add_scheme theory -> scheme -> osyn -> subst -> osyn -> goal -> goal -> o.`

Synopsis

Adds a new induction scheme to the database.

---

query_induction_scheme

`type query_induction_scheme theory -> scheme -> o.`

Synopsis

Prints out the scheme information.

List of Rewrites for Symbolic Evaluation

Symbolic Evaluation is done with reference to a (theoretically) confluent set of rewrite rules. By default this list is empty.

---

add_to_sym_eval_list

`type add_to_sym_eval_list (list rewrite_rule) -> o.`

Synopsis

Adds the named rewrite rules to the symbolic evaluation list.

---

delete_from_sym_eval_list

`type delete_from_sym_eval_list (list rewrite_rule) -> o.`

Synopsis

Removes the named rules from the symbolic evaluation list.

---

set_sym_eval_list

`type set_sym_eval_list (list rewrite_rule) -> o.`

Synopsis

Sets the symbolic evaluation list to the given list.

add_theory_to_sym_eval_list

```
type add_theory_to_sym_eval_list theory -> o.
```

Synopsis

Adds all the rewrite rules in the named theory to the symbolic evaluation list.

add_theory_defs_to_sym_eval_list

```
type add_theory_defs_to_sym_eval_list theory -> o.
```

Synopsis

Adds all the definitional rewrite rules in the named theory to the symbolic evaluation list.

delete_theory_from_sym_eval_list

```
type delete_theory_from_sym_eval_list theory -> o.
```

Synopsis

Removes all the rewrite rules in the named theory from the symbolic evaluation list.

set_theory_sym_eval_list

```
type set_theory_sym_eval_list theory -> o.
```

Synopsis

Sets the symbolic evaluation list to contain all the rewrite rules in the named theory and nothing else.

query_sym_eval_list

```
type query_sym_eval_list o.
```

Synopsis

Returns the symbolic evaluation list.

List of Rewrites for Rewriting

General rewriting can also be attempted. This is done with reference to a set of rewrite rules. By default this list is empty.

add_to_rewrite_list

```
type add_to_rewrite_list (list rewrite_rule) -> o.
```

Synopsis

Adds the named rewrite rules to the general rewriting list.

delete_from_rewrite_list

```
type delete_from_rewrite_list (list rewrite_rule) -> o.
```

Synopsis

Removes the named rules from the general rewriting list.

set_rewrite_list

```
type set_rewrite_list (list rewrite_rule) -> o.
```

Synopsis

Sets the general rewriting list to the given list.

add_theory_to_rewrite_list

```
type add_theory_to_rewrite_list theory -> o.
```

Synopsis

Adds all the rewrite rules in the named theory to the rewriting list.

delete_theory_from_rewrite_list

```
type delete_theory_from_rewrite_list theory -> o.
```

Synopsis

Removes all the rewrite rules in the named theory from the rewriting list.

set_theory_rewrite_list

```
type set_theory_rewrite_list theory -> o.
```

Synopsis

Sets the general rewriting list to contain all the rewrite rules in the named theory and nothing else.

query_general_rewrites_list

```
type query_general_rewrites_list o.
```

Synopsis

Returns the general rewriting list.

List of Rewrites for Rippling

Rippling is done with reference to a set of rewrite rules. By default this list is empty.

add_to_wave_rule_list

```
type add_to_wave_rule_list (list rewrite_rule) -> o.
```

Synopsis

Adds the named rewrite rules to the wave rule list.

delete_from_wave_rule_list

```
type delete_from_wave_rule_list (list rewrite_rule) -> o.
```

Synopsis

Removes the named rules from the wave rule list.

set_wave_rule_list

```
type set_wave_rule_list (list rewrite_rule) -> o.
```

Synopsis

Sets the wave rule list to the given list.

add_theory_to_wave_rule_list

```
type add_theory_to_wave_rule_list theory -> o.
```

Synopsis

Adds all the rewrite rules in the named theory to the wave rule list.

add_theory_defs_to_wave_rule_list

```
type add_theory_defs_to_wave_rule_list theory -> o.
```

Synopsis

Adds all the definitional rewrite rules in the named theory to the wave rule list.

delete_theory_from_wave_rule_list

```
type delete_theory_from_wave_rule_list theory -> o.
```

Synopsis

Removes all the rewrite rules in the named theory from the wave rule list.

set_theory_wave_rule_list

```
type set_theory_wave_rule_list theory -> o.
```

Synopsis

Sets the wave rule list to contain all the rewrite rules in the named theory and nothing else.

set_wave_rule_to_sym_eval

```
type set_wave_rule_to_sym_eval o.
```

Synopsis

Sets the wave rule list to whatever is currently in the symbolic evaluation list.

query_wave_rule_list

```
type query_wave_rule_list o.
```

Synopsis

Returns the wave rule list.

Induction Scheme List

Induction is done with reference to a set of induction schemes. By default this list is empty.

add_to_induction_scheme_list

```
type add_to_induction_scheme_list (list scheme) -> o.
```

Synopsis

Adds the named schemes to the induction scheme list.

delete_from_induction_scheme_list

```
type delete_from_induction_scheme_list (list scheme) -> o.
```

Synopsis

Removes the named schemes from the induction scheme list.

set_induction_scheme_list

```
type set_induction_scheme_list (list scheme) -> o.
```

Synopsis

Sets the induction scheme list to the named schemes.

add_theory_to_induction_scheme_list

```
type add_theory_to_induction_scheme_list theory -> o.
```

Synopsis

Add all the schemes in a given theory to the induction scheme list.

delete_theory_from_induction_scheme_list

```
type delete_theory_from_induction_scheme_list theory -> o.
```

Synopsis

Removes all the schemes in a given theory from the induction scheme list.

set_theory_induction_scheme_list

```
type set_theory_induction_scheme_list theory -> o.
```

Synopsis

Sets the induction scheme list to contain all the schemes in the given theory and nothing else.

query_induction_scheme_list

```
type query_induction_scheme_list o.
```

Synopsis

Returns the induction scheme list.

User Lemma List

The user lemma list is only allowed to contain rewrite rules declared in a theory as a `lemma`.

add_to_user_lemma_list

```
type add_to_user_lemma_list (list rewrite_rule) -> o.
```

Synopsis

Adds the named lemmas to the user lemma list. The command will fail if you attempt to add rewrite rules which have not been declared as lemmas (i.e. definitions, axioms, or hypotheses).

delete_from_user_lemma_list

```
type delete_from_user_lemma_list (list rewrite_rule) -> o.
```

Synopsis

Removes the named lemmas from the user lemma list.

set_user_lemma_list

```
type set_user_lemma_list (list rewrite_rule) -> o.
```

Synopsis

Sets the user lemma list to the named lemmas. The command will fail if you attempt to add rewrite rules which have not been declared as lemmas (i.e. definitions, axioms, or hypotheses).

User Definition List

The user definition list is only allowed to contain rewrite rules declared in a theory as a `definition`.

add_to_user_defn_list

```
type add_to_user_defn_list (list rewrite_rule) -> o.
```

Synopsis

Adds the named definitions to the user definition list. The command will fail if you attempt to add rewrite rules which have not been declared as definitions (i.e. lemmas, axioms, or hypotheses).

delete_from_user_defn_list

```
type delete_from_user_defn_list (list rewrite_rule) -> o.
```

Synopsis

Removes the named definitions from the user definition list.

set_user_defn_list

```
type set_user_defn_list (list rewrite_rule) -> o.
```

Synopsis

Sets the user definition list to the named definitions. The command will fail if you attempt to add rewrite rules which have not been declared as definitions (i.e. lemmas, axioms, or hypotheses).

Chapter 5

The Structure of the λ*Clam* Proof Planning Mechanism

5.1 Introduction

The basic planning mechanism at the heart of λ*Clam* is not about induction or rippling or any specialised methods or heuristics for automated proof. It is intended to provide some machinery upon which such methods and heuristics can be built.

This basic machinery consists of the notions of terms, types, methods, critics and plans. At present λ*Clam* doesn't provide much by way of support for tactics — although this may change in the future, when we hopefully will attach a theorem prover to λ*Clam*.

This chapter considers each of the aspects of the planning mechanism in turn in a general way. A person developing their own strategy (method waterfall), logic, etc. in λ*Clam* should not have to alter any of the files contained in the core planning part of λ*Clam* (i.e. any of the subdirectories of `src` apart from `theories`). However, it is useful to have a general understanding of what they do.

5.2 The Plan Data Structure

Our plans looks something like figure 5.1. A proof plan is a labelled tree — the labels contain a lot of information *possibly redundant?*. Only the goal is shown in the figure, but they are also labelled with the name of the method applied at the node, the methodical continuation etc. All the nodes in this tree are *and* nodes, that is for the plan to be successfully completed, each branch below a node must also be successfully completed. As well as the standard *and* nodes the plan can contain quantified nodes (indicated by the `EXISTS X` in the figure). These are usually generated by the elimination of quantifiers from goals and replacing with arbitrary constants (for universally quantified goals) or meta-variables (for existentially quantified goals). This quantification happens at the meta-level. In the goals, `X` appears as a meta-variable (for an *exists* node) or an arbitrary constant (for a *forall* node) as it does in any other part of the label (e.g. the Method or the Tactic) which use it. It is also treated uniformly in all the children of the node since they fall within the scope of the meta-level quantifier. The basic plan constructors are therefore *and* nodes, *forall* nodes and *exists* nodes.

You'll also notice in the figure a two step process by which a goal produces two subgoals as its child and then these two subgoals are split into two child nodes as a separate step. This no doubt seems a bit excessive. This arises because of the large

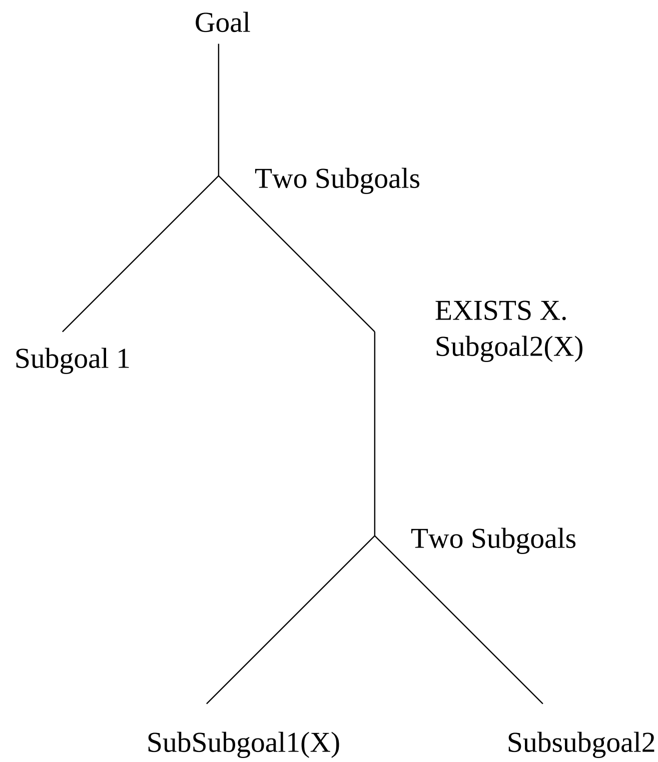

Figure 5.1: A Proof Plan

amount of information on the plan node labels. When a method is applied to a goal and it produces two subgoals the method writer is only interested in specifying what those subgoals are *not* in working out all the details of how those goals should be labelled. He therefore uses the `**` constructor to join the two goals together. For instance the `and_i` method in the constructive logic theory is as follows:

```
atomic constructive_logic and_i
           (seqGoal (H >>> (app and (tuple [A, B]))))
           true
           true
           ((seqGoal (H >>> A)) ** (seqGoal (H >>> B)))
           notacticyet.
```

This represent the sequent calculus rule:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \tag{5.1}$$

i.e. to prove $A \wedge B$ you need to prove both A and B. `**` is used to indicate this conjunction of goals. The next node in the proof plan of this is therefore labelled with the goal expression $\Gamma \vdash A{*}{*}\Gamma \vdash B$ which is then split into two with proper labelling for the other parts of the plan added at the next step. This is more than a little inelegant and other presentations have been tried - e.g. getting method application and splitting to happen at once. It is unclear which version is to be preferred.

5.3 The Partial Data Structure Planner

5.3.1 Basic Planning algorithm

The basic building blocks of the planning algorithm are two predicates one for planning one step of a plan - i.e. taking a goal, applying a method to it and producing a subgoal, and one for recursively repeating this process to produce a plan. After each single step a new agenda of open plan nodes is created and this determines which goal is attempted next by the planner. A record is also kept of the next *action* to be attempted. The default action is to add new nodes onto an agenda as a result of method application, but in certain situations - where designated methods have failed the action is an instruction to attempt to apply a critic to the node.

```
PLAN Agenda Action
      If the Agenda is empty succeed.
PLAN Agenda Action
      PLAN_ONE_STEP by applying a method or a critic to the
first node on the Agenda (as specified by the Action).  Get a
NewAction as a result.
      CREATE_AGENDA a NewAgenda using the results of PLAN_ONE_STEP
      PLAN NewAgenda NewAction.
```

`PLAN_ONE_STEP` plans one step of a plan by expanding the methodical definition to get a first method and applying it at a node to generate a new node. Similarly if the instruction is to apply a critic `PLAN_ONE_STEP` does this and modifies the existing plan accordingly.

`CREATE_AGENDA` determines whether planning will proceed depth-first, breadth first or by some other more sophisticated process. By default planning is depth-first but the predicates described take an *agenda predicate* as an argument and it is this predicate that is used to create the new agenda. This allows users to customise the search process.

It is during `CREATE_AGENDA` that any user intervention in terms of selecting the next node for planning or forcing backtracking can occur.

Applying Methods

The `PLAN_ONE_STEP` algorithm essentially applies a method to an open node in the plan and outputs a new plan. The step plan selects a node, gets a first method and the method continuation and then applies the method to the node to generate child nodes.

Recall that methods can be either *atomic* or *compound*. An atomic method has the traditional five slots (input goal, preconditions, postconditions, output goals, tactic). A compound method is an expression built up of chaining together methods with *methodicals*. So the step case of an induction is expressed as

```
compound induction step_case
        (then_meth
           (repeat_meth (wave_method outward R))
              (orelse_meth fertilisation_strong
                (then_meth
                   (try_meth (repeat_meth
                      (patch_meth (wave_method inout R1) wave_critic_strat)))
                   (try_meth fertilise)
                )
```

```
            )
        )
        _
        true.
```

NB. This is a simplified version of the `step_case` method which is in actuality more than a little baroque in order to cope with a number of special cases. This method instructs the planner to ripple outwards then attempt fertilisation. If that fails it ripples inwards and tries fertilising again. The `patch_meth` methodical around the ripple in method says that if this method fails the critic `wave_critic_strat` should be applied to the node.

Julian and Alan S. have shown in BBNote 1349 that all these methodical expression can be transformed to give the first method to be applied and then a *method continuation*, a new methodical expression indicating how to deal with the rest of the expression. For example the first method to be applied in `then_meth M1 M2` is the first method of `M1` (found by recursing through the expression) and the continuation is `then_meth M1Cont M2` where `M1Cont` is the continuation of `M1` found by recursing through the expression. This does not concern itself with whether a method within this expression is compound or atomic, e.g. `fertilise` above is compound but will be returned as the first method of the expression once `post_ripple` has been applied. The sequence of methods and methodical expressions leading up to the "top method" applied to a planning problem is referred to as the *method waterfall*.

`APPLY_METHOD`, used after the methodical expression has been transformed into a first method and a continuation, distinguishes between the atomic and compound methods and treats them differently in method application.

```
PLAN_ONE_STEP Agenda Action
     If Action says to apply a method
     Pick the first node on the Agenda and find the Goal and
Methodical Expression
     TRANSFORM The Methodical expression into a First Method and a
Method Continuation as specified in BBNOTE 1349
     APPLY the First Method to the Goal to get some Subgoals,
     ADD a new node to the plan of the the subgoal annotated with the
Method Continuation as the Methodical Expression.

APPLY Method to Goal return the Subgoal
     Method is atomic and its slots are:
              Goal Preconditions Postconditions Subgoal
     Preconditions succeed
     Postconditions succeed
APPLY_METHOD Method to Goal return the Subgoal
     Method is compound with Preconditions and a MethodExpansion
     Preconditions succeed
     The Subgoal is the same as the Goal but with the Methodical
Expression being (then_meth MethodExpansion MethodContinuation)
```

`APPLY` has special cases for splitting up compound goals and moving goal quantification to plan quantification.

Applying Critics

I also mentioned that `PLAN_ONE_STEP` deals with critics.

Like methods, critics are expressed using a waterfall of compound and atomic critics. So the `wave_critic_strat` mentioned above is a compound critic:

```
compound_critic wave wave_critic_strat
        (then_crit (pop_critic FailAd)
        (then_crit (subplan_crit FailAd (then_crit (wave_failure Failure Rule)
                                         open_node))
                   (wave_patch Failure FailAd Rule))).
```

This instructs the planner to find the address of the node at which the plan failed `pop_critic`, then at that address (`subplan_crit`) analyse the cause of failure (`wave_failure`) and patch the plan accordingly (`wave_patch`).

The critic waterfall is worked through in a very similar way to the method waterfall. Consider the situation where a rewrite has failed because the condition, C, on the rewrite $C \rightarrow L \Leftarrow\!\!:\!\!\Rightarrow R$ can not be shown to be true. One patch is to create two new subgoals, one in which C is true and one in which $\neg C$ is true in the hypotheses lists and attempt to prove both. The instantiation of `wave_patch` in this case is

```
compound_critic wave (wave_patch (failed_cond Rule) Ad Rule)
        (subplan_crit Ad
           (continue_crit (m\ (then_meth (wave_case_split Rule)
                (map_meth m))))).
```

This tells the planner to continue planning using a new method `wave_case_split` that does not appear in the usual waterfall. This method performs the casesplit dictated by the rule. It then maps the existing methodical continuation at that plan node, `m`, over all the subgoals produced by the new method.

Atomic critics have slots for the address at which they are applied, the current agenda, their preconditions and effects, any changes they make to the plan and the new agenda. So, for instance the built-in critic `roll_back_to_crit` rolls back the plan to the last occurrence of a specified method. To do this its preconditions examine the current plan for the nearest address to the input address at which the method occurred. It then calculates all the nodes that appear after that method and returns a list of these marked for deletion. It also removes them from the agenda and puts the node where the method was applied back on the agenda marking it as an open node.

Usually atomic critics examine preconditions of the failed method. For instance the `wave_failure` critic shown below:

```
atomic_critic wave (wave_failure Failure Rule)
        Address
        Agenda
        (get_preconditions Address Pre,
         failed_pre Pre Failed,
         wave_failed Pre Failed Failure)
        true
        nil
        Agenda.
```

In this case it makes no changes to the plan and leaves the agenda as it was. However it finds the preconditions, `Pre` at the input address and analyses them to get a diagnosis of the sort of failure - e.g. `failed_cond` discussed above.

In order for it to be possible to analyse preconditions in this way the planner annotates plan nodes with precondition information (along with method and method continuation etc. etc.). Usually the preconditions are simply marked as successful, but if a method has failed to apply and it occurs within a `patch_meth` expression then its preconditions are evaluated one at a time. Those that succeed are marked `success` and those that fail are marked `failed`. These can then be analysed.

Chapter 6

Creating Theories as Library Files or λProlog files

6.1 Introduction

At present, the best way to add your own goals, methods, rewrite rules, etc. to λ*Clam* is to compile a theory file into its bytecode. This chapter aims to overview the main components of a theory file and discuss how to compile and debug it.

6.2 Theory File Components

6.2.1 Theory Name

In most cases it is advisable to name your theory by introducing a constant of type `theory` in your theory's signature file.

6.2.2 Syntax Constants

NOTE: We have a utility for writing syntax in a more natural format than that required by λ*Clam*'s internals. See chapter 8.

Syntax constants can be any function symbols or constants that you wish to include in your theory. These should be declared in the signature file of your theory (as items of type `osyn`). You should also include a clause of the predicate `has_otype` in the module file for your theory. `has_otype` takes as arguments the name of the theory, the name of the constant and the type. New types should be declared using `is_otype`. `is_otype` takes the name of the theory, the new type and two lists of constructors one for "base" elements of the type and one for "constructor" functions. Types are given the type `universe`. For example the following is taken from the arithmetic theory and declares the type `nat` with the constructors `zero` and `s` and the function `plus`.

```
is_otype arithmetic nat [zero] [s].

has_otype arithmetic zero nat.
has_otype arithmetic s (nat arrow nat).
has_otype arithmetic plus ((tuple_type [nat, nat]) arrow nat).
```

The signature file for the theory contains the declarations

```
type    nat     osyn.

type    zero    osyn.
type    s       osyn.
type    plus    osyn.
```

6.2.3 Goals

New goals are declared using the predicate `top_goal`. This takes four arguments: the theory name; the name of the goal; a list of hypotheses; and finally a goal conclusion. So, for example, again from arithmetic, we have in the signature

```
type assp           query.
```

and in the module

```
top_goal arithmetic assp []
    (app forall (tuple [nat,
     (abs z\ (app forall (tuple [nat,
      (abs y\ (app forall (tuple [nat,
       (abs x\ (app eq (tuple [
        (app plus (tuple [
         (app plus (tuple [x, y])), z])),
        (app plus (tuple [x,
         (app plus (tuple [y, z]))]))])))]))))]))))])).
```

6.2.4 Atomic Methods

Atomic methods are declared by writing clauses for the predicate:
`atomic:  theory -> meth -> goal -> o -> o -> goal -> tactic -> o.`
It takes as arguments the name of the theory file, the name of the method, the input goal, the preconditions and postconditions (written in λProlog) the output goal, and the corresponding tactic (at present this is always the placeholder `notacticyet`). Most internal goals will be sequent goals created using the `seqGoal` constructor: `seqGoal (H >>> G)`.

If the method will instantiate some variable (input as part of the method name) (e.g. rewriting methods frequently instantiate a variable with the name of the rewrite rule used) then an additional predicate `change_method_vars` must also be used. This simply relates the method name to itself but without enforcing unification of any variables that will need instantiating. This predicate is used internally to ensure that the variable is not accidentally instantiated for subsequent method calls during planning.

