The Clam Proof-Planner

User Manual

and

Programmer Manual

Clam version 2.8.4, July 2006

The DReaM Group

Mathematical Reasoning Group
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh, Scotland

Abstract

This note describes the Clam proof-planning system. It is intended as a user manual for people who want to use Clam without knowing too much about the insides, and as a programmer manual for people who want to change and improve Clam. For quick reference, the note provides appendices which summarise the most frequently used predicates in Clam.

If you are only an interested reader and do not intend to actually use Clam on a machine, then you should read only sections1 on page 1 (Introduction) and2.1 on page 9 (The Methods). This will give you a general idea of the capabilities of Clam.

If you are a novice user and want to start playing straight away without ploughing through 50 pages of manual, read section1 on page 1 (the introduction), and section3.3 on page 92 on the library mechanism.

Acknowledgements. Clam is the result of a collaborative effort between a number of members of the DReaM group. The main ideas originated from Alan Bundy and were first implemented by Frank van Harmelen; Frank also wrote the first version of this manual. Alan Smaill contributed through many suggestions and discussions, and wrote some of the methods and tactics, Jane Hesketh wrote some of the early tactics, and Geraint Wiggins was adventurous enough to be one of the first users. Andrew Stevens, Andrew Ireland and Ian Green developed Clam over the years. Andrew and Ian made extensive contributions to this manual.

In addition to the above developers, the following people have made contributions to the current Clam release: David Basin, Richard Boulton, Jason Gallagher, Predrag Janičić, Helen Lowe, and Santiago Negrete. Other users and contributors include Francisco Cantú Ortiz, Ina Kraan, Raúl Monroy and Julian Richardson. This work is supported in part by EPSRC grants GR/H/23610, GR/J/80702 and GR/M/45030.

Contents

 CONTENTS

Chapter 1

Introduction

1.1 The purpose of this document and how to read it

This document describes the Clam proof-planning system. It is intended as a user manual for people who want to use Clam without knowing too much about the insides, and as a programmer manual for people who want to change and improve Clam. These aims are of course sufficiently conflicting to warrant two separate documents, but the state of flux of Clam and related systems means it will be difficult enough to maintain one document, let alone two. In order to satisfy both goals in one document, this document is separated into two parts. Part I, which is a *User Manual*, contains the information that will be needed by new users who want an introduction to Clam and by people using the system without wanting to know too much of the inside. Part II, the *Programmer Manual*, contains more information about the insides of Clam and is useful for people who want to change and improve Clam.

For quick reference, appendices summarise the most frequently used predicates in Clam.

If you are only an interested reader and do not intend to actually use Clam on a machine, then you should read only chapter A and sections1 and2.1 on page 9. This will give you a general idea of the capabilities of Clam.

1.2 What is Clam?

Clam is an implementation of the notion of *proof-plans.* It is built on top of the Oyster proof development system. Oyster is an interactive environment for developing proofs in Martin-Löf's Intuitionistic Type Theory[1], and is a redevelopment of the Nuprl system that was implemented at Cornell University. Clam is an extension of Oyster to support the idea of proof-plans. It provides a representation mechanism for *methods*, and provides a language for formulating methods. It also provides a number of *planners* which allow the automatic construction of proof-plans out of combinations of methods. Corresponding to each of the methods Clam has a *tactic* which allows the execution of the method, and consequently the execution of a plan consisting of individual methods.

Clam's development started in September 1988, and the first version version of Clam was documented and available for general use in February 1989. As is to be expected, Clam is in a state of flux and will no doubt change frequently and

[1] For literature references, see section1.4 on the following page.

drastically in the near future. However, the current version to which this document applies (version 2.8.4) will remain available unchanged until a new version will be released, together with a new version of this document.

Since Clam is built on top of Oyster, the mode of interaction is the same as in Oyster: Users type commands to the Prolog interpreter, and some of these commands will be special Oyster or Clam commands. Everything that holds for Oyster will also hold for Clam.

This document expects both Clam and Oyster to be installed on your system as top-level commands, so that typing `clam` or `oyster` to your operating system will start up the corresponding system.

The current implementations of both Clam and Oyster run under

- Quintus Prolog (tested with versions 3.1, 3.2 and 3.3).
- SWI Prolog (version 2.7).
- SICStus Prolog, versions 2.1 and 3.

1.3 Required knowledge

This document is written under the assumption that readers will have a basic knowledge about certain topics. If you lack this knowledge, then section1.4 will tell you where to go and look things up before you continue reading this document or using Clam. Both this note and the Clam program assume knowledge about the following topics:

- A familiarity with Prolog as a programming language.
- (To a certain extent) Martin-Löf type theory, and in particular the version of it used in the Cornell Nuprl system and its Edinburgh derivative Oyster.
- (To an even lesser extent) ability to use Oyster. In particular, I expect you to be familiar with the syntax used in the Oyster logic, with inference rules of the Oyster logic, and with the following predicates:

```
create-thm/2       load-thm/2         save-thm/2
create-def/2       create-def/1       add-def/1
save-def/2         select/[0;1]       pos/[0;1]
top/0              up/0               down/[0;1]
next/[0;1]         display/0          snapshot/[0;1]
goal/[0;1]         hypothesis/1       hyp-list/[0;1]
refinement/[0;1]   status/[0;1]       extract/[0;1]
eval/2             autotactic/[0;1]   universe/[0;1]
apply/1            repeat/1           then/2
try/1              complete/1.        idtac/0
```

- The general notion of proof-plans, methods and tactics.

1.4 Related reading

Below are a number of references that will supply more information about the topics mentioned above:

- [12] for a basic introduction to Prolog.

- [29] for a more advanced book on Prolog.
- – [24, 11] for the reference manual of Quintus and SICStus Prolog..
- [20] about Intuitionistic Type Theory in general
- [13] about the version of this logic used in the Nuprl and Oyster systems.
- [4] for a general introduction to the notion of proof plans (Notice however that much of the technical details of that paper are now out of date, but the original ideas still stand firmly).
- [17] for a gentle introduction to Oyster.
- [18] for detailed information about Oyster.
- [5] for a short overview of the Oyster-Clam systems.
- [7] for early experiments using Clam to construct inductive proof-plans.
- [8] for a detailed analysis of one of the important methods in Clam.
- [10, 6] for a detailed description of the concept of wave-rules.
- [3] for early work in the field of automated inductive theorem proving.

1.5 Structure of this note

The rest of this document has the following structure. Part I is the *User Manual* and describes:

- The mechanism for representing methods and the language that can be used for formulating them, plus the methods that are currently implemented in Clam (section2.1 on page 9).
- The mechanism for storing methods and submethods (section4.6 on page 117).
- A number of planners that can be used to build proof-plans out of these methods (section2.4 on page 71).
- The tactics that can be used to execute proof-plans (section3.1 on page 87).
- A number of utilities that make daily life with Clam bearable, such as a pretty-printer, a tracer and a simple library mechanism (section3.2 on page 88).

Part II is the *Programmer Manual*, and contains more technical information about the insides of Clam:

- The representation of induction schemes. In the Oyster logic it is necessary to justify induction schemes and this is done by proving a higher-order theorem of the appropriate form (section4.1 on page 109).
- The mechanism for constructing iterative methods (section4.2 on page 113).
- The mechanisms used for storing theorems, lemmas, definitions etc. (section4.3 on page 114).
- A number of utilities that make life as a Clam programmer bearable (section6 on page 121).
- AppendixC.1 on page 143 describes the organisation of Clam's source code.
- AppendixC.2 on page 144 describes the changes that were made in each release of Clam.

1.6 Notation

I have tried as much as possible to be consistent in the use of different type-faces, etc. Normal text will be in normal Roman font, except where new terms get introduced, or where emphasis is needed, when I use *italic Roman font*. Whenever I refer to pieces of Prolog code or type theory (either whole predicates, or terms or variables), I use `typewriter font`.

However: Because underscores _ are a pain to print in LaTeX, I have used hyphens `-` instead of underscores in many places. Underscores occur in the Prolog code inside functor-names and as anonymous variables. It should always be clear from the context when a `-` in the manual is actually a _ in the code.

Predicates are denoted by `f/n`, which stands for the Prolog predicate `f` of arity n. The notation `f/[n;m]` stands for the Prolog predicates `f` of arity n and m.

Predicate documentation is headed by the name of the predicate which is surrounded by horizontal lines. All references to predicates are included as entries in the predicate index on the last pages of this note. Many other key words are listed in the normal index printed on the pages just before the predicate index. The defining entry for a predicate is distinguished in the predicate index by an underlined page number.

Whenever a feature of Clam is discussed that is not to my liking, and which is a prime candidate for improvement, a frowny symbol will be printed in the margin, like here.

1.7 Version control

Clam is large programming project. There are many versions of Clam, some of which are fixed and some of which are evolving slowly, and some of which are highly experimental and liable to sudden change. In some cases, radically different concepts and implementation paradigms have to be integrated into a particular Clam version and these changes are typically done with some degree of overlap with older, perhaps buggy code.

In an effort to control the multiplicity of versions, the Mathematical Reasoning Group has instituted a version control system for Clam. This allows us to retrieve, compare and develop Clam versions. The version control system is called 'CVS'; some information on using CVS and Clam is given in AppendixC.2.1 on page 145.

It is useful to say a little about version numbering so that users have a picture about what is and what is not assigned a version number, and how that number can be used to describe a particular Clam system.

First, there is a top-level Clam 'version' number: it has the form $R.I.P$ where R is a release number, I is an instance of that release and P is the patchlevel of that instance. When Clam starts, it prints the RIP number. This number uniquely identifies all parts of Clam: the predicate `clam-version/1` may be used to show the RIP version.

Release numbers increase slowly over time and would correspond to significant alterations in Clam's architecture, operation, or user-interface, for example.

Instance numbers are increased for significant changes that are not large enough to warrant a new release: for example, a new collection of methods, or an essentially different implementation of some existing technique may well receive a new instance number.

Patchlevel increases are for all other changes: small extensions, bug-fixes, and so on.

In addition, the many files of which Clam is composed each has a version number, but it is important to note that this version number is not particular to any single

RIP (e.g., two Clam's with different RIPs may share a file with the same version number) and furthermore, the version numbering of files is not of the form $R.I.P$. Please do not confuse the two. File versions may be retrieved using the predicate `file-version/1` of all source files constituting Clam (at the moment this does not yet include methods and submethods).

Part I

User Manual

Chapter 2

Proof-planning and methods

2.1 Simple Methods

Methods are the basic stuff that make up proof-plans. They are specifications of tactics, which are procedures which execute a (large) number of proof steps as a single operation. As described in [4], a method is a structure with 6 "slots":

1. A *name-slot* giving the method its name, and specifying the arguments to the method.
2. An *input-slot*, specifying the object-level formula to which the method is applicable.
3. A *preconditions-slot*, specifying conditions that must be true for the method to be applicable.
4. A *postconditions-slot*, specifying conditions that will be true after the method has applied successfully.
5. An *output-slot*, specifying the object-level formulae that will be produced as subgoals when the method has applied successfully.
6. A *tactic-slot*, giving the name of the tactic for which this method is a specification.

These structures are represented as a Prolog `method/6` term. Each of the slots corresponds to an argument of the `method/6` term, in the order listed above. The general form of a `method/6` term is shown in figure 2.1.

1. The *name-slot* is a Prolog term of the form `name(...args ...)`, corresponding to the name and the arguments of the method.

```
method(name(...Args...),      % name slot: Prolog term
       H==>G,                 % input slot: sequent
       [...Preconditions...], % preconditions-slot: list of conjuncts
       [...Postconditions...],% postconditions-slot: list of conjuncts
       [...Outputs...],       % output slot: list of sequents
       tactic(...Args...)     % tactic slot: Prolog term
      ).
```

Figure 2.1: The general form of a `method/6` term.

2. The input-slot is a Prolog term that should unify with the sequent to which the method applies. Sequents are represented as terms `H==>G` where `H` unifies with the hypothesis-list of the input sequent and G with the goal of the input sequent.

3. The preconditions-slot is a list of Prolog goals, each of which should succeed after the input-slot has been unified with the input sequent. A method is said to be *applicable* if the input-slot unifies with the input-sequent and all of the preconditions are true.

4. The postconditions-slot is a list of Prolog goals, specifying properties that will hold after the method has applied successfully. It is an error when the postconditions do not succeed if the method is applicable.

5. The output-slot is a list of sequents that are the subgoals which remain to be proved after the method has been applied to the input sequent. Again, each of these output sequents is represented as a term `H==>G` with `H` representing the hypothesis list and `G` representing the goal of the sequent.

6. The tactic-slot is a Prolog term of the form `tactic(...args ...)`, corresponding to the name and the arguments of the method. Although this is not strictly necessary, at the moment the name of a method and the name of the corresponding tactic are identical. Thus, the name-slot and the tactic-slot will be the same Prolog term. This is not strictly necessary but makes execution of proof-plans easier, since we don't have to dereference the method-name to the tactic-name, because they are the same.

Thus, a method defines a mapping from an input sequent to a list of output sequents. Notice that this is different from the discussion in [4], where methods were described as mappings from formulae to formulae.

The applicability of a method is specified by the input- and preconditions-slots, and the results of a method are specified by the output- and postconditions-slots. The input- and output-slots are *schematic representations* of conditions on the input and output of a method, and the preconditions- and postconditions-slots are *linguistic representations* of conditions on the input and output of a method.

An example of a particular method (the `eval-def/2` method, which will be further discussed in §2.3.1 on page 42) is shown in figure2.2 on page 84.

All slots can share Prolog variables. In particular, variables which are bound while unifying the input-slot with the input sequent can be referred to in the pre- and postconditions-slots (and of course in the other slots if so desired). Similarly, variables which become bound during the execution of the pre- and postconditions can be referred to in the subsequent slots. In order to understand the binding rules for Prolog variables in the slots of a `method/6` term it is important to understand how these slots are used when a planner tests for the applicability of a method:

1. First the input-slot is unified with the input sequent.

2. Then the preconditions-slot is evaluated.

3. Then the postconditions-slot is evaluated.

4. Then the output-slot is constructed using the variable bindings resulting from the preceding operations.

Thus, the preconditions-slot will never be evaluated without the input-slot having been unified with the input sequent. Similarly, the postconditions will never be evaluated without the preconditions having been evaluated.

An important distinction can be made between terminating and non-terminating methods. A method is said to be *terminating* if it does not produce any further subgoals (i.e., its output slot is the empty list). If a method produces further subgoals (i.e., its output slot is a non-empty list) it is said to be *non-terminating.* It is encouraged programming style to indicate as much as possible in the formulation of a method whether or not the method is terminating. Thus, if at all possible, output-slots should not consist of just a Prolog variable, which will be instantiated to a (possibly empty) list of sequents after evaluation of pre- and post-conditions. Instead, an output-slot should be a term which indicates whether the list of output sequents can possibly be empty or not. An example of this usage is shown in the `eval-def/2` method of figure 2.2, where the output slot is indicated to be a non-empty list of sequents. This makes it much cheaper to recognise this method as a non-terminating one than when the output list could only be computed after evaluation of preconditions and postconditions.

2.2 The method language

Although it is possible to use arbitrary Prolog code in the formulation of the pre- and post-conditions slots, users should only use a designated language for this purpose. Part of the goal of the whole proof-plan enterprise is to try and formulate a good language for controlling proof search, and this language should exactly be the one used inside the method-slots. This *method language* consists of a set of predicates and a set of logical connectives. Below we first discuss the predicates of the method language, followed by a discussion of the logical connectives of the method-language. Note that the definitions below extend and override the earlier definitions from [4].

In the specification of a predicate, variables are annotated with mode annotations: +, − or ?. This notation is borrowed from the Quintus Prolog Reference Manual and the following is freely copied from [24]:

`+` This argument is an input to the predicate. It must initially be instantiated or the predicate fails (or behaves unpredictably).

`-` This argument is an output. It is returned by the predicate. That is, the output value is unified with any value which was supplied for this argument. The predicate fails if this unification fails.

`?` This argument does not fall into either of the above categories. It may be either input or output, and may be instantiated or not, as required by its application.

2.2.1 The method language: predicates

The predicates are listed in alphabetical order.

`active-inductive-hypothesis(V:H,Hs)`

`V:H` is an *active* hypothesis appearing in hypothesis list `Hs`. An active hypothesis is one which is a viable inductive target: it has status `raw` or status `notraw(_)`. (cf. `induction-hypothesis/[3;5]`). See §4.5 on page 116 for a more information on hypothesis status.

`adjust-existential-vars(+EV,+Bs,-NewEV,-S)`

`EV` is an association list with elements of the form `Term-Var:Typ` where `Term` denotes the instantiation for the existential variable `Var` of type `Type`. The instantiation may be partial so additional existential variables may be introduced. To prevent the introduction of name clashes the list of current bindings, `Bs`, is required. `NewEV` and `S` denote the refined list of existential variables and the substitution list for the partially instantiated existential variables respectively.

`annotations(?SinksSpec,?FrontsSpec,+T1, ?T2)`

`T1` and `T2` are identical except that all meta-level annotations are absent from `T2`. Wave-fronts and holes of `T1` are those indicated in `FrontsSpec`; sinks of `T1` are indicated in `Sinks` (using the representation described in `wave-fronts/3` and `sinks/3`) and in §4.4 on page 116.

`ann-exp-at(+TopLevel, ?Flag, +Exp, ?Pos, ?SubExp)`

Expression `Exp` contains `SubExp` at position `Pos`. `TopLevel` and `Flag` are either `in_hole` or `in_front`, indicating outermost annotations being exclusively wave-fronts *or* exclusively wave-holes, respectively. `TopLevel` refers to the status of `Exp`; `Flag` refers to the status of `SubExp`.

```
?- ann-exp-at(in_hole,in_hole,‘‘f(x,{y})’’, [2], y)
```

succeeds, but

```
?- ann-exp-at(in_hole,in_front,‘‘f(x,{y})’’, _, y).
```

fails since `y` is not in a wave-front, it is in a wave-hole. Similarly,

```
?- ann-exp-at(in_front,in_front,f(x,{y}), [], f(x,{y})).
?- ann-exp-at(in_front,in_front,f(x,{y}), [1], x).
```

both succeed, but

```
?- ann-exp-at(in_front,in_front,f(x,{y}), _, y).
```

fails.

(See `exp-at/3` for more details of positions etc.)

`ann-exp-at(+Exp, ?Pos, ?SubExp)`

Is a equivalent to `ann-exp-at(in-hole,in-hole,Exp, Pos, SubExp)`.

`cancel-rule(?Exp,?RuleName:?Rule)`

`RuleName` is the name of an equation which allows us to replace `Exp` with some term that is an instance of one of its proper subterms. A cancellation rule corresponds to an instance of a substitution rule.

`canonical-form(+Exp, +Rules, ?NewExp)`

`NewExp` is the result of applying to `Exp` the rewrite rules from `Rules` as often as possible to as many subexpressions as possible. In mode `(+,+,-)` this generally only makes sense if `Rules` is terminating. The elements of the list `Rules` are supposed to be of the form `RuleName:Rule`, where each `Rule` is a universally quantified equality.

Example

```
| ?- canonical_form(plus(s(0),y),
                    [plus1:x:pnat=>plus(0,x)=x in pnat,
                     plus2:x:pnat=>y:pnat=>
```

```
                    plus(s(x),y)=s(plus(x,y)) in pnat],P).

P = s(y)
```

`casesplit-suggestion(H,G,Scheme)`

`Scheme` describes an casesplit scheme, that would, according to the heuristics (see below), be a good choice. This casesplit scheme is assumed to be valid—this must be checked separately.

`Scheme` is actually a description of a substitution of terms for universally quantified variables appearing in the sequent `H==>G`. Let `G'` be the necessarily different goal resulting from the application of the the substitution `Scheme` to `G`. (In contrast to `induction-suggestion/3`, which is otherwise very similar to `casesplit-suggestion/3`, no wave-fronts are introduced by `casesplit-suggestion/3`.)

Each term introduced at each occurrence of each substituted variable is said either to be *unflawed* or *flawed*, according to whether or not a reduction-rule can be used to rewrite this term or one of its superterms. (Again, notice the difference between induction analysis and casesplit analysis: induction uses *wave-rules* whereas casesplit uses *reduction-rules*.)

`casesplit-suggestion/3` builds a so-called unflawed/flawed table according to almost all the various combinations of substitutions and variables; substitutions are computed by narrowing with the reduction-rules in the database. Not all substitutions will make sense as casesplits—this must be checked separately (see `scheme/3` for one way of doing this).

`induction-suggestion/3` generates suggestions by considering all permutations of all universally quantified variables with all possible induction terms. Induction terms are generated lazily, subject to the criterion that they can be rippled by some wave-rule present in the database. Hence the term *ripple analysis*. In this way, each occurrence of all variables can be tagged with a collection of candidate inductions which describes:

- the terms to be substituted for the variables and the way in which these 'induction terms' are to be annotated: we call this an annotated substitution.
- any variables which must remain as sinks
- how well this annotated substitution fares. That is, the unflawed/flawed assignment for all variables in the domain of the substitution. An occurrence for which a wave-rule is applicable is unflawed; otherwise it is flawed.

All of this information is then processed by a heuristic which implements the following:

1. prefer inductions on a variable which *minimizes* the number of flawed occurrences (in the case of `unflawed-induction-suggestion/3` this number must be zero);
2. prefer inductions on a variable which *minimizes* the (sum of the) depth in the term tree of all flawed occurrences;
3. inductions on a variable which *minimizes* the number of unflawed occurrences;

These criteria are used (in lexicographic order) in order to rank various suggestions.

Example The conjecture for the associativity of multiplication

```
a:pnat=>
  b:pnat=>
    c:pnat=>
      times(a,times(b,c)) = times(times(a,b),c) in pnat
```

together with the definitions of multiplication and addition, has the following unflawed/flawed table:

Annotated substitution		Position	Status	Sinks
Ind. var.	*Ind. term*			
`a`	$\boxed{\texttt{s}(\underline{\texttt{v0}})}$	`[1,1,2,1]`	unflawed	{}
		`[1,1,1]`	unflawed	{}
`b`	$\boxed{\texttt{s}(\underline{\texttt{v0}})}$	`[2,1,2,1]`	flawed	
		`[1,2,1,1]`	unflawed	{}

The heuristic chooses induction on `a` since that minimizes the number of flawed occurrences. See also: `unflawed-induction-suggestion/3`, `casesplit-suggestion/3` and `unflawed-casesplit-suggestion/3`.

`complementary-sets(?CNames)`

`CNames` is a list of the form

$$[C_1\text{–}R_1\text{–}Dir_1\text{–}Name_1, \ldots, C_n\text{–}R_n\text{–}Dir_n\text{–}Name_n]\text{–}LHS$$

such that $LHS \Rightarrow R_i$ is a rewrite with name $Name_i$ conditional upon C_i (a single condition, not a list of conditions), and direction Dir_i, where $n > 1$. Pictorially:

$$\begin{array}{rccc} Name_1: & C_1 \rightarrow LHS & \Rightarrow & R_1 \\ & & \vdots & \\ Name_n: & C_n \rightarrow LHS & \Rightarrow & R_n \end{array}$$

`complementary-sets/1` draws upon rewrites present in the rewrite database when it is called. Notice that the $Name_i$'s need not belong to the same family of definitions (by family I mean e.g., `member1`, `member2`, `member3`).

For example, given usual definitions of member and insert we have:

```
:- complementary_sets(CNames).
CNames =
 [[(A2<B2=>void)-B2::insert(A2,C2)-equ(left)-insert3,
   (A2<B2)-A2::B2::C2-equ(left)-insert2]-insert(A2,B2::C2),
  [(A4=B4 in int=>void)-member(A4,C4)-equ(left)-member3,
   (A4=B4 in int)-{true}-equ(left)-member2]-member(A4,B4::C4)]
```

`complementary-sets(+Names,?CNames)`

As `complementary-sets/1`, but first parameter is a list of rewrite names to consider. For example, `complementary-sets([member1, member2, member3], CNames)` restricts attention to the family of theorems defining `member`.

See also `print-complementary-sets/1`.

`complementary-sets(?CName)`

As `complementary-sets/1`, but just return a single element, others on backtracking.

`consistent-registry(+Tau,+Prec)`

The registry described by `Tau` and `Prec` is consistent. (See §A.4.2.1 on page 135.)

`contains-wave-fronts(+T)`

Succeeds once only iff `Term` contains a wave-front.

`cooper(+F,?V)`

Runs Cooper's decision procedure for Presburger arithmetic (see §B.2 on page 141). If `cooper/2` fails, `F` is not a sentence of Presburger arithmetic.

If `cooper/2` succeeds, `V` indicates the validity of `F`. `V == yes` means `F` is valid, otherwise `V == no` and `F` is invalid.

Example The statement of the associativity of addition is Presburger and is valid:

```
| ?- cooper(x:pnat=>
              y:pnat=>
                z:pnat=>
                  plus(x,plus(y,z))=plus(plus(x,y),z)in pnat,V).
V = yes
```

The following fails since `times` is not part of Presburger arithmetic.

```
| ?- cooper(a:pnat=>
         b:pnat=>
           c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat,V).

no

| ?- cooper(10 = plus(9,1) in int,P).

P = yes
```

`copy(+Term, -NewTerm)`

`Term` and `NewTerm` are identical except for the renaming of prolog variables.

`equal-rule(?Exp,?RuleName:?Rule)`

`RuleName` is the name of an equivalence which allows us to replace some sentence with an equality. `Rule` is the equivalence expressed as directional rewrite rule, with all universally quantified variables replaced by prolog variables.

`ev(+Term,?Value)`

Ground `Term` evaluates to value `Value`. More accurately, `Value` is the result of computing using the computation rules until no more computation is possible. 'Computation rules' are in fact those definitional equality and biimplication rules in the rewrite database, used from left-to-right. Rules for propositional evaluation are hard-wired. Evaluation strategy is topmost-leftmost.

Normally, applications call `ev/2` with `Term` being a quantifier-free variable-free term.

`exp-at(+Exp, ?Pos, ?SubExp)`

Expression `Exp` contains `SubExp` at position `Pos`. Positions are lists of integers representing tree coordinates in the syntax-tree of the expression, but in reverse order. A coordinate 0 represents the function symbol of an expression, thus, for example:

```
:- exp_at(f(g(2,x),3), [2,1], x).
:- exp_at(f(g(2,x),3), [0,1], g).
```

Due to the generous mode, this predicate can be used to either find the `SubExp` at a given `Pos`, or to find the `Pos` of a given `SubExp`.

The definition from [4] is extended by defining `[]` as the position of `Exp` in `Exp`. Fails if `Pos` is an invalid position in `Exp`.

Furthermore, the position specifications (or tree coordinates, or path expressions) are transparent for any possible wave-front annotations in `Exp`. (see §A.3.1 on page 131 for an explanation on wave-fronts). Thus the position of `x` in `f(x)` and in `''f({x})''` is `[1]` in both cases. wave-fronts (`''...''`) are regarded a part of the embedded expression, whereas wave variables (`{...}`) are not. Thus, all possible values `Pos` and `Sub` in `exp-at(f1(''f2({x})''),Pos,Sub)` are:

```
Pos = [1]
Sub = ''f2({x})'' ;
Pos = [0]
Sub = f1 ;
Pos = [1,1]
Sub = x;
Pos = [0,1]
Sub = f2 ;
```

Notice that neither `f2(x)` nor `{x}` are subexpressions, and that the position specifiers are as if the wave-front in `exp-at(f1(''f2({x})''),Pos,Sub)` had not been there.

See also `ann-exp-at/3` and `ann-exp-at/5`.

`exp-at(+Exp,?Pos,?SubExp,?SupExp)`

This is as `exp-at/3` (expression `Exp` contains `SubExp` at position `Pos`). The additional fourth argument `SupExp` is bound to the expression immediately surrounding `SubExp`. Thus, for example:

```
:- exp_at(f(g(2,x),3), [2,1], x, g(2,x)).
:- exp_at(f(g(2,x),3), [0,1], g, g(2,x)).
```

Notice that `exp-at(_,[],_,_)` will always fail (since what would the value of `SupSubExp` be?). Of course `exp-at/4` can be trivially expressed in terms of `exp-at/3` but the implementation of `exp-at/4` is more efficient since it avoids descending down the same subterms twice in a row.

The same rules for wave-fronts hold as described above for `exp-at/3`. Pursuing the same example as above, we get as all possible values for `Pos`, `Sub` and `Sup`:

```
:- exp_at(f1(‘‘f2({x})’’),Pos,Sub,Sup).

Pos = [1]
Sup = f1(‘‘f2({x})’’)
Sub = ‘‘f2({x})’’ ;
Pos = [0] Sup = f1(‘‘f2({x})’’)
Sub = f1 ;
Pos = [1,1]
Sub = x
Sup = ‘‘f2({x})’’ ;
Pos = [0,1]
Sub = f2
Sup = ‘‘f2({x})’’ ;
no.
```

extending-registry

This is a flag which the user can use to control the `reduction/2` method. If defined so as to succeed, `reduction/2` will attempt to extend the registry during symbolic evaluation, otherwise this behaviour is prevented. See the description of the `reduction/2` method for further detail.

The default is that `extending-registry` fails.

extend-registry-prove(+T-+P,?T'-?P',+Prob)

This predicate is a convenient interface to `rpos-prove/5`—it deals with lifting to variables, and ensures that all the function and constant symbols are mentioned in the resulting registry.

`extend-registry-prove/4` succeeds iff `Prob` can be proved to hold in registry `T'-P'`. `Prob` is an ordering problem as described under `rpos-prove/5`. Registry `T'-P'` is a consistent extension of registry `T-P`.

`T` is a status function, and `P` is a quasi-precedence, as described under `rpos-prove/5`.

The registry defined by `T` and `P` is assumed to be consistent. (This is decided by `consistent-registry/2`.) `S` and `T` are possibly non-ground Prolog terms. All the function and constant symbols in `S` and `T` are present in `T'-P'`.

Example Here is a simple example which computes a registry suitable to show that the step case definition of `plus` is terminating when used from left to right:

```
| ?- extend_registry_prove([]-([]-[]), T-P,
                               plus(s(X),Y) > s(plus(X,Y))).

P = [plus>=s]-[plus=\=s],
T = [s/_7243,plus/_7234]
```

In the example above the term `[]-([]-[])` is the empty registry. Notice that no commitment has been made to the status function, although entries been added for `plus` and `s`; the precedence commits to `plus` $\succ$ `s`.

A more realistic illustration of the use of `extend-registry-prove/4` can be found in the preconditions of the `reduction/2` method.

groundp(+Term)

Succeeds iff `Term` is ground. Since Clam uses Prolog as the meta-language (whereas

Type Theory is the the object-language), Prolog variables play the role of meta-variables. Thus, `groundp(Term)` can be used to check if `Term` does or does not contain any meta-variables. Arguably, this predicate should have another name.

`ground-sinks(+Instan, +Lhs, +Rhs, ?SubTerm)`

`Instan` is a list of sink instantiations. For all members of `Instan` which are prolog variables an instantiation is calculated using `Lhs` and `Rhs`, the left and right hand sides of the current goal. `SubTerm` is a subexpression of `Rhs` in which uninstantiated sinks may occur.

`hyp(?Hyp, ?HypList)`

Hypothesis `Hyp` is among the annotated hypothesis list `HypList`. `hyp/2` behaves much like list membership but it also recognises annotations and filters these.

Example

```
| ?- hyp(H,
         [x:pnat,
          ih:[ihmarker(_,_), v0:plus(s(x),x)=s(plus(x,x)) in pnat]]).

H = (x:pnat) ? ;

H = (v0:plus(s(x),x)=s(plus(x,x))in pnat)
```

`inductive-hypothesis(?S,?Hyp,+Hyps)`

`Hyps` is a list of hypothesis containing an inductive hypothesis `Hyp` having status `S`. (cf. §4.5 on page 116 and `inductive-hypothesis/5` and `active-induction-hypothesis/2`.)

`inductive-hypothesis(S,V:H,H1,NS,H2)`

`V:H` is an inductive hypothesis in both `H1` and `H2` (each a hypothesis list), with status `S` in `H1` and status `NS` in `H2`. `H1` and `H2` agree on all other elements.

This predicate is useful when changing the status of some induction hypothesis: see `weak-fertilization/4` for an example.

`induction-suggestion(H,G,Scheme)`

`Scheme` describes an induction scheme, that would, according to the heuristics (see below), be a good choice. This induction scheme is not known to be valid—this must be checked separately.

`Scheme` is actually a description of a substitution of terms for universally quantified variables appearing in the sequent `H==>G`. Let `G'` be the necessarily different goal resulting from the application of `Scheme` to `G`, and in which wave-fronts capture these differences. Thus the skeleton of `G'` is equal to `G`.

Each wave-front introduced at each occurrence of each substituted variable is said either to be *unflawed* or *flawed*, according to whether or not a wave-rule can be applied to that wave-front or one of its superterms. Since the ability to ripple a wave-front may depend on the presence of sinks, each unflawed/flawed assignment may demand that certain of the universal variables in `G'` are not induced upon, so that they may be used as sinks. Equally, certain ripples will require *other* variables to be induced upon, in the case of simultaneous induction.

`induction-suggestion/3` builds a so-called unflawed/flawed table according to almost all the various combinations of substitutions and sink assignments; substitutions are computed by narrowing with the rewrite rules in the database. Not all substitutions will make sense as inductions—this must be checked separately (see `scheme/3` for one way of doing this).

`induction-suggestion/3` generates suggestions by considering all permutations of all universally quantified variables with all possible induction terms. Induction terms are generated lazily, subject to the criterion that they can be rippled by some wave-rule present in the database. Hence the term *ripple analysis*. In this way, each occurrence of all variables can be tagged with a collection of candidate inductions which describes:

- the terms to be substituted for the variables and the way in which these 'induction terms' are to be annotated: we call this an annotated substitution.
- any variables which must remain as sinks
- how well this annotated substitution fares. That is, the unflawed/flawed assignment for all variables in the domain of the substitution. An occurrence for which a wave-rule is applicable is unflawed; otherwise it is flawed.

All of this information is then processed by a heuristic which implements the following:

1. prefer inductions on a variable which *minimizes* the number of flawed occurrences (in the case of `unflawed-induction-suggestion/3` this number must be zero);
2. prefer inductions on a variable which *minimizes* the (sum of the) depth in the term tree of all flawed occurrences;
3. inductions on a variable which *minimizes* the number of unflawed occurrences;

These criteria are used (in lexicographic order) in order to rank various suggestions.

Example The conjecture for the associativity of multiplication

```
a:pnat=>
  b:pnat=>
    c:pnat=>
      times(a,times(b,c)) = times(times(a,b),c) in pnat
```

together with the definitions of multiplication and addition, has the following unflawed/flawed table:

Annotated substitution		Position	Status	Sinks
Ind. var.	*Ind. term*			
`a`	$\boxed{\texttt{s}(\underline{\texttt{v0}})}$	`[1,1,2,1]`	unflawed	{}
		`[1,1,1]`	unflawed	{}
`b`	$\boxed{\texttt{s}(\underline{\texttt{v0}})}$	`[2,1,2,1]`	flawed	
		`[1,2,1,1]`	unflawed	{}

The heuristic chooses induction on `a` since that minimizes the number of flawed occurrences. See also: `unflawed-induction-suggestion/3`, `casesplit-suggestion/3` and `unflawed-casesplit-suggestion/3`.

`instantiate(+G1, ?G2, ?G2Vals)`

`G2` is the result of instantiating all the universally quantified variables in `G1` with the values in `G2Vals`. We require that *all* variables quantified in `G1` are instantiated in `G2`, thus:

```
:- instantiate(x:pnat=>y:pnat=>f(x,y), y:pnat=>f(1,y), [1])
```

will *not* succeed because of `y`, whereas

```
:- instantiate(x:pnat=>y:pnat=>f(x,y), f(1,2), [1,2])
```

will.

`instantiate(?Frees,+G1, ?G2, ?G2Vals)`

Similar to `instantiate/3`. (Exactly the same if `Frees` is a variable.) This predicate is more generous in that it allows universal variables of `G1` that do not take part in the instantiation to be supplied with instantiating terms.

For example, the following call to `instantiate/3` fails to give a ground term for `A`:

```
:- instantiate(x:pnat=>p(x)=0 in pnat, A = 0 in pnat,P).
A = p(X),
P = [X]
```

since any instantiation of `x` will do. (Notice that `instantiate/3` does not fail in this case, only when instantiation to a bound variable is required.)

There are cases where it is necessary to provide a binding for `x`, e.g., using a rewrite rule left-to-right when the set of variables on the left is a proper subset of those on the right. The rewrite is legitimate providing a term can be supplied of the correct type. For example, using

```
z:pnat, x:pnat=>f(x)=s(0) in pnat ==>x:pnat=>f'(x)=s(0) in pnat
```

during weak-fertilization from right-to-left (`x` is unbound).

At the meta-level, Clam ignores this problem and simply uses `x` as the resulting instance of `x`. At the object level the tactic must work to mimic this and ensure that the variable naming remains aligned.

`instantiate/4` allows these eigenvariables from the context to be supplied and used to instantiate what would otherwise be unbound variables: `Frees` is a list of terms which are to be used to supply the instantiation of otherwise 'floating' variables.

```
:- instantiate([y],x:pnat=>p(x)=0 in pnat, A = 0 in pnat,P).
A = p(y),
P = [y]
```

Typically, the `Frees` will correspond to the matrix of the goal being weak-fertilized, since that is the instantiation which preserves the structure of the hypothesis (see the implementation of the weak-fertilization tactic for further programmer-level information).

`issink(?T,?S)`

`T` is a sink whose contents is the term `S`. Informally, we might write `T` $= \lfloor$`S`$\rfloor$.

`iswh(?T,?S)`

`T` is a wave-hole whose contents is the term `S`. Informally, we might write `T` = $\underline{\texttt{S}}$.

`iswf(?T,?Type,?Dir,?S)`

`T` is a wave-front around the term `S`. Notice that `T` is not necessarily a well-annotated term: `iswf` does not enforce any restrictions on the term `S`.

`Dir` indicates the direction of the wave-front: it is either `in` or `out`; the flag `Type` is reserved for future expansion and must always be set to the atom `hard`.

`join-wave-fronts(+Term,?PosL,?JTerm)`

Recall that Clam always manipulates well-annotated terms, which are, by definition (see §4.4.1 on page 116) in a maximally-split normal form. This predicate can be use to transform annotated terms into a form in which wave-fronts are *not in this normal form: some wave-fronts are joined together. Note that this form is not well-annotated!*.

The maximally joined form (see `maximally-joined/2`) is typically used to present annotations through some user-interface: it is easier to read than the well-annotated form.

`JTerm` will be as `Term`, but with a number of small wave fronts joined into larger ones. `PosL` will contain the positions of the wave-fronts in `Term` which were joined. This predicate generates on backtracking all possible joins of wave-fronts.

`mark-potential-waves(+Goal, -NewGoal)`

`Goal` and `NewGoal` are identical except that all top-level existential variables in `NewGoal` are annotated as potential wave-fronts. For example:

```
mark_potential_waves(x:pnat=>y:pnat#p(g:pnat#h(g),p(y,g)),
x:pnat=>"y":pnat#p(g:pnat#h(g),p("y",g))).
```

Notice that the deepest existential (g) is not annotated.

`mark-sinks(+Bindings, +Term, -NewTerm)`

`Bindings` is a list of bindings. The `Term` and `NewTerm` are identical except that all variables in `Bindings` which occur in `Term` are annotated as sinks in `NewTerm`.

`matrix(?VarList,?Matrix,?Formula)`

`Matrix` is the matrix of `Formula`, that is: `Matrix` is as `Formula`, except all prefixing quantifiers. `VarList` is the list of variables involved in these quantifiers (in the same order as the quantifiers occurred in `Formula`).

`matrix(?VarList,?EVarList,?Matrix,?Formula)`

`matrix/4` is as `matrix/3` except it is extended to deal with existential quantification. `EVarList` is a list with elements of the form `MetaVar-ObjVar:Typ` where `MetaVar` is the prolog variable which replaces the object-level variable `ObjVar` in `Formula` to give `Matrix`. Possible modes are `matrix(+, +, +, -)` and `matrix(-, -, -, +)`.

`maximally-joined(+S,?T)`

`T` is the maximally-joined form of `S`. Note that (excepting the trivial case when `S` is identical to T), the term T will not be well-annotated.

Terms depicted in maximally-joined form are usually easier to read. For example, the following two terms are the 'same', but the second one is depicted in the maximally-joined form.

```
''s({''s({f(g(''s({''s({f(g(x))})''<out>})''<out>))})''<out>})''<out>
```

and

```
''s(s({f(g(''s(s({f(g(x))}))''<out>))}))''<out>
```

`metavar(?Var)`

Succeeds if `Var` is a meta-variable.

`matches(?S,?T)`

`T` matches `S`. That is, `T` can be instantiated to `S` without substituting for variables in `S`. No instantiation is carried out. See also `unify/2` and `unifiable/2`.

`nr-of-occ(?SubExp, +SupExp, ?N)`

`SubExp` occurs exactly `N` times in `SupExp`. Failure indicates that `SupExp` does not occur in `SupExp`, thus: `N` is never bound to `0`. Generous mode allows use to find the number of occurrences `N` of a given `SubExp`, or to find all `SubExp`s that occur `N` times

---

`notraw-to-used(H1,H2)`

`H1` and `H2` are identical hypothesis lists but for the fact that each hypothesis in `H1` having status `notraw(Ds)` has status `used(Ds)` in `H2`. (cf. `raw-to-used/2`.)

See `pwf-then-fertilize/2` for an example.

`object-level-term(+T)`

Succeeds iff `T` does not contain any meta-variables (i.e., is ground), and does not contain any wave-fronts (or parts thereof).

`occ(+Term, ?SubTerm, ?Pos, ?F)`

`SubTerm` occurs in `Term` at `Pos`, immediately surrounded by term `F`. This is equivalent to (but more readable and faster) than:

```
occ(Term, SubTerm, [N|P], SupTerm) :-
    exp-at (Term, [N|P], SubTerm),
    exp-at(Term, P, F).
```

`polarity(?O1,?O2,?F,?N,?P)`

Function `F` has polarity `P` in argument number `N` under orderings `O1` and `O2`. `P` is one of `+`, `-` or `0`. `F` is positive in argument number `N` under `O1` and `O2` if `O1(X1,X2)` $\rightarrow$ `O2(F(X1),F(X2))` where: `X1` and `X2` obey: `exp-at(F(Xi),N,Xi)`, and `O1` is a partial ordering on the domain of `F` (in its `N`th argument), and `O2` is a partial ordering on the codomain of `F`. A positive polarity means that `F` is monotonic in its `N`th argument, a negative polarity means that `F` is anti-monotonic in its `N`th argument. A zero polarity means that `F` is neither monotonic nor anti-monotonic in its `N`th argument.

Polarity of nested functions is calculated according to the obvious transitivity rules: $+ = +(+)$ or $+ = -(-)$, and $- = -(+)$ or $- = +(-)$.

This predicate is implemented via a lookup table, thereby making Clam not theory free (see §6.4 on page 127), and should eventually be implemented in a proper, theory free, way, as suggested there.

`polarity-compatible(+G, +P, ?Dir)`

This predicate holds if the subterm at position `P` within the goal `G` is compatible with respect to the rewrite orientation `Dir`.

See `rewrite-rule/6` for a description of the `Dir` parameter.

`precon-matrix(?TypedVarList,?PreCond=>?T1,?T2)`

Let a *prefix* R be either $v:t$ or p; then T_2 is of the form $R \to \cdots \to R \to T_1$, where `TypedVarList` is the collection of $v:t$'s for all R of the first form, and `PreConds` are the p's for all R of the second form. Note that there is much backtracking here, since T_2 may be of the form $R \to T_2'$.

Useful information: This predicate is used heavily by the rewrite database mechanism (see §4.3.3 on page 115). All theorems having a `T1` which is an equality or an implication are considered as possible rewrite rules (i.e., dynamic wave-rules) by `ripple/6`, being conditional upon the appropriate `PreCond` list.

`raw-to-used(Hs,Hyps,NHs)`

`NHs` is as `Hs` but all `raw` hypotheses in `Hyps` appearing in `NHs` are marked as `used([strong])` in `NHs`. (cf. `notraw-to-used/2`.)

See §4.5 on page 116 for a more information on hypothesis status.

`reduction-rtc(+S,?T)`

Roughly, this is the reflexive transitive closure of the reduction relation described by the current TRS.

More precisely: Let `S` rewrite under REDUCTION to some term `U` in a finite number k of steps. `reduction-rtc/2` succeeds iff `T` matches `U` and there is no $k' > 0$ such that `U` rewrites to `U'` different from `U` in k' steps and `T` matches `U'`.

Example Assuming that `plus` is in the reduction rules we have that:

```
| ?- reduction-rtc(plus(s(s(x)),y),T).
T = s(s(plus(x,y)))
```

which is the only solution. However, note that

```
| ?- reduction-rtc(plus(s(s(x)),y),s(plus(s(X),Y))).
X = x,
Y = y
```

succeeds only once since there is no normal form of `s(plus(s(x),y))` which is an instance of `s(plus(s(X),Y))`.

```
| ?- reduction-rtc(plus(x,y),T).
T = plus(x,y)
```

shows that `reduction-rtc/2` includes the reflexive case.