For instance, the wave method is declared in the wave theory files. In the signature we have:

```
type wave_method rewrite_rule ->  meth.
```

and the module itself contains the method definition:

```
atomic wave (wave_method Rule)
        (ripGoal (Hyps >>> Goal) Skel E1)
        (ripple_rewrite Rule Skel (Goal @@ E1) (NewGoal @@ E2) Cond,
         (embeds_list E2 Skel NewGoal,
         (measure_check red Dir E2 E1 nil Embedding,
         (for_all Embedding sinkable,
```

```
         (trivial_condition Cond Hyps)))))
        true
        (ripGoal (Hyps >>> NewGoal) Skel Embedding)
        notacticyet.
change_method_vars (wave_method Rule) (wave_method _) :- !.
```

6.2.5 Compound Methods

Compound methods are declared by writing clauses for the predicate:
`compound: theory -> meth -> meth -> (list int) -> o -> o.`
It takes as arguments the name of the theory , the name of the method, the methodical expression which makes up the method, the address where the method was applied (typically blank) and preconditions (in λProlog).

If the method will instantiate some variable then, as with atomic methods, the predicate `change_method_vars` will have to be extended.

So, for instance, a simplfied version of the step case method[1] in the induction theory could be declared in the signature file as

```
type   step_case  meth.
```

and in the module as

```
compound induction step_case
        ( cut_meth
           ( then_meth set_up_ripple
               (then_meth (repeat_meth (wave_method R))
                  (then_meth post_ripple
                        (try_meth fertilise
                         )
                  )
               )
           )
         )
        _
        true.
```

Compound methods use methodicals to build up more complex proof strategies from other methods. The methodicals available in *λClam* are listed below.

Methodical	Type
id_meth	`meth`
triv_meth	`meth`
orelse_meth	`meth -> meth -> meth`
cond_meth	`(goal -> o) -> meth -> meth -> meth`
try_meth	`meth -> meth`
repeat_meth	`meth -> meth`
then_meth	`meth -> meth`
then_meths	`meth -> meth -> meth`
map_meth	`meth -> meth`
complete_meth	`meth -> meth`
cut_meth	`meth -> meth`
pair_meth	`meth -> meth -> meth`
best_meth	`list meth -> meth`

[1]The actual `step_case` method used in *λClam* is more complex than this in order to deal with a number of special cases.

A brief description of the methodicals follows:

id_meth Automatically succeeds. This is used in the implementation of methods such as `try_meth` but can also be useful when dealing with more than one subgoal. For instance the `induction_meth` returns two subgoals a step case and a base case. The step case is given to the `step_case` method while the base case is to be treated like most other goals by having the default `induction_top_meth` applied to it. The method, `ind_strat`, does this as follows:

```
compound induction (ind_strat IndType)
      ( then_meths (induction_meth IndType Scheme Subst)
                   (pair_meth (map_meth (step_case IndType)) (map_meth id_meth)) )
        _
        true.
```

This applies the `induction_meth` and then applies `step_case` method to the first subgoal (the step case) and `id_meth` to the second subgoal (the base case). Since `id_meth` does nothing the result is subgoal is unchanged an is passed as a result of `ind_strat` to the top level repeat.

triv_meth Only succeeds if the goal is `trueGoal`. Is used in the implementation of `complete_meth`.

orelse_meth Succeeds if one of the two methods succeeds.

cond_meth If the condition evaluates to true then it tries the first method otherwise it tries the second.

try_meth Attempts the listed method but does not fail if the method fails.

repeat_meth Attempt the method as many times as possible. The method must succeed once.

then_meth Tries the first method and then apples the second method to the result of the first.

then_meths In the current implementation this is identical to `then_meth` but the intention is that it should provide finer control over multiple subgoals while `then_meth` should be more brute force.

map_meth Maps the method does a list of subgoals.

complete_meth Succeeds only if the output of the method is `trueGoal`.

cut_meth Indicates that if the method succeeds then if the proof subsequently fails alternative successes of the method should not be considered. This is not implemented in the current version of λ*Clam*. This is intended to be like prolog cut.

pair_meth Applies to two subgoals and applies the first method to the first subgoal and the second method to the second subgoal.

best_meth Takes a list of methods (each of which may be either atomic or compound), computes a heuristic score for each method using the current subgoal, and applies the method with the lowest score to the subgoal. If that fails, the method with the next lowest score is attempted, and so on. The heuristic scoring predicate is `score_method: goal -> meth -> int -> o`. To write a scoring function for a method, one just writes a clause for this predicate along with the method definition. The file theories/best_first_test.mod contains an example best-first top method and some fixed method scores (which the user should feel free to experiment with).

Methodicals can thus be used to create complex strategies tailored to attempting specific sorts of problem. More and more complex strategies can be created out of the blocks of simpler strategies. It is such a strategy that is applied to a goal as its "top method".

6.3 Critics

The critics code is still in development and has yet to be properly documented. The modules `core_theories/wave_critics.mod` and `theories/ireland_critics.mod` contain the currently implemented critics and we advise the interested reader to consult these files.

Critics are invoked when a method within a `patch_meth` fails. `patch_meth` takes two arguments, a method and a critic, and applies the critic to the goal when the method fails. As a side effect it also prints out the failing methods preconditions — this can be used for debugging by replacing the appearance of the method in your strategy with `patch_meth Method id_crit`.

There are four built-in basic critics that can be used to build up critic strategies:

```
type    pop_critic      (list int) -> crit.
type    open_node       crit.
type    continue_crit   (meth -> meth) -> crit.
type    roll_back_to_crit       meth -> (list int) -> crit.
```

`pop_critic` returns the address of the top node on the agenda. `open_node` makes the node to which the critic is currently applying open and `continue_crit` transforms the methodical expression at a node as specified. `roll_back_to_crit` deletes all nodes in the plan after the last occurence of the specified method and places the node at which the method was applied at the top of the agenda.

Compound critics are glued together using *criticals*, which are analogous to methodicals for lguing together methods. λ*Clam*'s criticals are shown in the following table.

Critical	Type
fail_crit	`crit`
id_crit	`crit`
orelse_crit	`crit -> crit -> crit`
cond_crit	`(plan_state -> o) -> crit -> crit -> crit.`
try_crit	`crit -> crit.`
repeat_crit	`crit -> crit.`
then_crit	`crit -> crit -> crit.`
subplan_crit	`(list int) -> crit -> crit.`
some_crit	`(_ -> crit) -> crit.`

Most of these criticals work analogously to the methodical with the associated name. However, `subplan_crit` and `some_crit` require explanation. `subplan_crit` apples the critic to a particular address in the plan – these need not be the address at which a method failed. `some_crit` takes a function which produces a critic as its output. This is then applied to a meta-variable and the resulting critic applied – *to be honest I'm not clear what function this is supposed to serve - Louise*. There is more discussion of critics in chapter 5.

WARNING: in Teyjus, the comma used to sequence predicates associates to the left. The way the critics planner works means that you will have to explicitly bracket your method preconditions in order to get the planner to consider them one at a time (as shown in the wave method above).

6.4 Compiling your Theory

Say your theory is called `mytheory` and is in the files `mytheory.sig` and `mytheory.mod`.

If your theory depends on no other theories add

`accum_sig lclam.` At the top of your theory's `.sig` file and

`accumulate lclam.` At the top of your theory's `.mod` file.

It your theory depends upon other theories then you must should replace the `lclam` accumulation headers with the name of these theories. You should try and minimise the number of theories accumulated into your modules and you should make sure that no theory can be accumulated twice - e.g. by being accumulated into your module and into one of the other theories accumulated into the module.

The `lclam` command runs λProlog executable called `test.lp` in the `planner/` directory. If you want to incorporate your theory file into those used by `test.lp` then you will need to add accumulation commands for your theory to `test.sig` and `test.mod`, and similarly if you wish to modify an existing theory to use predicates in your theory.

Otherwise you may run your theory without all the other stuff which is included in test. First you must compile your theory into an executable.

If your theory is in the `theories/` subdirectory of the distribution. Type `make mytheory.lp` in the `theories/` subdirectory. Otherwise you need to include the location of your theory on your `TJPATH`, and then type `tjcc mytheory.lp` in that directory.

The command:

`$LCLAM_HOME/bin/lclam mytheory`

will then act just like `lclam` except that the only methods etc. available will be those in your theory.

6.4.1 If your theory will not Compile

. The first thing to do is to check for warnings and error messages in the output generated by the compiler. Many such errors will look like the following:

```
./mytheory.mod:57.1: Error: clash in operator and operand type
```

This tells you the name of the module in which the error occured, `mytheory` in this case, the line in that module which is causing the error, 57 in this case, and the character in the line, 1 in this case. It will then give some details of the error, usually these will be type checking failures and the compiler will give both the expected type and the existing type.

It is always worth checking for warnings even if compilation has successfully completed. For instance the compiler warns if it meets an undeclared constant (i.e. one for which no type has been given). This is often a sign of mispelling somewhere. The compiler does not warn about single occurences of variable names in predicate clauses – unlike most prolog compilers which do warn of this.

If the compiler is no longer generating errors or warning and your module still will not compile or run there are a number of possible causes. In the case of failing to compile common messages are `Access to unmapped memory!`, `Signal 10` or `Signal 6`. All of these errors can be fixed by the addition or removal (seemingly at random) of print statements or by commenting in or out extra predicates which are not used in the code. These error messages are caused by a bug in the Teyjus compiler which hopefully will become less and less frequent as Teyjus becomes more stable.

If your program compiles but fails to run with the message of the form `Loader error: X overflow` then it has exceeded built-in limits in the Teyjus simulator. It is possible to extend these limits in the Teyjus source and then recompile, contact `dream@dream.dai.ed.ac.uk` for details.

6.5 Some Useful Built-In Predicates

There are a number of standard list and map predicate you may want to use in developing λ*Clam* theories. You can find this by looking at the files in the `utils/` subdirectory of the λ*Clam* distribution.

Other predicate you may find useful are:

`obj_atom` indexobj_atom

```
type obj_atom osyn -> o.
```

Synopsis

Succeeds if the input is an atomic piece of syntax (i.e. not a tuple, function application, λ-abstraction etc.).

`subterm_of`

```
type subterm_of osyn -> osyn -> (list int) -> o.
```

Synopsis

Succeeds if the first argument is a subterm of the second, appearing at the position in the term indicated by the third argument.

`has_otype_`

```
type has_otype_ osyn -> osyn -> o.
```

Synopsis

Returns the object type of the syntax given as its second argument.

`env_otype`

```
type env_otype osyn -> (list osyn) -> osyn -> o.
```

Synopsis

Mostly used to return (as its third argument) the object type of the syntax given as its first argument, given an environment (list of hypotheses, some of which may contain typing information) given as its second argument.

`replace_at`

```
type replace_at osyn -> osyn -> osyn -> (list int) -> o.
```

Synopsis

Third argument is the first argument with the subterm indicated by the fourth argument replaced by that indicated by the second argument.

---

replace_in_hyps

```
type replace_in_hyps (list osyn) -> osyn -> (list osyn) -> (list osyn)
-> o.
```

Synopsis

Replaces one hypothesis with a list of hypotheses. The first argument is the original list of hypotheses, the second is the hypothesis to be replaced, the third the list of hypotheses to appear in its stead and the forth the resulting hypothesis list. This predicate appears in the `logic_eq_constants` theory and so this will need to be accumulated to use this predicate.

---

printstdout

```
type printstdout A -> o.
```

Synopsis

Prints its argument to standard out followed by a new line.

---

testloop

```
type testloop o -> o.
```

Synopsis

Allows the interactions at the command line to be simulated.

Example

The following piece of code provides a predicate which will set up a particular contect (e.g. it uses all the induction schemes in arithmetic and lists) and then plans a goal in that context. This saves the user typing in the list of instructions each time they want to test their methods.

```
benchmark_plan test Meth Query :-
        testloop (add_theory_to_induction_scheme_list arithmetic,
        (add_theory_to_induction_scheme_list objlists,
        (set_sym_eval_list [idty, s_functional, cons_functional, leq1, leq2, leq_ss, assAnd1,
        (add_theory_defs_to_sym_eval_list arithmetic,
        (add_theory_defs_to_sym_eval_list objlists,
        (set_wave_rule_to_sym_eval,
        pds_plan Meth Query)))))).
```

NOTE: bracketing of commands has had to be made explicit for this to work. It is possible to write your own clauses for `benchmark_plan` the first argument is the name of a theory and so can be used to determine precisely which context is wanted by the user. Obviously it also possible to write completely new predicates that involve `testloop` to perform other such tasks. Examples of `benchmark_plan` can be seen in the `list_benchmarks` theory, the `ordinal` theory and a number of others.

6.5.1 List and Mapping Utilities

A number of utility predicates for manipulating lists and mapping predicates down lists have been implemented in λ*Clam* on top of those already existing in Teyjus. These can be found in the files `lclam_list` and `lclam_map` in the `util/` subdirectory of the distribution.

6.6 Debugging Theories

It is described above how `patch_meth` can be used to provide information about the failure of a methods preconditions. There is also a predicate that can be embedded into the code to print out information on backtracking:

`on_backtracking`

`type on_backtracking o -> o.`

Synopsis

This can be supplied with a λProlog instruction which it will only execute when backtracked over. For example, it is used in the partial data structure planner to print information when a method is being backtracked over, using the call

```
on_backtracking
        (pprint_string "backtracking over",
        pprint_name Method),
```

Chapter 7

The Built-in Theories of λ*Clam*

7.1 Introduction

λ*Clam* is best known as a inductive proof planner. The object of this chapter is to give an overview of the implementation of methods for induction in λ*Clam*. It also discusses briefly some of the other built-in theory files.

7.2 The Proof Strategy for Induction and Rippling

The proof strategy for induction can be seen in figure 7.1. This is implemented as a selection of atomic and compound methods. The diagram shows a top level repeat which attempts a disjunction of methods. These include basic tautology checking, generalisation of common subterms and also symbolic evaluation and the induction strategy (`ind_strat`). Within the induction strategy, the induction method performs ripple analysis [5] to choose an induction scheme (from a selection specified in λ*Clam*'s theories) and produces subgoals for base and step cases. The base cases are passed out to the repeat of the top level strategy. The step cases are annotated with *skeletons* and *embeddings* (described below) and then the wave method is repeatedly applied to them followed by fertilisation (exploitation of the induction hypothesis). Annotations are then removed. The methods (`set_up_ripple` and `post_ripple`) which place and remove annotations are omitted from the diagram. The results are then passed out to the top level strategy again. The process terminates when all subgoals have been reduced to *true* (or *false* in the case of failure).

We will discuss the `wave_method` in more detail since it was a new formulation of this method we wished to investigate. The wave method embodies the rippling heuristic. Rippling was first introduced in [6]. We use the theory as presented by Smaill & Green [19] who proposed a version that naturally coped with higher-order features. Rippling steps apply rewrite rules to a target term which is associated with a skeleton and an embedding that relates the skeleton to the target term (e.g. rippling rewrites an induction conclusion which has an induction hypothesis embedded in it). In the present context, we make use of higher order rewriting, in the style of [10]. After rewriting a new embedding of the skeleton into the rewritten term is calculated. There is a measure on embeddings and any rewriting step must reduce this *embedding measure* (written as $<_\mu$). This is a generalisation of the original version of rippling that used annotated *wave rules* to rewrite annotated

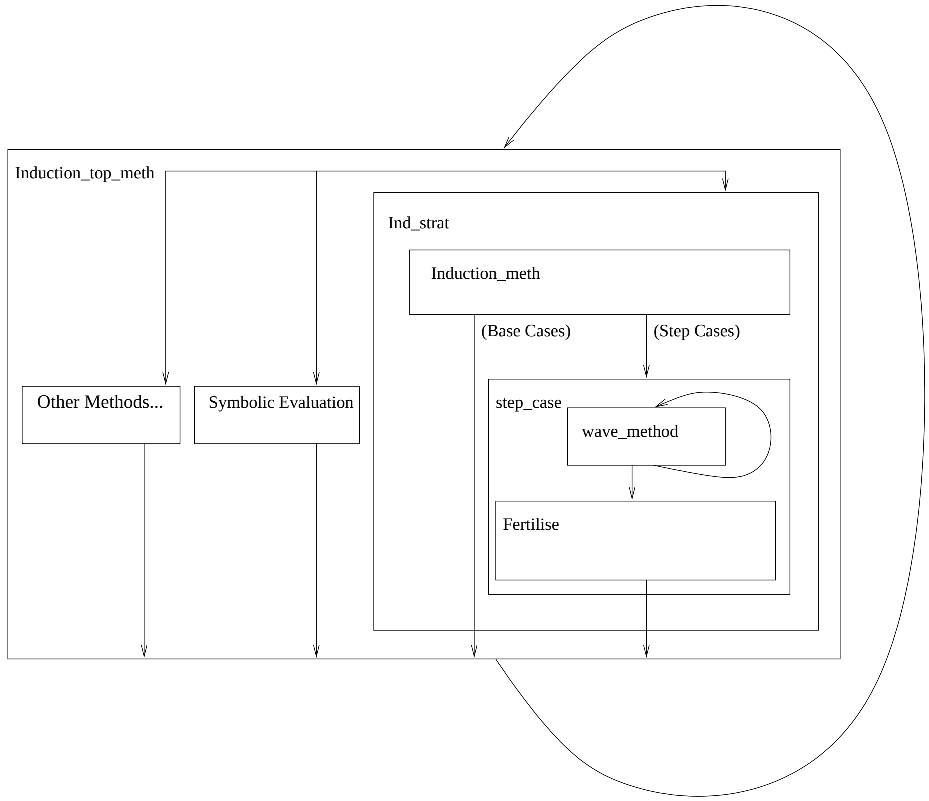

Figure 7.1: The Proof Strategy for Induction

terms.

Rippling is terminating [1]. Rippling either moves differences outwards in the term structure so that they can be cancelled away or inwards so that the differences surround a universally quantified variable (or *sink*). If it is possible to move differences inwards in this way the embedding is said to be *sinkable*. The measure on embeddings allows differences that are being moved outwards to be moved inwards but not vice versa – this is at the heart of the guarantee of termination.

The `wave_method` method has five preconditions. It finds a rewrite rule that rewrites the goal. It then checks that there is still an embedding of the skeleton into the rewritten goal and that this new embedding is less, according to the embedding measure, than the original embedding. It checks that the embedding is sinkable and that any conditions for the application rule are trivial. This is shown in figure 7.2. The method will backtrack in order to try to satisfy all requirements, and if it is successful returns a new goal.

7.3 Embeddings

We review here the notion of embeddings from Smaill & Green [19]. These provide the higher-order framework for rippling used in this paper.

Embeddings are described by a tree data structure. Embedding trees describe how a *skeleton* embeds in a term, called the *erasure*. The nodes in an embedding tree can be viewed as labels on the nodes in the term tree of the skeleton. These labels contain addresses and directions. The directions are used during rippling as outlined above. The addresses are the addresses of nodes in the term tree of the erasure which

Input

$$\textit{ripple\_goal}(H \vdash G, S, E).$$

where *ripple_goal* is a triple of a sequent, a skeleton, S and an embedding, E, of that skeleton into the goal, G.

Preconditions

1. The conditional rewrite rule *Rule*, $Cond \Rightarrow X \leftarrow:\rightarrow Y$ instantiated with some substitution σ, applies to G and rewrites it to G'.
2. There is an embedding E' that embeds S in G'.
3. $E' <_\mu E$.
4. $\sigma(Cond) = True$ or $\sigma(Cond) \in H$.
5. E' is sinkable.

Output

$$\textit{ripple\_goal}(H \vdash G', S, E')$$

Figure 7.2: The Wave Method

correspond to that node in the skeleton. Term trees represent function application and λ-abstraction explicitly as nodes with constant and variables symbols appearing only at the leaves of the tree. Our implementation also contains tuple nodes for lists of arguments to functions but these are not necessary to the theory. Embeddings do not annotate λ-abstraction nodes. Where an embedding matches a variable in the skeleton to one in the erasure it indicates that they are α-convertible. It is the ability to coherently handle λ-abstractions which was particularly valuable in this experiment. The ability to handle difference occuring within functions as well as the arguments to functions is also an extension of the previous calculus.

Example 7.1 *Consider embedding the term $\lambda x.\ f(x)$ into the term $\lambda y.\ \lambda x.\ (f(y)+x)$. We do this as in figure 7.3. The two terms are shown as trees with branches represented by solid lines. The address of each node is given (λ-abstraction nodes do not carry addresses). The embedding appears between them as an embedding tree with dashed lines – the address label of the nodes is also shown. The dotted arrows illustrate how the embedding tree links the two terms.*

The embedding tree for this is (node [1, 2] [(leaf [1, 1, 1]) (leaf [2, 1, 2])])). This states that the function application at the top of $f(x)$ matches with the node at address [1, 2] of $f(y)+x$ (i.e. the application involving $+$ has been bypassed), that the function symbol f matches the sub term at [2,1,1] (i.e. f) and x matches y (i.e. the bound variables can be consistently renamed in either (or both) terms so that they are identical).

The annotations originally used in rippling are still useful for presentation. Annotations consist of contexts (expressions with holes) indicated by a wave front (box) with a directional arrow. The holes in the context are wave holes (i.e. they are filled with an expression which is underlined). The skeleton is everything that appears

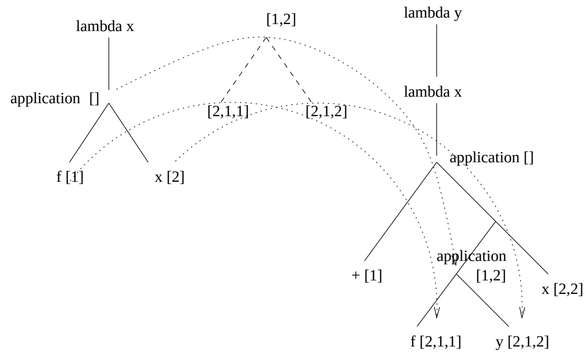

Figure 7.3: An Embedding

outside wave fronts, or in wave holes. So the above embedding can be presented as

$$\lambda x.\ \boxed{\lambda y.\ \underline{f(y)} + x}^{\uparrow}\ .$$

NB. It is important to remember that the annotations do not actually exist in the implementation which records instead the skeleton and embedding[1]*. In some cases it is less easy to use the traditional annotations to represent embeddings, particularly when dealing with λ-abstrations which are ignored by the embedding trees. You can see that in the above the bound variables in the skeleton are assumed to have been appropriately renamed already. These annotations are just a presentational convenience.*

7.4 Logics

You will note that the proof strategy for induction calls the `taut` method for tautology checking. This may vary depending upon whether constructive or classical logic (for instance) is being used. As a result the basic "build" of the induction theory in λ*Clam* does not specify how the `taut` method or logical constants such as `forall` or `and` should be treated by the system. It simply asserts that they exist. Additional files, of which the `constructive_logic` theory can then be created to provide the necessary details.