For the non-reflexive case, use `reduction-tc/4`.

See `reduction-rule/6`, `registry/4`, `canonical-form/3` and §3.3.1 on page 92 for related information.

Use `reduction-rtc/4` to introduce hypotheses etc, as may be required for conditional rules.

`reduction-rtc(+S,?T,?Tactic,+Hyps)`

As `reduction-rtc/2`, but `Tactic` and hypothesis (for conditional rewriting) are made explicit. `reduction-rtc(S,T)` is the same as `reduction-rtc(S,T,_,[])`.

`reduction-tc(+S,?T,?Tactic,+Hyps)`

`T` is the normal form of `S`, with respect to the transitive closure of the rewrite relation defined by the current TRS. `polarity-compatible/3` is used to determine which of the two registries, positive or negative, should be used to ensure terminating rewriting.

`Hyps` is a hypothesis list—this may be used to establish conditions of conditional rules. `Tactic` is a tactic which justifies the normalization.

See `reduction-rule/6`, `registry/4` and §3.3.1 on page 92, and `normalize-term/1` for an example.

`rpos-prove(+Problem,+TP,?NTP,+Vars,?Proof`

`Proof` is a proof of `Problem` under the registry described by `NTP`; furthermore, `NTP` is a consistent extension of `TP`. (`TP` is assumed to be consistent, as described by `consistent-registry/2`.)

`TP` and `NTP` are of the form `Tau-Prec`; `Tau` is a status function, represented by a list having elements of the form `F/Fs`, assigning status `Fs` to function symbol `F`. See §5.3.2.2 on page 120 for more information on the status function. `Prec` is the representation of a quasi-precedence, as described in §5.3.2.1 on page 119.

The status function of `TP` must mention all the function symbols and constants appearing in `Problem`.

`Proof` is simply for information—it has nothing to do with proof-planning or tactics!

`Problem` is an ordering problem, having one of the following forms (cf. §5.3.4 on page 120 and §A.4.2.1 on page 135):

`S >= T` Iff `S = T` or `S > T`, that is, `S` $\geq_\rho$ `T`.

`S = T` Iff `S` and `T` are equivalent under RPOS; that is, `S` $\sim_\rho$ `T`.

`S > T` Iff `S` is greater than `T` under RPOS; that is, `S` $>_\rho$ `T`.

`S < T` Iff `T > S`.

`S =< T` Iff `T >= S`.

In these ordering problems `S` and `T` must be ground Prolog terms. Atoms appearing in `Vars` indicate which of the atoms in `S` and `T` are to be considered variables by RPOS.

Example.

```
| ?- rpos-prove(plus(s(v0),v1) > s(plus(v0,v1)),
           [plus/P,s/S]-([plus>=s]-[plus=\=s]),
           NTP,[v0,v1],Prf).
NTP = [plus/P,plus/ms,plus/lex(rl),plus/lex(lr),plus/_A,s/S]
       -([plus>=s]-[plus=\=s]),
Prf = decomp(plus>s) then
       [extn(plus/P=plus/P) then
         [[multiset(s(v0)>v0,_B)],[_C],[identity,_D]]]
```

Compare this with the example given under `extend-registry-prove/4`.

`reduction-rule(?LHS,?RHS,?Cond,?Dir,?RuleName,?Ref)`

`RuleName` is the name of a rewrite `Cond=>LHS:=>RHS` that has been proven to be measure decreasing under some well-founded termination order. `Dir` describes the polarity restrictions in using the rule. The rule may be based on implication, equality or equivalence. See §A on page 129 for background information and `rewrite-rule/6` for documentation on `Dir`.

Clam supports two different reduction rule sets in order to permit implicative rewrites to be used in both directions. This is terminating because the polarity restriction on the use of the rules (which must be maintained by the caller) ensures that cycles are prevented. Equality rules must be oriented either left-to-right or right-to-left since they may be applied in positions of either polarity.

Thus, there are two registries, labelled "`positive`" and "`negative`", establishing termination of all reduction rules whose `Dir` is `imp(left)` and `imp(right)`, respectively. (Equality/equivalence rules are accounted for in both registries.)

`Ref` is the recorded database reference.

`registry(+TRS,?Tau,?Prec,?Ref)`

The registries under which reduction rules are shown to be terminating are stored in the Prolog database, and accessed via `registry/4`. The registry evolves over time as it is extended (for an illustration of how to extend the registry and add it to the database, see `reduction/2`).

`TRS` is the name of the registry—currently, Clam supports only two registries, `positive` and `negative`, for reduction rules at different polarities. Equivalence rules are equality rules are present in both. Together, these two registries define Clam's terminating rewrite system, `trs(default)`. (See `lib-load/[1;2]` and §3.3.1 on page 92.)

`replace(+Pos, ?NewSub, +OldExp, ?NewExp)`

`NewExp` is the result of replacing the subexpression in `OldExp` at position `Pos` with `NewSub`. Either `NewSub` or `NewExp` must be instantiated.

Similar rules for dealing with wave-front hold as for `exp-at/3`: position specifiers are transparent to wave-fronts, and wave-fronts (`''...''`) are deemed part of the embedding expression, while wave variables are part of the surrounding expression. For example:

```
:- replace([1],new,f1(''f2({x})''),T).
T = f1(new)

:- replace([1,1],new,f1(''f2({x})''),T).
T = f1(''f2({new})'')
```

`replace-all(+OldSub, +NewSub, +Exp, ?NewExp)`

`NewExp` is the result of replacing all occurrences of `OldSub` with `NewSub` in `Exp`.

`rewrite(?Pos, +Rule, ?Exp, ?NewExp)`

`NewExp` is the result of rewriting the subexpression in `Exp` at position `Pos` using equation `Rule`. Only one of `Pos`, `Exp` and `NewExp` has to be instantiated, so this can also be used to detect if and where a rewrite rule has been applied (but not to generate all possible applications of a rewrite rule).

`rewrite-rule(?LHS,?RHS,?Cond,?Dir,?Rn,?Ref)`

Accesses the currently available rewrite rules.

$$\texttt{Cond} \Rightarrow \texttt{LHS} \rightarrow \texttt{RHS}$$

is a sound rewrite rule at polarity described by `Dir`. `Dir` is one of

`equ(Type,Orient)` `Orient` is either `left` or `right`; `Type` is an object-level equality. The rule is based upon a typed equality, used from left-to-right (`Orient==left`) or from right-to-left (`Orient==right`). It can be used at positions of any polarity. Notice that Clam stores commuted variants of rules based on equality and equivalence.

`equiv(Orient)` `Orient` is as above. The rule is based on a logical equivalence (and so can be used at positions of any polarity).

`imp(Orient)` `Orient` is as above. The rule is based on an object-level implication, used from left-to-right or from right-to-left (as identified by `Orient`).

Soundness requires that:

`imp(left)` These rules may only be used at positions of positive polarity.

`imp(right)` These rules may only be used at positions of negative polarity.

SeeA.2 on page 129 for further details.

oading the definition of list membership with tracing set to level 40 (via `trace-plan/3`) shows how each of the formulae are processed in to rewrite rules. (Additional information, also shown at this tracing level, is elided below.)

```
| ?- lib_load(def(member)).
...
member1/equ(u(1),left): [] => member(B,nil) :=> void
member2/equ(u(1),left): B=C in int => member(B,C::D) :=> {true}
member2/imp(right):
      [] => member(B,C::D)={true}in u(1) :=> B=C in int
member3/equ(u(1),left):
      B=C in int=>void => member(B,C::D) :=> member(B,D)
member3/imp(right):
      [] => member(B,C::D)=member(B,D)in u(1) :=> B=C in int=>void
...
Loaded def(member)

yes
```

Each of the three equations making up `member` are processed in turn and the rewrite rules extracted, in accordance with §A.2 on page 129. Here, `member3` is the object-level formula

$$\forall x.\forall h.\forall t.x \neq h \supset \texttt{member}(x, h :: l) \equiv \texttt{member}(el, l)$$

which yields two rewrite rules:

```
member3/equiv(left):
      B=C in int=>void => member(B,C::D) :=> member(B,D)
member3/imp(right):
      [] => member(B,C::D)=member(B,D)in u(1) :=> B=C in int=>void
```

the first of which is based on an equivalence '`equiv(left)`' and so can be used at either positive or negative polarity (notice that this rewrite is based on a left-to-right reading. The right-to-left direction is not a legal rewrite rule because of the variable condition on rules). This rule is conditional: the condition is `B=C in int=>void`.

The second rewrite rule is unconditional (depicted by '`[]`' to the left of the meta-level implication '`=>`'), and is based on the implication also present in the formula. The resulting rewrite can only be used at positions of negative '`imp(right)`' polarity. Again, there is no `imp(left)` rule because of variable restrictions.

`ripple(?Kind,+WTT,?NWTT,?Cond,?Rn,?Dir)`

`NWTT` is the result of applying rewrite-rule `Rn` as a wave-rule to the well-annotated term `WTT`, subject to condition `Cond`. `WTT` and `NWTT` are in normal form—errors may result if `WTT` is not in normal form (see `join-wave-fronts/3` for description of normal form).

`Dir` indicates the orientation of the lemma `Rn` on which the rewrite is based: `Dir` is either `equ(right)` (for right-to-left based upon an equality), `equ(left)` (for left-to-right based upon an equality), `imp(right)` (for right-to-left based upon an implication), or `imp(left)` (for left-to-right based upon an implication).

`Kind` is one of `direction_in` (only rippling-in is allowed) `direction_out` (only rippling-out) `direction_in_or_out` (either), depending upon the type of rippling permitted. (If `Kind` is uninstantiated the default is `direction_in_or_out`.)

Useful information: `ripple/6` is the core of the static wave-rule parsing mechanism. Both eager and lazy are implemented. When a term is to be rippled, `ripple/6` first examines a cache of previously parsed wave-rules and uses those if they are applicable. This corresponds to eager parsing, since those cached rules are already available. If none of the cached wave-rules are applicable (or others are required on backtracking), lazy parsing is used.[1]

The skeleton preservation is modulo sinks. Weakening *is not* carried out by `ripple/6`: weakening must be done explicitly (see `wave/4` for how that can be done). `ripple/6` *does not* carry out meta-rippling (see `unblock/3` for a description of that).

See `wave/4` for an illustration of how `ripple/6` is used to do term rewriting.

`scheme(?Thm,?Term,?Scheme)`

`scheme/3` is a schematic database of all of the inductions loaded into the environment. At present, `scheme/3` is only suited to the representation of structural inductions which has single step cases.

[1] Strictly speaking, this caching of wave-rules is not eager static parsing, since not all possible parses of the rewrite rules are necessarily computed.

There are two different representations of induction schemes, an object-level one and a meta-level one. `Thm` is the name of the object-level theorem which is a statement of the validity of the induction. `Scheme` is the meta-level version of this induction scheme. The translation is done automatically by the library mechanism when the object-level scheme is loaded with `lib-load(scheme(Thm))`. In fact for some schemes, Clam is either unable to do this translation automatically or no such translation exists. (See the relevant section on `lib-load` for more details.)

Induction schemes are indexed via a substitution of terms for induction variables: since the meta-level scheme makes these variables explicit, it is only necessary to index on the terms. `Term` is a list of unannotated terms in implicit correspondence with each of the variables to be induced upon.

`Scheme` is a Prolog term, a list of the elements. Each element has the form `Sequents ==> Conclusion`, where `Conclusion` is a schematic term of the form `phi(...)` where `...` is a list of arguments. Each argument has the form `V:T`, where `V` is a Prolog variable and `T` is a possibly non-ground Prolog term. The set of variables in the `Conclusion` is called V, the set of free variables in the `T` are called T.

As indicated above, the third argument of `scheme/3` reads as an induction scheme:

$$\frac{\texttt{Hyps1==>Goal1}\quad\texttt{HypsN ==> GoalN}}{\texttt{==> Goal}}\ \text{induction}$$

Sequents is a list of sequents of the form `Hyps==>Goal`. When `Hyps` is empty, the sequent may be written `==>Goal`. `Goal` is again a term of the form `phi(...)` (of the same arity as appeared in `Conclusion`), where each argument is an object-logic term, possibly containing Prolog variables. Let the set of Prolog variables be called G.

`Hyps` is a list of the (binding) form `B:BT` or the (induction hypothesis, or indhyp) form `phi(...)`. (Again `phi` must have the same arity as in `Conclusion`.) Let the set of binding variables be B, and the set of variables in the indhyp form be IH. We insist that $B = IH \cup G$, so that all variables in a sequent (both goal and hypotheses) are mentioned in the bindings.

The terms `BT` may mention Prolog variables in T (`scheme/5` allows these variables to be instantiated when a scheme is applied to a goal. This freedom for type variables is useful: see the example below.

Internally, B, (and hence IH and G) are kept in a different name space from V (appearing in `Conclusion`), but it might be confusing (on a casual perusal of the `scheme/3` database) to rely on this, and so the user should discipline itself to keeping these sets disjoint, $B \cap V = \emptyset$.

Example Here are two example induction schemes both of which are present in the standard Clam library.

```
| ?- lib_load(scheme(list_primitive)).
Loaded scheme(list_primitive)
Added induction scheme for list_primitive

yes
| ?- scheme(A,B,C).

A = list_primitive,
B = [_A::_B],
C = [[]==>phi(nil),[_C:_D,_E:_D list,phi(_E)]==>phi(_C::_E)]
      ==>phi(_F:_D list)
```

The library mechanism loads the object-level theorem and then translates it into a meta-level scheme.

```
| ?- lib_load(scheme(treeind)).
Loaded def(node)
Loaded def(leaf)
Loaded def(tree)
Loaded scheme(treeind)
Added induction scheme for treeind

| ?- scheme(A,B,C).

A = treeind,
B = [node(_A,_B)],
C = [[_C:_D]==>phi(leaf(_C)),
     [_E:_D tree,_F:_D tree,phi(_E),phi(_F)]
       ==>phi(node(_E,_F))]
    ==>phi(_G:_D tree)
```

Notice that there are two induction hypothesis in the second of the two subgoals: `phi(E)` and `phi(F)`. Induction using this rule will normally try to exploit both of these hypothesis and so the constructor `node(.,.)` in the goal will be a multi-hole wave-front.

`scheme/5` will create the appropriate annotations automatically.

`scheme(?SchemeSource,?Sch,+H==>+AnnGoal,?BaseSequents,?StepSequents)`

Induction scheme `Sch` is applicable to the sequent `Sequent` giving a list of `BaseSequents` and `StepSequents`. `SchemeSource` is the source of the induction scheme, e.g., `lemma(Thm)` where `Thm` is the name of the object-level induction theorem. `scheme/5` is the meta-level partner to the corresponding induction inference rule since it also adds necessary meta-level annotation. For some logics (e.g., Oyster) it is possible to justify the induction rule by proving some (typically) higher-order scheme lemma. These scheme lemmas are used by the induction tactic. (See §4.1 on page 109 for information on this topic.)

`H` and `AnnGoal` may be annotated, although any inductive conclusions in `StepSequents` will not have sinks preserved from `AnnGoal`: they are removed first.

The `Sch` term identifies the induction scheme albeit in a crude and non-unique way (see also `scheme/3`. It is a list of the form `(V:T)-IT` where `V:T` is an essentially universal variable from `H==>AnnGoal` (i.e., either a universally quantified variable in `AnnGoal` or a parameter occurring in `H`). `IT` is the term which will be substituted for all occurrences of that `V` in `AnnGoal` when the induction is performed.

It is a list of the induction terms used in the induction. The order of these terms is that which appears in the `scheme/3` database, considering a left-to-right traversal of the list of sequents. For example, for the `nat-list-pair` scheme:

```
| ?- scheme(nat_list_pair,B,C).

B = [s(_A),_B::_C],
C = [[_D:pnat]==>phi(_D,nil),
     [_E:int list]==>phi(0,_E),
     [_F:pnat,_G:int,_H:int list,phi(_F,_H)]
       ==>phi(s(_F),_G::_H)]
    ==>phi(_I:pnat,_J:int list)
```

`Sch` would be (assuming induction on `x` and `y`) `[(x:pnat)-s(X), (y:pnat list)-h::t]`. In this induction, `scheme/5` would insist upon induction parameters `h` and `t` for the `list` part of the induction, whilst for the `pnat` part the induction parameter is left unbound and will be automatically chosen.

`scheme/5` is a uniform way of exploiting the `scheme/3` database, which is the raw form in which induction schemes are stored in Clam.

Although recursion schemes can have any number of step-cases, the schemes are still characterised (indexed) by a single term (the `IT` argument). This is obviously not good enough. Furthermore, even for induction schemes with only one step-case, simple classification (or characterisation or indexing) of induction schemes by a single induction term is not rich enough: different induction schemes can have the same induction scheme, but still be very different.

Example `scheme/5` has a generous mode so that it can be used to test for and to generate applicable induction schemes.

```
| ?- scheme(_,Scheme,[]==>x:pnat=>y:int list=>p(x,y),B,S).

Scheme = [(y:int list)-v0::v1],
B = [[y:int list]==>x:pnat=>p(x,nil)],
S = [[v0:int,v1:int list,ih:[RAW,v2:x:pnat=>p(x,v1)],y:int list]
       ==>x:pnat=>p(\x/,‘‘v0::{v1}’’<out>)] ;

Scheme = [(x:pnat)-s(v0)],
B = [[x:pnat]==>y:int list=>p(0,y)],
S = [[v0:pnat,ih:[RAW,v1:y:int list=>p(v0,y)],x:pnat]
       ==>y:int list=>p(‘‘s({v0})’’<out>,\y/)] ;

Scheme = [(x:pnat)-s(v0),(y:int list)-v1::v2],
B = [[v0:pnat,x:pnat,y:int list]==>p(v0,nil),
     [v0:int list,x:pnat,y:int list]==>p(0,v0)],
S = [[v0:pnat,v1:int,v2:int list,
      ih:[RAW,v3:p(v0,v2)],x:pnat,y:int list]
       ==>p(‘‘s({v0})’’<out>,‘‘v1::{v2}’’<out>)] ;

no
| ?-
```

(The last solution here is produced only when `scheme(nat-list-pair)` has been loaded.)

Example Induction on a parameter

```
| ?- scheme(_,Scheme,[v0:pnat]==>x:pnat=>p(x,v0),B,S).

Scheme = [(x:pnat)-s(v1)],
B = [[x:pnat,v0:pnat]==>p(0,v0)],
S = [[v1:pnat,ih:[RAW,v2:p(v1,v0)],x:pnat,v0:pnat]
       ==>p(‘‘s({v1})’’<out>,v0)] ;

Scheme = [(v0:pnat)-s(v1)],
B = [[v0:pnat]==>x:pnat=>p(x,0)],
S = [[v1:pnat,ih:[RAW,v2:x:pnat=>p(x,v1)],v0:pnat]
       ==>x:pnat=>p(\x/,‘‘s({v1})’’<out>)] ;
```

Example Similar to the example above but in which the scheme prescribed is inapplicable due to variable conflicts

```
| ?- scheme(_,[VT-s(s(v0))],[v0:pnat]==>x:pnat=>p(x,v0),B,S).

no.
```

Example The induction rule (schematic in ϕ)

$$\frac{\phi(0) \quad \phi(x) \rightarrow \phi(s(x))}{\forall x\!:\!pnat.\phi(x)}$$

Could be used in a proof of associativity of *plus* via

```
:- scheme(_,[s(v0)],[a:pnat],
          []==>a:pnat=>b:pnat=>plus(a,b)=plus(b,a)in pnat,B,S).
B = [[a:pnat]==>b:pnat=>plus(0,b)=plus(b,0)in pnat]
S = [[v0:pnat,
      ih:[RAW,v1:b:pnat=>plus(v0,b)=plus(b,v0)in pnat],
      a:pnat]
     ==>
      b:pnat=>plus(``s({v0})''<out>,\b/)=
              plus(\b/,``s({v0})''<out>)in pnat]
```

`sinks(?T1, ?SinksSpec, ?T2)`

`T2` is as `T1`, except that `T2` has sinks in the positions specified by, `SinksSpec`, a list of term positions. Due to the generous mode of this predicate, `sinks/3`, can be used to insert (mode `sinks(+,+,-)`) or to delete sinks (mode `sinks(-,+,+)`), or locate sinks (mode `sinks(-,-,+)`). At least one of `T1` of `T2` must be instantiated.

`sink-proper(?T1, ?T2)`

`T1` is identical to `T2` except that `T1` is enclosed in a sink. `sink-proper/2` can be used to retrieve the contents of a sink (mode `sink-proper(+,-)`) or package up a term in a sink (mode `sink-proper(-,+)`). Either `T1` or `T2` must be instantiated.

`skeleton-position(+GPos, +G, -HPos )`

`GPos` is a position inside the annotated term `G`, and `HPos` is the same position with respect to the skeleton of `G`.

For example, in an inductive proof, one can use this predicate to compute positions in the hypothesis which correspond to position in the goal, providing those positions are in the skeleton.

`split-wave-fronts(+Term,?PosL,?STerm)`

`STerm` will be as `Term`, but with a number of complex wave-fronts split into smaller ones. `PosL` will contain the positions of the wave-fronts in `Term` which were split. It generates on backtracking all possible splits of all wave-fronts; it returns the possible splits in a sensible order: it returns the splits in bigger chunks (i.e., few splits) before splits in smaller chunks.

See the description of `join-wave-fronts/3` for further details.

`strip-meta-annotations(+T1, -T2)`

Identical to `unannotated(T1,T2)`. See `unannotated/2`.

`strip-redundant-sinks(+T1, -T2)`

`T1` and `T2` are lists of goal sequents. The corresponding goal sequents from each list are identical except that for each goal in `T1` which contains sinks but no wave-fronts the associated goal in `T2` contains no sinks.

`strip-redundant-waves(+T1, -T2)`

`T1` and `T2` are lists of goal sequents. The corresponding goal sequents from each list are identical except that for each goal in `T1` for which a nested induction would not be profitable then the wave-fronts in the associated goal in `T2` are not present.

`term-instance(+Context,T,S)`

`T` is a quantifier-free formula (term) in some context `Context` and `S` is the result of instantiating variables appearing in `T` with values of the appropriate type. Values are ground constructor terms. The choice of these values is unspecified: one can think of them as being selected from a known, finite set randomly (cf. `random-term-instance/3`). On backtracking, generates other term instances, no two of which are identical.

```
?- term_instance([x:pnat,y:int list],f(x,y),P)
P = f(0,0::nil) ;
P = f(0,-3::nil) ;
P = f(0,3::nil) ;
P = f(0,0::0::0::nil) ;
P = f(0,0::0:: -3::nil) ;
P = f(0,0::0::3::nil)
```

This predicate is used by `trivially-falsifiable/2`.

`theorem(?Theorem, ?Goal)`

`Theorem` is an Oyster theorem or lemma with top level goal `G`. See §3.3 on page 92 for more information on the nature of theorems and lemmas.

`trivially-falsifiable(+C,+F)`

`F` is a formula in context `C` which can readily be shown to be false. 'Readily shown to be false' means that one of (up to) five randomly chosen ground instances of `F` evaluate to 'false'. (Of course, 'five' is a special number: no fewer, no more.)

Evaluation is performed using `ev/2` and the defining equations present in the current environment.

Example applications of this predicate can be found in `weak-fertilize/4` and `generalise/2`.

`unannotated/1(+T1)`

`T1` contains no meta-level annotations.

`unannotated/2(+T1, ?T2)`

`T1` and `T2` are identical except that all meta-level annotations (wave-fronts, holes, and sinks) which appear in `T1` are absent from `T2`.

`unannotated-hyps(+T1, -T2)`

`T1` and `T2` are identical hypothesis lists except that `T2` does not contain any annotation.

`unifiable(?S,?T)`

`S` and `T` are could be unified with each other. No variable instantiation takes place. See also `unify/2` and `matches/2`.

`unify(?S,?T)`

`unify/2` computes the most general unifier of terms `S` and `T` and instantiates them with that unifier. See also `unifiable/2` and `matches/2`.

`universal-var(+Seq,?Var)`

`Var` is a variable occurring universally in the sequent `Seq`. This can be because `Var` occurs among the hypotheses in `Seq`, or because `Var` appears as an explicitly universally quantified variable in the goal of `Seq`.

`unflawed-casesplit-suggestion(+H,+G,?Scheme)`

`Scheme` is an unflawed casesplit for goal `G` in context `H`. See `casesplit-suggestion/3` for more information.

`unflawed-induction-suggestion(+H,+G,?Scheme)`

`Scheme` is an unflawed induction scheme for goal `G` in context `H`.

This predicate carries out the same analysis as `induction-suggestion/3` but it further restricts `Scheme` to be such that there are no flawed variable occurrences. See `induction-suggestion/3` for more information.

`wave-fronts(?T1,?FrontsSpec,?T2)`

`T2` is as `T1`, except that `T2` has wave-fronts in the positions specified by `FrontsSpec`.

See [9] for a precise description of wave-fronts; see `split-wave-fronts/3` for a discussion of this normal form.

To make this document self-contained, §A.3.1 on page 131 gives a brief description of wave-fronts and their representation. `FrontsSpec` is a list of terms, each specifying the position of a wave-front as described in §A.3.1 on page 131.

Due to the generous mode of this predicate, `wave-fronts/3` can be used to insert wave-fronts (mode `wave-fronts(+,+,-)`), or to delete wave-fronts (mode `wave-fronts(-,+,+)`), or to find wave fronts (mode `wave-fronts(-,-,+)`). At least one of `T1` and `T2` must be instantiated.

wave-fronts are pretty printed by Clam as follows: the wave-front B_i is surround by double quotes (`‘‘...’’`), and all wave hole in B are surrounded by braces (`{...}`). Thus, a formula like `f(g(x),y)`, with wave-fronts specified by `[[1]-[[1]]]` would be printed like `f(‘‘g({x})’’,y)`.

(But see also `idplanTeX[0;1]` and `dplanTeX[0;1]` to get LaTeX=LaTeX source files of proof-plans.)

`wave-terms-at(+Term, ?Pos, ?SubTerm)`

`SubTerm` is a subterm of `Term` at position `Pos` such that `SubTerm` is a wave-term or contains wave-term(s).

`well-annotated(+Term)`

`Term` is is a well-annotated term. Clam will only manipulate well-annotated terms (as defined in §4.4.1 on page 116) (with the exception of `join-wave-fronts/3` and `maximally-joined/2`.

2.2.2 Oyster specific predicates

There are a great many ways in which Clam implicitly assumes that its object-logic is type theory, Oyster in particular. This section documents some predicates which are very specific to Oyster.

`canonical(?Term,?Type)`

`Term` is a canonical member of `Type` (see readings on Type Theory (e.g. [13]) for a precise definition of canonical). Although it is possible to call this predicate in mode `canonical(-,+)`, to generate all canonical members of a given type, this is in general not very useful (typically, types have an infinite number of canonical members), and the only useful mode is of course `canonical(+,?)`, to check whether a given element is canonical in a type.

Currently, this predicate does not contain a full definition canonical members of all types in Oyster, and should be extended when and if needed.

`constant(?C,?T)`

`C` is a constant of type `T`. The only useful mode is of course `constant(+,-)`, since mode `constant(-,?)` would just generate an infinite number of constants of type `T`xxxxs.

`elementary(+S,?T)`

`T` is an Oyster tactic which proves sequent `S`. All annotations appearing in `S` are ignored (removed). The proof is restricted to a limited amount propositional reasoning and a limited amount of non-propositional reasoning.

It is hard to characterise the class of sequents provable by `elementary/2`:

- It is almost a decision procedure for intuitionistic propositional sequents.
- It spots identities (`X=X in T`).
- It removes universally quantified variables from the goal and sees if the remainder is a tautology (`x:pnat=>f(x)=>f(x)`).
- It knows a little bit about the structure of types such as uniqueness properties, e.g., that no number is equal to its own successor and that no number's successor is equal to zero.

If `S` is a list of sequents then `T` is the corresponding list of tactics.

`elementary/2` requires that certain lemmas are loaded from the library: this dependency is reflected in the needs file.

`propositional(+S,?T)`

`S` is a derivable sequent of intuitionistic propositional logic, and `T` is an Oyster tactic to prove this to be the case.

`type-of(+H,+Exp, ?Type)`

Guesses (or checks) the `Type` of `Exp`, presumed to be well-typed in context `H`. This is of course in general undecidable in Martin Löf's Type Theory, which is why I used the word guess. On backtracking it enumerates all educated guesses (see `guess-type/3`).

```
| ?- type_of([],lambda(u,lambda(v,member(u,v))),T).
T = int=>int list=>u(1)

| ?- type_of([],lambda(u,app(u,u)::nil),T).
T = _7317 list=> (_7317 list)list
```

2.2.3 The method language: connectives

This section discusses the connectives that can be used in the method-language. Obviously, we can use any Prolog connective (such as `;`, `\+` etc) but we rather develop our own principled (?) set of connectives (as with the predicates discussed above). The only "native" Prolog connective that we recommend is conjunction: "`,`". Again, the connectives are discussed in alphabetical order.

`thereis {?Var\+List}:+Pred`

Extensional bounded existential quantification: succeeds if `Pred` succeeds for some element of `List`. `Var` will become bound the first such element. All other variables mentioned in `Pred` will remain unbound. `Var` can occur in `Pred`. `Var` can also be a general term containing variables. This term should then be the element of `List` for which `Pred` succeeds, and all variables occurring in this term will be bound. The `thereis/1` predicate produces only the first element in `List` for which `Pred` succeeds, and fails on backtracking.

`thereis {?Var\+Pred1}:+Pred2`

Intensional bounded existential quantification: succeeds if `Pred2` succeeds for some element in the set of values for `Var` specified by the predicate `Pred1`. `Var` becomes bound to the first such element. All other variables in `Pred1` and `Pred2` will remain unbound. `Var` can occur in both `Pred2` in `Pred1`. `Var` can also be a general term containing variables. `Pred1` must then specify values for each of these variables such that `Pred2` holds. Only variables occurring in `Var` can be referred to in `Pred2`. The `thereis/1` predicate does not backtrack to find further elements in `Pred1` for which `Pred2` succeeds. This is a *bounded* quantifying construct: the set of values for `Var` specified by `Pred1` must be *finite* (this is needed to ensure finite failure of this predicate). The intensional bounded existential quantification can be expressed in terms of the extensional bounded quantification as follows:

```
thereis {Var\Pred1}:Pred2 :-
    findall(Var, Pred1, List), thereis {Var\List}:Pred2.
```

`thereis {?Var}:+Pred`

Minor variation on the existential quantification constructs. Succeeds if there is a value of `Var` for which `Pred` succeeds. `Var` becomes bound to the first such value. `Var` can occur in `Pred`. All other variables in `Pred` remain unbound. `Var` can also be a general term containing variables. Does not backtrack over alternative values of `Var` for which `Pred` holds.

`forall {?Var\+List}:+Pred`

Extensional bounded universal quantification: `forall {Var\List}:Pred` succeeds if `Pred` succeeds for each element of `List`. `Var` can occur in `Pred`. This succeeds for any `Pred` if `List` is empty. All variables mentioned in `Pred` (including `Var`) will remain unbound. `Var` does not have to be a uninstantiated variable, but can be an arbitrary term containing variables, each of which can then be used in `Pred`. The following examples all succeed without any variables getting bound:

```
:- forall {X\[1,2,3]}:number(X).  :- forall
{f(X,Y)\[f(1,a),f(2,b)]}:(number(X), atomic(Y)).
```

`forall {?Var\+Pred1}:+Pred2`

Intensional bounded universal quantification: succeeds if `Pred2` succeeds for each element in the set of values for `Var` specified by `Pred1`. `Var` can occur in `Pred2` and `Pred1`. This succeeds if `Pred1` specifies the empty set (i.e., if `Pred1` fails for any value of `Var`). All variables occurring in `Pred1` and `Pred2` (including `Var`) remain unbound. Again, `Var` can also be an arbitrary term containing variables. Only the variables occurring in `Var` can be referred to in `Pred2`. This is a *bounded* quantifying construct: the set of values for `Var` specified by `Pred1` must be *finite*. The intensional bounded universal quantification can be expressed in terms of the extensional bounded quantification as follows:

```
forall {Var\Pred1}:Pred2 :- findall(Var, Pred1, List), forall
{Var\List}:Pred2.
```

`listof(?Term, +Pred, ?TermSet)`

This predicate is as the Prolog built-in `setof/3`, except that this predicate does not fail. If `Pred` never succeeds, `TermSet` will be the empty list `[]`, instead of the predicate failing (as `setof/3` does). In this respect, `listof/3` is like the Quintus library predicate `findall/3`, except that `TermSet` is indeed a set, and not a list. Thus, `listof/3` can be thought of as:

```
listof(Term,Pred,TermSet) :- setof(Term,Pred,TermSet),!.
listof(_,_,[]).
```

`map-list(?OL, +OE:=>NE, +Pred, ?NL)`

The predicate maps `OL` into `NL` by applying `Pred` to each element. `OE` and `NE` must occur in `Pred`, and are regarded as input- and output-argument respectively. Values for `OE` are taken from `OL` and values for `NE` are used to for `NL`. If `Pred` is bidirectional, then `map-list` works bidirectionally as well. (N.B.Variables in `Pred` which do not appear in `OE` or `NE` get uniformly renamed.)

Essentially this predicate produces the extension of a function (`Pred`) to a set (`OL`). Example:

```
:- map_list([1,2,3], I:=>O, O is I+10, L).
L = [11,12,13].
:- map_list([a+1,b+2,c+3], C+N:=>C+NN, NN is N+10, L).
L = [a+11,b+12,c+13]
```

`map-list-filter(?OL, ?OE:=>?NE, +Pred, ?NL)`

This is the same as `map-list/4` but elements for which `Pred` fails are dropped from `NL`.

`map-list-history(?OL,?X-?OE:=>?NE,+Pred,?NL,?Hist)`

This is the same as `map-list/4` with the exception that the list of so-far-computed `NL` is unified with `X` before `Pred` is called. `Hist` is the 'initial' value of this so-far-computed list—normally the empty list. This means that the mapping can be dependent upon the mapping of previous elements.

Here is an example which enumerates the elements of a list:

```
?-  map_list_history([a,b,c,d],  (X-N):=>L-N, length(X,L), New, [] ).

New = [0-a,1-b,2-c,3-d]
```

`map-list-history-filter(?OL,?X-?OE:=>?NE,+Pred,?NL,?Hist)`

This is the merge of `map-list-filter/4` and `map-list-history/5`. It filters and gives access to the history too.

Here is an example which removes duplicates from a list:

```
?-  map_list_history_filter([a,b,c,c,b],
                             (X-N):=>N,
                             \+member(N,X),
                             New, []).
New = [a,b,c]
```

`not +Goal`

Meta-linguistic negation by failure. Exactly as Prolog's `\+`. Variables in `Goal` will not be bound.

`+G1 orelse +G2`

Committed disjunction. `G1 orelse G2` will execute `G1` but if this fails will execute `G2`. The only difference between `G1 orelse G2` and `G1 or G2` is that the `orelse` construct does not allow backtracking over `G1`. For Prolog hackers: `G1 orelse G2` is shorthand for `(G1,!);G2` or equivalently `G1->true;G2`.

2.2.4 Compound methods

A distinction can be made between *simple methods* and *compound methods*. Simple methods use only the predicates and connectives described in §2.2 to formulate their preconditions and postconditions. A method is called compound when it calls other methods from its pre- or postconditions. Thus, compound methods can be seen as built out of other methods (together with the predicates and connectives from the method language). The `ind-strat` method (as described in [4]) is an example of such a compound method. The method-language from §2.2 on page 11 offered no constructs for calling other methods from a method's pre- or postconditions, and the first subsection below describes how this can be done.

Calling other methods from a method's pre- or postconditions can also be done in a fail-safe way if that is required, using the `try/1` connective. Methods can also be combined in a sequential way using the `then/2` connective or disjunctively using the `or/2` connective. All these connectives are described below. (All these connectives are very close to the Oyster tacticals of the same names).

A final way of constructing compound method are the *iterating methods*. Such an iterating method (also called an *iterator*) is build by exhaustively iterating another method (or set of methods). The final subsection below describes how to build iterating methods.

2.2.4.1 Calling other (sub)methods

The main predicates for calling a (sub)method from the pre- or post-conditions from another method are `applicable/[2;4]` and `applicable-submethod/[2;4]`, described in §2.4.1 on page 71. Figure 2.3 shows part of the `normal/1` submethod which checks as part of the preconditions whether there is a lemma that can be used to prove the antecedent of an implication occurring in the hypothesis-list. An alternative (and equivalent) way of formulating this method would of course have been to include the code for the `apply-lemma/1` and `backchain-lemma/1` methods in the preconditions of the `normal/1` submethod, but the formulation of figure 2.3 has all the obvious advantages of modularity, readability etc. In the preconditions of the `normal/1` submethod it is only necessary to know if the `apply-lemma/1` or `backchain-lemma/1` methods apply, but it is not necessary to know what the output sequent and postconditions of these methods are if they apply. Therefore, the predicate `applicable/2` is used. Had it been necessary to know the output sequent and/or the postconditions of the `apply-lemma/1` or `backchain-lemma/1` methods, the predicate `applicable/4` should have been used instead.

As shown in figure 2.3, methods can also be represented using a `submethod/6` term. This `submethod/6` representation should be used if the method is not to be used in its own right during the plan formation, but only as a submethod to be called from other methods (which should then use the `applicable-submethod/[2;4]` predicate for this purpose). Thus, both methods represented by `method/6` and as `submethod/6` can be called from the pre- and postconditions of other methods, but only methods represented by `method/6` will be used as building blocks by the plan-formation programs.

2.2.4.2 Iterating (sub)methods

It is possible to create a new method out of a given set of (sub)methods by constructing an *iterator*. This iterator will exhaustively apply the elements of the given set until none of them apply any longer. Thus, the application of an iterator is equivalent to a maximally long chain of applications of the given (sub)methods. The preconditions of the newly constructed iterator will state that at least one of the methods in the given set is applicable, and the postconditions will state that none of the methods in the given set is applicable.

The representation of an iterating method is as an `iterator/4` term:

`iterator(+IteratorType,+Name,+IteratedType,+MethodList)`

When this term is read when loading methods, a new (sub)method `Name` will be constructed which iterates the (sub)methods specified in `MethodList`. The tag `IteratorType` must be one of the atoms `method` or `submethod`, and specifies whether the resulting iterator `Name` is a method or a submethod. `Name` must be an atom, and the elements of `MethodList` must be specified as skeletal functors[2]. `IteratedType` must be one of the atoms `methods` or `submethods`, and specifies whether the iteration is over methods or over submethods. The resulting method will be `Name` of arity 1 (`Name/1`), and will exhaustively apply the elements of `MethodList` until none of them apply any longer. Thus, the application of a `Name/1` is equivalent to a maximally long chain of applications of the elements of `MethodList`. The preconditions of the newly constructed iterator method `Name/1` will state that at least one element of `MethodList` is applicable, and the postconditions will state that none of the elements of `MethodList` is applicable.

Using various combinations of the values for `IteratorType` and `IteratedType` it is possible to construct a method that iterates submethods, a submethod that iterates methods, etc.

For example, the following `iterator/4` term constructs an iterating method `sym-eval/1` which iterates 3 submethods:

```
iterator(submethod,sym_eval,submethods,
        [ equal(_,_),
          normalize_term(_),
          existential(_,_) ]).
```

An example application of the newly constructed `sym-eval/1` method would be:

```
| ?- applicable(M,_,_).
   M = sym_eval([normalize_term(
                 [reduction([1,1],[plus1,equ(left)]),
                  reduction([1,2,1],[plus1,equ(left)])])])
```

Thus, the single argument of the newly constructed iterator (sub)method will become bound to the list of applications of the iterated (sub)methods. Clam's pretty-printer (described in §3.2.1 on page 88) treats iterator (sub)methods specially. It suppresses the single argument of `Name/1`, since this argument gets very long and complicated in general and does not carry much information that a user would want to see. It prints this list as `[...]`, where the number of .s represents the number of iterations applied.

The construction of an iterating method using the `iterator/4` predicate will also result in the construction of a tactic of the same name as the iterating method for execution purposes. Notice that this automatic construction of a tactic (i.e., a Prolog predicate) might result in overwriting an existing Prolog predicate that accidentally has the same name.

The method constructed with `iterator/4` iterates the given set of methods exhaustively. Thus, only the longest possible chain of consecutive applications of methods in the given set will be generated by the iterator. No subsequences of this chain of applications will be generated. Also, no permutations of this chain will be generated (the elements in the given set are tried in the order in which the set was specified). Small modifications in the code of `iterate-methods/4` would make it possible to generate subsequences or permutations or both. See §4.2 on page 113 in the *Programmer Manual* for details.

`repeat(+IGs,+CG:=>SGs,+PS,+FS,+SPs,+OGs)`

`repeat/6` can be used to construct sub-planners, that is methods which construct a possibly branching sub-plan depth first. For each sub-goal in `IGs` `repeat` unifies it with `CG` and invokes `FS`. This, if it succeeds binds the plan step to be recorded to `PS`

[2] A *skeletal functor* is a functor with all arguments uninstantiated. For instance `f(_,_)` is the skeletal functor corresponding to `f/2`.

and the subgoals (if any) to `SGs`. `repeat/6` then recursively invokes itself (depth-first) on the sub-goals introduced until there are no further subgoals to which `FS` can be successfully applied. When it completes the plans found for each goal in `IGs` they appear in the corresponding position in `SPs`. An empty plan is denoted by the token `idtac`. Any open sub-goals left over are unified with `OGs`.

`iterate(+IArg,+CArg :=> NArg,+Proc,+CutCond,-OArg)`

`iterate/5` is used to iterate a given procedure, `Proc`, over some argument, `IArg`, with control over backtracking. When invoked `iterate` unifies `IArg` with `CArg` and then invokes `Proc`. If either this unification or the procedure invocation fails it finishes and unifies `IArg` with `OArg`. If `Proc` succeeds `iterate` then invokes `CutCond`. If this fails `NArg` is bound to `OArg` and `iterate` succeeds and `Proc` may resatisfy on backtracking. If `CutCond` also succeeds `iterate` recursively invokes itself with `IArg` replace with `NArg`, but any resatisfaction of `Proc` is cut. This rather specialised behaviour turns out to be just what you need for rewriting methods that should apply some submethods exhaustively except at points where looping might be introduced.

`iterate-lazy(+IArg,+CArg :=> NArg,+Proc,+CutCond,-OArg)`

`iterate-lazy/5` is as `iterate/5` but it prefers shorter rather than longer iterations. It tries sequences of length 0, then 1 on backtracking, then 2 and so on.

This form of iteration is useful for controlling unblocking in the inductive proof plan: we only want a minimal amount of unblocking rather than a maximal amount. See `unblock-lazy/1` for an example.

2.2.4.3 Calling fail-safe (sub)methods

Sometimes it is desirable to apply a method if applicable, but to not fail if the method is not applicable. An example of the need for such a fail-safe mechanism of applying (sub)methods is the application of the `eval-def/2` method in the base-case of an induction. If the `eval-def/2` is applicable we want to apply it, otherwise we just want to leave the base-case sequent unchanged. The `try/1` connective implements this mechanism:

`try +Method`

The construct `try Method` is always applicable, whether `Method` is applicable or not. If `Method` is applicable, the postconditions and output-sequent of try Method are as the postconditions and output-sequent of `Method`. If `Method` is not applicable, the postconditions of `try Method` are empty and the output-sequent is a singleton list consisting of the input-sequent. Thus: if `Method` is applicable, then `try Method` is like `Method`, if `Method` is not applicable, then `try Method` is the identity operation.

An example of the use of `try/1` is in combination with an iterator. As explained in the previous section, an iterator succeeds only if at least one of the iterated methods can be applied. In combination with the `try/1` connective, an iterator can be seen as a chain of 0 or more applications, instead of a chain of 1 or more applications.

2.2.4.4 Disjunctively combining (sub)methods

Methods and submethods can be combined disjunctively (that is: either one (sub) methods is applied or the other), using the `or/2` connective:

`M1 or M2`

The construct `M1 or M2` is applicable whenever `M1` is applicable to the given input sequent or `M2` is applicable to the given input sequent. The system will first try the applicability of `M1` , and, on failure or backtracking, try the applicability of `M2`. The output sequents of the `or/2` construct are the output sequents of the chosen applicable method.

2.2.4.5 Sequentially combining (sub)methods

Methods and submethods can be combined sequentially (that is: one (sub)method is applied to the output sequent of another (sub)method) using the `then/2` connective:

`M1 then M2`

`M1 then [`$\mathtt{M2}_1$`,...,`$\mathtt{M2}_n$`]`

The construct `M1 then M2` is applicable whenever `M1` is applicable to the given input sequent and `M2` is applicable to each output sequent of `M1`.[3] The construct `M1 then [`$\mathtt{M2}_1$`,...,`$\mathtt{M2}_n$`]` is applicable whenever `M1` is applicable to the given input sequent and each element of `[`$\mathtt{M2}_1$`,...,`$\mathtt{M2}_n$`]` is applicable to the corresponding output sequent of `M1` (thus, `M1` is required to have n output sequents). The output sequents of the `then/2` construct are the output sequents of the applications of `M2` or `[`$\mathtt{M2}_1$`,...,`$\mathtt{M2}_n$`]`.