7.5 Arithmetic, Natural Numbers and Lists

There are a number of built-in theories dealing with natural number and list theorems. Standard functions and goals can be found in the `arithmetic` theory and the `objlists` theory. Further functions can be found in `list_benchmarks`, `map_benchmarks` and the `clam_corpus` theories (there are 3 clam corpus theory files).

7.6 Mutual induction

This module was written for the MFOTL group based at Liverpool. The method `alexei_meth` in `theories/mutual_induction.mod` implements the following proof

[1]In fact λ*Clam* maintains a list of possible embeddings during rippling since there may be more than one way to embed the induction hypotheses into the conclusion. For convenience we assume here there is only one.

rule. It is a variant of a mutual induction rule suitable for treatment of arithmetical translations of temporal formulae:

$$\frac{\begin{array}{l}\forall y \forall \bar{x} A_1(y, \bar{x}) \Rightarrow B_1(s(y), \bar{x}) \\ \forall y \forall \bar{x} A_2(y, \bar{x}) \Rightarrow B_2(s(y), \bar{x}) \\ \ldots \\ \forall y \forall \bar{x} A_n(y, \bar{x}) \Rightarrow B_n(s(y), \bar{x})\end{array}}{\forall u.u > t \Rightarrow \varphi(u)}$$

with the side conditions:

$$\exists \bar{x}(A_1(t, \bar{x}) \vee A_2(t, \bar{x}) \vee \ldots \vee A_n(t, \bar{x}))$$

$$\forall y \forall \bar{x}(B_i(y, \bar{x}) \Rightarrow \varphi(y)), i = 1, \ldots, n$$

$$\forall y \forall \bar{x} \exists \bar{z}(B_i(y, \bar{x}) \Rightarrow \bigvee_{j=1\ldots n} A_j(y, \bar{z})), i = 1 \ldots, n$$

where A_i and B_i are arbitrary first-order formulae, t is an arbitrary arithmetical term, s is the successor function symbol, $\Rightarrow$ is implication, $\bar{x}$ is a sequence of variables, and y is a (numerical) variable of type `nat`. n is not fixed, so we can deal with arbitrary lists of premises in the rule.
Since the rule is generally used in backward reasoning, one can present its application as follows:

Given a goal sequent $H \vdash \psi$ with ψ being of the form of the conclusion of the above rule, the method generates some subset H' of H matching the premises of the rule and generates the associated side conditions as new subgoals. These subgoals are then resolved by application of some other methods (for first-order reasoning).

At present the method simply chooses the maximal matching subset of hypotheses. Probably some optimization for iteration over subsets will be needed later.
Example:

Given:

$$\begin{array}{l}\forall x, y(A(y, x) \Rightarrow B(s(y), x)) \\ \forall x, y(C(y, x) \Rightarrow D(s(y), x)) \\ \forall x(A(0, x) \vee C(0, x)) \\ \forall x, y(B(y, x) \Rightarrow C(y, x)) \\ \forall x, y(D(y, x) \Rightarrow (A(y, x) \vee C(y, x)))\end{array}$$

Prove:

$$\forall y > 0 \exists x(B(y, x) \vee D(y, x))$$

Side conditions to prove (taking the set of first two hypotheses and the conclusion):

$$\exists x(A(0, x) \vee C(0, x))$$

$$\forall y \forall x(B(y, x) \Rightarrow \exists z(B(y, z) \vee D(y, z)))$$

$$\forall y \forall x(D(y, x) \Rightarrow \exists z(B(y, z) \vee D(y, z)))$$

$$\forall y \forall x \exists z(B(y, x) \Rightarrow A(y, z) \vee C(y, z))$$

$$\forall y \forall x \exists z(D(y, x) \Rightarrow A(y, z) \vee C(y, z))$$

Chapter 8

Compiling Theories to λ*Clam*

8.1 Introduction

This chapter describes the current state of the λ*Clam* theory compiler. It develops the idea that a logic can be presented in a high-level declarative style, and then *compiled* down into a more procedural implementation-specific language.

The reasons for wanting to do this are:

- the user-unfriendliness of the current λ*Clam* format
- to avoid errors that can arise when entering axioms and inferences as low-level rewrites and methods
- (ultimately) to aim for a logic-independent planner by factoring out logic-specific code
- since a high-level representation of logics and theories, rather than one which is hard-wired into a low-level implementation, is easier to communicate.

In our case, of course, the procedural language is λ*Clam*'s format for the paraphernalia of proof planning (expressed in HOAS) in λProlog; for declarative language, we simply make up our own user-friendly syntax for declarations, inference rules, and the like.

The compiler has two modes of use: online, using the `add_theory` command during λ*Clam*'s normal command loop, and offline, generating a separate λ*Clam* theory file. The offline mode is currently deprecated but is described here as it helps understand the general compilation process and may prove useful in the future.

8.2 Online Mode

The command

`add_theory` *"theory-file"*

will read in the file *theory-file*, check its well-formedness, and generate the appropriate user-defined constants.

For example, the file `simpletheory`:

```
Theory Nsa.
Use Logic arithmetic.
```

```
type real.
type hyperreal.
$
```

gives the following results:

```
lclam:
add_theory "simpletheory".
Trying ...Scanning for tokens...
Tokenizer stopped after line 6
Parsing...
Successfully parsed
Checking well-formedness...
Type real
Type hyperreal
Theory is well-formed
Importing lclam declarations...
Successfully processed
Done
Done
lclam:
```

We can now use the query commands to check that the definitions have been made correctly.

```
lclam:
query_osyn (user_theory "nsa") (user_object "real") X.
default
user_theory "nsa"
real
universe
lclam:
```

A more complex example of a theory file can be seen in Section 8.3.5.

8.3 Offline Mode

We describe here how a user can use the files `envgrammar.{sig,mod}` and `postprocess.{sig,mod}` in the `compiler` directory to parse a theory file written in concrete syntax and generate a λ*Clam* theory file.

We first explain how to use the compiler, and in the following sections, explain the workings of the compiler in case the user wants to alter it.

The phases of such a logical compilation are much the same as in the compilation of programs: parsing of concrete into abstract syntax (Section 8.3.1), a well-formedness check (Section 8.3.2), and then code generation (Section 8.3.3). We give the grammar for theory files (Section 8.3.4), and examples of concrete syntax (Section 8.3.5) and the corresponding λ*Clam* (Section 8.3.6).

Using the Compiler

First, edit the `Makefile`, if necessary, to alter the paths appropriately.

Then, generate the parser by

```
make envparser
```

Next, do

```
make postprocess
```

If the theory file is `theoryfile`, then the command

```
parsefile "theoryfile" X.
```

will bind to `X` the abstract syntax that results from parsing this file. This might be useful to you if you just want access to the higher-order abstract syntax form of an expression.

Here, `theoryfile` is the file with concrete syntax which you can, of course, change and rename. Note that the underlying parser, λyacc, requires this file to terminate with a `$`.

The command

```
process_file "theoryfile"
```

will parse the file, check its well-formedness, and output the appropriate *λClam* files. The names of the files are determined by the line

```
Theory <name>.
```

in the theory file, which results in `<name>theory.{sig,mod}`.

Finally, you then have to fit these files in with your version of lambda clam (by setting the appropriate accumulates) and compile the whole system. Note that the compilation files themselves should not be compiled with *λClam*.

8.3.1 Parsing

The logic file is first parsed into abstract syntax. The parser is implemented using λyacc[1], a parser-generator written in λProlog, due to Chuck Liang. It generates bottom-up *shift-reduce* parsers, which can be less intuitive than top-down parsers, but are more efficient.

The user specifies their logic by writing various predicates and rules in a λProlog file, which accumulates the λyacc files.

1. Declare grammar symbols for the abstract syntax for each nonterminal.
2. Declare the terminals.
3. Give concrete syntax for the terminals using `printname`.
4. List the terminals, and nonterminals.
5. Set `ntnum` to the number of nonterminals plus 1.
6. Give the parse rules, with semantic actions to construct the abstract syntax.

   For example, rules giving the syntax for axiom declarations are

   ```
   rule ((syn_decl_gs D5) ==>
          [axiom_decl_gs S7 H1 Z7]) (D5 = axiom_decl S7 H1 Z7),

   rule ((axiom_decl_gs S5 H2 Z5) ==>
          [axiomt, axiom_name I6, qprop_gs Z3, periodt])
                                 (S5 = I6, Z5 = Z3, H2 = []),
   ```

[1] See `http://www.cs.hofstra.edu/~cscccl/parsergen/`.

The `rule` construct combines a production with a semantic action.

7. Give precedence rules.

8. Supply freshcopy clauses for non-nullary nonterminals.

This file is compiled, and then executed. Then, a generate parser command is called, which, assuming λyacc accepts the file, generates *another* file. This file is then compiled to give the parser, all of which takes about a minute in total.

8.3.2 Well-formedness

A consequence of the fact that the semantic actions attached to λyacc parse rules cannot contain implications (and that λProlog does not have assertions), is that well-formedness of the resulting abstract syntax has to be checked in a separate pass.

However, logically inspired manipulation of terms such as this is what λProlog does best. For example, well-formedness of existentials and abstractions is checked by

```
well_formed_assertion (app exists (tuple [T, abs P])) :-
        well_formed_type T,
        pi X \ (new_const X T) => well_formed_assertion (P (user_object X)).

well_typed_term (ty_abs T L) (T arrow U) :-
        well_formed_type T,
        pi X \ (new_const X T) => well_typed_term (L (user_object X)) U.
```

Since declarations are 'first-class' in λProlog, λProlog's context mechanism can be used for managing declarations.

```
well_formed_decls ((inference_decl S Hyps H2 C2)::Ds) :-
        well_formed_decl (inference_decl S Hyps H2 C2),
        ((new_inference S) => well_formed_decls Ds).

well_formed_decl (inference_decl S Hyps H2 C2) :-
        not (new_inference S),
        map_pred2 group_assertions Hyps Ass,
        all_pred well_formed_assertions Ass,
        ly_append H2 [C2] Ps2,
        well_formed_assertions Ps2,
        M is "Inference " ^ S ^ "\n",
        print M.
```

There are no error messages at present.

8.3.3 Generation of λ*Clam*

Once the theory file has been parsed into abstract syntax and checked for well-formedness, it is converted to λ*Clam* format and written to a file. Constructing the appropriate higher-order abstract syntax was done during parsing, so this stage is mainly concerned with abstracting over the variables, and converting propositions to rewrites and methods.

The syntax comprises a list of declarations, each of which is converted to the corresponding λ*Clam* entity.

% `decl2lclam` *+sig-file +mod-file +theory-name +logic-name +declaration*

```
type decl2lclam  out_stream -> out_stream -> string -> string -> syn_decl -> o.
```

A syntactic declaration is one of: type, typed constant, axiom, inference, conjecture, predicate symbol, definition. The interesting cases are axiom and inference rules. We convert axioms to rewrite rules, and inferences to methods.

An example of an axiom is

```
axiom ax1 [y:nat |- forall f:nat->nat. f = (x:nat \ (f y))].
```

This is parsed as

```
axiom_decl "ax1" (otype_of (user_object "y") nat :: nil)
   (app forall (tuple ((nat arrow nat) ::
         abs (W1\ app eq
          (tuple (W1 :: ty_abs nat (W2\ app W1 (user_object "y")) :: nil))) ::
 nil)))
```

and converted to the rewrite rule

```
axiom nsa ax11 ltor (trueP)
   (app forall
     (tuple ((nat arrow nat) ::
       abs (W1\ app eq (tuple (W1 :: ty_abs nat (W2\ app W1 _107597) :: nil)))
 :: nil)))
    (trueP).
```

For inference rules, the system generates preconditions to check for required antecedents in the conclusion, and to construct the antecedents of the hypotheses. The succedent of the conclusion is recorded in the form of the method input, and the succedents of the hypotheses appear in the output goal.

For example, the inference rule

```
inference and_l [G,A,B,G' |- P / G, A /\B |- P]
```

parses to abstract syntax

```
(inference_decl "and_l"
[(pair [user_object "G", user_object "A", user_object "B", user_object "G'"]
       (user_object "P"))]
[user_object "G", (app and (tuple [user_object "A", user_object "B"])),
user_object "G'"] (user_object "P"))
```

which is converted to the atomic method

```
atomic nsa and_l
        (seqGoal (H >>> (_9055)))
        (sublist (app and (tuple (_9072 :: _9088 :: nil)) :: nil) H,
        replace_in_hyps H (app and (tuple (_9072 :: _9088 :: nil)))
                             (_9072 :: _9088 :: nil) H1)
        true
        seqGoal (H1 >>> _9055)
        notacticyet.
```

The inference rule

```
inference and_r [G |- A ; G |- B / G |- A /\B]
```

parses to abstract syntax

```
(inference_decl "and_r"
[(pair [user_object "G"] (user_object "A")),
(pair [user_object "G"] (user_object "B"))]
[user_object "G"] (app and (tuple [(user_object "A"), (user_object "B")]))
)
```

and converts to

```
atomic nsa and_r
        (seqGoal (H >>> (app and (tuple (_5035 :: _5052 :: nil)))))
        (H =  H1, H =  H2)
        true
        seqGoal (H1 >>> _5035) ** seqGoal (H2 >>> _5052)
        notacticyet.
```

8.3.4 Theory Grammar

Identifiers can contain underscores but not hyphens. A theory begins with a declaration of its name, and the logic it uses. The logic name is not used, but must be entered for syntactic correctness.

```
Theory-declaration ::=
        'Theory' Id '.'
        'Use Logic' Id '.'
        (Declaration '.')*

Declaration ::= Constant-declaration
                | Type-declaration
                | Axiom-declaration
                | Inference-declaration
                | Conjecture-declaration
                | Predicate-declaration
                | Definition


Constant-declaration ::= 'const' Id Type

Type-declaration ::= 'type' Id

Axiom-declaration ::= 'axiom' Id Prop | 'axiom' Id '[' Sequent ']'

Inference-declaration ::= 'inference' Id '[' Sequents '/' Sequent ']'

Conjecture-declaration ::= 'conjecture' Id '[' Sequent ']'

Predicate-declaration ::= 'predicate' Id Type+

Definition ::= 'define' 'const' Id ':' Type '=' Term
             | 'define' 'type' Id '=' Type

Sequents ::= (Sequent ';')* Sequent

Sequent ::= Assumptions '|-' Prop
```

```
Assumptions ::= (Assumption ',')* Assumption

Assumption ::= Prop | Id ':' Type

Prop ::= 'forall' Id ':' Type '.' Prop
       | 'exists' Id ':' Type '.' Prop
       | Prop '/\' Prop
       | Prop '\/' Prop
       | Prop '=>' Prop
       | '<' Term '=' Term '>'
       | '(' Prop ')'
       | Id
       | Id '{' Term '}'

Type ::= nat | bool | '(' Type ')' | Type '->' Type | Type '#' Type | Id

Term ::= Id
       | (Id ':' type '\' Term ')'
       | '<' Term Term '>'
```

8.3.5 Theory file

```
Theory Nsa.
Use Logic Thol.

type real.
type hyperreal.
const z nat.
const f nat->nat#bool.

axiom sch2 [x:nat |- <x=x>].
axiom sch3 [x:nat, y:nat |- <x=y>].
axiom sch4 [x:nat, y:nat, <x=y> |- <y=x>].
axiom ax1 [y:nat |- forall f:nat->nat. <f = (x:nat \ <f y>)>].
axiom exax exists x:nat. <x=x>.
axiom badax [true, x:nat |- <x=x>].

conjecture conj1 [x:nat |- forall y:nat . <x=y>].
inference inf1 [true |- false / true, x:nat |- <x=x>].

axiom qs (forall x:nat. (exists y:nat. <x=y>)) => true.

conjecture conj2 [x:hyperreal |- exists y:hyperreal . <x=y>].

predicate even : nat.
predicate close_to : hyperreal, real.

inference easy_inf [true|-true; false|-false / x:nat |- <x=x>].

axiom pred_in_prop even{z}.

define type real_pair = real#real.
```

```
define const n:nat#bool = <f z>.

define const m:nat = z.

axiom false_e [G1, false, G2 |- P].

inference and_r [G |- A; G |- B / G |- A /\ B].

inference and_l [G1,A,B,G2 |- P / G1, A /\B,G2 |- P].

inference or_l [G,A |- P; G,B |- P / G, A\/B |- P].
```

8.3.6 Generated λ*Clam*

```
module nsatheory.

all_pred P nil.
all_pred P (H::T) :- P H, all_pred P T.
sublist Xs Ys :- all_pred (X\ (member X Ys)) Xs.

has_otype nsa real universe.
has_otype nsa hyperreal universe.
has_otype nsa z (nat).
has_otype nsa f (nat arrow tuple_type (nat :: bool :: nil)).
axiom nsa sch21 ltor (trueP) (_135695) (_135695).

axiom nsa sch22 ltor (trueP) (app eq (tuple (_137129 :: _137129 :: nil)))
(trueP).

axiom nsa sch31 ltor (trueP) (_137732) (_137749).

axiom nsa sch41 ltor (app eq (tuple (_139400 :: _139383 :: nil))) (_139383)
(_139400).

axiom nsa sch42 ltor (app eq (tuple (_144856 :: _144839 :: nil))) (app eq
(tuple (_144839 :: _144856 :: nil))) (trueP).

axiom nsa ax11 ltor (trueP) (app forall (tuple ((nat arrow nat) :: abs (W1\
app eq (tuple (W1 :: ty_abs nat (W2\ app W1 _147505) :: nil))) :: nil)))
(trueP).

axiom nsa exax1 ltor (trueP) (app exists (tuple (nat :: abs (W1\ app eq (tuple
 (W1 :: W1 :: nil))) :: nil))) (trueP).

axiom nsa badax1 ltor (trueP) (_148474) (_148474).

axiom nsa badax2 ltor (trueP) (app eq (tuple (_149926 :: _149926 :: nil)))
(trueP).

top_goal nsa conj1 [(otype_of (user_object "x") nat)] (app forall (tuple (nat
:: abs (W1\ app eq (tuple (user_object "x" :: W1 :: nil))) :: nil))).
atomic nsa inf1
        (seqGoal (H >>> (app eq (tuple (_152632 :: _152632 :: nil)))))
        (sublist (trueP :: otype_of (user_object "x") nat :: nil) H)
```

```
        true
        (seqGoal (H1 >>> falseP))
        notacticyet.

axiom nsa qs1 ltor (trueP) (trueP) (app forall (tuple (nat :: abs (W1\ app
exists (tuple (nat :: abs (W2\ app eq (tuple (W1 :: W2 :: nil))) :: nil))) ::
nil))).

axiom nsa qs2 ltor (trueP) (app imp (tuple (app forall (tuple (nat :: abs (W1\
 app exists (tuple (nat :: abs (W2\ app eq (tuple (W1 :: W2 :: nil))) ::
nil))) :: nil)) :: trueP :: nil))) (trueP).

top_goal nsa conj2 [(otype_of (user_object "x") (user_object "hyperreal"))]
(app exists (tuple (user_object "hyperreal" :: abs (W1\ app eq (tuple
(user_object "x" :: W1 :: nil))) :: nil))).
atomic nsa easy_inf
        (seqGoal (H >>> (app eq (tuple (_156487 :: _156487 :: nil)))))
        (sublist (otype_of (user_object "x") nat :: nil) H, replace_in_hyps H
(otype_of (user_object "x") nat) (falseP :: nil) H1, replace_in_hyps H
(otype_of (user_object "x") nat) (trueP :: nil) H2)
        true
        (seqGoal (H1 >>> trueP) ** seqGoal (H2 >>> falseP))
        notacticyet.

axiom nsa pred_in_prop1 ltor (trueP) (app (user_object "even") (user_object
"z")) (trueP).

definition nsa real_pair0 trueP (tuple_type (user_object "real" :: user_object
 "real" :: nil)) real_pair.
definition nsa n0 trueP (app (user_object "f") (user_object "z")) n.
definition nsa m0 trueP (user_object "z") m.
atomic nsa false_e
        (seqGoal (H >>> (_25170637)))
        (sublist (falseP :: nil) H)
        true
        trueGoal
        notacticyet.

atomic nsa and_r
        (seqGoal (H >>> (app and (tuple (_162845 :: _162862 :: nil)))))
        (true, H =  H1, H =  H2)
        true
        (seqGoal (H1 >>> _162845) ** seqGoal (H2 >>> _162862))
        notacticyet.

atomic nsa and_l
        (seqGoal (H >>> (_186518)))
        (sublist (app and (tuple (_186535 :: _186551 :: nil)) :: nil) H,
replace_in_hyps H (app and (tuple (_186535 :: _186551 :: nil))) (_186567 ::
_186535 :: _186551 :: _186583 :: nil) H1)
        true
        (seqGoal (H1 >>> _186518))
        notacticyet.
```

```
atomic nsa or_l
        (seqGoal (H >>> (_197214)))
        (sublist (app or (tuple (_197231 :: _197247 :: nil)) :: nil) H,
replace_in_hyps H (app or (tuple (_197231 :: _197247 :: nil))) (_197247 ::
nil) H1, replace_in_hyps H (app or (tuple (_197231 :: _197247 :: nil)))
(_197231 :: nil) H2)
        true
        (seqGoal (H1 >>> _197214) ** seqGoal (H2 >>> _197214))
        notacticyet.

end
```

Chapter 9

λ-Clam in the MathWeb Software Bus

Jürgen Zimmer[1] (e-mail: jzimmer@mathweb.org)
This chapter summarises the work done by Jürgen Zimmer visiting the DREAM group as a CALCULEMUS Young Visiting Researcher (Oct. 2001 - Mar. 2002).