2.3 The method database

Clam provides a database for storing methods and submethods. Methods and submethods can be individually loaded and stored in the database. The distinction between methods and submethods only arises when they are loaded into the database. As stored in the library, and as described in this manual, there are no submethods. However, a method may be loaded in such a way that it enters the database as a submethod.

The order of the database can be changed by the user, and the contents of the database can be inspected. It is also possible to remove methods from the database. The manipulation of the (sub)methods databases (adding and deleting methods) is integrated with Clam's general library mechanism, and is described in more detail in §3.3 on page 92.

The order in which the methods occur in the database is significant. A number of planners (for instance the depth-first planner, see §2.4.2 on page 74) try to apply methods in the order in which they appear in the database. As a result, methods which are cheap should occur at the top of the database. Cheap can mean a number of things, for instance that the preconditions are easily tested, or that the preconditions lead to failure quickly if the method is not applicable. Another reason for putting methods early in the database used to be when they lead to termination of plans. Such methods are in general a good thing to choose if they are applicable, and thus should be tried early on in the search for applicable methods. However, most planners described in §2.4 are smart enough to first look for terminating methods themselves before looking for other methods, so that the place of terminating methods in the database no longer matters.

Some methods assume the presence of other methods, and not all combinations of methods are meaningful or effective. The dependencies between methods can

[3] Note that this is applicable with the *same* instantiation: i.e., `M2` is instantiated.

be expressed using the `needs/2` predicate that is provided by the Clam's library mechanism (see §3.3).

Apart from the general library predicates, the following predicates are available for inspecting the current databases for methods and submethods:

`method(?M,?I,?Pre,?Post,?O,?T)`

This is the main predicate for accessing the elements in the database of methods: `M` is a method with input sequent `I`, preconditions `Pre`, postconditions `Post` and output sequent list `O`. `T` is the tactic associated with `M`. (Current convention is that `M==T`).

`submethod(?M,?I,?Pre,?Post,?O,?T)`

As the `method/6` predicate, but for the database of submethods.

`list-methods(?L)`

Unifies `L` with a list representing the current order of methods in the database. Each element of `L` is a method specification of the form `Functor/Arity`.

`list-methods`

As `list-methods/1`, but instead prints the result on the current output stream.

`list-submethods(?L)`

As `list-methods/[0;1]`, but for the database of submethods.

2.3.1 Current repertoire of (sub)methods

This section will briefly described the current set of methods in Clam. The purpose of describing the current set of methods is instead to give a brief introduction-by-example into the art of method-writing.

`elementary(I)`

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: elementary,v 1.10 1998/11/10 16:08:20 img Exp $
 *
 * $Log: elementary,v $
 * Revision 1.10  1998/11/10 16:08:20  img
 * Extract Presburger information from goal; Added elementary(fertilize(_))
 *
 * Revision 1.9  1998/09/15 16:00:33  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.8  1998/07/27 12:43:38  img
 * switch on Presburger
 *
 * Revision 1.7  1998/06/10 09:23:34  img
 * *** empty log message ***
 *
 * Revision 1.6  1998/06/10 08:26:28  img
 * rationalisation: prohibit backtracking
 *
 * Revision 1.5  1997/10/09 17:17:15  img
 * Unify treatment of annotations
 *
 * Revision 1.4  1997/05/08 12:56:16  img
 * Use lpa_dp/2
 *
 * Revision 1.3  1997/05/05 16:13:39  img
 * Place-holder for decision prodecures.
 *
 * Revision 1.2  1997/01/14 10:48:50  img
 * Integration of decision procedure.
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 */

/* ELEMENTARY METHOD: applies if current method is a trivial instance
```

```
   of the hypotheis, a propositional tautology, or a true statement of
   Presburger arithmetic.  */

method(elementary(fertilize(Hyp)),
       H==>G,
       [erase(G,GE),matrix(_,Matrix,GE),
hyp(Hyp:Hypothesis,H),
        instantiate(H,[],Hypothesis,Matrix,_) ],
       [],
       [],
       elementary(fertilize(Hyp))).

method(elementary(I),
       H==>G,
       [erase_sequent(H==>G,HH==>GG),
once(((propositional(HH==>GG,I) -> Seqs = []);
      (using_presburger,
       presburger_context(H,Arith),
       precon_matrix([],Arith=>GG,ArithGG),
       universal_closure(HH,ArithGG,Gclosed),
       ex(lpa_dp(Gclosed,yes),60),
       /* can only conclude {false} when entire goal is Presburger */
       Seqs = [], I = presburger)))],
       [],
       Seqs,
       elementary(I)).
```

This terminating method deals with simple goals via (`elementary/1`), and terminates the plan iff this succeeds; `I` will become bound to the sequence of Oyster inference rules which are needed to prove the input sequent. Because this sequence becomes very long and boring very quickly, Clam's pretty-printer (see §3.2.1 on page 88) treats the term `elementary(I)` specially, and prints the `I` as `...`. See `elementary/2` for more details.

---

propositional(I)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: propositional,v 1.2 1997/10/09 17:17:16 img Exp $
 *
 * $Log: propositional,v $
 * Revision 1.2  1997/10/09 17:17:16  img
 * Unify treatment of annotations
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 */

method(propositional(I),
       H==>G,
       [erase_sequent(H==>G,HH==>GG),
matrix(Vs,M,GG),
append(Vs,HH,Hyps),
propositional(Hyps==>M,I)],
       [],
       [],
       propositional(I)).
```

This terminating method calls a decision procedure for intuitionistic propositional logic. (Quantification over propositions is removed if present.) The predicate that does all the work is `propositional/2`, which is an implementation of Dyckhoff's algorithm.

If the method is applicable, `I` will become bound to the sequence of Oyster inference rules which are needed to prove the input sequent. Because this sequence becomes very long and boring very quickly, Clam's pretty-printer (see §3.2.1 on page 88) treats the term `propositional(I)` specially, and prints the `I` as `...`.

Notice that we need to remove meta-level annotations from the formula before running the decision procedure.

---

equal(HName,Dir)

```
/* -*- Mode: Prolog -*-
 * @@(#)$Id: equal,v 1.5 1998/11/10 16:08:49 img Exp $
 *
 * $Log: equal,v $
 * Revision 1.5  1998/11/10 16:08:49  img
 * Use inductive_hypothesis/3
 *
 * Revision 1.4  1998/07/30 16:07:57  img
 * drop junk
 *
 * Revision 1.3  1998/07/27 12:51:24  img
```

```
 * Only use hyps in non-inductive branches otherwise may spoil fertilization
 *
 * Revision 1.2  1997/04/07 11:02:38  img
 * Document conditions on annotation
 *
 * Revision 1.1  1996/12/11 15:09:18  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

        % Use equalities of the form v1=v2 or v1=t where v1 and v2 are
        % vars and t is a constant term.
        % We always substitute vars by constant terms, and we rewrite
        % var-equalities depending on alphabetic order. (Boyer and More
        % use the same hack).
        %
        % After having done the substitution, it's not clear if it's
        % safe to throw away the equal from among the hypotheses
        % (this is what B&M do).
        % The goal is assumed to be unannotated
method(equal(HName,Dir),
       H==>G,
       [((hyp(HName:Term=Var in T,H), Dir=left)
     v
     (hyp(HName:Var=Term in T,H), Dir=right)),
\+ inductive_hypothesis(_,HName:_,H),
(not freevarinterm(Term,_)
     orelse
     (atomic(Var), not atomic(Term),
 not exp_at(Term,_,Var))
     orelse
     (atomic(Var), atomic(Term), Term @< Var)),
freevarinterm(G,Var)],
       [replace_all(Var,Term,G,GG),
del_hyp(HName:_,H,HThin) ],
       [HThin==>GG],
       equal(HName,Dir)).
```

This method checks if there is any equality among the hypotheses. If so, we use the equality to rewrite all hypotheses and the goal. To give the equality a unique direction as a rewrite rule, we always rewrite towards the alphabetically lowest term (using the Prolog term-comparison predicate `@<`). After this rewriting is done, we can throw away the equality. This is essentially the same approach to equalities as taken in [3].

This method is normally applied as part of the `sym-eval/1` iterator.

`reduction(Pos,[Thm,Dir])`

```
/* -*- Prolog -*-
 * @(#)$Id: reduction,v 1.3 1998/09/15 16:00:41 img Exp $
 *
 * $Log: reduction,v $
 * Revision 1.3  1998/09/15 16:00:41  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.2  1997/10/09 17:18:48  img
 * trivia
 *
 * Revision 1.1  1996/12/11 14:08:47  img
 * Merge mthd and smth libraries.
 *
 * Revision 1.3  1996/05/23  11:20:39  img
 * incorrect argument order in reduction_rule/6
 *
 * Revision 1.2  1996/05/14  14:50:43  img
 * Updated in line with revisions to the reduction rule mechanism.  This
 * version allows the registry to be extended on the fly, providing
 * extending_registry/0 succeeds.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

/* Normalize the goal by exhaustive application of rewrite rules from
   a terminating rewrite system.  Rules are applied in an order
   determined by exp_at (outermost-rightmost at the time of writing).

   (In fact, this method only applies a single rule, so it normalizes
   only when used inside an iterator: we call this one-step relation
   "reduction".)

   Rewrite rules come from the recorded database of reduction rules.
   These rules are known to be terminating under Clam's global EPOS
   registry.  There two clauses: the first attempts to apply an
   existing rule and the second will extend the registry (and the
   reduction rule database) if a rewrite rule can be oriented.

   NB: this method does not attempt to clear the rewrite system nor
   does it initialize the registry.  Such things are left to other
   parts of the plan (perhaps in a calling method).  */

/* First try to use a known reduction rule, otherwise (if enabled)
   attempt to extend the registry with an arbitrary rewrite rule.  */
```

```
method(reduction(Pos,[Thm,Dir]),
  H==>G,
          [matrix(Vars,Matrix,G),
     exp_at(Matrix,Pos,LHSground),
   \+ Pos = [0|_],
   not metavar(LHSground),
   ((reduction_rule(LHSground,RHSground,Cground,Dir,Thm,_),
     polarity_compatible(Matrix, Pos, Dir))
     -> true
     ;  (extending_registry,%do we want to do this?
 rewrite_rule_speedy(LHSground,RHSground,Cground,Thm,Dir,RuleRef),
 polarity_compatible(Matrix, Pos, Dir),
 %% fetch the same rule, uninstantiated and make it
 %% into a reduction rule, if possible
 rewrite_rule(LHS,RHS,C,Thm,Dir,RuleRef),
 registry(Tau,Prec,RegistryRef),
 (extend_registry_prove(Tau-Prec,TauExt-PrecExt,LHS,RHS),
  (Tau-Prec == TauExt-PrecExt
       -> true % no extension required
       ;  (erase(RegistryRef),% delete the old
   recorda(registry,registry(TauExt,PrecExt),_),
   numbervars(C-LHS-RHS,1,_),
   clam_info('Registry extended to %t\n',
     [C=>LHS:=>RHS])))))),
   immediate(H==>Cground)],
          [replace(Pos,RHSground,Matrix,NewMatrix),
           matrix(Vars,NewMatrix,NewG)],
          [H==>NewG],
          reduction(Pos,[Thm,Dir])).

/* cancellation rules are used during post-fertilization rippling, but
   this is not yet fully implemented in this version.  */
/* method(reduction(Pos,Rule),
          H==>G,
          [matrix(Vars,Matrix,G),
           cancel_rule(Exp,Rule:PC=>Exp:=>NewExp),
           exp_at(Matrix,Pos,Exp),
           object_level_term(Exp),
           immediate(H==>PC) ],
          [replace(Pos,NewExp,Matrix,NewMatrix),
           matrix(Vars,NewMatrix,NewG) ],
          [H==>NewG],
          reduction(Pos,[Rule,imp(left)]) ).
*/
```

This method attempts to apply reduction rules. If an applicable reduction rule cannot be found in the current environment, an attempt is made, providing `extending-registry/0` succeeds, to extend the set of reduction rules. This is done by proving that a rewrite rule is measure decreasing under RPOS—if this is successful, the new rule is added to the reduction rule database.

See `extend-registry-prove/4` for more details on extending the reduction rule database. See §4.3.2 and §A.4 for more information.

---

eval-def(Pos, Rule)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: eval_def,v 1.2 1998/09/15 16:00:33 img Exp $
 *
 * $Log: eval_def,v $
 * Revision 1.2  1998/09/15 16:00:33  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1996/12/11 15:09:20  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.4  1996/05/23 11:20:36  img
 * incorrect argument order in reduction_rule/6
 *
 * Revision 1.3  1996/05/14  16:00:09  img
 * Use reduction rules
 *
 * Revision 1.2  1995/12/04  14:11:57  img
 * evaluation order documented;  don't eval_def functor positions
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

/* Symbolically evaluates a term in the goal by applying one of its
 * defining equations.  In order to prevent interference with rippling
 * it will not apply when waves are present.
 *
 * Evaluation is in a outermost/rightmost reduction strategy.  This
 * ordering is as result of exp-at/3 term traversal.  */
method(eval_def( Pos, [Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),
        wave_fronts(_, [], Matrix),
        exp_at(Matrix,Pos,Exp),
\+ Pos = [0|_],%don't eval functors
        not metavar(Exp),%or meta-variables
reduction_rule(Exp,NewExp,C,Dir,Rule,_),
        polarity_compatible(Matrix, Pos, Dir),
immediate(H==>C)],
```

```
[% Once have applied a base-case ignore wave-fronts
 replace(Pos,NewExp,Matrix,NewMatrix),
 matrix(Vars,NewMatrix,NewG)
],
[H==>NewG],
eval_def(Pos,[Rule,Dir])).
```

This method looks for applicable base and step-equations instead of wave-rules. Furthermore, we require that the expression to be rewritten does not contain any wave-fronts, and does not consist solely of a meta-variable. This is not strictly logically needed (applying a base or step rule to a meta-variable can produce a legal proof step), but is introduced to restrict the applicability of this method. Without this restriction, every base and step rule will always apply to every occurrence of every meta-variable in `G`, thus exploding the number of possible applications of this method.

`existential(Var:Type, Value)`

The existential method is designed to deal with existentially quantified base case proof obligations and form part of the `sym-eval/1` iterator. As they stand this submethods is not very general. Ideally the submethods `equal/2`, `reduction/2` and `eval-def/2` should be modified to deal with rewriting within existential quantification in the same way rippling has been extended.

`existential/2` deals with existentially quantified equalities where the existential variable occurs isolated on one side of the equality:

```
/* -*- Prolog -*-
 * @(#)$Id: existential,v 1.4 1998/09/15 16:00:34 img Exp $
 *
 * $Log: existential,v $
 * Revision 1.4  1998/09/15 16:00:34  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.3  1997/04/07 11:38:32  img
 * Document assumption about goal annotation
 *
 * Revision 1.2  1996/12/11 14:07:12  img
 * Merge mthd and smthd libraries.
 *
 * Revision 1.4  1996/05/23  11:20:38  img
 * incorrect argument order in reduction_rule/6
 *
 * Revision 1.3  1996/05/14  16:01:53  img
 * Use reduction rules
 *
 * Revision 1.2  1995/06/06  14:34:27  img
 *  * Corrected bug in which the method succeeded but with a
 *    faulty result when the variable substition is on R in L = R
 *    (debugged by Julian Richardson).
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 *
 */

/* The goal is assumed to be unannotated */
method(existential(Var:Typ1,Value),
          H==>G,
          [matrix(Vars,Var:Typ1#(LG = RG in Typ2),G),
    ((LG = Exp1,RG = Exp2,NewMat = (NewExp = Exp2 in Typ2)) v
     (RG = Exp1,LG = Exp2,NewMat = (Exp2 = NewExp in Typ2))),
    not wave_fronts(_,[_|_],Exp1),
    not atomic(Exp1),
    replace_all(Var,Value,Exp1,NewExp),
    reduction_rule(NewExp,Exp2,C,_,Rule,_)],
          [matrix(Vars,NewMat,NewG)],
          [H==>NewG],
          existential(Var:Typ1,Value)).
```

`normalize-term(Tac)`

Normalize the goal by exhaustive application of rewrite rules from a terminating rewrite system (as described by the reduction rule database). Uses labelled rewriting for speed; flattens rule application into a single tactic invocation. Conditions are decided inside `reduction-tc/4`.

This method will fail if the goal is already in normal form.

```
/* -*- Prolog -*-
 * @(#)$Id: normalize_term,v 1.6 1998/09/15 16:01:09 img Exp $
```

```
 *
 * $Log: normalize_term,v $
 * Revision 1.6  1998/09/15 16:01:09  img
 * *** empty log message ***
 *
 * Revision 1.5  1998/09/15 16:00:40  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.4  1998/06/10 09:31:00  img
 * clause for rewriting with definitional eqns
 *
 * Revision 1.3  1997/10/09 17:18:10  img
 * new clause for cancellation (disabled)
 *
 * Revision 1.2  1997/04/07 11:04:01  img
 * Document condition on annotation
 *
 * Revision 1.1  1996/12/11 14:08:46  img
 * Merge mthd and smth libraries.
 *
 * Revision 1.1  1996/06/12  10:48:43  img
 * Normalize the goal using nf_plus/4 (which used labelled term
 * rewriting).
 *
 */

/* Normalize the goal by exhaustive application of rewrite rules from
   a terminating rewrite system (as described by the reduction rule
   database).  Uses labelled rewriting for speed; flattens rule
   application into a single tactic invocation.  Conditions are
   decided inside reduction_tc/4.

   This method will fail if the goal is already in normal form.

   The goal is assumed to be unannotated.  */
method(normalize_term(Tactic),
       H==>G,
       [matrix(Vars,Matrix,G), append(H,Vars,Context),
reduction_tc(Matrix,MatrixNF,Tactic,Context)],
       [matrix(Vars,MatrixNF,NewG) ],
       [H==>NewG],
       normalize_term(Tactic) ).


/* This clause deals with terms of the form f(X) = f(Y) --> X = Y.
   Note that this rule is terminating under any simplification
   ordering: it cannot be oriented from right-to-left under any
   registry, and it can be oriented from left-to-right under the empty
   registry.  Deals only with unary f.  */
/*
method(normalize_term(Tac),
       H==>Goal,
       [matrix(GVs,G,Goal),
exp_at(G,Pos,Term1 = Term2 in Type),
Term1 =.. [F,A1], Term2 =.. [F,A2],
append(GVs,H,GVsH),
once(((type_of(GVsH,A1,Atype), ground(Atype));
      (type_of(GVsH,A1,Atype), ground(Atype)))),
Tac = cancel(Pos,Term1 = Term2 in Type, A1 = A2 in Atype) ],
       [replace(Pos,A1 = A2 in Atype, G,NewG),
matrix(GVs,NewG,NewGoal) ],
       [H==>NewGoal],
       normalize_term(Tac) ).
*/

/* Use definitional rewrite rules
method(normalize_term(blank),
       H==>G,
       [matrix(Vars,Matrix,G), append(H,Vars,Context),
        exp_at(Matrix,Pos,Exp),
\+ Pos = [0|_],%don't eval functors
        not metavar(Exp),%or meta-variables
rewrite_rule(Exp,NewExp,C,equ(_,left),Origin,Rule,_),
\+ Origin == Rule,% force use of defs from left-right
        polarity_compatible(Matrix, Pos, Dir),
        immediate(Context==>C)
       ],
       [% Once have applied a base-case ignore wave-fronts
        replace(Pos,NewExp,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG) ],
       [H==>NewG],
       normalize_term(blank)
      ).
*/
```

The second clause is a generalized cancellation method. Goals of the form $f(x) = f(y)$ are reduced to $x = y$. This is disabled by default.

sym-eval(SymEvals)

This method is an iterator over submethods that effects symbolic evaluation.

```
/*
 * @(#)$Id: sym_eval,v 1.6 1997/10/09 17:20:35 img Exp $
 *
 * $Log: sym_eval,v $
 * Revision 1.6  1997/10/09 17:20:35  img
 * specify type of casesplit
 *
 * Revision 1.5  1997/04/07 11:39:17  img
 * Only applicable when goal is unannotated (to prevent rewriting to
```

```
 * ill-annotated terms)
 *
 * Revision 1.4  1997/04/07 10:34:49  img
 * Allow branching proofs via repeat methodical
 *
 * Revision 1.3  1996/12/11 15:07:21  img
 * New methods for lazy application of unblocking.
 *
 * Revision 1.2  1996/06/12 10:44:19  img
 * Use new reduction rule machinery.
 *
 * Revision 1.2  1996/05/14  16:00:58  img
 * Fast symbolic evaluation using nf/2.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

method(sym_eval(SubPlan),
       H==>G,
       [unannotated(G),
       repeat([H==>G],
               Goal :=> SubGoals,
               Method,
               (member(Method, [equal(_,_),
                                normalize_term(_),
                                casesplit(disjunction(_)),
                                existential(_,_)]),
                applicable_submethod(Goal, Method, _, SubGoals)),
               [SubPlan],
               SubGoals
              ),!,
        SubPlan \= idtac ],
       [],
       SubGoals,
       SubPlan ).
```

base-case(Plan)

The `base-case/1` method constructs a plan for base case proof obligations using the submethods `elementary/1` and `sym-eval/1`.

```
/*
 * @(#)$Id: base_case,v 1.3 1998/09/15 15:30:02 img Exp $
 *
 * $Log: base_case,v $
 * Revision 1.3  1998/09/15 15:30:02  img
 * unfold sym_eval into base cases in order to avoid reappling equal,
 * normalize_terms (etc) when elementary is applicable.  Consider
 * base-case of assp: normalize_term is applied, then all others in
 * sym_eval fail (including testing normalize_term again), before
 * dropping back to elementary.
 *
 * Revision 1.2  1997/04/07 10:34:48  img
 * Allow branching proofs via repeat methodical
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 */

method(base_case(SubPlan),
       HG,
       [repeat([HG],
               Goal :=> SubGoals,
               Method,
               (member(Method, [elementary(_),
equal(_,_),
                                normalize_term(_),
                                casesplit(disjunction(_)),
                                existential(_,_)]),
                applicable_submethod(Goal, Method, _, SubGoals)),
        [SubPlan],
        SubGoals
       ),!,
       SubPlan \= idtac ],
    [],
    SubGoals,
    SubPlan).
```

wave(Pos, Rule, Subst)

Rippling. The first clause of the `wave` method deals with rippling. `Type` restricts the rippling to outwards wave-fronts (`direction_out`), inwards (`direction_in`) or either of these (`direction_in_or_out`). This control may be useful when writing plans. Note that `ripple/6` carries out a check on the sink-ability of any inward fronts in `NewWaveTerm` and so this does not need to be checked here.

We may only rewrite annotated terms: skeleton preserving steps are *not* sufficient since they do not guarantee termination. One example of this would be rewriting beneath a sink, inside a wave-front etc. These are all 'unblocking' operations (cf. `unblock/3`) since they require a termination justification from something other than the wave-rule measure.

```
/* -*- Prolog -*-
 * @(#)$Id: wave,v 1.4 1999/01/07 16:30:25 img Exp $
 *
 * $Log: wave,v $
 * Revision 1.4  1999/01/07 16:30:25  img
 * Support for ripple-and-cancel reintroduced (currently disabled,
 * however).  weak_fertilize uses larger_size/2 to try to replace larger
 * with smaller terms during weak fertilization
 *
 * Revision 1.3  1998/09/15 16:00:43  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.2  1998/06/10 08:32:01  img
 * extend context & type instantiation
 *
 * Revision 1.1  1996/12/11 14:08:52  img
 * Merge mthd and smth libraries.
 *
 * Revision 1.11  1996/07/10  09:06:30  img
 * Cosmetic changes
 *
 * Revision 1.10  1995/10/03  13:09:13  img
 * remove annotations in non-recursive case of complementary wave;  use a
 * different rule in the non-recursive case than that used in the
 * recursive case.
 *
 * Revision 1.9  1995/05/10  18:21:08  img
 *  * cc -> complementary_sets;  tidying up
 *
 * Revision 1.8  1995/05/10  03:33:34  img
 *  * Complementary wave.  Checks that a ripple is possible via a
 *    conditional rewrite, and that there are complementary
 *    rewrites.  Only the complementary rewrites are performed:
 *    wave-rewrites are left to the other clause.  The mechanism
 *    by which compelentary rules are applied is unsatisfactory
 *    since they are dealt with singly rather than n at a time
 *
 * Revision 1.7  1995/04/26  09:20:46  img
 *  * Weakening moved from wave into unblock
 *
 * Revision 1.6  1995/03/01  03:23:33  img
 *  * Dynamic rippling method
 *
 * Revision 1.5  1995/02/09  23:47:47  img
 *  * Simple wave smthd.  Only ripples out!
 *
 * Revision 1.4  1995/01/30  09:14:55  dream
 * meta-rippling is disabled.  this must be dealt with in a secure manner
 *
 * Revision 1.3  1994/09/22  12:03:01  dream
 *  * change regular wave to perform rippling dynamically
 *      * removed joining and splitting since dynamic rippling uses normal
 *    form only
 *
 * Revision 1.2  1994/09/20  15:03:28  dream
 *  * use mark_potential_waves/2 instead of potential_waves/2
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 *
 */

/* Dynamic rippling method.  "Type" restricts the rippling to outwards
   wave-fronts ("direction_out"), inwards ("direction_in") or either
   of these ("direction_in_or_out").  This control may be useful when
   writing plans.  Note that in these cases ripple/6 carries out a
   check on the sinkability of any inward fronts in NewWaveTerm.

   Type == ripple_and_cancel is similar to "direction_in" but the
   check on the sinkability of wave-fronts is not carried out.

   We may only rewrite annotated terms: skeleton preserving steps are
   NOT sufficient since they do not guarantee termination.  One
   example of this would be rewriting beneath a sink, in a wave-front
   etc.  These are all "unblocking" operations since they require a
   termination justification from something other than the wave-rule
   measure.  */

method(wave(Type,Pos,[Rule,Dir],[]),
  Hyps==>Conc,
         [matrix(Vars, Matrix, Conc),append(Hyps,Vars,Context),
   wave_terms_at(Matrix, Pos, WaveTerm),
   ripple(Type, WaveTerm, NewWaveTerm, Cond, Rule, Dir,TypeInfo),
   polarity_compatible(Matrix, Pos, Dir),
   instantiate_type_variables(TypeInfo,Context),
   immediate(Context==>Cond)],
  [replace(Pos, NewWaveTerm, Matrix, NewMatrix),
   matrix(Vars, NewMatrix, NewConc)],
  [Hyps==>NewConc],
  wave(Type,Pos,[Rule,Dir],[])).

/* Proof plan should be organised to do all n cases in one go, not
   just one at a time.

   A complementaty wave-rule is not skeleton preserving: the
   postconditions remove all annotation from the subgoals.  Really,
   all this method has to do is identify sequents which are in the
   non-recursive branches of rippling proofs.  It would be sufficient
   to leave the sequent untouched, but for removing annotation;
   however, since it is cheap to apply a rewrite, we do that here as
   well.  */
```

```
method(wave(Type,Pos,[Rule,complementary,Dir],[]),
  Hyps==>Conc,
         [matrix(Vars, Matrix, Conc), append(Hyps,Vars,Context),
   wave_terms_at(Matrix,Pos,M),
   %% one of them is a wave-rule, ...
   ripple(Type, M, _, WaveCond, _, _,_),% recursive case
   unannotated(M,Term),
   complementary_set(Cases-Term),
   member(WaveCond-_-Dir-_-RecRule,Cases),% recursive case
   %% ... which as already be dealt with in another branch.
   %% Next clause ensures we are being used as a complementary wave-rule
   member(Cond-RHS-Dir-TI-Rule,Cases),% non-recursive case
   \+ Rule == RecRule,% and a different rule
   polarity_compatible(M, Pos, Dir),% check polarity ok
   instantiate_type_variables(TI,Context),
   immediate(Context==>Cond) ],
  [replace(Pos, RHS, Matrix, NewMatrix),
   matrix(Vars, NewMatrix, NewConcAnn),
   unannotated(NewConcAnn,NewConc) % remove annotation since this
% is a non-ripple case
   ],
  [Hyps==>NewConc],
  wave(Type,Pos,[Rule,Dir],[])).
```

The third argument of `wave/4` is used to record the incremental instantiation of existential variables during the application of existential wave-rules—here it is unused.

`wave(Pos, Rule, Subst)`

Complementary rewriting. The second clause of the `wave` method (see `wave/4` above) deals with the initial rewriting of non-inductive branches of a casesplit using *complementary rewrite rules*.

Complementary rewrite rules are not skeleton preserving and so the postconditions remove all annotation from the each subgoal. Really, all this method has to do is identify sequents which are in the non-recursive branches of rippling proofs, and it does this via `complementary-set/1`. It would be sufficient to leave the sequent untouched, but for removing annotation; however, since it is cheap to apply a rewrite (the right-hand-sides are in `Cases`, we do that here as well.

`casesplit(Conds)`

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: casesplit,v 1.10 1998/09/15 16:00:32 img Exp $
 *
 * $Log: casesplit,v $
 * Revision 1.10  1998/09/15 16:00:32  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.9  1998/08/26 12:54:21  img
 * update prefix; duplicate clause removed
 *
 * Revision 1.8  1998/07/30 15:59:00  img
 * Not necessary to adjust sink annotation
 *
 * Revision 1.7  1998/07/27 12:57:13  img
 * Do not touch sink markers
 *
 * Revision 1.6  1998/06/10 08:30:37  img
 * remove sink marking when universal variable is involved in split
 *
 * Revision 1.5  1997/10/10 09:17:26  img
 * uncomment datatype split
 *
 * Revision 1.4  1997/10/09 17:16:44  img
 * New clause added for splits on datatypes
 *
 * Revision 1.3  1996/12/11 14:07:11  img
 * Merge mthd and smthd libraries.
 *
 * Revision 1.2  1996/07/10  09:01:35  img
 * from submethod
 *
 * Revision 1.1  1995/05/11  16:21:25  img
 *  * from submethod
 */

/* We introduce a case-split in the proof when there are "applicable"
 * conditional rules (either reduction or wave), none of whose
 * conditions are known as true among the hypotheses.  This will then
 * enable the application of the conditional rules in the next step of
 * the proof.  We need to check that the free variables in each of the
 * Cases are in the context.  If they are not, we have to introduce
 * the corresponding quantifiers.  It may not be possible to do this
 * because (i) the quantification may not be universal, and/or (ii)
 * the quantifer may not be in the prefix.  To add to the confusion,
```

```
 * we need to check to see which binding operator is indeed binding
 * that occurrence of the variable in the Case(s).  I have code
 * written for this already (in tactics.pl, for rewriting beneath
 * binding operators) but I haven't used it here yet.  In the
 * meantime, the following approximation (in the post-conditions) will
 * suffice.  */

method(casesplit(disjunction(Cs)),
       H==>G,
       [matrix(Vars, Matrix, G),append(H,Vars,Context),
%% Removing annotations allows this method to be used in
%% base-case as well as step-case situations.
strip_meta_annotations(Matrix,Matrixstripped),
complementary_set(Cases-LHS),
exp_at(Matrixstripped,Pos,LHS),
map_list(Cases, Cond-RHS-Dir-TI-Name :=> Cond,
 \+ immediate(H==>Cond), Cs)],
       [%% Build the new goals according to each Case: i.e., stick
%% the case in the hypothesis.   V are the inhabitants: we
%% instantiate them at the end when we know how many there
%% are.
        %% Find maximal Vrest
freevarsinterm(Cs,CsFVs), append(Vpre,Vrest,Vars),
\+ (member(A,CsFVs), member(A:_,Vrest)),
delete_all(Vrest,Vars,Vlost),
append(Vpre, H, NewH), hfree([V],NewH),
%% need to strip the corresponding elements of the prefix of
%% any raw induction hypotheses
findall(VH,inductive_hypothesis(raw,VH,H),VHs),
matrix(Vrest, Matrix, NewG),
map_list(Cases,
 Cond-RHS-Dir-TI-Name :=> (FinalHV==>NewG),
 (instantiate_type_variables(TI,Context),
  append(NewH,[V:Cond],FinalHV)), NewGoals)],
       NewGoals,
       casesplit(disjunction(Cs))).

method(casesplit(datatype(Kind,[V:Type,CSD])),
       H==>G,
       [/* Do some analysis to suggest case splits.  */
/* Only works on unannotated goals in non-inductive branches  */
/* Currently fixed for lists and pnat */
unannotated(G),
\+ inductive_hypothesis(raw,_,H),
\+ inductive_hypothesis(notraw(_),_,H),

(Kind == unflawed -> unflawed_casesplit_suggestion(H,G,S);
    flawed_casesplit_suggestion(H,G,S)),
member((V:Type)-CSD,S),
\+ member(nosplit(V),H) ],%dont split repeatedly
       [matrix(Vars,Matrix,G),
/* if V is in the prefix, adjust the prefix of the resulting
           matrix. otherwise, leave it unchanged  */
once((select(V:Type,Vars,Vs); Vs = Vars)),

((CSD = s(VV),atomic(VV)) ->
 (VVdec = [VV:pnat],
  replace_all(V,0,Matrix,NM1),matrix(Vs,NM1,NG1),
  replace_all(V,s(VV),Matrix,NM2), matrix(Vs,NM2,NG2));
 ((CSD = Hd::Tl,atomic(Hd),atomic(Tl)) ->
  (Type = Ty list,
   VV = Tl,
   VVdec = [Hd:Ty, Tl:Type],
   replace_all(V,nil,Matrix,NM1),matrix(Vs,NM1,NG1),
   replace_all(V,Hd::Tl,Matrix,NM2), matrix(Vs,NM2,NG2)))),
append(H,[nosplit(V),nosplit(VV)|VVdec],VVdecH)],
       [ H==> NG1,
 VVdecH==>NG2],
    casesplit(datatype(Kind,[V:Type,CSD]))).
```

This method introduces a casesplit in a proof, based on the notion of complementary sets (cf. `complementary-set/1`). In the preconditions, we test if there is a conditional wave-rule that could be applicable, except that it comes from a complementary set of conditional rules, none of whose preconditions holds. This is then sign to introduce a casesplit in the proof, based on the preconditions from the complementary set. We have to take care to remove any universally quantified variables from the goal which are involved in the conditions of the casesplit.

Repeated application of casesplit is prevented by the test in the preconditions that none of the cases are already provable.

In order to justify a casesplit we need a *complementary set* of wave-rules, that is: a set of wave-rules whose conditions disjunctively amount to 'true'. (However, this condition is not checked at the proof-planning level.)

`unblock(Typ,Loc,Rewrite)`

The `unblock/2` method provides a conservative set of rewrites for unblocking rippling. There are a number of variants, all embodied in the following method:

```
/* -*- Prolog -*-
 * @(#)$Id: unblock,v 1.8 1998/09/15 16:00:42 img Exp $
```

```
 *
 * $Log: unblock,v $
 * Revision 1.8  1998/09/15 16:00:42  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.7  1998/06/10 09:32:27  img
 * extra context for type instantiation; new clause for conditional weak fert
 *
 * Revision 1.6  1996/12/11 14:07:17  img
 * Merge mthd and smthd libraries.
 *
 * Revision 1.9  1996/12/04 13:20:57  img
 * Deleted (highly) redundant calls to wave_terms_at/3: this unnecessary
 * since meta_ripple/3 and weaker/2 both account for subterm relation.
 *
 * Revision 1.8  1996/07/10  09:04:27  img
 * Need to ensure that unblocking in a wave-front is free of all
 * annotation, otherwise ill-formed terms may result.
 *
 * Revision 1.7  1996/06/19  08:32:44  img
 * Explicit unused flag (simplifies comparison of proof plans).
 *
 * Revision 1.6  1996/05/23  11:20:41  img
 * incorrect argument order in reduction_rule/6
 *
 * Revision 1.5  1996/05/14  15:59:35  img
 * Cleaner version, using reduction rules (no labelled rewriting).
 *
 * Revision 1.4  1995/10/03  10:30:49  img
 * position of subterm was incorrect
 *
 * Revision 1.3  1995/04/26  09:20:48  img
 *  * Weakening moved from wave into unblock
 *
 * Revision 1.2  1995/03/01  02:55:36  img
 *  * Unblocking for in dynamic rippling:  meta-rippling is
 *    generalized to arbitrary measure-decreasing manipulations
 *    excepting those of the object-logic;  skeleton-invariant
 *    rewrites inside sinks are legitimate under the new notion of
 *    skeleton.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

/* Meta-rippling refers to any measure-decreasing manipulation of the
 * goal which does not involve manipulation of the object-logic.  This
 * includes weakening and inverting outward wave-fronts to inwards
 * wave-fronts.  */
method(unblock(meta_ripple,unused,unused),
         H==>G,
         [matrix(Vars,Matrix,G),
   meta_ripple(direction_out,Matrix,NewMatrix) ],
         [matrix(Vars,NewMatrix,NewG) ],
         [H==>NewG],
         idtac ).

/* Weakening is not performed by ripple/6 so we do it here explicitly
   rather than in the wave method (see wave_rules.pl).  This is
   arguable; I chose to do it this way because I felt that dropping a
   skeleton was something we ought not to take lightly.  THIS IS
   DISABLED AT THE MOMENT BECAUSE unblock(meta_ripple) SUBSUMES IT.
   However, I expect that meta_ripple/3 will be replaced by
   fast_meta_ripple/2 which is much faster, but does not weaken.  In
   this eventuality, unblock(weaken) may be needed.  */
/* method(unblock(weaken,unused,unused),
  Hyps==>Conc,
         [matrix(Vars, Matrix, Conc),
     weaker(Matrix, NewMatrix)],
    [matrix(Vars, NewMatrix, NewConc)],
    [Hyps==>NewConc],
    idtac).  */

/* Apply a measure-reducing rule to a part of the wave-front.
   Rewriting in the wave-front ensures that the skeleton is not upset,
   and, thanks to the measure being insensitive to the terms
   contained in a wave-front, does not upset the rippling termination
   measure.   Notices that _certain_ ripple measures are not invariant
   under such wave-front manipulation!

   Rewriting in the skeleton (i.e., altering the skeleton) is possible
   if (i) the hypotheses are changed to reflect that, or (ii) the
   skeleton is in a sink position.  In case (i), there is some
   question as to whether this is a good idea: certainly in some
   proofs it is.  Note that there are two cases: (a) rewriting a subterm
   which is wholly contained in the skeleton (this is readily
   implemented), and (b), the subterm is in both the wave-front and
   the skeleton.  It is this second case that might not permit a
   corresponding rewrite in the hypotheses, and so it is not always
   possible to maintain skeleton preservation.

   unblock(sink,...) rewrites inside a sink;
   unblock(wave-front,...) rewrites inside a wave-front.
   unblock(skeleton,...) rewrites a term wholly contained within the
                         skeleton: this is currently unimplemented.
   All do so using reduction-rules, which are terminating.

   Note that a further complication concerning rewriting in the
   hypotheses arises with the use of rewrites bases on implications
   where polarity restrictions prevent the rule being applied.  */

/* Unblock in a sink.  This is quite simply a rewrite beneath a sink.
   Since the skeleton of a sink is arbitrary, there is no need to
   preserve skeleton in such a position.  */
method(unblock(sink,AbsSinkPos,[Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),append(H,Vars,Context),
```

```
        sinks(_,Sinks,Matrix),% fetch a sink
        member(SinkPos,Sinks),
        exp_at(Matrix,SinkPos,Sink),
issink(Sink,Contents),
        exp_at(Contents, ExpPos, Exp ),
\+ExpPos = [0|_],
        \+ var(Exp),% not in MOR
reduction_rule(Exp,NewExp,C,Dir,TI,Rule,_),
append(ExpPos,SinkPos,AbsExpPos),
        polarity_compatible(Matrix, AbsSinkPos, Dir),
instantiate_type_variables(TI,Context),
        immediate(Context==>C) ],
       [replace(ExpPos, NewExp, Contents, NewContents ),
issink(NewSink,NewContents),
        replace(SinkPos,NewSink,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG) ],
       [H==>NewG],
       reduction(AbsSinkPos,[Rule,Dir]) ).

/*  Unblock in a wave-front.  */
method(unblock(wave_front,ExpPos,[Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),append(H,Vars,Context),
        ann_exp_at(in_hole,in_front,Matrix,ExpPos,Exp),
%% and there is no skeleton inside here
\+ ExpPos = [0|_],
        \+ var(Exp),% not in MOR
%% Use of well_annotated/1 is tricky here: assuming that
%% Matrix is well-annotated, and Exp is inside a wave-front;
%% if Exp contains annotation, it must have a hole as the
%% uppermost annotation, in which case it is ill-annotated.
%% If it has a wave-front uppermost, Matrix would be
%% ill-annotated (contra the assumption).  So we see that if
%% Exp is well-annotated, it in fact does not contain any
%% annotations.  This is precisely what we want to ensure.
%% (We can't use unannotated/1, since that assumes
%% well-annotation.)
well_annotated(Exp),
reduction_rule(Exp,NewExp,C,Dir,TI,Rule,_),
        polarity_compatible(Matrix, ExpPos, Dir),
instantiate_type_variables(TI,Context),
        immediate(Context==>C) ],
       [replace(ExpPos,NewExp,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG) ],
       [H==>NewG],
       reduction(ExpPos,[Rule,Dir]) ).

/* very simple unblockling to simulate conditional weak fertilization.
   This method is applicable to sequents of the form:

   v: x:t=>y:u=>p(x)=>q(x,y)
   ==>
   x:t=>y:u=>p'(x)=>q'(x,y)        where q' is q with some annotation
                                         p' is an instance of p with
                                            some sink annotation

   the effects of the method are to strip away the quantifier prefix x
   and the antecedent p' so that weak fertilization may apply inside
   q'.  Since p' may contain sinks we need to take the appropriate
   instance of p (there are no other annotations in p').

   v: y:u=>q*(x,y)
   x: t
   w: p''(x)   We might want to drop this?
   y:u=>q'*(x,y)        where p'' is the erasure of p'
                        and * denotes the instantiating subst

   Note that the instance q* in the new hypothesis may not be
   suitable for fertilization with the skeleton of q'* in the goal.
*/

% method(unblock(cwf,unused,unused),
%        H==>G,
%        [
%  matrix(Vars,Pp=>Qp,G),append(H,Vars,Context),
%  annotations(_,[],PpErasure,Pp),
%  inductive_hypothesis(Status,Hyp:IndHyp,H),
%  (Status = raw; Status = notused(_)),
%         matrix(IndHypVars,P=>Q,IndHyp),% find hyp to use for fertilization
%
%  if(\+IndHypVars==Vars,
%     clam_warning('strange.... skel vars are different.')),
%
%  freevarsinterm(P,PFV),% fetch x
%  untype(PFVVT,PFV,PFVT),
%  subset(PFVVT,Context),% find types
%  delete_all(PFVVT,Vars,NewVars),
%  delete_all(NewVars,Vars,Decls),
%  matrix(PFV,P,Pf),
%  instantiate(H,PFV,Pf,PpErasure,Pvals),
%  /* Qp may contain sinks due to variables in Decls which are no
%     longer valid.  These need to be removed.  */
%  updated_sinks(Decls,Qp,Q,QpQp),
%  s(Q,Pvals,PFV,NewQ),
%  s(QpQp,Pvals,PFV,NewQp),
%  matrix(NewVars,NewQp,NewG),
%  matrix(NewVars,NewQ,NewHyp),
%
%  inductive_hypothesis([Hyp:IndHyp],[Hyp:NewHyp],H,RevisedH),
%  hfree(PV,Context),%slight overkill here
%  append(Decls,[PV:P],ExtraH),
%  append(RevisedH,ExtraH,NewH)],
%        [ ],
%        [NewH==>NewG],
%        unblock(cwf,unused,unused) ).
%
%
```

```
%
%
%
%
%
%
%
```

`unblock(weaken, Pos, [])` Weakens some wave-front in the annotated term at position `Pos`. *Weakening* is the removal of one (or more) wave-holes from a wave-front. This is subject to the condition that at least one wave-hole remains in the weakened term. This method is not active by default because weakening is covered by `unblock(meta-ripple,_,_)`.

`unblock(meta-ripple, Pos, [])` *Meta-rippling* is the rewriting of an annotated term t with the (object-level) rewrite rule $t \Rightarrow t'$, such that (i) t and t' differ only in their annotation (i.e., they are the same when annotations are erased), and, (ii) they have the same skeleton, and finally, (iii) t' is smaller in the measure than t.

When meta-rippling is needed. In a proof that $half(x) \leq x$, $s(s(x'))/x$ induction gives a step-case residue of $x' \leq s(x')$. $s(x'')$ induction on x' is required but rippling analysis finds that both occurrences of $\boxed{s(\underline{x''})}$ in

$$\boxed{s(\underline{x''})} \leq s(\boxed{s(\underline{x''})})$$

are flawed.[4] The reason is that the wave-rule for $\leq$:

$$\boxed{s(\underline{X})} \leq \boxed{s(\underline{X})} \Rightarrow X \leq \boxed{s(\underline{X})}$$

doesn't match. It is necessary to meta-ripple the above term so that the wave-front of the double successor is moved up the term. Then the $\leq$ wave-rule matches.