9.1 Introduction

The MathWeb Software Bus [11] is a platform for distributed automated reasoning that supports the connection of a wide range of mathematical services by a common software bus.[2] The MathWeb-SB provides the functionality to turn existing reasoning systems into mathematical services that are homogeneously integrated into a networked proof development environment. Client applications can access 23 reasoning systems, for instance, the CASs Maple, MagMa, and CoCoA, the constraint solver $\mathcal{C}o\mathcal{SIE}$, mediators, model generators, such as Mace and Satchmo, and the automated theory formation system HR. Moreover, the MathWeb-SB integrates nine first order ATPs, such as Otter, Spass and Bliksem, and E.

The architecture of the MathWeb system is depicted in Fig. 9.1. Local *brokers* provide routing and authentication information to the mathematical services. MathWeb-SB wrappers encapsulate the actual reasoning systems and offer the mathematical services to their local broker. Client applications such as the Ωmega system, HR, or λ-Clam, connect to a MathWeb broker and request services. In case the requested service is found, the client application receives a reference to a newly created *service object*. The client can then send messages directly to this service object.

9.2 Integration of λ-Clam

We integrated λ-Clam into the MathWeb-SB using the methodology that has been successfully used for the integration of other systems like the Ωmega proof planner or the higher order theorem prover TPS: A MathWeb-SB *wrapper* implements the interface to the services offered by λ-Clam and has full control over λ-Clam by sim-

[1]The author was supported by the CALCULEMUS European Union IHP grant HPRN-CT-2000-00102.

[2]Further information about the MathWeb-SB is available at http://www.mathweb.org/mathweb/.

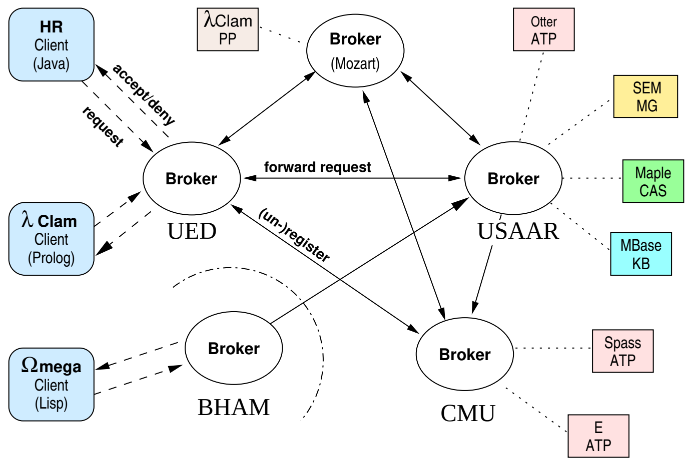

Figure 9.1: The MathWeb Software Bus

ulating user input using socket communication[3]. λ-Clam itself can use the wrapper to access other services as external reasoning systems.

The implementation of λ-Clam, Teyjus λ-PROLOG, does not allow programming with multiple threads. Therefore, we had to define a new interaction mode for λ-Clam in which the system reads commands from the MathWeb-SB socket:

`sock_read_write`: The top-level loop of a λ-Clam process in `sock_read_write` mode exclusively reads commands from the `lclam_server_socket`. The user cannot interact with the system via the usual toplevel loop.

9.2.1 Starting λ-Clam in the MathWeb-SB

To use λ-Clam within the MathWeb-SB the system has to be started by the λ-Clam MathWeb wrapper which establishes a socket connection to the Teyjus process. This means, a MathWeb-SB installation is required. At the moment, installing and configuring the MathWeb-SB is still a bit tricky. A detailed description of how to install and configure the MathWeb-SB can be found on the system's homepage `http://www.mathweb.org/mathweb`. In the following we only sketch the necessary steps to get an example session running on DREAM machines:

Add the MOZART directory of the user `jzimmer` to your path:

- `export PATH=${PATH}:/hame/jzimmer/.oz/bin`

You can put this line in your `~/.bashrc` or your `~/.benv` file for the sake of convenience.
Copy MathWeb-SB's configuration files for the DREAM environment:

- `cd ~`
- `mkdir .mathweb`
- `cp -bi /hame/jzimmer/.mathweb/*config* .mathweb/`

[3]Note: In the current implementation of Teyjus, socket communication only works on Solaris/SunOS machines.

The MathWeb-SB expects that the environment variable `LCLAM_HOME` is set to some installation of λ-Clam (Version 3.2. or higher). If you don't have λ-Clam installed, please set the variable to the installation of some other user, for instance, `export LCLAM_HOME=/hame/jzimmer/lclam`. The same holds for the variables `TEYJUS_HOME` which should point to the implementation language Teyjus, and `TJPATH` which defines the load-path of Teyjus (cf. section 2).
Start the MathWeb-SB and the λ-Clam system[4] :

- `lclam-mathweb`

An emacs window should appear with a λ-Clam process running in `command_pretty` mode. The system should behave like af "normal" λ-Clam except that the socket connection to the MathWeb-SB wrapper is established before the user is allowed to type in commands. You can try to plan a problem which uses the simplification of the Computer Algebra System MAPLE during the planning process by typing in the following commands:

```
add_theory_defs_to_sym_eval_list analytica.
set_wave_rule_to_sym_eval.
add_to_induction_scheme_list twostep2.
pds_plan (induction_top normal_ind) fib_goal.
```

You should leave your λ-Clam system by typing `halt.` in the λ-Clam prompt and then `C-x C-c` in the emacs window.

9.3 λ-Clam using MathWeb Services

For the use of external reasoning systems, we extended λ-Clam by the modules `src/io/sockets.mod` and `src/plan/mathweb.mod` which abstract from socket communication details and offers a convenient interface to the MathWeb-SB wrapper. The main predicates of the `mathweb` module are the following:

`mathweb_get_service:`
`string → string → o.`

takes the service name as first argument (e.g. "MAPLE", "OTTER") and returns a string uniquely identifying the service object in the second argument, if the service is available. If the service is not available, the predicate returns the string `"error"`.

`mathweb_apply:`
`string → string → (list (list string)) → int → (list (list string)) → o.`

applies a method to a service object identified by the first argument. The second argument is the name of the service method. The third argument is a list of lists of strings representing the parameters for the method call. The fourth argument is an integer giving the timeout for the method call in seconds. The result of the method call is again encoded in a list of lists of strings and is unified with the fifth argument.

The following example is taken from the module `src/theories/maple.mod` and illustrates how to call the `simplifyTerm` method of a MAPLE service:

```
mathweb_get_service "MAPLE" MapleService,
print_open_math Formula OMFormula,
OMObj is "'OMOBJ'(" ^ OMFormula ^ ")",
```

[4]Note that to be able to start `lclam-mathweb` properly you should be a member of the user group `dreamer`. Ask `gordon@dai.ed.ac.uk` for details.

```
mathweb_apply MapleService "simplifyTerm" [["1", OMObj]] 100 Result.
```

First, a MAPLE service object is requested. Then a given `Formula` is translated into OPENMATH representation. The OPENMATH object is given as the first (and only) argument to the `simplifyTerm` service of MAPLE.

`mathweb_leave_service`:

`string -> o.`

leaves the service object identified by the first argument. The service object and the corresponding reasoning system (e.g. a MAPLE process) is terminated. MathWeb-SB clients should always leave service objects when they are no longer needed.

We implemented the module `src/io/print_open_math.mod` to translate λ-Clam's formulae into the OPENMATH standard using the core OPENMATH Content Dictionaries (CDs). Thus, the module forms one half of an OPENMATH phrase-book for λ-Clam. The translation table for λ-Clam's symbols is distributed over the theory files `src/theories/arithmetic`, `src/theories/analytica`, and `src/theories/constructive_logic` to keep the OPENMATH representation close to the definition of the symbols.

Using the `mathweb` module, λ-Clam can now access every reasoning system available in the MathWeb-SB. Possible applications are:

Using ATPs to prove simple subgoals: This approach has already been used in the ΩMEGA proof planner to restrict the search space. In some cases, even higher order problems can be transformed to first order problems and then sent to one or more first order ATPs in the MathWeb-SB.

Using ATPs on the control level: It is a known shortcoming of λ-Clam that it does not check the consistency of hypotheses after performing a case split. This leads to λ-Clam missing easy proofs by inconsistency. Modern ATPs are very efficient and could detect trivial inconsistencies in a few milliseconds. We therefore try to prune inconsistent search paths in proof planning with the help of ATPs like OTTER.

CAS computation in proof planning: Due to our positive experience with CAS computations in many proof planning domains, we think that the use of CASs in inductive proof planning can enable λ-Clam to solve problems it couldn't solve before. The rewriting capabilities of a CAS can complement the rewriting of λ-Clam and can thus enhance the reasoning capabilities of λ-Clam.

Until now, we focused on the latter and formalised problems taken from the *Analytica* system [8, 2] and the CLAM-Lite system [20][5]. The *Analytica* problems can be found in the theory file `src/theories/analytica.mod` and the summation problems in the file `src/theories/sums.mod`. During the planning for these problems, λ-Clam uses MAPLE's `simplifyTerm` service which gets a term (formula) in OPENMATH representation as an input argument. It returns the OPENMATH representation of the term (formula) resulting from the call of MAPLE's `simplify` function. For the access of MAPLE, we defined the `maple_simplify_meth` which calls the `simplifyTerm` service of MAPLE on the current subgoal and introduces the resulting formula as a new subgoal. The method is defined in the module `src/theories/maple.mod`. It should fail if λ-Clam is in standalone mode and no MathWeb-SB connection can be established. We added `maple_simplify_meth` to the compound methods `induction_top` and `fertilise` defined in the module `src/theories/induction.mod`.

[5]NOTE: Due to a bug in Teyjus's memory management, most of the examples do not run with Teyjus Version 1.0-b30p2.

9.4 Services offered by λ-Clam

All services offered by λ-Clam are based on the OpenMath [7] and the OMDoc [14] standard. We implemented a translation service (the second half of a phrase-book) to translate incoming formulae, definitions, and conjectures into λ-Clam's higher order abstract syntax. For the sake of efficiency, this translation is performed by the MathWeb-SB wrapper. λ-Clam currently offers two services to the MathWeb-SB:

`planProblem`: This service takes an OMDoc document, containing a single conjecture, as an argument. The service starts the λ-Clam proof planning mechanism on the conjecture. In our current implementation, the service expects the conjecture to be about natural number arithmetic. We plan an extension of the service such that clients can also provide the theory the conjecture is defined in.

Client applications using the `planProblem` service can use optional arguments to determine which proof strategy (compound method) λ-Clam should use for the planning attempt, and to give a time limit in seconds. In the current implementation, the service simply returns the OpenMath symbol *true* if λ-Clam could find a proof plan within the given time limit, and $false$ if no proof plan could be found.

`ripple`: Rippling is one of the most successful heuristics for guiding (inductive) proof planning. Therefore, λ-Clam offers its rippling mechanism as a separate service to the MathWeb-SB. The service is given a single OMDoc document as an input. The OMDoc must contain a non-empty set of rewrite rules, formalised as lemmas, and a goal sequent $H \vdash \phi$ as a conjecture. The `ripple` service tries to reduce the difference between ϕ and the best suitable hypothesis in H using the rewrite rules. The `ripple` service also tries to apply fertilisation to reduce the goal ϕ to the trivial goal *true*. The service returns the OpenMath symbol *true* if this was successful and $false$ otherwise. In a future version, the `ripple` service should us the partial planner, implemented by Louise Dennis, to ripple as far as possible and return an OMDoc which contains the resulting proof planning goal as a sequent $H \vdash \phi'$.

The services offered by λ-Clam can be used by other reasoning systems via the MathWeb-SB. We made first experiments with the use of the `planProblem` service and the `ripple` service within the Ωmega proof planner. We formalised problems in the natural number theory of Ωmega and implemented proof planning methods that call the `planProblem` and the `ripple` service of λ-Clam to close open subgoals or to reduce a subgoal to a simpler one, respectively.

To see Ωmega calling λ-Clam's services, do the following:
Start an omega by typing in a shell:

- `loui-local`

A window with the GUI of Ωmega should appear. Read in a problem and plan it:

- choose the menu `File`→`Read`→`Problem`
- Click `Choose File...`
- choose `/hame/jzimmer/xmp/plus2right.post`
- execute one planning step with the menu `Planner`→`Step Plan` or with `CTRL+t`.

During the planning step, the ΩMEGA proof planner tries to apply methods for the call of both the `ripple` and the `planProblem` service. This is why you see two windows of λ-Clam services popping up on the screen. In our example only the `planProblem` leads to success and the proof planning goal in ΩMEGA is closed with the justification `LCLAM-M`, i.e. the proof planning method that called λ-Clam's `planProblem` service. You can also see a window with title "Oz Browser" which shows the lemmas ΩMEGA sends to λ-Clam for the rippling[6].

9.5 Possible Future Work

Possible directions for future work include the use of other external reasoning systems such as ATPs or decision procedures in λ-Clam as described in section 9.3.

Our experiments with the use of Computer Algebra computations in proof planning could be extended to a full case study. Especially the summation problems of [20] are interesting for this application.

The services offered by λ-Clam could be refined and extended. The `ripple` service, for instance, should return the subgoal after one application of the rippling method (and fertilisation). The service could also return the list of rewrite rules applied, and the positions of the subterms the rules were applied on. The `planProblem` service could be extended to take the definition of the theory a problem is formulated in, and to return a (partial) proof plan in OMDoc encoding.

Furthermore, λ-Clam offers the means to formalise and prove theorems in nonstandard analysis (NSA) [16]. Using NSA, λ-Clam could already find proof plans for the limit theorems LIM-PLUS and LIM-TIMES and some other analysis theorems, e.g., the mean-value theorem. In contrast to the classical ϵ-δ proofs, NSA proofs tend to be much shorter and more intuitive. Hence, it would be interesting to use λ-Clam's NSA theory to construct alternative proofs for some of the complex theorems proved by the *Analytica* system using the computational power of CASs available in the MathWeb-SB.

The proper use of context in the communication between reasoning systems is still an open research question. Context can not only reduce the amount of information that has to be transfered between systems, it is also crucial to establish more complex forms of collaboration and coordination between reasoning systems. The λ-Clam proof planner, for instance, offers the powerful *critics* mechanism which analyses the failure of a proof attempt and gives feedback to the user about possible ways of correcting the proof. This feedback can include generalising the original goal, modifying the chosen induction scheme, or speculating a new rewrite rule. Potentially this feedback could also be given to another reasoning system using λ-Clam and this would involve far more complex interactions in terms of context. For instance, if λ-Clam were to suggest modifying the induction scheme this new proposed scheme might have to be transmitted back to the client system for verification in terms of its own logic. To enable this form of fine-grained interaction between reasoning systems we need to develop a general notion of context in intersystem communication. λ-Clam could then be used as a prototypical reasoning systems that builds up a context when communicating with other systems.

[6]Change the representation mode in the Browser menu "Options" to see show strings and virtual strings strings properly. Then select the whole Browser output and type `C-b`. Further information can be found at `http://www.mozart-oz.org/documentation/browser`.

Chapter 10

Developing λ-Clam

10.1 Introduction

This chapter is intended mainly for the primary developer(s) of λ-Clam . It contains information on the system architecture of the current version (v4.0.0) of λ-Clam , an discussion of the core planning modules, an overview of the contents of the theory library, and lastly some information on benchmarking.

This chapter may seem a little disjointed , having been written by two different developers at different times. In particular, section 10.2 on the λ-Clam architecture was written by James Brotherston (Aug 2001 – Oct 2002); the rest is mostly due to Louise Dennis (Aug 1999 – Aug 2001) but has been updated.

10.2 λ-Clamv4 schematics

10.2.1 The Teyjus module system

It is important to understand the meanings of the various types of arc in the schematic diagrams we present, since they correspond to rather different types of module accumulation in Teyjus.

Dotted lines correspond to *signature extension*. In other words, if there is a dotted arc from M2 upwards to M1, then in the signature of M1 we have `accum_sig M2.` Thus all the predicates and constants declared in the signature of M2 are "pasted" into the signature of M1, and therefore are visible to other modules that accumulate M1. However, the actual predicate definitions in M2 (if any) are *not* accumulated into M1. This may seem strange, but in most cases we do this to avoid the situation where the same predicate definitions are accumulated more than once into a higher-level module. In that situation, those predicates start to succeed once for each copy of the definition accumulated — which is not desirable. We therefore import the signature only, in order to avoid type errors, and make sure that the code for each module is accumulated only once.

Solid lines correspond to *global accumulation* of modules — if there is a solid line upwards from M2 to M1, then in the body of M1 we have `accumulate M2` and in the signature of M1 we have `accum_sig M2`. So in this case, as with signature extension, all the declarations in the signature of M2 are visible in M1 and outwards from it, but the predicate definitions of M2 are now also present in M1 and visible outwards from it.

Lastly, dashed lines correspond to *local accumulation* of modules, with the possibility of partial outwards visibility. If there is a dashed line upwards from M2 to M1 then in the body of M1 we have both `accumulate M2` and `accum_sig M2`. No

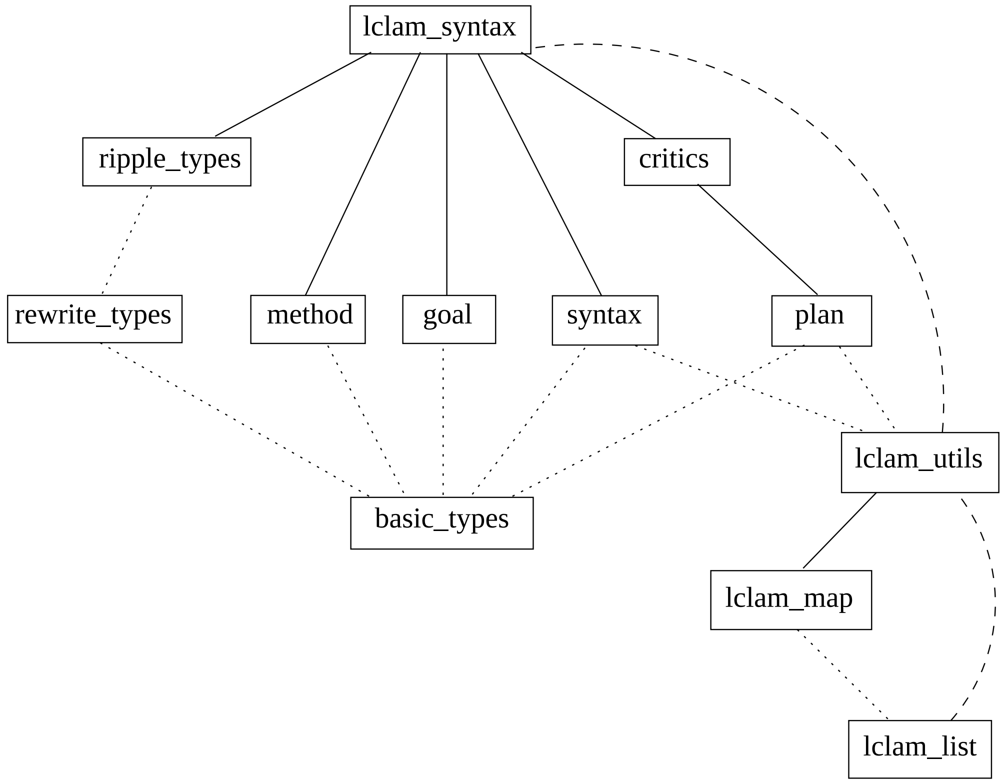

Figure 10.1: λ-Clam syntax declarations and utility functions

mention of M2 is made in the *signature* of M1, however. Doing this treats all the declarations and definitions in M2 as local to M1, and thus not visible to modules that accumulate M1. However, we can allow a part of the signature of M2 to percolate outwards by repeating the relevant part of M2's signature in the signature of M1. Then those declarations and any associated predicate definitions are visible to modules that accumulate M1. Local accumulation is thus useful when we wish to conceal module code from higher-level modules.

We now proceed to present the schematics for λ-Clamv4, along with brief descriptions of the component modules.

10.2.2 Syntax and utilties (Fig. 1)

lclam_list All the usual list utilities including member, append and so on.

lclam_map Map utility predicates.

lclam_utils Utility supermodule which includes the list and map predicates, plus miscellaneous utilities such as findall.

basic_types Declarations for the most fundamental types in λ-Clam such as methods, goals, plan states and so on.

rewrite_types Declarations for types and predicates used in rewriting.

ripple_types Declarations for types and predicates used to support rippling.

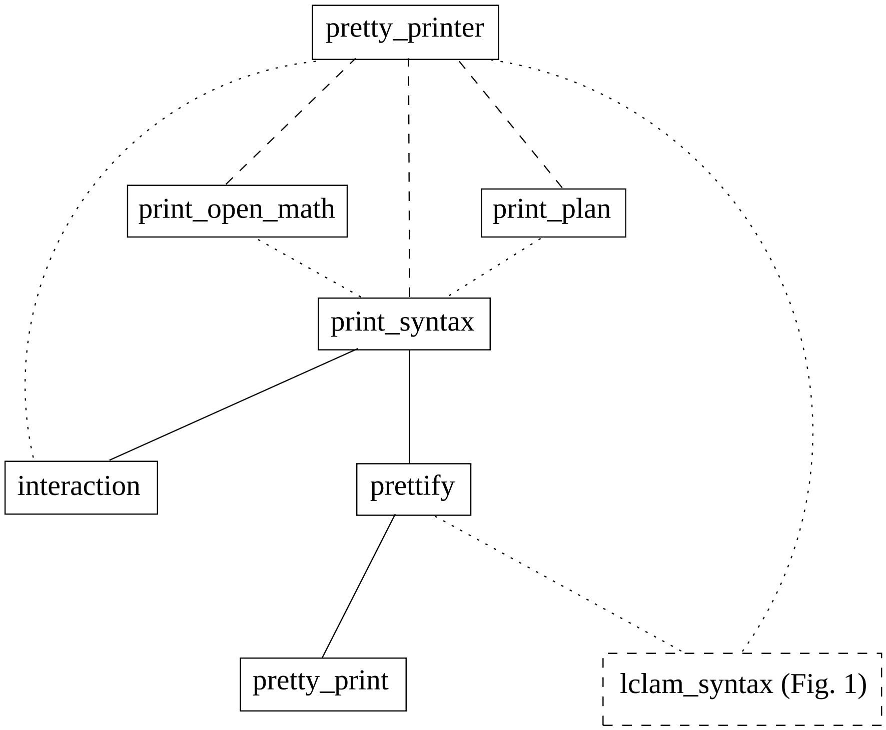

Figure 10.2: The λ-Clam pretty printer

method Basic method predicates atomic and compound, plus declarations for the various methodicals.

goal Declarations for the various goal constructors, plus a couple of utility functions on goals.

syntax Declarations for the constructors of osyn, which is the metalanguage used for goals in λ-Clam, plus utility functions for typechecking etc.

plan Constructors for plan states plus utility functions on plans.

critics Basic critics plus declarations for the criticals.

lclam_syntax Syntax / utility supermodule which includes all of the modules above, plus predicates for querying the various rewrite rule lists.