The meta-rippling rule is created dynamically as needed. For the standard wave-rule measure, t' can be 'smaller' than t if at least one of the following hold:

1. t' is weakening of t. Weakening is very expensive when there are multiple numbers of multi-hole wave-fronts because there is a combinatorial problem. Coloured rippling flattens the combinatorial problem since then all terms are effectively single-holed (see [33] for more on coloured rippling).
2. Wave-fronts in t' are moved up the term tree (or down for inward fronts) in comparison with the corresponding fronts in t. This is less expensive to carry out but we have to ensure skeleton preservation. This can be expensive unless some partial evaluation goes on. (For example, some of the skeleton remains unchanged.) Often, skeleton preservation is trivial since the wave-fronts are identical, e.g., $s(\boxed{s(\underline{x})}) \Rightarrow \boxed{s(\underline{s(x)})}$.
3. Outward wave-fronts become inward wave-fronts. (For the purposes of this version of Clam this type of meta-rippling is not needed since it is part of rippling.)

[4] I think there is little sense here in saying that the first occurrence is unflawed, although one could imagine pursuing that distinction.

See also `unblock-lazy/1`.

unblock-lazy(Plan)

This iterating method applies the `unblock/3` method lazily.

```
/* -*- Prolog -*-
 * @(#)$Id: unblock_lazy,v 1.2 1998/06/10 09:33:44 img Exp $
 *
 * $Log: unblock_lazy,v $
 * Revision 1.2  1998/06/10 09:33:44  img
 * specifiy types of unblocking
 *
 * Revision 1.1  1996/12/11 15:09:30  img
 * Merge of mthd and smthd libraries.
 *
 */

iterator_lazy(method,unblock_lazy,submethods,
[unblock(meta_ripple,_,_),
 unblock(weaken,_,_),
 unblock(sink,_,_),
 unblock(wave_front,_,_)]).
```

It succeeds first with zero applications of `unblock/3`, then with 1 application on backtracking, then 2 and so on.

ripple(Dir,SubPlan)

The `ripple/2` builds possibly branching proofs using `wave/4`, `casesplit/1` and `unblock-then-wave/2`. The `Dir` parameter is passed to `wave/4` to control the type of rippling.

```
/* -*- Prolog -*-
 * @(#)$Id: ripple,v 1.8 1997/10/16 10:32:53 img Exp $
 *
 * $Log: ripple,v $
 * Revision 1.8  1997/10/16 10:32:53  img
 * restrict casesplitting
 *
 * Revision 1.7  1997/10/09 17:19:56  img
 * removed duplicated header
 *
 * Revision 1.6  1996/12/11 14:07:13  img
 * Merge mthd and smthd libraries.
 *
 * Revision 1.5  1996/11/02 14:00:41  img
 * Drop redundant argument in ripple/3.
 *
 * Revision 1.4  1996/07/10  08:53:15  img
 * Slight error in the implemementation of fertilization: it is
 * conditional on rippling-in, not on the type of fertilization expected.
 *
 * Revision 1.3  1995/05/11  16:19:30  img
 *  * upgraded from submethod
 *
 * Revision 1.2  1995/03/01  02:45:50  img
 *  * updates to cope with dynamic rippling
 *
 * Revision 1.1  1994/09/16  09:33:29  dream
 * Initial revision
 */
method(ripple(Dir, SubPlan),
       HG,
       [repeat([HG],
       Goal :=> SubGoals,
       Method,
       (member(Method, [wave(Dir,_,_,_),
casesplit(disjunction(_)),
unblock_then_wave(Dir,_)]),
applicable_submethod(Goal, Method, _, SubGoals)),
       [SubPlan],
       SubGoals),!,
SubPlan \= idtac],
       [strip_redundant_sinks(SubGoals,SubGoalsRS)],
       SubGoalsRS,
       SubPlan).
```

cancellation([],Rule)

```
/*
 * @(#)$Id: cancellation,v 1.2 1998/09/15 16:00:31 img Exp $
 *
 * $Log: cancellation,v $
 * Revision 1.2  1998/09/15 16:00:31  img
 * use immediate/[1,2]; indentation changes
 *
```

```
 * Revision 1.1  1996/12/11 15:09:16  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.1  1994/09/16 09:34:27  dream
 * Initial revision
 */

method(cancellation([],Rule),
       H==>G,
       [matrix(Vars,Matrix,G),
wave_fronts(Exp,WSpec,Matrix),
cancel_rule(Exp,Rule:PC=>Exp:=>NewExp),
object_level_term(Exp),
immediate(H==>PC)],
       [modify_wave_ann(WSpec,WWSpec),
wave_fronts(NewExp,WWSpec,NewMatrix),
matrix(Vars,NewMatrix,NewG) ],
       [H==>NewG],
       cancellation([],Rule)).
```

This method controls the cancellation of outer term structure during post-fertilization rippling.

fertilize(Type,Ms)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: fertilize,v 1.4 1999/01/07 16:30:24 img Exp $
 *
 * $Log: fertilize,v $
 * Revision 1.4  1999/01/07 16:30:24  img
 * Support for ripple-and-cancel reintroduced (currently disabled,
 * however).  weak_fertilize uses larger_size/2 to try to replace larger
 * with smaller terms during weak fertilization
 *
 * Revision 1.3  1998/11/10 16:10:03  img
 * Reorganised methods and integrated piecewise fertilization.
 *
 * Revision 1.2  1998/09/15 16:00:36  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 */

/* Strong and weak are pretty much the same but strong does not have a
   ripple then cancel phase. */
method(fertilize(strong,Fertilize),
       H==>G,
       [Fertilize = pwf_then_fertilize(strong,_),
applicable_submethod(H==>G,Fertilize,_,Seqs)],
       [map_list(Seqs,(Hy==>NS):=> (Hy==>NSNA),
 strip_meta_annotations(NS,NSNA),NewSeqsNA)],
       NewSeqsNA,
       fertilize(strong,Fertilize)).

method(fertilize(weak,FertPlan),
       H==>G,
        [Fertilize = pwf_then_fertilize(weak,_),
         applicable_submethod(H==>G,Fertilize,_,Seqs),
 map_list(Seqs, Seq :=> (Plan-NewSeq),
          (PostFertRipple = ripple_and_cancel(_),
   ((fail,applicable_submethod(Seq,PostFertRipple,_,[NewSeq]),
     Plan = PostFertRipple)
           orelse
    (NewSeq = Seq, Plan = idtac))),PlanNewSeqs),
         zip(PlanNewSeqs,Plans,NewSeqs),
         FertPlan = (Fertilize then Plans)],
        [map_list(NewSeqs,(Hy==>NS):=> (Hy==>NSNA),
          strip_meta_annotations(NS,NSNA),NewSeqsNA)],
    NewSeqsNA,
       fertilize(weak,Ms)).
```

This method controls the use of induction hypotheses once rippling has terminated.

fertilization-strong(Hyp)

```
/* -*- Prolog -*-
 * @@(#)$Id: fertilization_strong,v 1.7 1999/02/04 16:40:19 img Exp $
 *
 * $Log: fertilization_strong,v $
 * Revision 1.7  1999/02/04 16:40:19  img
 * adjustment for tactic
 *
 * Revision 1.6  1999/01/26 10:37:11  img
 * corrected comment
 *
 * Revision 1.5  1999/01/11 12:26:27  img
 * Strong fertilization is terminating method
 *
 * Revision 1.4  1998/11/10 16:10:02  img
 * Reorganised methods and integrated piecewise fertilization.
 *
```

```
 * Revision 1.3  1998/09/15 16:00:34  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.2  1998/06/10 08:37:21  img
 * Multiple s.f.'s may be possible, so do them all.  This means that
 * s.f. is no longer closing (see step_case which must now account for
 * this possibility).
 *
 * Revision 1.1  1996/12/11 15:09:21  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.2  1996/12/04  13:14:51  img
 * Use inductive_hypothesis.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

/* Look for an instance of an inductive hypothesis in the goal.  For
   such a term to exist it must either be the case that the goal does
   not contain wave-fronts, or, the top of the the matrix is a
   wave-front and this front has one or more holes, each of which is
   an instance of some hypothesis.  */

method(fertilization_strong(Hyp),
       H==>G,
       [matrix(_,Matrix,G),
/* simple instance */
        inductive_hypothesis(Status,Hyp:Hypothesis,H),
        instantiate(H,[],Hypothesis,Matrix,_) ],
       [],
       [],
       fertilization_strong(Hyp)).

method(fertilization_strong(HypsNs),
       H==>G,
       [matrix(Vs,Matrix,G),
findall(AHyp-[N],
(ann_exp_at(Matrix,[N],Hole),
 inductive_hypothesis(Status,AHyp:Hypothesis,H),
 instantiate(H,[],Hypothesis,Hole,_)),HypsNs),
        \+ HypsNs = []],
       [zip(HypsNs,Hyps,Ns),
raw_to_used(H,Hyps,NewH),
erase(Matrix,UMatrix),
/* each of the positions N is S.F. */
replace_foldr(Ns,{true},UMatrix,NewMatrix),
matrix(Vs,NewMatrix,NewG)],
       [NewH==>NewG],%step case will remove ann. from H
       fertilization_strong(HypsNs)).
```

This terminating method will trigger when the current goal has been rewritten to match with one of the hypotheses (typically an induction hypothesis), possibly after instantiating some universally quantified variables.

fertilization-weak(Plan)

When strong fertilization does not apply, we attempt weak fertilization. This consists of using the induction hypothesis to rewrite all the wave terms inside a wave-front when that wave-front is the only one present, and it has bubbled all the way up to the top of one side of the formula:

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: fertilization_weak,v 1.3 1998/11/10 16:10:02 img Exp $
 *
 * $Log: fertilization_weak,v $
 * Revision 1.3  1998/11/10 16:10:02  img
 * Reorganised methods and integrated piecewise fertilization.
 *
 * Revision 1.2  1998/09/15 16:00:35  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1996/12/11 15:09:23  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.1  1994/09/16 09:34:27  dream
 * Initial revision
 */

method(fertilization_weak(Type,FertPlan),
       H==>G,
         [Fertilize = pwf_then_fertilize(Type,_),
          applicable_submethod(H==>G,Fertilize,_,Seqs),
 map_list(Seqs, Seq :=> (Plan-NewSeq),
           (PostFertRipple = ripple_and_cancel(_),
   ((applicable_submethod(Seq,PostFertRipple,_,[NewSeq]),
     Plan = PostFertRipple)
           orelse
    (NewSeq = Seq, Plan = idtac))),PlanNewSeqs),
         zip(PlanNewSeqs,Plans,NewSeqs),
         FertPlan = (Fertilize then Plans)],
        [map_list(NewSeqs,(Hy==>NS):=> (Hy==>NSNA),
          strip_meta_annotations(NS,NSNA),NewSeqsNA)],
       NewSeqsNA,
```

```
      fertilization_weak(Type,Ms)).
```

fertilize-then-ripple(Plan)

Once the fertilization process is complete post-fertilization rippling is attempted. Post-fertilization rippling has been shown to be a useful lemma conjecturing mechanism. (For further details refer to [6].)

This is not yet fully implemented in this version of Clam.

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: fertilize_then_ripple,v 1.3 1998/11/10 16:10:03 img Exp $
 *
 * $Log: fertilize_then_ripple,v $
 * Revision 1.3  1998/11/10 16:10:03  img
 * Reorganised methods and integrated piecewise fertilization.
 *
 * Revision 1.2  1998/09/15 16:00:37  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1996/12/11 15:09:27  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.1  1994/09/16 09:34:27  dream
 * Initial revision
 */

        % See the weak_fertilize/4 submethod for details on fertilization.
%
method(fertilize_then_ripple(FertPlan),
H==>G,
        [Fertilize = pwf_then_fertilize(Type,_),
         applicable_submethod(H==>G,Fertilize,_,Seqs),
 map_list(Seqs, Seq :=> (Plan-NewSeq),
          (PostFertRipple = ripple_and_cancel(_),
   ((applicable_submethod(Seq,PostFertRipple,_,[NewSeq]),
     Plan = PostFertRipple)
            orelse
    (NewSeq = Seq, Plan = idtac))),PlanNewSeqs),
         zip(PlanNewSeqs,Plans,NewSeqs),
         FertPlan = (Fertilize then Plans)],

        [map_list(NewSeqs,(Hy==>NS):=> (Hy==>NSNA),
          strip_meta_annotations(NS,NSNA),NewSeqsNA)],
        NewSeqsNA,
        fertilize_then_ripple(FertPlan)).
```

ripple-and-cancel(Plan)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: ripple_and_cancel,v 1.3 1999/01/07 16:30:24 img Exp $
 *
 * $Log: ripple_and_cancel,v $
 * Revision 1.3  1999/01/07 16:30:24  img
 * Support for ripple-and-cancel reintroduced (currently disabled,
 * however).  weak_fertilize uses larger_size/2 to try to replace larger
 * with smaller terms during weak fertilization
 *
 * Revision 1.2  1998/11/10 16:10:05  img
 * Reorganised methods and integrated piecewise fertilization.
 *
 * Revision 1.1  1996/12/11 15:09:29  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.1  1994/09/16 09:34:27  dream
 * Initial revision
 */

iterator(method,ripple_and_cancel,submethods,
 [cancellation(_,_),
  wave(ripple_and_cancel,_,_,_)]).
```

fertilize-left-or-right(Dir,Ms)

Since there can be more than one wave-term (or: wave-hole) in the top wave-front, we construct a method that rewrites just one of the wave terms, and a method that iterates this method in order to rewrite all of the wave terms. Since the rewriting can be done either left to right or right to left, the iterating method distinguishes these two cases in a disjunction, and looks as follows:

```
/* -*- Prolog -*-
 * @(#)$Id: fertilize_left_or_right,v 1.3 1998/11/10 16:10:03 img Exp $
 *
 * $Log: fertilize_left_or_right,v $
 * Revision 1.3  1998/11/10 16:10:03  img
 * Reorganised methods and integrated piecewise fertilization.
 *
 * Revision 1.2  1998/09/15 16:00:36  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1996/12/11 15:09:25  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.2  1996/12/04 13:16:15  img
 * Use notraw_to_used to tidy up after fertilization.
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

iterator(method,fertilize_left_or_right,submethods,
 [weak_fertilize(_,_,_,_)]).
```

The methods `weak-fertilize-right/1` and `weak-fertilize-left/1` iterate the submethod that does the actual rewriting on single wave fronts.

`weak-fertilize(Dir,Conn,Pos,Hyp)`

This method, called iteratively from `weak-fertilize/2` via `weak-fertilize-left/1` or `weak-fertilize-right/1`,[5] performs the actual rewrite on one of the wave terms (wave variables) inside a wave-front that has bubbled all the way up to the top of one side of the formula. It gets called iteratively to do the operation on all wave terms in the wave-front. A special case is when the wave-front hasn't been rippled to the top of one side of the formula, but has been rippled right out of one side. In that case we can do fertilization on the whole side of that formula (the case where S below is empty). Although fertilization was originally invented only for equalities, and later patched for implications, [32] generalised the operation to arbitrary transitive functions: fertilization performs the following transformations on sequents, where $\sim$ is a transitive predicate:

$$
\begin{aligned}
&L \sim R \vdash X \sim S(R) \textit{ into } L \sim R \vdash X \sim S(L) \textit{ if } R \textit{ occurs positive in } S, \textit{ or}\\
&L \sim R \vdash S(L) \sim X \textit{ into } L \sim R \vdash S(R) \sim X \textit{ if } L \textit{ occurs positive in } S, \textit{ or}\\
&L \sim R \vdash X \sim S(L) \textit{ into } L \sim R \vdash X \sim S(R) \textit{ if } L \textit{ occurs negative in } S, \textit{ or}\\
&L \sim R \vdash S(R) \sim X \textit{ into } L \sim R \vdash S(L) \sim X \textit{ if } R \textit{ occurs negative in } S.
\end{aligned}
$$

where the wave-fronts must be $\boxed{S(\underline{R})}$ or $\boxed{S(\underline{L})}$ where appropriate.

A term "occurs positively" in a function if that function is monotonic in that argument. A function f is monotonic monotonic in argument x if:

$$x_1 \preceq_D x_2 \rightarrow f(x_1) \preceq_{CD} f(x_2)$$

where $\preceq_D$ and $\preceq_{CD}$ are partial orderings on the domain and codomain of f respectively. A term occurs x positively in a set of nested functions $f_1(\ldots(f_n(x))\ldots)$ if either x occurs positively in f_n and or x occurs negatively in f_n and $f_n(x)$ occurs negatively in $f_1(\ldots(f_{n-1}(\_))\ldots)$. If $\sim$ is also symmetrical, we can drop the requirements on polarity (as is the case with equalities, for instance).

After all this, the code for the `weak-fertilize/4` method that does these operations is as follows:

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: weak_fertilize,v 1.11 2005/04/30 14:30:40 smaill Exp $
 */

        % Weak fertilization consists of using the induction hypothesis
        % to rewrite all the wave terms inside a wave front once that
        % wave front is the only one present, and it has bubbled all the
        % way up to the top of one side of the formula. This method
        % applies a fertilization to just one wave term of the top wave
```

[5] To enforce a uniform direction.

```
        % front. It gets called iteratively to do the operation on all
        % wave terms in the wave front.  A special case is when the wave
        % front hasnt been rippled to the top of one side of the
        % formula, but has been rippled right out of one side. In that
        % case we can do fertilization on the whole side of that formula
        % (the case where S below is empty).
        %
        % For a function symbol ~ for which transitivity holds, we can
        % perform the following transformations on sequents:
        %
        % 1. L~R |- X~S(R) into L~R |- X~S(L) if R occurs positive in S, or
        % 2. L~R |- S(L)~X into L~R |- S(R)~X if L occurs positive in S, or
        % 3. L~R |- X~S(L) into L~R |- X~S(R) if L occurs negative in S, or
        % 4. L~R |- S(R)~X into L~R |- S(L)~X if R occurs negative in S.
        %
        % where the wave fronts must be: ``S({R})'' etc.
        % These can be reprased as:
        %
        % 1. substitute R by L in rhs when R occurs positive, or
        % 2. substitute L by R in lhs when L occurs positive, or
        % 3. substitute L by R in rhs when L occurs negative, or
        % 4. substitute R by L in lhs when R occurs negative
        %
        % If ~ is also symmetrical, we can drop the requirements on polarity.
        % The method below implements 1-2 in one method. It knows about
        % a set of function symbols which are transitive. It then always
        % does a fertilization replacing R by L, but it can assign L and
        % R to either lhs and rhs or vice versa, so we get the symmetry
        % we want.
        %
        % REMARKS:
        % 1. The set of transitive function symbols this method knows
        % about might of course have to be extended in the future
        % (similary, it knows about one symmetrical function symbol
        % (equality)).
        %
        % 2. The case where S below is empty can possibly loop (for
        %    instance y:t=>f(g(y))=>f(y) |- z:t=>f(``g({2}''))=>f(z)
        %    gives    y:t=>f(g(y))=>f(y) |- z:t=>f(``g({2}''))=>f(g(z))
        %    (according to 1. above), which in turn gives:
        %             y:t=>f(g(y))=>f(y) |- z:t=>f(``g({2}''))=>f(g(g(z)))
        %    (again according to 1.) above, etc.
        %    The problem with this is of course that there is no
        %    "reducing measure" to stop the iteration of fertilization
        %    (as there is in the wave front case, namely the reducing
        %     number of wave fronts).
        %    When there is no wave front (S empty), then we want to do
        %    only one fertilization step, and not iterate. This is
        %    achieved by always deleting the wave fronts in L after one
        %    fertilization. This will then stop the next iteration
        %    (since L having wave fronts is a condition for
        %    fertilization when R has none). The immediate deletion of
        %    wave front from L will not affect the iteration when R has
        %    wave fronts, since the presence of wave fronts in L is then
        %    immaterial.
        %
        % 3. The below doesn't implement cases 3.-4. above (negative
        %    fertilization), but it can't be very difficult. It would
        %    look very much like the below. I wonder if it can
        %    reasonably be done with one piece of code (once method).
        %
        % 4. Notice that "in" is used instead of "=" to represent (_=_ in _),
        %    since it makes the code below more uniform (in particular,
        %    it allows the exp_at(_,[0],_) expression).
        %
        % 5. See the file schemes.pl and Blue Book note 539 for a sermon
        %    about "theory free" theorem provers, and how the weak
        %    fertilization method is one of the few places where CLaM
        %    violates this requirement.
        %
        % 6. The main idea of and motivation for weak fertilization is
        %    also described in Blue Book note 538.

method(weak_fertilize(Dir,Connective,Pos,Hyp),
       H==>G,
       [matrix(Vars,InitM,G),
        maximally_joined(InitM,M),
transitive_pred( M, [LR,RL], [LRN,RLN], NewG_M ),
        exp_at(M,[0],Connective),
/* Do fertilization right-to-left or left-to-right; in some
      situations w.f. can be in either direction for a given
      hypothsis, and in this situation we prefer the direction
      which removes worse terms in favour of better ones.  Here,
      larger is worse and smaller is better.  There will be
      better metrics.  */
         ((Dir=right,GL=LR,GR=RL, GLNew=LRN, GRNew = RLN) v
  (Dir=left,GL=RL,GR=LR, GLNew=RLN, GRNew = LRN )),
        (
          (wave_fronts(GR1,[[]-PosL/[Typ,out]],GR), % We must have 1 wave-front in GR,
           select(Pos,PosL,OtherPosL),              % which is on the top ([]-...) and
           NewWFspec = [[]-PosL/[Typ,in]]     % out-bound. Note change in wave
          )      % direction.
        v
          (wave_fronts(GR1,[[]-PosL/[Typ,in]],GR),  % or if we have 1 wave-front in GR
   PosL = [_,_|_],    % which is on top and in-bound
           select(Pos,PosL,OtherPosL),    % then we must have mutiple holes.
           NewWFspec = [[]-PosL/[Typ,in]])
         v
  (wave_fronts(GR1,[],GR),                 % or we have no wave front in GR,
           wave_fronts(_,[_-_/[_,out]|_],GL),      % but we require out-bound
           PosL=[],Pos=[],OtherPosL=[[]],          % wave-fronts in GL,
           NewWFspec = [Pos-OtherPosL/_])
         v
          (wave_fronts(GR1tmp,[WFPos-[WHPos]/[Typ,_]],GR), % or all wave fronts
           sinks(GR1,[WFPos],GR1tmp),                      % are sunk in GR.
           sinks(GL1,_,GL),
   append(WHPos,WFPos,RSinkPos),
```