10.2.3 Pretty printer (Fig. 2)

pretty_print Declarations for the pretty-printing markup syntax, plus predicates for pretty-printing primitives.

prettify Predicates for marking up various syntax expressions for the pretty printer.

interaction Various output mode switches for controlling how an expression is printed.

print_syntax Predicates for pretty-printing terms, goals, methods etc.

print_open_math Predicates for pretty-printing OpenMath terms.

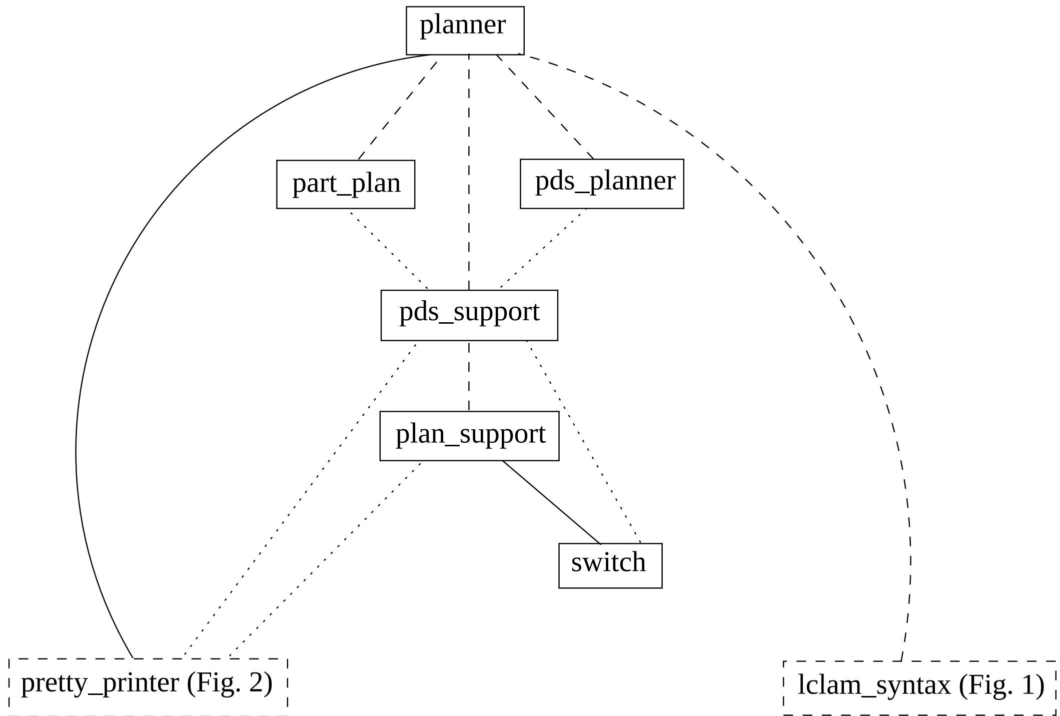

Figure 10.3: The λ-Clam planner core

print_plan Predicates for pretty-printing plans.

pretty_printer Interface for the pretty-printer. Notice that the other pretty-printing modules are locally accumulated into this module, allowing careful control over which printing predicates are visible to higher modules.

10.2.4 Planner core (Fig. 3)

switch Declarations for various switches which affect planner operation.

plan_support Predicates for manipulating the agenda and for tidying method continuations.

pds_support Most of the important planning predicates are in this module, including the predicates for applying methods and critics, and for transforming compound methods into a first method and continuation.

pds_planner Main predicates for the Partial Data Structure planner, in depth-first and iterative deepening flavours.

part_plan Main predicates for a partial planner.

planner Planner interface, including wrappers for calling the various planners. The actual code for the syntax and pretty-printer supermodules is accumulated into this module. Notice that the other planner modules are accumulated locally, ensuring that the planners can only be called via the interface provided by this module.

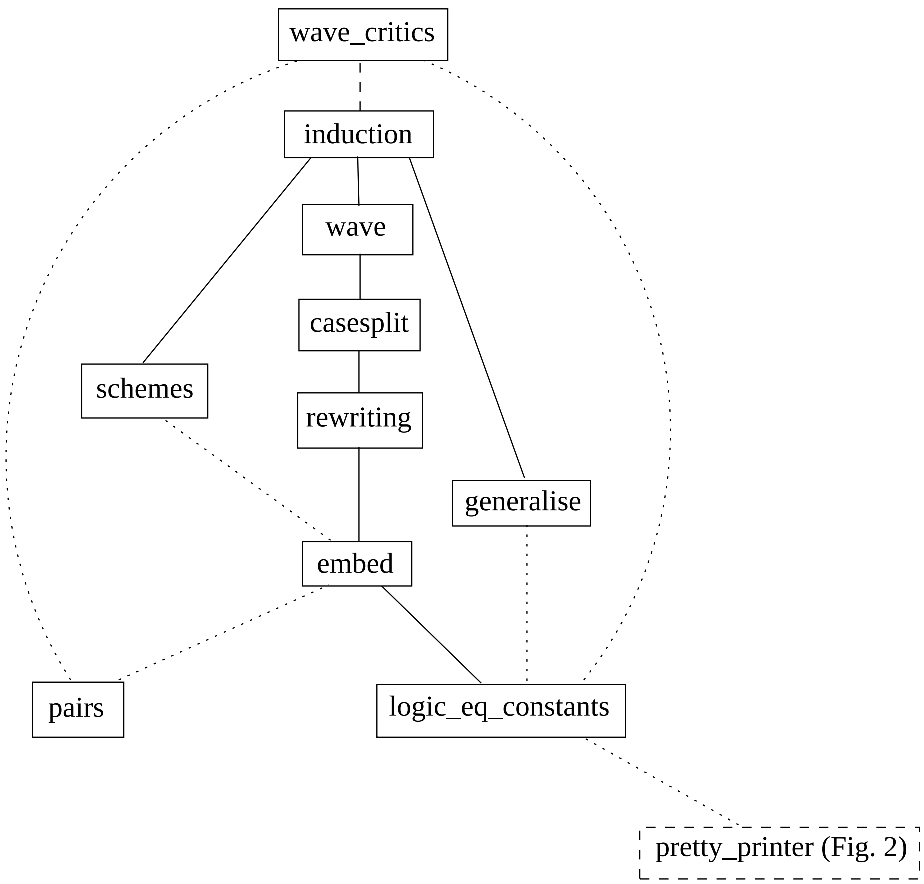

Figure 10.4: The λ-Clam theory core

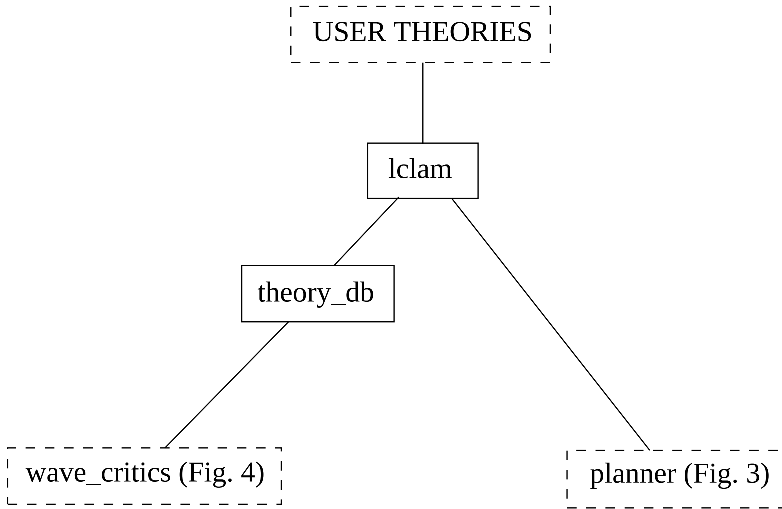

Figure 10.5: The top level of λ-Clam

10.2.5 Core theories (Fig. 4)

logic_eq_constants Declarations for logic constants and logical methods needed for the higher core theories.

pairs Declaration and syntactic sugar for pairing type.

generalise Generalisation method and supporting predicates.

embed Support for embeddings.

rewriting Generic rewriting methods and supporting predicates.

casesplit Support for case-splitting.

wave Support for rippling, including atomic wave methods.

schemes Support for induction schemes, including ripple analysis.

induction Methods for induction and supporting predicates.

wave_critics Supermodule for the core theories of λ-Clam with the usual control over the interface in order to hide the internal supporting predicates.

10.2.6 Top-level interface (Fig. 5)

theory_db Predicates for querying the λ-Clam theory database.

lclam Top-level module which accumulates the theory core and the planner and contains all of the various user commands.

10.3 The Proof Planning Core

This section goes through the files in the core proof planning making notes on anything of interest in those files. Much of the following will make little sense unless you have the code to hand to refer to.

10.3.1 Planner

The "top" module of the core proof planner is `planner.mod/planner.sig`. The planner is supposed to provide a generic top level predicate for applying any proof planner built into the system to any goal — which is what `plan_this` does. So `plan_this` takes as arguments a proof planner predicate, a method, a goal and what is called an *agenda predicate* which is supposed to define a search strategy. The idea is that `plan_this` applies the proof planner to the given goal using the given method and search strategy. In practice λ-Clam has two proof planners; the PDS planner (in `pds_planner.mod`, in iterative deepening and "Claudio" varieties), and a partial planner (in `part_plan.mod`), and only one agenda strategy which is the depth-first strategy. (However, it should be no problem to code up other stratgeies if required.) In order to speed up and facilitate use of the system we have wrappers for the various planners in `planner.mod` which call `plan_this` with the appropriate planner and strategy.

```
plan_this Proof_planner Method Query AgendaPredicate:-
        top_goal _ Query  Hyps Osyn, !,
        announce_start Query Osyn,
        Proof_planner (pstate (active_agenda [nil])
          (and_node (seqGoal (Hyps >>> Osyn))
                     nil
                     open_status
                     Method
                     nomethodyet
                     []
                     []
                     notacticyet
                     []))
          AgendaPredicate
          Plan, !,
        ((Plan = (pstate _ VerbosePlan), !, pprint_plan VerbosePlan);
        (Plan = (cpstate _ CPlan), pprint_cplan CPlan)),
        announce_end, !.
```

`announce_start` and `announce_end` just print out information at the start and end of planning. `top_goal` retrieves the actual sequent (`Osyn`) which must be proved. The proof planner `Proof_planner` to an empty agenda (`pstate (active_agenda [nil])`), an initial `and_node` of a plan (notice this has lots of empty slots because nothing has happened yet), and the agenda predicate, getting back a `Plan` as output. There is then a disjunction on how the plan should be treated. If the plan is a `pstate` then it is verbose (i.e. it contains all the plan information) otherwise it has been stripped down and contains just method and goal information (`cpstate`). This is because full plans were difficult to read and took up too much memory. (Hopefully somebody will implement some decent pretty printing for plans soon).

10.3.2 pds_planner

So what does the PDS planner actually do? The predicate `pds_planner` actually just passes everything on to `context_planner` adding an *action* and asserting the first node in the plan (`pplan_node`).

The heart of this planner is the clause

```
context_planner ActionIn Agenda AgendaPredicate PSOut:-
        context_plan_step ActionIn ActionOut Agenda AgendaOut NewNodes,
```

```
%% AgendaOut is containd in ActionOut
create_agenda ActionOut (pliststate AgendaOut NewNodes)
        AgendaPredicate (pliststate NewAgenda NewNewNodes),
context_planner_nodes ActionOut NewNewNodes NewAgenda AgendaPredicate PSOut.
```

The predicate context_plan_step performs one planning step. This takes the current agenda and action and applies the action appropriately to the agenda (the action tells the planner whether to apply a method or a critic) and returns a new action and agenda as a result. The reason the context_plan_step predicate can alter the agenda is in the case of critics etc. where it may want to delete nodes from the agenda. It also returns a list of the nodes in the plan that have been altered as a result of the action. create_agenda uses the new agenda, the list of changed nodes and the agenda predicate to create a further new agenda. For example, in depth-first planning it would put any new nodes on the front of the agenda while in breadth-first planning it would put them at the end. Note that this just handles the order in which subgoals are treated. There is also a question about handling different outcomes for method application on a given subgoal. The planner then recurses. There are extra clauses in context_plan_step apart from the base case. These are attempts to deal with cuts which have never worked satisfactorily.

This planner (and its name) require some explanation. The first pass at implementing a partial data structure planner in Teyjus resulted in an algorithm that was immensely space hungry. The planner failed to find plans for the associativity of append, exceeding of the simulator's allowed heap size of 120 Megabytes.

While the memory needed by the planner was not so large or so catastrophic in the Mali version of λ-Clam, it was of concern. Jeremy Gow had processes that were taking up as much as a gigabyte and there were a couple of theorems we were failing to plan for supposed space reasons (though this was never satisfactorily proved).

An alternative way of communicating plan information within the program has been devised which seems to have substantially solved the space problems in λ-Clam.

λ-PROLOG's => Symbol

λ-PROLOG does not contain the assert and retract predicates which allowed Prolog programmers to essentially modify the predicates (and hence data) of their program dynamically at run time. Instead λ-PROLOG achieves this using =>, which has roughly the same semantics as implication. Below a simplified section of λ-Clam's top level loop is shown. lclam is the predicate that controls the loop. It waits for an input from the user and then executes the command they have given. In this case the user has specified that a new piece of syntax is to be used in future proof attempts. λ-Clam is most interested in the type of syntax (especially in the generalisation method) and refers to a predicate has_otype when it needs to use this. The code fragment below shows what happens when the user wants to dynamically add a piece of syntax. Essentially the program just returns to the lclam loop, but this time there is an extra clause of has_otype added using =>.

```
execute_command (add_osyn Syntax Type):-
        (has_otype Syntax Type) => lclam.
```

The scope of this implication is the right hand side, i.e. just the execution of the predicate lclam. So in:

```
execute_command (add_osyn Syntax Type):-
        ((has_otype Syntax Type) => lclam),
        has_otype Syntax Type.
```

the second has_otype Syntax Type will fail because it's outside the scope of the => expression.

`=>` is used to "assert" the nodes in the plan as they were generated rather than passing the partial plan around internally. This had a dramatic effect on the size of the process — reducing attempts that had easily topped 100 Megabytes to under 10 Megabytes.

Implementation Details

Plan nodes are stored as an address and a plan using the predicate `pplan_node`. By default all addresses have an empty plan associated with them unless stated otherwise. Plans are retrieved using `retrieve_node`. This dual structure is used so that only the most recently stored plan for an address is retrieved – this is done with a cut in `retrieve_node`.

Since nodes can also be quantified there are similar predicates for storing and retrieving the bindings of the quantified variables (do I mean bindings? I think maybe I mean instantiations).

```
local nobinder osyn.

%% support for context
pplan_node _ emptyPlan.
retrieve_node Address Plan:-
        pplan_node Address P, !,
        %% delay unification until after the cut to make sure _only_ the
        %% last plan node succeeds
        (Plan = P).

address_binder _ nobinder.
retrieve_binder Address Binder:-
        address_binder Address B, !,
        (Binder = B).
```

The Planning Algorithm The basic planning algorithm[1] is:

```
context_planner Agenda Out:-
        context_plan_step Agenda AgendaOut NewNodes,
        create_agenda AgendaOut NewNodes NewAgenda NewNewNodes,
        context_planner_nodes NewNewNodes NewAgenda Out.
```

`context_plan_step` applies a method to the first node in the `Agenda` and returns a list of child nodes. A new agenda is then created (users may intervene at this point) and then planning continues. `build_child_nodes` handles the addition of the new nodes to the context.

```
%%%  "Asserting" the new nodes
context_planner_nodes nil Agenda Out:-
        context_planner Agenda Out.
```

If there are no new nodes carry on planning.

```
context_planner_nodes ((add_node Ad (and_node ...)::T)) A Out:-
        ((pplan_node Ad (and_node ...)) =>
                context_planner_nodes T A Out).
```

If an ordinary node is to be added to the tree, add it to the context and continue. NB. I've obscured a lot of the content of `and_node`s here.

[1] NB. The actual code is unfortunately rather more complex than this

```
context_planner_nodes ((add_node Ad (forall_node Ad Type AtoPlan))::T) A Out:-
        pi x\ (
        all_add_nodes (AtoPlan x) (NewN x),
        append (NewN x) T (NT x),
                ((pplan_node Ad (forall_node Ad Type AtoPlan)) =>
                  ((address_binder Ad x) =>
                        context_planner_nodes (NT x) A Out))).
```

For a universally quantified node a new constant `x` is introduced and substituted into `AtoPlan` to create a list of nodes, all of which are marked as nodes to be added. The `forall_node` is stored, as is the constant that was introduced. The case for existentially quantified nodes is very similar except a variable is used instead of a new constant. The stored binder is only used when reconstructing the plan at the very end to make sure the correct constant is abstracted away from the children of the node.

```
context_planner_nodes ((delete_node Ad)::T) A Out:-
        ((pplan_node Ad emptyPlan =>
                context_planner_nodes T A Out)).
```

Lastly if a node is marked for deletion (if for instance a critic has been invoked) then the empty plan is stored once more at that address.

Once stored, plan nodes are accessed using `retrieve_node` when they are extended by a method or critiqued.

Performance

No coherent performance testing has been carried out, but a couple of figures are available at time of writing *NB. this was in August 2001 and these figures are almost certainly no longer accurate - JB.* The associativity of plus plans in under 6000 K in the new Teyjus version of λ-Clam, when the data structure was passed round this plan took over 40 M. The Mali version of λ-Clam passed the data structure as an argument but did so less than the current version (the planner recursed under the plan step rather than after the plan step) and, of course, memory management was one of Mali's big things. In Mali the associativity of plus takes up 7320 K.

This makes the new Teyjus version and the Mali version appear largely equivalent in terms of space. However if we look at a large plan such as that for `expplus`, $X^{Y+Z} = X^Y * X^Z$, then the Mali version takes up 85 M while the Teyjus version takes 19 M. Better still most of that space is taken up when Teyjus constructs the plan data structure for printing out once the plan is found, up until that point the process is under 8512 K in size. The actual plan output is 267 K in size (give or take).

Theorems like `exptimes`, $X^{Y^Z} = X^{Y*Z}$, which produce larger plans crashed the Mali version of λ-Clam every time they were attempted, but go through the Teyjus version.

10.3.3 pds_support

This contains lots of essential predicates for supporting the pds planner, such as `pds_method_trans` for applying the methodical transformation rules, `context_plan_step` for applying one step in the planning process, and various predicates for applying methods and critics at a particular address.

The major predicates in this module are as follows.

context_plan_step This accesses the methodical expression contained in the current plan node. It transforms this expression (using `pds_method_trans` and tidy_continuation) into a first method and an subsequent methodical expression. It then applies the first method to the current plan node (using apply_method_at_address) to gain new nodes that need to be added to the plan.

apply_method_at_address This predicate has several clauses trying to deal with occurrences of triv_meth and id_meth and triv_tf_meth which is a method I introduced when investigating counterexample generators. This last method is not used in regular builds of the system. The two main clauses deal with compound and atomic methods. The atomic method case applies the method to the goal and gets a compound goal. I think the clauses for triv_meth are probably redundant since they have been superseded by the clause that checks for occurrences of trueGoal which appears directly before it in the code.

At this point things get a bit complicated. A compound goal is a conjunction (or disjunction - though this has never been investigated) of goals - e.g. the base and step case of an induction. Since a node in a plan contains a lot of additional labels (e.g the method the continuations etc. etc.) we don't want method writers to worry about this when their method outputs two subgoals. So they use the constructor ** to indicate a conjunction of goals. This must then, somehow, be turned into a list of child nodes. We've experimented with doing this immediately the goals were calculated and the details of how this worked can be found by looking through the CVS history. We rejected this in favour of transforming compound goals into lists of nodes as a separate application of apply_method_at_address and this is the current implementation. You will notice that special cases are set aside to deal with pair_meth when different methods are to be applied to different subgoals.

It should be noted that we would like then_meth to be considered as then_meths X (map_meth Y), i.e. then_meth is a special case of then_meths. then_meths is intended to require that it be made explicit what method is to be applied to which subgoal generated by its first method. This is not currently the case. Dealing with map_meth in an elegant way is quite hard. At present occurrences of map_meth are simply stripped where they occur and the planner assumes that a method applies to all plan nodes if there is a list of them at some point. Probably the clause of apply_method_at_address which deals with compound goals should be adapted so that it works out the appropriate continuations for its subgoals in some suitable fashion. At the moment we rely on repeated application of this clause if there are more than two subgoals (i.e. a tree of compound goals) to use map_meth properly apply_method_at_address would probably have to work through this tree in one go – this might also allow a more natural labelling of proof node positions which currently list each splitting of a conjunct as a separate node. Alternatively there would need to be special cases for map_meth applied to single proof nodes.

apply_critic This is very similar to a combination of pds_method_trans and apply_method_at_address. It works through the critical expression, unpacking compound critics until it reaches an atomic critic which it applies to the current plan and agenda. It involves a sub-predicate apply_critic_nodes which works much like context_plan_nodes. This is needed for uses of then_crit and the like where two critics may be applied one after another and we want the second to work on the plan as it is after the application of the first. Essentially apply_critic deals with a whole critical expression in one lump supplying at the end a list of nodes to be added or deleted and a new agenda (included in the resulting action) unlike the planner's handling of methods which it does in a much more step-by-step fashion. This is by accident rather than design and maybe should be thought on more carefully.

pds_method_trans This performs methodical transformations as outline in BB-Note 1349 (more or less). It treats `then_meth` and `then_meths` identically. I inherited this from Julian who thought there was no need for two methodicals here. I tried putting them back as they are supposed to be (see Alan Smaill for details) but fell foul of the handling of compound goals discussed above. This all needs rationalisation.

pds_method_trans also contains some extra clauses beyond those that are strictly necessary to remove occurrences of `id_meth`. `id_meth` has a habit of "bulking out" methodical continuations and it just makes things smoother to eliminate it where at all possible.

build_plan This transformation the plan (in the context) into an expression that can be output and printed to a user (or passed to a theorem prover). Even in compact form this uses a lot of memory.

10.3.4 plan_support

This module in theory contains predicates which are of use in general planner design. `tidy_continuation` is yet another predicate for the removal of unwanted occurrences of `id_meth`.

`apply_method_to_goal` is the basic method application predicate. It contains two cases for critics and one for normal method application. The normal application is straightforward. It checks the method is non-trivial (i.e. not `id_meth`), extracts its definition and then checks the preconditions and postconditions. It returns the preconditions and the new goals. The first case for critics (where the action is `addmeth` but the method has a surrounding `patch_meth`) treats the method in exactly the same way. It is only if the action is `critique` that something difference happens. In this case the preconditions are evaluated one at a time by `evaluate_in_turn`. This marks each precondition individual as a success or a failure and returns this marked up list for consideration by the critic. Essentially if a "patchable" method is attempted and fails (i.e. the first clause fails) then it is attempted in "critiquing mode". The action is used to determine new agendas and in `context_plan_step` to apply a critic instead of a method. The action informs the planner of what it should do next. `evaluate_in_turn` is largely uninteresting except to note that the nature of Teyjus prolog binding means that you have to explicitly bracket method preconditions – (See the `wave_method` – See also comments on `testloop` in § 10.4.1).

`create_agenda` simply applies the agenda predicate to the plan state. It has a second clause to allow some user interaction at this point. `modify_agenda` deals with use intervention in step by step planning mode. More on this in the next section.

Lastly `depth_first_plan` implements a depth-first search strategy. There are two records of plan state that can be passed to this. `pstate` records the agenda and the current plan while `pliststate` records the agenda and a list of nodes to be added or deleted. The first is a hang-over from before the context planner and has mostly been cleaned up. The fourth clause of `depth_first_plan` is used after the firing of critics when frequently much of the plan has been rolled back or deleted and the new agenda may bear little relation to the current agenda.

Editing Methodical Continuations: Allowing Users to Choose the Next Method

One of the features built into the Teyjus version of λ-Clam is the ability for a user to determine which method to apply next. This turned out to have some subtleties (caused by the design of the partial data structure planning mechanism (PDS Planner)) which I outline below.

The Planning Mechanism

A plan in λ-Clam is a labelled tree. The labels of the nodes contain information such as the goal at that node, the method that was applied and a *method continuation*. A method continuation is an expression of the methods to be attempted next. The planner works by extracting a “first method” and a new method continuation from such expressions, applying the first method to the goal and using the continuation to determine the method to be applied to the subgoals. This extraction process is determined by the rules shown in figure 10.6 (taken from BBNote 1349).

$$\frac{M \Rightarrow F \oplus L}{repeat\_meth\ M \rightarrow F \oplus (then\_meth\ L\ (orelse\_meth\ (repeat\_meth\ M)\ id\_meth))} \tag{10.1}$$

$$\frac{M_1 \Rightarrow F \oplus L}{then\_meth\ M_1\ M_2 \rightarrow F \oplus (then\_meth\ L\ M_2)} \tag{10.2}$$

$$\frac{M_1 \Rightarrow F \oplus L}{orelse\_meth\ M_1\ M_2 \rightarrow F \oplus L} \tag{10.3}$$

$$\frac{M_2 \Rightarrow F \oplus L}{orelse\_meth\ M_1\ M_2 \rightarrow F \oplus L} \tag{10.4}$$

$$\frac{M \Rightarrow F \oplus L}{try\_meth\ M \rightarrow F \oplus L} \tag{10.5}$$

$$\frac{}{try\_meth\ M \rightarrow id\_meth \oplus id\_meth} \tag{10.6}$$

$$\frac{C \qquad M_1 \Rightarrow F \oplus L}{cond\_meth\ C\ M_1\ M_2 \rightarrow F \oplus L} \tag{10.7}$$

$$\frac{\neg C \qquad M_2 \Rightarrow F \oplus L}{cond\_meth\ C\ M_1\ M_2 \rightarrow F \oplus L} \tag{10.8}$$

$$\frac{(then\_meth\ M\ triv\_meth \Rightarrow F\ L}{complete\_meth\ M \rightarrow F \oplus L} \tag{10.9}$$

$$\frac{method\_name(M)}{M \rightarrow M \oplus id\_meth} \tag{10.10}$$

Figure 10.6: The Methodical Transformation Rules

The PDS planner is currently implemented so that planning has two distinct stages. First a proof node is expanded with a method to produce children. Then the user is given the opportunity to interact with the planner, for instance to force backtracking. It is here that a user can intervene to suggest that a particular method be tried first effectively overriding the method continuation which would otherwise be applied there.

The obvious way to achieve this is to edit the current plan by modifying the method continuation appearing at the target node.

Attempts at Modifying the Method Continuation

It is hard to do a good evaluation of what exactly is needed since I have essentially experimented with only one proof strategy/method waterfall - i.e. that for induction. The strategy for induction is characterised by a large *orelse_meth* expression encased in a *repeat_meth* expression. This *orelse_meth* expression contains not only the induction method but also symbolic evaluation and tautology checking which are used to handle base cases and to finish off step cases after fertilisation. The order in which methods appear in the *orelse_meth* expression essentially determines the order in which they are attempted (for implementational reasons).

One of the first tests I performed was to specify that I wanted to try induction first, thus bypassing tedious backtracking over tautology checking and symbolic evaluations (a common frustration when attempting a proof).

I investigated a number of ways of doing this:

Replacing the Continuation The most naive approach was to completely replace the continuation with the new method (or methodical expression). I quickly rejected this. The problem with replacing the whole strategy with the induction method alone as a first step was that I lost the information for coping with base cases etc. Hence I deduced that the new method needed to be "embedded" somehow in the expression to preserve this information.

Placing the New Method as an Alternative This would be achieved by replacing the continuation, *Cont*, with *orelse_meth M Cont* where M is the new method. This had the added potential of allowing the "shuffling" of *orelse_meth* expressions to allow the user to prioritise one option over the other. This was my preferred solution when I started the work since I foresaw the possibility of an interface which could list all the alternatives in an *orelse_meth* expression and present them to the user to choose between. However the enclosing *repeat_meth* expression foiled this.

Say I have the expression *repeat_meth (orelse_meth M N)* and I want N to be attempted first. If I "shuffle" the order in the *orelse_meth* I end up with the expression *repeat_meth (orelse_meth N M)* which essentially preserves the "shuffle" in the subsequent repeats. Almost the only time you want to attempt induction first is at the start of the proof. In the base case and after rippling you want to attempt symbolic evaluation and tautology checking before starting a new induction off. So this approach caused unnecessary backtracking/user intervention later in the proof.

Ditching the recursion through the *repeat_meth* and editing the expression to *orelse_meth N (repeat_meth (orelse_meth M N))* didn't help since, if N succeeds I effectively loose the *repeat_meth (orelse_meth M N)* expression and don't recover it until N is backtracked over. In this case I was back in the situation I was in when I replaced the continuation with a new one[2].

It is tempting to think that at some point a user will reach the expression *orelse_meth M N* without the enclosing *repeat_meth* and will be able to edit it. This does not happen if intervention occurs before or after the methodical transformation process (since it will be modifying the result of the process). The continuation of *repeat_meth (orelse_meth M N)* is *then_meth L (orelse_meth (repeat_meth M N) id_meth)* where L is the continuation produced by transforming M or N – once again the alternatives are "trapped" under the *repeat_meth* expression. Obviously if the user were allowed to intervene *during* the transformation process they could affect the resulting methodical in a more controlled way. Methodical transformation is recursive so

[2]except that this at least allows recovering to the original strategy if the alternative one fails.

a user would need to be prompted for intervention every time the predicate recursed. I deemed that this would rapidly become tedious and it wouldn't be at all clear to a novice user how to "correctly" direct the process. It might be possible to pass information about user changes to the methodical transformation process rather than editing the methodical continuation in the plan node. I've not investigated this but I fear it would be very fiddly and complex.

Placing the New Method in sequence before the Continuation This is the solution upon which I finally settled. This assumes a user wants to preserve the existing strategy more or less intact but that they want a new method inserted into the plan at just one point in such a way that after the method is applied planning continues according to the original strategy. *try_meth* is used to make sure the whole planning process doesn't fail if the suggested method fails. So a methodical expression *Cont* will be transformed to *then_meth (try_meth M) Cont* when a user intervenes.

There are a couple of exceptions I implemented to this last rule. It is common to start a plan with the expression *complete_meth* M (*complete_meth* ensures the plan only succeeds if M succeeds with trivial subgoals) but it isn't common to use this anywhere else in the method waterfall for a strategy. It seemed likely that someone proposing an alternative first method at the start of a planning process would in fact intend that method to be encompassed within the *complete_meth* expression. In this case the procedure recurses through *complete_meth* and tries to add the new method to M.

I also recurse into the first argument of *then_meth* expressions. I'm not sure this is strictly necessary but I think it makes no difference to the behaviour described.

An Observation Motivating Recursion into *then_meth* Expressions

I observed that once *then_meth* appears at the top of a method continuation it appears at the top of all subsequent method continuations for children of that node. This is because it generates a *then_meth* expression as its continuation. *complete_meth* and *repeat_meth* also generate *then_meth* expressions as their continuations so once any of these have appeared in the method waterfall all method continuations thereafter start with *then_meth*. Furthermore the first method in the *then_meth* expressions will not be a method name it will be a methodical expression (unless it is *id_meth* or *triv_meth*). You can see this by an inductive argument on the rules. I originally recursed into *then_meth* expressions when I was trying out the various options involving *orelse_meth* since it got me to the "interesting" part I wanted to change. I saw no reason to change this after I'd settled on a solution although it was no longer clear this was strictly necessary.

Alternatives

The fact that I ran through several options for method choice suggests to me that different people may expect rather different things when using the facility. I can certainly see that people might actually want to replace the next method rather than insert a new method in front of it. This would indicate that a wider range options should be implemented.

What I learned in doing this is that the methodical transformation rules mean that care needs to be taken when creating or transforming plans on the fly otherwise important information can be lost. In particular *repeat_meth* will reproduce changes throughout a plan unless treated with care and this could well be undesirable behaviour.

10.3.5 critics

It is a good idea to look in both `critics.sig` and `critics.mod`. The various *criticals* are all constructors for the type `crit` - the type of critics. The construction of the signature of this module is very similar to the method module in that it defines criticals for creating critiquing strategies. It also defines a type `action` which is used to give instructions to the planning mechanism. This type has three constructors:

```
type critique   crit -> (list int) -> action.
type addmeth (list int) -> (list (list int)) -> action.
type endplan action.
```

`critique` is used to instruct the planner to critique the last method application attempt, `addmeth` is used to instruct it to try a method and `endplan` is used to instruct it to stop planning (NB. not sure if this last is implemented).

It also defines a new methodical `patch_meth` which is used in a strategy to indicate a method that the user would like critiqued.

The module also contains four atomic critics – `pop_critic`, `open_node`, and `roll_back_to_crit`.

I'll discuss the criticals first. Criticals are used by `apply_critic` in the pds_support modulepds_support module. Mostly they should be fairly self-explanatory (e.g. `orelse_crit C1 C2` succeeds if one of the critics `C1` or `C2` succeeds). The only ones I feel need any comment are `subplan_crit Ad C` which applies the critic `C` at address `Ad` in the plan and `some_crit C` which applies `C` to a meta-variable and continues - this can be read as $\exists x.C(x)$.

Atomic critics have 7 slots. The theory in which the critic is used (in this case `general_critics`), the address at which the critic is applied, the current agenda, preconditions, postconditions a list of plan nodes to be added or deleted as a result of the critic and a new agenda. The atomic critics are mostly straightforward `pop_critic H` instantiates `H` with the top address on the agenda. `open_node` updates a nodes status to `open_status`. Status information is used to control the colour of nodes in the Xλ-Clam interface and has not really been implemented in this version of λ-Clam (but should be). `roll_back_to_crit Meth Ad` looks for the nearest occurrence of the method, `Meth`, in the plan and deletes all nodes between the current node and the address, `Ad`, of that method occurrence and puts `Ad` at the top of the agenda. `continue_crit C` applies `C` to the continuation at the address to which the critic is applied. This essentially updates the continuation in some way (e.g. changing the next method to be attempted) and places that address at the top of the agenda.

Using these atomic critics and criticals it is possible to build up critiquing strategies using compound critics. A compound critic has 3 slots: the name of the theory, and two critic slots. You can think of these as the "name" of the critic and then the critical expression associated with the name.

Example 10.1 *So, for instance, the critic strategy for rippling is expressed (in the* `wave` *module) as:*

```
compound_critic wave wave_critic_strat
        (then_crit (pop_critic FailAd)
        (then_crit (subplan_crit FailAd (then_crit (wave_failure Failure Rule)
                                         open_node))
                  (wave_patch Failure FailAd Rule))).
```

So the critic `wave_critic_strat` *pops the current address off the agenda. At this address it then applies the critic* `wave_failure` *- this simply analyses the failed preconditions at this address and instantiates* `Failure` *to the precondition that failed* *and* `Rule` *to the rewrite rule being attempted. It then opens the current node and applies the critic* `wave_patch` *which does the work. An example of* `wave_patch` *is the generalisation critic of Andrew Ireland.*

```
compound_critic wave (wave_patch (failed_sink Emb) Ad _Rule)
        (subplan_crit Ad
        (then_crit
                (generalisation_AI Emb Lemma)
                (then_crit (roll_back_to_crit (induction_meth _ _) Address)
        (subplan_crit Address
                (continue_crit
                        (m\ (then_meth (introduce_gen Lemma)
                            induction_critics))))))).
```

This says that in the event of a failed sink (the name of the critic is `(wave_patch (failed_sink Emb) Ad _Rule)` *which tells us something about how the method failed. At the address where the failure occurred (*`subplan_crit`*) we apply the atomic generalisation critic then we roll back the plan (*`roll_back_to_crit`*) to the last occurrence of the induction method and then at that address we apply the* `introduce_gen` *method followed by our* `induction_critics` *method. If you want to know what Andrew Ireland's generalisation critic does see [13].*

Lastly, in `critics.sig` there is a type constructor `user_crit` this is for naming critics at the command line so it takes a string and returns a critic.

10.3.6 plan

The signature for the plan node defines a number of important types: `plan`, `pnode_status`, `agenda`, `plan_state`, `preconditions` and `changed_node`.

`plan` has four constructors `and_node`, `or_node`, `forall_node` and `exists_node`. Of these `or_nodes` and `exists_node` have never been used in anger (`or_nodes` have never been used at all). The intention is that a plan is a tree structure - so `and_node` is the basic constructor representing a node in the tree and its children. An `or_node` is so people can experiment with multiple plans, especially in step-by-step mode - but no one ever did this. `forall_node`s and `exists_node`s are used when a quantifier is removed from a goal and a constant introduced. λ-PROLOG handles this by quantifying the plan. `exists_node`s are going to be important when anyone gets to grips with synthesis.

The `pnode_status` type is only used in conjunction with Xλ-Clam to provide the information needed to colour nodes. As Xλ-Clam is not up and running with this version of λ-Clam yet I suspect all the status information is incorrectly implemented.

There are three constructors for `preconditions`. The type is used for marking which preconditions have succeeded or failed when critics are in use. `nopreconditions` is used for nodes where no method has yet been applied. `success` and `failed` are used for individual preconditions. There are two predicates `failed_pre` and `success_pre` which can be used with a list of preconditions to return one that has failed or succeeded respectively.

Agendas are another aspect of λ-Clam that could use some more thought. Mostly we only use `active_agenda` which contains a list of the nodes currently needing expansion by the plan. `finished_agenda` is used once in the program which is after all planning as finished. `failed_agenda` is used nowhere. I think Jeremy Gow has a cleared idea of plans for the use of agendas than I do.

`plan_state` is used for passing around agendas and "plans". The constructor `pstate` is a hangover from the days when the whole plan was passed around as an argument and is arguably redundant. `pliststate` is a pair of an agenda and a list of plan nodes that need changing.

`changed_node` is a list of nodes that either need to be added to or deleted from the current context.

There are also a number of predicates in the plan module. `retrieve_node` and `retrieve_binder` are used to access the plan stored in the context. They are a little contorted in an attempt to only access the last node stored for any address. `get_open_nodes` is not used anywhere to my knowledge. Similarly get_open_nodes_at. `transform_node` applies a function to a given address in a plan. `rationalise_addresses` is used in the construction of child nodes during method application. It is supposed to work out what the addresses of these child nodes should be. After this there is a commented out function `remove_AllGoals` - this was at Jeremy's request and was used to transform any `allGoal`s into `forall_node`s. I removed it because it took an unreasonable amount of time. In the new context planner it might be worth reinstating this. There are also a selection of predicates (e.g. `get_method`) for extracting information from plan nodes.

10.3.7 Method

The module is pretty straightforward. It defines the methodicals and `user_method` which works like `user_crit`. The only interesting bit here is `change_method_vars` which is used to keep the arguments to methods uninstantiated when they appear within `repeat_meth` expressions. NB. despite various attempts `cut_meth` has never been got to work. Apparently Julian has some clever ideas about this.

10.3.8 Goal

This module defines types for goals, and queries. We distinguish between goals and queries since we use the query type for names of theorems we wish to prove while goals are for the actual syntactic expression in proofs.

The constructors for goals are `trueGoal` and `falseGoal` which, in theory, are used to indicate that a branch of the proof has completed. In practice only `trueGoal` is recognised by the planner. I originally implemented this so that λ-Clam and required something like `triv_meth` to pick them up. I am now convinced this was a design flaw and that the proof should stop the moment `trueGoal` or `falseGoal` appear. The reason I implemented it alternatively was because of the definition of `complete_meth` given in Alan S. and Julian's work – i.e. that `complete_meth` only succeeds if `trueGoal` is reached. Implementing a `trueGoal` finder as well sort of makes the checking in `complete_meth` redundant since the proof stops before it gets that far.

`**` and `vv` are used for conjunctions and disjunctions of goals. Disjunctions of goals are not used anywhere in the system at the moment. Conjunctions are only used in method application and almost immediately split into lists of proof nodes. This is undesirable but I'm not clear exactly what the structure of interaction between conjunctions of goals and list of proof nodes should be.

`allGoal` and `existsGoal` indicate points where a quantifier is introduced into the proof. This is usually as a result of eliminating a quantifier from the syntactical expression of the goal conclusion. Only `allGoal` has been used in anger.

`seqGoal` is slightly different to the other constructors and is used to indicate that an expression is a sequent goal - the standard format for goals. It is similar to the `ripGoal` constructor used in rippling to indicate annotated goals. I am a little uneasy about these. Clearly arbitrarily more such constructors could be introduced in individual theories and potentially we may want some methods to apply to goals with these new constructors. The fertilize method for instance already has cases for both forms of goal which are largely identical except for which input goal they pattern match to. I think the solution may be to introduce destructors for goals which extract the hypothesis list and the conclusion (common to both forms used so far) then instead of pattern matching on `seqGoal (H >>> G)` and `ripGoal (H >>> G) Skel Embedding` in the input slot of a method and then manipulating `H` and/or `G` we could pattern match on a variable `Goal`, say, and then extract the hypotheses and conclusion with the destructor. This would make the could more complex however.

`>>>` is a constructor for `osyn` it is intended to represent the mathematical symbol $\vdash$ and so is used to construct sequents.

Three predicates are also declared: `reduce_goal` and `list2goal` are simple support predicates the first is used to simplify conjunctions and disjunctions of goals involving `trueGoal` the second is used to transform a list of goals into a conjunction of goals. `top_goal` does not actually have any clauses in the goal module. It is the predicate used elsewhere to declare theorems for attempted proof.

10.3.9 Syntax

Our basic structure for syntax consists of constants and variables, function application and λ-abstractions. So a term or expression of type `osyn` can be thought of as:

$$osyn ::= c|v|osyn(osyn)|\lambda x.osyn(x)$$

In λ-PROLOG we declare new constants as expression of type `osyn`, function applications is indicated by the `app` constructor and λ-abstraction by the `abs` constructor. Since λ-PROLOG has λ-abstraction in the language the type of `abs` is `(osyn -> osyn) -> osyn` - i.e. it applies to a function from `osyn` to `osyn` not to an expression of type `osyn` itself.

We have complicated this basic format however. Firstly we have introduced types and type attribution. So we have `otype_of` which associates a term with its type. Term types are also of λ-PROLOG type `osyn` so we have an `arrow` constructor for function types.

We also introduce tuples of arguments, `tuple` (with associated `tuple_type`). This is sugar and has been the source of much debate. Alan Smaill has argued against doing away with it since it is frequently useful when manipulating terms to be able to distinguish between function and argument symbols. I am now in favour of altering the type of `app` to `osyn -> (list osyn) -> osyn` and doing away with the `tuple` constructor by subsuming it within the `app` constructor.

We have a constructor `user_object` for introducing syntax at the command line and "base constants" `obj` and `universe`.

There are a number of support predicates `obj_atom` succeeds if the argument is a constant or variable and fails if it is a function application or λ-abstraction. `subterm_of` succeeds if the first argument is a subterm of the second. It's third argument is the position of the subterm so it can be used to locate a subterm at a given position as well. `replace_at` replaces the subterm of the first argument appearing at the position indicated by its fourth argument with its second argument and returns its third argument (I suggest you look at the code here if you're interested). `copy_term` copies a term but replaces all the λ-PROLOG variables - this is useful if you want to avoid things getting instantiated.

`has_otype` and `env_otype` are used to relate terms to their types. `env_otype` takes an extra argument which is the hypothesis list so it can deduce types involving constants introduced in the hypotheses. `is_otype` has no clauses in this module it is used for declaring the types of constants introduced in theories.

`headvar_osyn` succeeds if the argument is a variable.

10.4 Input and Output

I can't comment on the pretty printing code but I will discuss the basic methodology I have used in implemented input and output mechanisms.

There are potentially a number of ways users will want to interact with the system: at the command line, through our custom Xλ-Clam interface. We also anticipate that external systems may wish to interact with λ-Clam possibly via PROSPER or MATHWEB. This means that λ-Clam needs to be able to receive input and generate output in a number of formats. The module `switch` in the `io` directory defines a type `interaction`. Three values `command`, `command_pretty` and `xbarnacle` have already been defined for this type. There is also a type `switch` which has the values `on` and `off`. There is a predicate `interactive_mode` which returns the current interaction mode being used. This works much as `retrieve_node` described above. The default interaction mode is `command_pretty`. We distinguish between `command` and `command_pretty` because the pretty printer is very memory hungry.

There are a large number of predicates defined in the `print_syntax` module and `print_plan` module which all work much the same way. I shall consider `pprint_term` as an example.

```
pprint_term Term:-
        interactive_mode Interaction,
        pprint_syn_for Interaction Predicate,
        Predicate Term.

pprint_syn_for command_pretty (X\ (pretty_show_term X, print "\n")):-!.
pprint_syn_for _ printstdout.
```

As can be seen `pprint_term` first uses `interactive_mode` to determine the appropriate style of interaction. It then uses the predicate `pprint_syn_for` to retrieve a predicate. If the interaction mode is `command_pretty` then the predicate is one of the pretty printing predicates. Otherwise the predicate is the default `printstdout` which prints the current λ-PROLOG expression to standard out. It then applies this predicate to the term.

The `switch` also defines a predicate `step_by_step_mode` which can return `on` or `off` (again working at `retrieve_node`). This is used in the `plan_support` module to determine whether the planner is working in step by step mode or not. There is also a predicate `plan_step_prompt` which determines what the prompt should be for the next user interaction depending upon the interaction mode.

We don't have any parsers yet so input predicates have not been defined.

10.4.1 Top Level Loop

I've talked in several places about the "top level loop" so I'm going to discuss here what I mean by this and how it's implemented. Relevant code can mostly be found in `lclam.mod`.

By a top level loop I mean the front end shown to the user or an external program. This front end is given an instruction (e.g. "prove this") executes the instruction returning any output to the user or external program and waits for another instructions. So conceptually its control flow looks like figure 10.7 (hence loop) This is actually fairly tricky to automate especially since we want some of the command to change the context (e.g. add new rewrite rules). The basic loop is as follows:

```
lclam:-
```

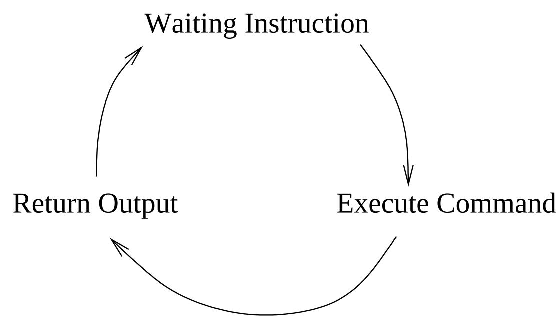

Figure 10.7: The Top Level Loop

```
        pprint_string "lclam:",
        read Command, !,
        execute_command Command lclam, !.
```

(NB. There are two versions of this one for interactive mode `command` and one for interactive mode `command_pretty` – the intention is, I think, to distinguish the situation from xlclam mode but I suspect this could be neatened).

This code prints the prompt

```
lclam:
```

reads in a command and then executes the command - *but* it also passes the name of the looping predicate to `execute_command` (more on this anon). It is this predicate `lclam` that is passed to Teyjus by the lclam script in the `bin/` subdirectory. The "basic" code for `execute_command` occurs at the end of the lclam module and is:

```
execute_command Command Loop:-
        Command, !,
        Loop.
execute_command Command Loop:-
        pprint_string "  Failed.\n",
        Loop.
```

So it can be seen this executes the command predicate and then executes the looping predicate - this is because there are several looping predicates like `lclam` but `execute_command` can be used by all of them. In between are a whole load of "special cases" for `execute_command`. These are all pretty similar so I shall only consider one, by way of example.

The user has a command `step_by_step` which can be set to `on` or `off` indicating whether they want to run in step by step mode or not. Internally in λ-Clam this means they will be altering the current context. Recall that step-by-step mode is used in the plan_support module to determine whether or not to prompt the user for intervention in between each step of the plan construction. This is done by checking a predicate `step_by_step_mode` which returns `on` or `off`. `step_by_step_mode` works by looking at the *first* success of a predicate call `step_by_step_m`. The code which allows the user to control what this first success will be is:

```
execute_command (step_by_step X) Loop:-
        (step_by_step_m X => (pprint_string "Done", Loop)).
```

Here we use λ-PROLOG's `=>` to put `step_by_step_m X` (where `X` will be the user's specified value) in the context and this "most recent" addition will be the first value returned by `step_by_step_m`. λ-Clam will then print `"Done"` and execute the loop predicate again.

I said that `lclam` was not the only possible top level loop predicate. There is also `testloop`. This automatically runs through a series of commands instead of prompting for user interaction between each one. So the code is

```
testloop (H, T):- !,
        execute_command H (testloop T).
testloop H:- !,
        execute_command H lclam.
```

This is much like `lclam` except that it doesn't print prompts or read input and it has an argument which is the commands to be executed. It does have a major flaw though (which is shared by the critics engine, incidentally) the association of the `,` in λ-PROLOG means that you have to explicitly bracket the command list `(Command1, (Command2, (Command3)))` to get it to work properly. Otherwise, given `Command1, Command2, Command3`, λ-PROLOG will match `Command1, Command2` to the `H` in the first clause and `Command3` to the `T`. `testloop` is used in the `benchmark_plan` predicate used in testing.

There is also a third loop predicate in `tjxb.mod` – our first attempt at Xλ-Clam. It is called `xbloop`. Off hand, I'm not sure if this is working.

10.5 Theories

I'm going to go through the theories in alphabetical order. Many have little of interest.

Arithmetic Definitions, rewrite rules and queries for arithmetic. It should be noted that originally the definition of plus and times used recursion on a non-standard argument (i.e. $X+s(Y) = s(X+Y)$ instead of $s(X)+Y = s(X+Y)$) this was so we could overload the operators and use them with the ordinals as well. However when we final wrote separate operators for the ordinals we switched it back. The "traditional" definition makes the proof of exptimes $(X^Y)^Z = X^Y * X^Z$ much harder (try it by hand if you don't believe me) and so we can no longer prove this on arithmetic or though we can still prove the ordinal version.

Casesplit This doesn't actually contain the casesplit method - some rationalisation could probably usefully happen here. This contains support predicate for the casesplit method. It should be noted that the method is only called by critics and it assumes the rewrite rule condition and its negation in the hypotheses lists of two new goals.

Challenges Contains definitions, rewrite rules and queries for the induction challenge corpus. Incomplete.

Clam Corpus Two files. Contains definitions rewrite rules and queries for the clam benchmarks.

Conjecture Tester Methods for testing the truth or falsity of conjectures. Not used anywhere.

Embed Support for embeddings. The main predicate is `embedding`. This follows the theory of embeddings quite closely though there is a certain amount of fancy footwork to try to prevent instantiation of variables. The measure checking code `check_measure_reducing` was a real nightmare to write and is not necessarily bug free. This is (for a change) fairly heavily commented in the code with my various attempts and thoughts. The real problem is controlling the turning inward of outward wave fronts, especially in the presence of critics.

GCD Goal for synthesising the greatest common denominator of three numbers. Proof doesn't work.

Generalise The generalise method. Seeks for common subterms and replaces them with a variable.

Induction This contains fertilisation, step case and induction methods.

The fertilisation method uses a predicate `sink_match_` to handle matching of sinks and universally quantified variables. Alan Smaill has pointed out that this is not well though out in a theoretical sense. In particular many (all?) the proofs using ordinals introduce a universal quantifier all the quantifiers originally appearing in the expression have been stripped off (so that case splitting works). The variable associated with this quantifier is then matched to a sink in the hypothesis even though the other variables matching sinks are free variables. There is some concern that if more than one quantifier were to appear then quantifier order might not be respected. Note that fertilisation calls various `rewrite_using` predicates described in the rewriting module.

The induction method is relatively straightforward since it hides all the nastiness in `all_ripple_analysis`. This returns an ordered list of suggested induction schemes (in theory if not in practice) and then uses `nth` to recurse down this list one at a time and attempt to apply the scheme.

The step case method is highly baroque. It is hard to describe the combination of plan attempts which lead to this but I would recommend considering the plans of `qrevcorrect` – using the set up for `benchmark_plan` from `list_benchmarks` –, `comm` and `memapp` as good litmus tests that everything is still working.

Note also that there are two versions of nearly all the methods in this theory: One using critics and one not. I considered parameterising the methods (since many are essentially duplicates) with whether critics were in use or not but never got round to thinking this through.

list_benchmarks Self-explanatory really. Definitions and theorems for a large number of list based theorems for benchmarking. It has its own version of `benchmark_plan` at the bottom. This is largely intended for testing `qrevcorrect` since it contains the associativity of append as a rewrite rule most of the other are intended for testing on the set up in `map_benchmarks`.

map_benchmarks Definitions etc. involving higher order functions such as map, fold and filter.

mlremovelists Stuff for my ML work.

objlists List theory.

ordinal Ordinal theory. Lots of this is discussed in [9] although as ever this discussion is a little out of date (it neglects to mention the treatment of quantifiers in the step case method).

rewrite_types I separated out the rewriting methods and the types used in them because I wanted all method/rewrite rule stuff to be compiled *after* all the planning code so that the precise composition of theories could be customised. However I found I needed to know the types for rewriting for the pretty printer which is compiled in with the planner. I now think the rewriting specific stuff could be separated out from the pretty printing code and probably should be included with the rewriting module in which case the rewriting and rewrite_types modules could be merged.

rewriting The major method in this file is the `rewrite_with` method. This rewrites a goal with an rule taken from a list specified by the user. If you look in `rewrite_types.mod` you'll see a couple of these set up. The two I've used are `sym_eval_rewrites_list` and `general_rewrites_list`. In practise there is no difference between these. In theory `sym_eval_rewrites_list` should only contain confluent rewrite rules, but this is not enforced in any way.

As well as `rewrite_with` there is an atomic method called `rewrite_with_hyp` which rewrites using rules from the hypotheses. Obviously this has implications for confluence and the like which have not been explored at all.

There are two compound methods `sym_eval` and `general_rewriting`. They both repeatedly apply `rewrite_with` just with different lists of rules so there is little to distinguish between them. `sym_eval` also uses `rewrite_with_hyp` largely because `general_rewriting` is hardly ever used in practice and so has not yet been extended. `sym_eval` should probably be renames `simplify` or some such since in practice it is used whenever a proof strategy is deemed to need to apply some rewriting in order to simplify a goal.

Of the support predicates `congr` is the most interesting and something similar is implemented for rippling. `congr` is higher-order and is intended to apply rewriting to some subterm of a term. It takes as an argument a predicate that is to be used to transform a term. It's other arguments are the rewrite rule used, a polarity argument, the term, the new term and any conditions on the rewrite rule. It then recurses through the term structure until it finds a sub-term to which the predicate applies.

The idea behind polarities are explained in Santiago Negrete's thesis [17]. Essentially the idea is that where a rewrite rule is an implication not an equivalence (e.g. $Y \rightarrow X :\rightarrow (X \vee Y) \rightarrow (X \vee Z)$) care needs to be taken in its application. We reason backwards in conclusions but (should anyone implement it ever) we reason forwards in hypotheses. Clearly if your goal is $\Gamma \vdash (A \rightarrow B)$ then the `imp_i` rule of inference can be used to deduce $\Gamma, A \vdash B$ - so if we apply rewriting we need to be sure that it is forwards in A even though we would normally reason backwards in hypotheses. This is represented in the code although it is confused by the fact that our abstract syntax has required the introduction of `polar_term` to match polarities to the arguments of functions and then there is some work to handle and remove these wrappers at later dates.

`bad_for_synthesis` is a hack put in there because certain rewrite rules e.g. beta reduction, $(\lambda x.F(x))Y \leftarrow:\rightarrow F(Y)$ can fire repeatedly in the presence of meta-variables - e.g. $G(z)$ instantiates the meta variable F to $\lambda x.F(x)$ and rewrites to $F(z)$ and so on. I think maybe the predicate should be declared in this module but the clauses should appear in the theory files where the offending rewrite rules are declared (this might affect compilation order though).

`rewrite_using` and `rewrite_using_prop` are predicates for stripping quantifiers off hypotheses and then using them as rewrite rules. They were originally written for use with the fertilisation method but have been adapted for more general use.

ripple_types My comments on this are pretty much identical to those on the rewrite_types module.

schemes Another nightmare of a module which is probably far from bug free. Induction schemes as they appear in various theory files consist of 7 slots: the name of the theory, the name of the scheme, the type to which the scheme applies, a *substitution* which is intended to indicate in some sense what the scheme does, some sort of expression indicating the measure on scheme (I know nothing about this nor have any idea what it does - I think it's something to do with Jeremy Gow's work) and the goals before and after scheme application. The predicate `ripple_analysis` in the schemes module is the major predicate that deals with all this. In theory ripple analysis considers all possible induction schemes that apply to a goal and orders them. This ordering is based on the number of flawed and unflawed variables and some measure of how "complicated" the scheme is. A flawed variable occurrence is one where the structure introduced by the scheme can not be moved by a rewrite rule. At the moment the measure on the "complexity" of the scheme just looks at how many constructors are introduced by the scheme. Back to the `ripple_analysis` predicate. The predicate starts out by assuming an `empty` substitution. If the goal is universally quantified (giving a candidate induction variable) it picks an induction scheme of the appropriate type. This provides a new substitution which is passed into the recursive step (the new goals are not). Substitutions have two main constructors `dom` and `repl` - `dom` is used to introduce a new constants, `repl` is the constructor that conveys informations expressing what a variable will be replaced by according the the induction scheme. If you look at the second two clauses for `ripple_analysis` you will see these at work. `repl` is used to introduce a substitution for the universally quantified variable, eliminating the quantifier in the process. `ripple_analysis` then recurses, once again provided an `empty` substitution. I am interested to note that `ripple_analysis` never successfully applies schemes to two variables in the one goal - needed for many proofs involving the order on natural numbers for instance - and I suspect the bug may lie here somewhere (in fact the `Max` argument - set to 1 by `induction_meth` looks like a likely culprit to me). `ripple_analysis` can skip over quantifiers (clause 4 of the predicate) and since it is called from within a `findall` this means induction schemes are tried out on all variables. There is a final case where instead of looking for universally quantified variables the scheme is applied to a variable declared in the hypothesis list - i.e. a previously introduced induction variable. This is needed for the proof of the commutativity of multiplication, `comm`. The final clause of `ripple_analysis` is called when all universal quantifiers have been stripped off. At this point the predicate counts the flaws.

`number_flaws` and `number_flaws_` which count flaws are pretty straightforward. The main problem is the need to make copies of the terms, changing all meta-variables to prevent them being instantiated by the rewriting process. This problem only shows up when attempting to prove goals containing meta-variables, e.g. the correctness of qrev which uses the generalisation critic.

At the top of the file is the predicate `all_ripple_analysis` which finds all applicable induction schemes and returns them as a list. This is then sorted using an insertion sort and a member of the list returned. `pick_quantifier` rearranges the quantifiers in the list allegedly so that the scheme applies to the right quantifier. I'm at a loss as to how it knows which quantifier was intended since the number supplied seems to come from the position of the scheme in the ordered list (see `induction_meth`). You can tell I don't really understand all this can't you!!!

Finally another problem is with schemes that have more than one step case (e.g. the scheme used by the ordinal arithmetic methods). The substitution only considers one possible replacement so it is impossible to check whether a scheme is flawed (or unflawed) on all its cases. I think an additional substitution constructor and some extra cases in `ripple_analysis` might solve this.

test_set_gen This is used for generated counter-examples. These are not at present used anywhere and are experimental.

testdatatype This contains a dummy datatype for use with counter-example finding.

wave This contains the contents of two modules joined together. It performs rippling. The `wave_method` itself should be fairly clear, based on the reconstruction of rippling Andrew performed for critics. The preconditions are: Find a wave rule that applies, check any conditions are trivial, check that the skeleton embeds into the new goal, check this embedding is measure reducing, and check it is sinkable.

The other methods in this file are `wave_case_split` which fires only if called by the casesplit critic. This happens if a condition on a rewrite rule was not trivial. This method assumes the condition and its negation in the hypothesis list of two new goals and then continues. I'm not at all sure this method should be here, support predicates for it are in the casesplit module; `set_up_ripple` and `post_ripple` which annotate and strip annotations from goals respectively.

There are a couple of other support predicates, `fun_abs`, which I've never been entirely happy with which is for handling λ-abstractions in skeletons, `measure_check` which uses `filter2` (once of the mapping predicates from the lclam_map module) to check that at least one embedding is measure reducing and return all those that are. `ind_hyp` which attempts to identify candidate induction hypotheses. (NB. In *Clam* I believe that induction hypotheses were explicitly annotated as such, we've not had a need to do this yet but it might be worth considering).

We then have some critics. The `wave_critic_strat` which analyses the failed preconditions and calls the `wave_patch` critic. `wave_patch` behaves differently depending on the precondition that failed. The atomic critic `wave_failure` is the one that analyses preconditions.

`ripple_rewrite` behaves much like `rewrite_with`. The addition is that it allows rewriting in either direction and it attempts, as it goes, to keep track of skeletons and embeddings and specifically preserve as much of the embedding as possible so that less work is later done in constructing and embedding between the new term and the skeleton.

`wcongr_ripple` deconstructs the term attempting to apply a rewrite rule. At the same time it deconstructs the embedding so as to preserve as much of it as is unaffected by the rewrite. This means sorting through possible skeletons and embeddings to reject those that are no longer applicable. This is done using `congr_ripple_skel`. It wouldn't surprise me if this could all be rationalised.

There are also `wave_rules_list` and `wave_rule_list` which work like `retrieve_node` etc. `embeds_list` maps `embeds` down a list and `step_forall_embeds` which replaces all universally quantified variables in the skeleton with wild cards used to represent sinks. `congr_ripple_skel` is part of the `congr`-style predicate `wcongr_ripple` used in rippling. Because rippling happens with reference to a list of skeletons and embeddings (so it can be treated as "coloured" although this has not been implemented) it can happen that some embeddings are excluded from consideration by a particular ripple. The purpose of `congr_ripple_skel` is to partition the list of embeddings into those that are applicable in the current situation and those that are not. `reform_emb` attempt to put these back together but replaces unused embeddings with `dummytree` because, err..., umm - I'll get back on this.

wave_critics This contains most of the wave critic code. Again, I vaguely feel this could all be organised better. At the moment there are only two critics. One for casesplitting and one for generalisation. There is some partially written code hanging around for a putative induction revision critic. The casesplit critic is is pretty straightforward. If the `trivial_condition` precondition fails then it creates two new subgoals one with the condition and one with its negation in the hypotheses list. The generalisation critic (called `generalisation_AI` since other generalisation critics have been proposed - this is the on from Andrew Ireland). Constructs a generalised version of the goal where the induction method fired and rolls back the plan to this point, introducing this generalisation (using the `introduce_gen` method) instead of applying induction.

whisky This contains an expression of the "Whisky Problem" from temporal logic. See Alan Smaill about the details. It also contains some functions I used as benchmarks to check that rewriting/rippling can take place within functions as well as their arguments.

10.6 Testing and Benchmarking

There are two methods for testing out the effects of modifications of λ-Clam in the context of the rest of the system. There is a λ-PROLOG module called test which runs quick checks of parts of the system and a executable called `benchmark` for running λ-Clam over its corpus. These are briefly described here.

10.6.1 Quick Testing

To quickly test the system on a couple of examples type

```
$LCLAM_HOME/bin/lclam
```

at the command line.

Then type :

```
testpds.
```

At the prolog prompt. This should run λ-Clam over a three or four examples giving brief information as to what each one is supposed to test.

10.6.2 Benchmarking

In the `bin/` subdirectory of the λ-Clamis an executable `benchmark`. This will run λ-Clam over all those problems in its built-in corpus that it can currently plan and write the results to a file `report.tex` in a subdirectory `tests` of that the benchmarking is attempted in. Some of these benchmarks run off specific theory files not included in the general build so a benchmark run should be preceded by

```
[prompt] make allclean
[prompt] make benchmarks
```

in the `src` directory. The tutorial subdirectory of the λ-Clam distribution should be added to your `$TJPATH`.

The name of this subdirectory can be altered using the flag

`benchmark -dir` *report_directory*

`benchmark` attempts most of the problems using `induction_critics` with all the definitional rewrite rules and induction schemes available to the system switched on and a few other rewrite rules. NOTE: All the benchmarks should succeed in under 10 minutes when run on methven except `falsearith`, which should fail, and `exptimesord` (which may well run out of memory).

There is also a similar script `fulltests` which will run λ-Clam over the complete corpus.

To add more theorems to the benchmark and fulltests set a developer should edit the file `benchmark` and `fulltests`. The best way to control the rewrite rules etc. available when benchmarking is to write a clause for the `benchmark_plan` predicate in your theory.

```
benchmark_plan theory -> meth -> query -> o.
```

Examples of this predicate can be seen in the `map_benchmarks` theory file and the `ordinal` theory file. The predicate `testloop` can be used to mimic the sequence of commands you would type in at the command line.

The script `lclam_test_lib` takes three arguments: the query name (*query*), the name of the top method to be used (*meth*) and the theory name (*theory*) plus an optional fourth argument `-timeout` *seconds*. It starts up λ-Clam on the theory file `theory.lp` and calls `benchmark_plan theory meth query`. NB. This will thus only work properly if the theory name is the same as the name of the module that contains it.

You should add the following lines to `benchmark` and `fulltests` where `query1`, `query2`, `query3` are the names of your new theorems, `meth` is the top method you want used and `theory` is the name of your theory.

```
foreach x (query1 query2 query3)
echo Trying to prove $x.
($LCLAM_HOME/bin/lclam_test_lib $x meth theory $timeout >& $report_dir/$x.tex ; $LCLAM_HOME/b
sleep $wait
end
```

As default `benchmark` (and `fulltests`) time out each attempt after 5 minutes. This can be altered by setting the flat

`benchmark -timeout` *seconds*

It waits for 30 seconds between each attempt This can be altered by setting the flat

`benchmark -wait` *seconds*

As a default `benchmark` and `fulltests` write their output to the file `report.tex` in the subdirectory `tests`. They also create individual output files for each problem which show the λ-Clam output generated. To change the directory where the output is placed (e.g. if you don't want your previous results overwritten) use the flag `-dir` *dir* with the scripts.

Appendix A

Benchmarking — λClam timings

A.1 λ-Clam v4.0.0 timings, 28th August 2002, Edinburgh

Note that the testing for v4.0.0 was done on one of the DREAM group's standard Sun Ultra 10s and not a Linux box. The two failed benchmarks all_list_cv and allList_and_dist run out of memory at the ripple analysis stage — apparently this is not a problem under Linux.

Run on clunie (sparc), started Wed Aug 28 15:00:20 BST 2002.
Timings in minutes and seconds. Fail means failed proof attempt. N means proof attempt did not fail cleanly (e.g. it gave a segmentation fault).

Theorem	Time (User CPU)	Note
taut1	0m0.16s	
taut2	0m0.26s	
exptimesord	4m21.06s	
expplusord	3m3.96s	
asspord	0m14.05s	
plus0ord	0m3.92s	
distord	0m48.68s	
assmord	1m33.33s	
times1ord	0m6.84s	
times2ord	0m4.73s	
exp2ord	0m8.26s	
times1leftord	0m6.87s	
plus1rightord	0m0.67s	
exp1ord2	0m7.73s	
assapp	0m11.67s	
all_list	0m25.53s	
all_list_cv	N 0m2.24s	
all_list_sy	0m0.50s	
allList_and_dist	N 0m2.19s	
allList_and_dist_cv	0m25.53s	
expplus	9m3.57s	
eqzero	0m0.13s	
simple	0m0.34s	

simple_taut	0m0.14s
assp	0m20.42s
comm	7m57.30s
comp	1m28.31s
falsearith	N 0m41.31s
plus2right	0m9.01s
dist	2m39.78s
distr	4m40.66s
assm	6m41.97s
plus1lem	0m3.33s
doubletimes	1m12.30s
times2right	2m26.68s
plusxx	0m0.54s
doubleplus	1m6.26s
doubletimes2	0m13.75s
app1right	0m6.27s
apporev	1m21.52s
cnc_cons	0m2.82s
cnc_cons2	0m2.50s
comapp	1m42.26s
lenorev	0m24.02s
lensum	0m10.68s
memapp2	0m39.31s
memapp3	1m4.29s
nthmem	3m35.76s
nthnil	0m0.65s
singleorev	0m0.91s
tailorev	0m23.36s
tailorev2	0m1.33s
appatend	0m23.04s
memapp	0m44.48s
simple_sy	0m0.33s
evenodd1	0m11.12s
evenodd2	0m11.12s
geqnotlessp	1m38.64s
greatercons	0m9.79s
greaters	0m7.04s
greaters2	1m36.02s
less_double_mono	1m39.68s
less_double_mono2	1m39.99s
lessleq1	1m37.94s
lessleq2	1m38.52s
lesss	0m7.38s
plusgeq	1m45.41s
minus_pred	0m13.04s
minus_succ	0m14.87s
cnc_pl1	0m30.32s
cnc_s	0m1.86s
dedthm	0m0.89s
evendouble	0m2.26s
evenp	0m24.11s
identrm	0m18.97s

comp2	0m54.40s
plus1right	0m5.94s
leqdupl	1m13.19s
foldapp	0m30.85s
qrev_correct_gen	2m44.84s
times1right	0m6.12s
qrevcorrect	0m59.58s
filapp	0m24.55s
mapcarapp	0m14.20s
mapthm	0m8.50s
lenmap	0m4.26s
lim_plus_thm	2m1.66s
synthesis_thm	4m1.00s
revqrevlemmaplan	0m6.44s
orevqrevplan	0m3.69s
dividesthm	0m1.18s
fun_iter_g1	0m0.73s
fun_iter_g2	0m18.79s
whisky_problem	0m7.94s
whisky_problem2	0m11.44s
whisky_problem3	0m15.84s
whisky_gen_problem	0m5.86s
icT6	0m18.44s
icT7	2m7.60s

A.2 λ-Clam timings, 2nd November 2001, Nottingham

I include these, not because I've done much to alter the scope or speed of the system but to note that it runs *significantly* faster on a Pentium 3 based Linux box than on the MRG groups dual processor Sun.

Run on bartok.cs.nott.ac.uk (i386), started Fri Nov 2 12:37:48 GMT 2001.

Timings in minutes and seconds. Fail means failed proof attempt. N means proof attempt did not fail cleanly (e.g. it gave a segmentation fault).

Theorem	Time (User CPU)	Note
taut1	0m0.05s	
taut2	0m0.08s	
exptimesord	1m21.76s	
expplusord	0m57.26s	
asspord	0m4.01s	
plus0ord	0m0.89s	
distord	0m14.63s	
assmord	0m29.03s	
times1ord	0m1.65s	
times2ord	0m1.17s	
exp2ord	0m1.93s	
times1leftord	0m1.64s	
plus1rightord	0m0.16s	
exp1ord2	0m1.85s	
assapp	0m1.91s	
all_list	0m5.08s	

all_list_cv	0m8.53s
all_list_sy	0m0.10s
allList_and_dist	0m8.51s
allList_and_dist_cv	0m5.08s
expplus	0m45.73s
eqzero	0m0.07s
simple	0m0.08s
simple_taut	0m0.04s
assp	0m1.95s
comm	0m24.58s
comp	0m4.27s
falsearith	N 2m0.75s
plus2right	0m1.15s
dist	0m23.51s
distr	0m18.34s
assm	0m29.97s
plus1lem	0m0.47s
doubletimes	0m10.00s
times2right	0m9.07s
plusxx	0m0.13s
doubleplus	0m9.11s
app1right	0m0.88s
apporev	0m14.38s
cnc_cons	0m0.72s
cnc_cons2	0m0.63s
comapp	0m11.18s
lenorev	0m5.50s
lensum	0m2.22s
memapp2	0m7.66s
memapp3	0m12.90s
nthnil	0m0.17s
singleorev	0m0.24s
tailorev	0m5.45s
tailorev2	0m0.33s
appatend	0m3.93s
memapp	0m8.59s
evenodd1	0m2.43s
evenodd2	0m2.45s
greatercons	0m2.10s
greaters	0m0.88s
lesss	0m0.87s
minus_pred	0m2.90s
minus_succ	0m3.30s
cnc_pl1	0m4.59s
cnc_s	0m0.40s
dedthm	0m0.17s
doubletimes2	0m7.30s
evendouble	0m0.44s
evenp	0m4.63s
identrm	0m3.10s
comp2	0m9.28s
plus1right	0m0.76s

foldapp	0m5.74s
qrev_correct_gen	0m30.47s
times1right	0m1.35s
qrevcorrect	0m10.34s
filapp	0m6.54s
mapcarapp	0m3.29s
mapthm	0m2.30s
lenmap	0m1.11s
lim_plus_thm	0m31.96s
synthesis_thm	0m23.28s
revqrevlemmaplan	0m1.68s
orevqrevplan	0m1.00s
dividesthm	0m0.33s
fun_iter1plan	0m0.12s
fun_iter2plan	1m31.96s

A.3 λClam timings, 21st March 2001

Timings in minutes and seconds. Fail means failed proof attempt. N means proof attempt did not fail cleanly (e.g. it gave a segmentation fault).

Theorem	Time (CPU mins:secs)	Note
plus0ord	6.1	
asspord	46.1	
times1ord	8.8	
times2ord	7.3	
exp2ord	10.4	
times1leftord	8.9	
plus1rightord	0.7	
exp1ord2	10.0	
eqzero	0.2	
simple	0.4	
simple_taut	0.1	
distord	2:26.6	
assp	10.2	
falsearith	Fail 0.8	
plus2right	1.0	
comp	16.7	
plus1lem	0.8	
dist	31.9	
doubletimes	7.2	
comm	1:11.2	
expplus	2:04.0	
assapp	10.9	
all_list_sy	1.2	
all_list	17.1	
times2right	39.7	
all_list_cv	23.5	
assm	1:03.1	
allList_and_dist	23.5	
allList_and_dist_cv	17.1	
exptimes	3:08.4	

app1right	6.3
distr	1:31.7
assmord	4:50.1
cnc_cons	3.9
cnc_cons2	3.3
allList_or_left	52.5
allList_or_right	52.6
apporev	1:06.5
lenorev	24.3

comapp	47.1
nthnil	0.8
lensum	37.0
memapp2	27.5
singleorev	1.3
memapp3	44.7
tailorev2	1.9
tailorev	25.1
appatend	22.7
expplusord	9:12.1
exptimesord	N
memapp	30.3
filapp	37.1
mapcarapp	20.1
mapthm	15.3
lenmap	5.9
revqrevlemmaplan	10.0
orevqrevplan	6.3
dividesthm	1.4
taut1	0.1
taut2	0.2

Comment. Extended benchmarking set and removed all the failures to a new fulltests set, all attempted using induction with critics and it is pleasing to note that this hasn't had a drastic effect on time or success rate.

A.4 λClam timings Prolog Mail

For comparison here are the timings for λ-Clam.v2.0.0 which was written in prolog mali.

allList_and_dist	11.5
allList_and_dist_cv	18.6
allList_omember	4.1
allList_or_left	2.2
allList_omember_imp1	Fail 40.6
all_list	18.5
all_list_cv	11.6
all_list_sy	3.9
allList_or_right	Fail 2:18.4
assapp	5.4
assm	29.0
assmord	45.5
assp	4.6

assp_sy	Fail 20.5
asspord	6.5
comm	Fail 11.1
comp	3.6
dist	26.9
distord	22.8
distr	29.2
expplus	56.1
falsearith	Fail 1.0
leqrefl	1.0
expplusord	1:06.7
leqreflord	1.4
exptimes	N 1:05.5
leqsuc	1.1
leqsucord	Fail 3.9
exptimesord	N 1:26.9
plus2right	1.4
simple	0.9
simple_sy	1.4
simple_taut	0.3
times1ord	2.1

A.5 Remarks

It should be noted that the new versions of λ-Clam are approximately 2 to 3 times slower than the Mali version. However the 29th November version is less memory hungry which allows it to plan exptimesord without running out of space. This seems to have degraded somewhat by the most recent version but we should attempt to recover it.

Note also that we can not yet manage synthesis proofs.

Appendix B

Working with λ-Clam within the Dream Group: The CVS Server

B.1 Introduction

We have had a serious problem in the past that each member of the Dream Group essentially worked on and developed their own version of *Clam*. In this way interesting and important additions, such as critics or relational rippling, never found their way into the main distribution.

We are anxious that a similar situation doesn't arise with λ-Clam and this guide is intended to set out some procedures for preventing this.

B.2 Trying Out the System, Running Case Studies with existing Methods and Strategies

It should be possible to run λ-Clam on your own theorems without having a personal version of the system or compiling any λ-PROLOG files yourself. This will done using λ-Clam's library mechanism[1] or the command line options for adding theory information discussed in chapter 4. If at all possible we would like to encourage users to work with λ-Clam through this mechanism. In this way any work they do consists solely of new library files which can be easily included in the distribution. This work can be done using the centrally installed executable for the latest version of λ-Clam[2].

This method of working should be sufficient for anyone experimenting with the existing capabilities of the system to see what it can do.

B.3 Developing Complex Methods, Heuristics and Theories

To customise λ-Clam with your own predicates you will need a copy of the source code in your own directory space. This doesn't mean, however, that you have to

[1] Not yet implemented

[2] There is currently no centrally installed version but there will be for version 0

work on it in isolation or that you should make changes to the existing files in the distribution.

Firstly, unless you do so in consultation with one of the λ-Clam developers[3], you should not edit any files in the distribution unless it is a Makefile. In this way all modifications made by individuals should appear in separate files which can be imported wholesale into the main distribution without hundreds of minor edits being needed throughout the system.

If you anticipate that you will be working with λ-Clam in this way for a significant period of time then you should seriously consider getting hold of the source code through CVS. This means that you can keep track of any changes being made by the developers on a regular basis so that at the end of your work your new code still works with any changes that may have been made with λ-Clam in the meantime. This stops you ending up with files that only work with a now defunct version of λ-Clam. There are a number of CVS guides around which will tell you how to use it, including a quick crib at the end of this appendix. The important thing for work of this nature is that you don't necessarily have to include your personal work in the main source directories that everyone sees until you feel ready to do so. You can however update your λ-Clam source files regularly from the main directory so that you keep track of any changes other people are making. Regular updating by users and hence regular bug reporting also allows the developers to identify quickly if they have made a change which has introduced bugs.

B.4 Developing the Core of λ-Clam

There shouldn't be a large number of people who have reason to change the core λ-Clam files. It should mainly be the two RAs, Alan Smaill and Gordon Reid, plus people working on the induction competition etc. Developers should try to update and commit their work regularly. We should probably also aim for something like 6 monthly releases of the code so that users not working on code in the CVS repository still working on pretty much up-to-date version of λ-Clam.

B.5 Undergraduates, MSc Students, Researchers visiting for less than 6 months

Hopefully many of people on short projects will be able to work with library files alone, in which case they need never worry about acquiring a copy of the code for themselves.

It is unreasonable to ask someone who will only work with the system for a few months to get to grips with CVS, nor will they want to cope with the vagaries of constantly changing code.

If you fall into this category and you need to write files for compilation you should take a snapshot of the most recent version of λ-Clam in the directory in `/usr/local/reason/dream/meta-level/lambda-clam/` and work with that. Please bear in mind the notes above about only altering specific files without consulting the developers.

At the end of the project get in touch with either of the two RAs about merging your work back into λ-Clam.

[3]One of the two RAs on the project, the computing officer or Alan Smaill

B.6 PhD Students, Researchers on long term projects or those wanting to work with the most up to date version of the system

If you are a PhD student or a researcher with a long term project (not counting those who can manage with library files alone) then you should either ensure that you take a new snapshot of the code every time there is a release or you should work with the CVS repository. While this second option makes you more vulnerable to changes in the code it makes it much more likely, at the end of your research, that the work you have done will be consistent with the the current version of λ-Clam and that λ-Clam will be consistent with your work. This makes it much more likely that your work will get out and be used by other people.

Working with the current version in CVS also means that you can get specific bug fixes within hours of them being made instead of waiting for the next release. In some cases this means you will be able to receive bug fixes in a very short space of time after first discovering the bug.

If using CVS, there is no need to put your work into the main distribution until you are ready. All that is required is that you regularly (say once a month) update the files in your directory (using `cvs update -d` - see appropriate CVS manual). This means that any changes the developers have made in the meantime will propagate themselves into the code you are using. At the end of your time using λ-Clam you simply type `cvs commit` to put your work into the main distribution for others to see and use.

We realise this is not an ideal situation. We would like to get to a situation where anyone developing strategies, methods, heuristics and theories can work through library files alone from a centrally installed version of λ-Clam. At the moment this simply isn't possible. While this system will undoubtedly cause many short term headaches we hope that in the long run it will mean less work for all concerned.

B.7 CVS Crib

CVS works on the principal of editing files in your own private working directory and then merging them back into a central repository. This is done by the use of the commands `cvs commit` and `cvs update`. `cvs commit` commits any changes you have made in your working directory back into the main tree, while `cvs update` merges any changes in the main tree into the files in your working directory. There are subsidiary commands to add and remove files from the CVS repository and to acquire a working directory (these are outlined in the crib below).

CVS commands generally work on everything in the directory where they are invoked and all its subdirectories. This can seem a little strange at first. E.g. typing `cvs update` in some sub-directory of the module will update all the files in that directory and its subdirectories, but will not update anything in any higher-level directories.

To use CVS you need to set the enviroment variable `CVSROOT`. For local Edinburgh this should be set to `/usr/local/reason/dream-src`.

B.7.1 Important CVS Commands and Flags

Acquiring a Working Directory

`cvs get` will copy a snapshot of the current state of the program in the repository into your filespace. For example

```
cvs get lambda-clam-teyjus
```

will copy into your directory the current working version of λ-Clam on the main branch.

Note that `get` is used to create directories. The top-level directory created is always added to the directory where checkout is invoked, and usually has the same name as the specified module.

If you do not intend to use cvs for working on your files. I.e. you don't intend to update or commit but will simply email your changes at the end of your time working on them to the developers or you do not want your changes to get into the main distribution, then you can acquire a snapshot of the working directory using `cvs export`. This will not allow you to use commit or update.

If you want to abandon a working directory for whatever reason and no longer use it for committing or updating you should type `cvs release .` in the directory you intend to abandon.

Members of the Dream Group who are not based at Edinburgh University can get a cope of λ-Clam from the Edinburgh CVS repository by typing

```
cvs -d :pserver:anonymous@broom.dai.ed.ac.uk:/usr/local/reason/dream-src
 login

cvs -d :pserver:anonymous@broom.dai.ed.ac.uk:/usr/local/reason/dream-src
 checkout lambda-clam-teyjus
```

The first of these will prompt you for a user name and password. The password is currently empty (I'm sure Gordon will change this soon though!!)

You will not be able to commit changes this way but will be able to track updates.

Committing Changes

Use `cvs commit` when you want to incorporate changes from your working source files into the general source repository. `cvs commit` will work recursively through all the files and subdirectories from the one in which it is typed. It will only commit changes in files and directories which *already exist* in the source repository. New files and directories should be added first by `cvs add`. Similarly files and directories you have removed should be removed by `cvs remove`.

`cvs commit` verifies that your files are up to date. I.e. that no one else has committed any changes since you last updated. If they are not up to date it will notify you and then exit without committing. You will need to do an update before you can commit your files.

When all is well, an editor is invoked to allow you to enter a log message that will be written to one or more logging programs and placed in the source repository file. You can control the editor used by setting the environment variable `CVSEDITOR` or invoking `cvs commit` with the flag `-e editor`.

Updating your working Directory

After you've run checkout to create your private copy of the source from the common repository, other developers will continue changing the central source. From time to time, when it is convenient in your development process, you can use the `cvs update -d` command from within your working directory to reconcile your work with any revisions applied to the source repository since your last checkout or update.

The `-d` option to creates (or deletes) any directories that exist in the repository if they're missing from the working directory. (Normally, update acts only on directories that were already enrolled in your working directory.)

Adding a file to the Repository

Use `cvs add` to create a new file or directory in the source repository. The files or directories specified with add must already exist in the current directory (which must have been created with the checkout command).

For example

```
cvs add mytheory.mod
```

schedules your file `mytheory.mod` to be placed in the repository next time you commit some changes. After this, anyone typing `cvs update` will get a copy of your file. Similarly if you have added a new directory you can add it in the same way.

NB. The file will not appear in the source repository until you commit it.

Removing a file from the Repository

Use `cvs remove` to remove a file you have deleted from the source repository. As with `cvs add` The files are not actually removed until you apply your changes to the repository with commit.

B.7.2 Final Word

This is only an overview of the major aspects of working with CVS. You can find much more detail in the man pages or in various guides and textbooks.

Appendix C

Credits

λ-Clam is a collaborative effort, and many people who are not listed below have participated in some way. Apart from ourselves, the major developers are: Alan Smaill (original version of λ-Clam, many of the existing queries and rewrite rules), Mike Jackson (graphical user interface), Andreas Schöter (early λ-Clamdevelopment), Ian Green (recursion analysis), David Lacey (conditional rewriting), Julian Richardson (persistent proof plan datastructure, many months development on λ-Clam 2), Jeremy Gow (critics mechanism), Ewen Maclean (nonstandard analysis), Ewen Denney (theory compiler), Jürgen Zimmer (MATHWEB-SB integration), and Claudio Castellini (temporal logic prover) as well as other members of the *DREAM* group for actual and suggested changes. Special thanks are also due to Gopalan Nadathur and Chuck Liang for their help with, and development on, Teyjus Prolog.

Bibliography

[1] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[2] A. Bauer, E. Clarke, and X. Zhao. Analytica — an Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning (JAR)*, 21(3):295–325, 1998.

[3] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *14th International Conference on Automated Deduction*, pages 252–255. Springer-Verlag, 1997.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[5] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[6] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *AI*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[7] O. Caprotti and A. M. Cohen. Draft of the Open Math standard. The Open Math Society, `http://www.nag.co.uk/projects/OpenMath/omstd/`, 1998.

[8] E. Clarke and X. Zhao. Analytica – A Theorem Prover for Mathematica. Technical Report CMU//CS-92-117, Carnegie Mellon University, School of Computer Science, October 1992.

[9] L. A. Dennis and A. Smaill. Ordinal arithmetic: A case study for rippling in a higher order domain. In P. Jackson, editor, *Theorem Proving in Higher-Order Logic*, 2000.

[10] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L-H. Eriksson et. al., editor, *Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 135–61. Springer-Verlag, 1992.

[11] A. Franke and M. Kohlhase. System description: MathWeb, an agent-based communication layer for distributed automated theorem proving. In Harald Ganzinger, editor, *Proc. of the 16th Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 217–221. Springer Verlag, 1999.

[12] A. G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1978.

[13] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available from Edinburgh as DAI Research Paper No 716.

[14] M. Kohlhase. OMDoc: An open markup format for mathematical documents. Seki Report SR-00-02, Fachbereich Informatik, Universität des Saarlandes, 2000. `http://www.mathweb.org/omdoc`.

[15] David Lacey, Julian Richardson, and Alan Smaill. Logic program synthesis in a higher-order setting. In J. Lloyd et al, editor, *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 87–100. Springer-Verlag, 2000.

[16] E. Maclean, J. Fleuriot, and A. Smaill. Proof-planning non-standard analysis. In *Proc. of the 7th International Symposium on Aritifical Intelligence and Mathematics*, 2002.

[17] S. Negrete. *Proof planning with logic presentations*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, May 1996.

[18] J. D. C. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with lambdaclam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.

[19] A. Smaill and I. Green. Higher-order annotated terms for proof search. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland, 1996. Springer-Verlag. Also available as DAI Research Paper 799.

[20] T. Walsh. Proof Planning in Maple. In *Proc. of the CADE-17 workshop on Automated Deduction in the Context of Mathematics*, 2000.

Index