```
   exp_at(GR1,RSinkPos,Sink),
   exp_at(GL1,LSinkPos,Sink),
   NewWFspec = [LSinkPos-[WHPos]/[Typ,out]],
   Pos=[])
         v
%% This does not correctly place ingoing
%% wave front on output
%% (when there are 2 fertilisation rewrites to do, anyway).
        (wave_fronts(GR1tmp,[[]-PosL/[Typ,out]|SunkFronts],GR), % We must have 1 wave-front in GR,
           select(Pos,PosL,OtherPosL),                       % which is on the top ([]-...) and
           NewWFspec = [[]-PosL/[Typ,in]],             % out-bound; with all other waves fronts sunk.
           sinks(GR1,SinkPosns,GR1tmp),
           (forall{PP \ SunkFronts}: (PP = FrontPos-_/[_,_], %  all fronts in Sunkfronts are in sinks
                    (thereis{ SS \ SinkPosns} : append(SS,_,FrontPos)))))),
        exp_at(GR1,Pos,GR1Sub),
                                      % check for positive occurrence
                                      % or symmetrical function symbol:
        (Connective = (in) orelse polarity(_,_,GR1,Pos,+)),
inductive_hypothesis(Status,Hyp:IndHyp,H),
((Status = raw,EarlierFs=[]);
 Status = notraw(EarlierFs)),% don't allow 'used'
        matrix(IndHypVars,IndHyp_M,IndHyp),% find hyp to use for fertilization
        replace_all(GL,GRNewSub,M,GSub1),
        replace_all(GR,GR1Sub,GSub1,GSub),
untype(IndHypVars,IndHypVarsNoTypes),
        instantiate(H,IndHypVarsNoTypes,IndHyp,GSub,Instan),
% nl,        write( GR1Sub :=> GRNewSub),nl,
larger_size(GR1Sub,GRNewSub),
!,
/* Rule is ground now, so we can check that it is being used appropriately */

        wave_fronts(GRNewSub,_,GRNewSub), % and check it doesnt introduce
                                      % more wave-fronts.
/* following replaces GR1_Pos with GRNewSub.  (that is, the ground rule
   GR1Sub :=> GRNewSub at position Pos */
replace(Pos,GRNewSub,GR1,GRNew1),% apply the fertilization
        wave_fronts(GRNew1,NewWFspec,GRNew),
        wave_fronts(GLNew,_,GL),        % squash the wave-fronts in L
% (see remark 2. above).
        sinks(NNewG_M,_,NewG_M),        % squash all sinks

        matrix(Vars,NNewG_M,NewG),
inductive_hypothesis(Status,Hyp:IndHyp,H,notraw([Dir|EarlierFs]),NewH),
ann_normal_form(NewG,NewGNF),
unannotated(NNewG_M,Testee),
append(NewH,Vars,Context),
if(\+ member(Connective,[in,<=>]),
   \+ trivially_falsifiable(Context,Testee))],
       [],
       [NewH==>NewGNF],
       weak_fertilize(Dir,Connective,Pos,Hyp)).



method(weak_fertilize(Dir,Connective,Pos,Hyp),
       H==>G,
       [matrix(Vars,InitM,G),
        maximally_joined(InitM,M),
transitive_pred( M, [LR,RL], [LRN,RLN], NewG_M ),
        exp_at(M,[0],Connective),
/* Do fertilization right-to-left or left-to-right; in some
      situations w.f. can be in either direction for a given
      hypothsis, and in this situation we prefer the direction
      which removes worse terms in favour of better ones.  Here,
      larger is worse and smaller is better.  There will be
      better metrics.  */
         ((Dir=right,GL=LR,GR=RL, GLNew=LRN, GRNew = RLN) v
  (Dir=left,GL=RL,GR=LR, GLNew=RLN, GRNew = LRN )),
        (
          (wave_fronts(GR1,[[]-PosL/[Typ,out]],GR), % We must have 1 wave-front in GR,
           select(Pos,PosL,OtherPosL),              % which is on the top ([]-...) and
           NewWFspec = [[]-PosL/[Typ,in]]     % out-bound. Note change in wave
          )      % direction.
        v
          (wave_fronts(GR1,[[]-PosL/[Typ,in]],GR),  % or if we have 1 wave-front in GR
   PosL = [_,_|_],    % which is on top and in-bound
           select(Pos,PosL,OtherPosL),    % then we must have mutiple holes.
           NewWFspec = [[]-PosL/[Typ,in]])
         v
  (wave_fronts(GR1,[],GR),                 % or we have no wave front in GR,
           wave_fronts(_,[_-_/[_,out]|_],GL),      % but we require out-bound
           PosL=[],Pos=[],OtherPosL=[[]],          % wave-fronts in GL,
           NewWFspec = [Pos-OtherPosL/_])
         v
          (wave_fronts(GR1tmp,[WFPos-[WHPos]/[Typ,_]],GR), % or all wave fronts
           sinks(GR1,[WFPos],GR1tmp),                      % are sunk in GR.
           sinks(GL1,_,GL),
   append(WHPos,WFPos,RSinkPos),
   exp_at(GR1,RSinkPos,Sink),
   exp_at(GL1,LSinkPos,Sink),
   NewWFspec = [LSinkPos-[WHPos]/[Typ,out]],
   Pos=[])
         v
%% This does not correctly place ingoing
%% wave front on output
%% (when there are 2 fertilisation rewrites to do, anyway).
        (wave_fronts(GR1tmp,[[]-PosL/[Typ,out]|SunkFronts],GR), % We must have 1 wave-front in GR,
           select(Pos,PosL,OtherPosL),                       % which is on the top ([]-...) and
           NewWFspec = [[]-PosL/[Typ,in]],             % out-bound; with all other waves fronts sunk.
           sinks(GR1,SinkPosns,GR1tmp),
           (forall{PP \ SunkFronts}: (PP = FrontPos-_/[_,_], %  all fronts in Sunkfronts are in sinks
                    (thereis{ SS \ SinkPosns} : append(SS,_,FrontPos)))))),
        exp_at(GR1,Pos,GR1Sub),
                                      % check for positive occurrence
                                      % or symmetrical function symbol:
        (Connective = (in) orelse polarity(_,_,GR1,Pos,+)),
inductive_hypothesis(Status,Hyp:IndHyp,H),
```

```
((Status = raw,EarlierFs=[]);
 Status = notraw(EarlierFs)),% don't allow 'used'
        matrix(IndHypVars,IndHyp_M,IndHyp),% find hyp to use for fertilization
        replace_all(GL,GRNewSub,M,GSub1),
        replace_all(GR,GR1Sub,GSub1,GSub),
untype(IndHypVars,IndHypVarsNoTypes),
        instantiate(H,IndHypVarsNoTypes,IndHyp,GSub,Instan),
% nl,        write( GR1Sub :=> GRNewSub),nl,
/* Rule is ground now, so we can check that it is being used appropriately */

        wave_fronts(GRNewSub,_,GRNewSub), % and check it doesnt introduce
                                         % more wave-fronts.
/* following replaces GR1_Pos with GRNewSub.  (that is, the ground rule
   GR1Sub :=> GRNewSub at position Pos */
replace(Pos,GRNewSub,GR1,GRNew1),% apply the fertilization
        wave_fronts(GRNew1,NewWFspec,GRNew),
        wave_fronts(GLNew,_,GL),        % squash the wave-fronts in L
% (see remark 2. above).
        sinks(NNewG_M,_,NewG_M),        % squash all sinks

        matrix(Vars,NNewG_M,NewG),
inductive_hypothesis(Status,Hyp:IndHyp,H,notraw([Dir|EarlierFs]),NewH),
ann_normal_form(NewG,NewGNF),
unannotated(NNewG_M,Testee),
append(NewH,Vars,Context),
if(\+ member(Connective,[in,<=>]),
   \+ trivially_falsifiable(Context,Testee))],
       [],
       [NewH==>NewGNF],
       weak_fertilize(Dir,Connective,Pos,Hyp)).
```

This method only implements positive weak fertilization (when L occurs positive in S). The negative weak fertilization is not implemented, but cannot be very difficult to do, either with an additional clause for this method, or by extending the current method.

Notice that `in` is used instead of `=` to represent `_=_ in _`, since it makes the code below more uniform (in particular, it allows the `exp-at(_,[0],_)` expression).

For this method to work, the system of course needs to know which functions are transitive, and what the monotonicity properties of functions are. Currently, these properties are hardwired into Clam for certain function symbols: the transitive functions are explicitly mentioned in the method (second predicate in the preconditions) and the polarity is explicitly encoded in the `polarity/5` predicate. Both of these mechanisms violate the *theory free* requirement formulated in §6.4 on page 127. See that section for a sermon on this topic, and also for suggestions on how to remove these violations of the theory free requirement from Clam.

`step-case(Plan)`

The `step-case/1` method constructs a plan for step case proof obligations. Here is an outline of that plan:

(1) Ripple out as much as possible, giving subgoals $S_1, \ldots, S_N$, and plans $P_1, \ldots, P_N$.

(2) Try fertilize and base-case on $S_1, \ldots, S_N$. If this is not possible stop.

(3) For each S_i for which fertilization and base-case are inapplicable:

(3.1) Try to ripple the S_i using rippling in, taking care not to ripple beyond the point of a fertilization, giving $S_{i_1}, \ldots, S_{i_{M_i}}$ and $P_{i_1}, \ldots, P_{i_{M_i}}$.

(3.2) Try fertilize/base-case on each of the $S_{i_{M_i}}$.

The subgoals of this plan are the subgoals remaining from each phase. `Plan` is the composition of all the plans for each of these subgoals.

```
/*
 * @(#)$Id: step_case,v 1.19 2005/05/09 18:13:26 smaill Exp $
 */
method(step_case(Plan),
       H==>G,
       [annotated(G),
RippleOutPlan = ripple(direction_out,_),
        applicable_submethod(H==>G,RippleOutPlan,_,RippleOutSeqsPrePWF),
```

```
map_list(RippleOutSeqsPrePWF, PrePWF :=> PostPWF-PostPWFPlan,
 ((PostPWFPlan=pw_fertilize(_),
     applicable_submethod(PrePWF,PostPWFPlan,_,PostPWF)); (PostPWFPlan=idtac, PostPWF=PrePWF)),
 RippleOutSeqsAndPlansPostPWF),
zip(RippleOutSeqsAndPlansPostPWF,RippleOutSeqsPostPWF,RippleOutPlansPostPWF),
write('RippleOutPlansPostPWF are '), write(RippleOutPlansPostPWF),nl,
flatten(RippleOutSeqsPostPWF,RippleOutSeqs),
write('RippleOutSeqs are '), write(RippleOutSeqs),nl,
write('RippleOutPlan is '), write(RippleOutPlan),nl,
map_list(RippleOutSeqs, RippleOutSeq :=>
     PostRippleOutSeqs-PostRippleOutPlan,
 orelse((/* STRONG FERTILIZATION */
 PostRippleOutPlan=unblock_then_fertilize(strong,_),
 applicable_submethod(RippleOutSeq,PostRippleOutPlan,
      _,PostRippleOutSeqs)),
(/* ADDITIONAL RIPPLING (WITH RIPPLING-IN) */
 /* It may be possible to W.F. at this stage. */
 EarlyWFPlan=unblock_then_fertilize(weak,_),
 (applicable_submethod(RippleOutSeq,
       EarlyWFPlan,_,EarlyWFSeqs)
   -> EarlyWFPossible = yes
   ;  EarlyWFPossible = no),
 RippleInPlan = ripple(direction_in_or_out,_),
 orelse(/* TRY RIPPLING-IN; THEN S.F. ORELSE W.F. */
     (applicable_submethod(RippleOutSeq,
   RippleInPlan,_,RippleOutInSeqs),
      map_list(RippleOutInSeqs,RippleOutInSeq :=>
   PostRippleOutInSeqs-PostRippleOutInPlan,
       orelse(
   /* Possible problem here in that any W.F. cannot be undone */
   /* A committment to S.F. seems appropriate here, but W.F.
      is probably something we want to be able to backtrack
      over (since it is not equivalence preserving).  */
   once((/* STRONG or WEAK FERTILIZE */
 /* This W.F. will not be undone (BUG?) */
 (FertType=strong; FertType=weak),
 PostRippleOutInPlan=
     unblock_then_fertilize(FertType,_),
 applicable_submethod(RippleOutInSeq,
      PostRippleOutInPlan,_,
      PostRippleOutInSeqs))),
   (/* IDTAC (no fertilization possible) */
    PostRippleOutInPlan=idtac,
    PostRippleOutInSeqs=[RippleOutInSeq])),
       PostRippleOutInSeqsPlan),
      /* If early W.F. were possible, only allow */
      /* rippling-in if it enabled some S.F */
      if(EarlyWFPossible=yes,
 member(_-unblock_then_fertilize(strong,_),
PostRippleOutInSeqsPlan)),
      zip(PostRippleOutInSeqsPlan,PostRippleOutInSeqs,
  PostRippleOutInPlans),
      flatten(PostRippleOutInSeqs,PostRippleOutSeqs),
      PostRippleOutPlan=
  (RippleInPlan then PostRippleOutInPlans)),
     (/* (NO RIPPING-IN POSSIBLE, TRY WEAK FERTILIZATION)
 If early W.F. were possible, try that, but allow idtac too
 (i.e., this W.F. may be undone).  */
      ((EarlyWFPossible=yes,
PostRippleOutPlan=EarlyWFPlan,
PostRippleOutSeqs=EarlyWFSeqs);
       (PostRippleOutSeqs=[RippleOutSeq],
PostRippleOutPlan=idtac)))))),
 PostRippleOutSeqsPlans),
zip(PostRippleOutSeqsPlans,PostRippleOutSeqs,PostRippleOutPlan),
write('PostRippleOutPlan is '), write(PostRippleOutPlan),nl,
flatten(PostRippleOutSeqs,OutputSequents),
Plan = ((RippleOutPlan then RippleOutPlansPostPWF) then PostRippleOutPlan)],
       [erase_sequents(OutputSequents,OutputSequentsErased)],
       OutputSequentsErased,
       step_case(Plan)).

/* The step-case method attempts to apply rippling techniques to the
   goal in order to acheive fertilization.  Two types of fertilization
   are possible in general, strong and weak (S.F. and W.F.).
   S.F. closes a proof branch, so is much preferred over W.F.  W.F. is
   not equivalence preserving, which suggests that backtracking over
   W.F. may be needed.  Two types of rippling are available, "in" and
   "out".  The ripple method is parametrized in order to determine
   this type.

   The method is as follows.  Try all possible ripplings (i.e.,
   in_or_out) in an effort to acheive S.F.  Being able to S.F. means
   that there are no wave-fronts left in the term.  Thus, one can see
   that rippling cannot ripple beyond the point of a S.F., since
   rippling requires the movement of wave fronts; if there are none,
   as is the case when S.F. is applicable, rippling is inapplicable.

   Hence S.F. is achieved by attempting: put the goal into normal form
   (N.F.) wrt rippling-out then try S.F.  (Note: the N.F. is not
   unique.)  If that SF fails, continue ripping, with rippling-in,
   then try S.F.  By the above argument, there is no need for the
   ripple method to "lazily" determine the applicability of S.F. after
   each wave rule application.  (Actually, this is not true if the
   annotation is left in place beneath sinks---however, this rippling
   beneath sinks cannot affect fertlilization except in bizzare
   circumstances.  Clam removes annotation from sinks.)

   In cases where S.F. cannot be applied, it may be possible to W.F.
   There is some choice here as to when to W.F.  Typically, W.F. will
   be applicable at multiple points in the course of rippling, since
   one side of the goal will be "weak-fertilizable" whilst the other
   will contain a ripple-redex. The choice is whether to W.F. or to
   reduce the redex.  In particular, notice that additional rippling
   in may not be required to enable W.F.: here this is referred to as
   "early W.F.".  When early W.F. is possible, tentative rippling-in
   (computed in the course of finding S.F.) is discarded iff S.F. was
```

```
   not successful.

   From the point of view of implementation, early W.F. is easier,
   since we W.F. as soon as we can.  However, this might not be
   sensible, since it may be possible to fully ripple out the other
   side of the goal.  Furthermore, there is an efficiency
   consideration.  Recall that we have established (by computing
   N.F. wrt rippling-out and rippling-in) that S.F. is
   inapplicable---it is quite wasteful to recompute (segments of)
   these two reduction sequences.

   Concering backtracking over W.F.  W.F. on the fully rippled goal
   (wrt in and out rippling) cannot be undone; this is a bug really.
   W.F. on the fully rippled-out goal can be undone.  */


%%% Local Variables:
%%% mode: prolog
%%% End:
```

generalise(Exp,Var:Type)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: generalise,v 1.8 1998/09/15 16:00:38 img Exp $
 *
 * $Log: generalise,v $
 * Revision 1.8  1998/09/15 16:00:38  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.7  1998/08/26 12:56:45  img
 * check FV condition
 *
 * Revision 1.6  1998/07/30 16:07:05  img
 * Dont drop unnecessary quantifiers
 *
 * Revision 1.5  1998/06/10 09:26:38  img
 * flag failure to type-guess
 *
 * Revision 1.4  1996/11/02 13:56:25  img
 * Don't generalize propositions/atomic definitions.
 *
 * Revision 1.3  1996/07/10  09:02:44  img
 * use type_of/3.
 *
 * Revision 1.2  1995/10/03  12:52:40  img
 * remove annotaion from goal in preconditions rather than postconditions.
 *
 * Revision 1.1  1994/09/16  09:33:29  dream
 * Initial revision
 *
 */

% GENERALISE METHOD:
% Replace a common subterm in both halves of an
% - equality, or
% - implication, or
% - inequality
% by a new variable.
% Disallow generalising over object-level variables, and over
        % terms containing meta-level variables (too dangerous), and
        % over constant object-level terms, and over terms containing
        % wave-fronts.  Remove annotations since we are giving up on
% the present induction (if any).
/* original
method(generalise(Exp,Var:Type),
       H==>GG,
       [strip_meta_annotations(GG,G),
matrix(Vs,M,G),
        member(M,[(L=R in _),(L=>R),geq(L,R),leq(L,R),greater(L,R),less(L,R)]),
        exp_at(L,Pos1,Exp), \+ Pos1 =[0|_],
        not atomic(Exp),        % disallow generalising object-level variables
        not constant(Exp,_),    % Exp must not be a constant term.
        not Exp = {_},          % don't generalise atomic definitions
        object_level_term(Exp), % Exp must not contain meta-vars or wave fronts
        exp_at(R,Pos2,Exp),\+ Pos2 =[0|_],
        type_of(H,Exp,Type),
\+ Type = u(_),         % don't generalise over propositions
ground(Type),
        append(Vs,H,VsH),
        hfree([Var],VsH)],
       [replace_all(Exp,Var,G, NewG)],
       [H==>Var:Type=>NewG],
       generalise(Exp,Var:Type)).
*/

/* generalized to uniform treatment of binary predicates */
method(generalise(Exp,Var:Type),
       H==>GG,
       [strip_meta_annotations(GG,G),
matrix(Vs,M,G),compound(M),
        append(Vs,H,VsH),        % extend the context for type-guessing
(member(M,[(L=R in _),(L=>R)]) orelse
     (M =.. [P,L,R],
      type_of(H,M,u(_)))),
        exp_at(L,Pos1,Exp), \+ Pos1 =[0|_],
not Exp = (_ = _),% want _=_ in _
        not atomic(Exp),        % disallow generalising object-level variables
        not constant(Exp,_),    % Exp must not be a constant term.
        not Exp = {_},          % don't generalise atomic definitions
        exp_at(R,Pos2,Exp),\+ Pos2 =[0|_],
        object_level_term(Exp), % Exp must not contain meta-vars or wave fronts
```

```
Exp =.. [Func|_],
        (type_of(VsH,Exp,Type) -> true;
 (clam_warning('Cannot guess type of %t(...)\n',[Func]),fail)),
\+ Type = u(_),         % don't generalise over propositions
ground(Type),% or if the type cannot be guessed
freevarsinterm(Exp,ExpFV),
hfree([Var],VsH),
replace_all(Exp,Var,M, NewM),
/* ensure that the generalized term contains only free
variables from the context: e.g., don't generalise double(z)
in
         (z:pnat#v0=double(z)in pnat)=>z:pnat#s(s(v0))=double(z)in pnat
        */
        forall(XXX,ExpFV,(member(XXX:_,Vs);member(XXX:_,H))),
/* Is there any advantage in minimising the prefix? */
% freevarsinterm(NewM,NewMFV),
% disjoint(NewMFV,ExpFV), % dont generalize a term if there are shared vars
% map_list_filter(Vs,(V1:T1):=>(V1:T1),member(V1,NewMFV),NewVs),
NewVs=Vs,
        append(VsH,[Var:Type],VsHVT),
\+ trivially_falsifiable(VsHVT,NewM)],
       [matrix(NewVs,NewM,NewG)],
       [H==>Var:Type=>NewG],
       generalise(Exp,Var:Type)).


/* experimental: generalize a term containing variables that don't
   appear elsewhere */
%%% method(generalise(Exp,Var:Type),
%%%        H==>GG,
%%%        [strip_meta_annotations(GG,G),
%%%  matrix(Vs,M,G),
%%%         append(Vs,H,VsH),
%%%         hfree([Var],VsH),
%%%         exp_at(M,Pos,Exp), \+ Pos =[0|_],
%%%         not atomic(Exp),        % disallow generalising object-level variables
%%%         not constant(Exp,_),    % Exp must not be a constant term.
%%%         not Exp = {_},          % don't generalise atomic definitions
%%%         object_level_term(Exp), % Exp must not contain meta-vars or wave fronts
%%%  freevarsinterm(Exp,FVs),
%%%  replace(Pos,Var,M, NewM),
%%%  freevarsinterm(NewM,MFVs),
%%%  disjoint(FVs,MFVs),
%%%         type_of(VsH,Exp,Type),
%%%  ground(Type),
%%%  \+ Type = u(_),         % don't generalise over propositions
%%%  true],
%%%        [matrix(Vs,NewM,NewG)],
%%%        [H==>Var:Type=>NewG],
%%%        generalise(Exp,Var:Type)).
```

Replace a common subterm `Exp` in both halves of an equality or implication or inequality by a new universal variable `Var` of `Type`. Disallow generalising over object-level variables (not very useful), over constants (not very useful), over terms containing meta-level variables (too dangerous), and over terms containing wave-fronts (messes up the rippling process).

The last 3 conjuncts of the preconditions will always succeed, and are not really needed for applicability test, so they could go in the postconditions, but we have them here to get the second arg of the method instantiated even without running the postconditions. . .

---

induction(Lemma-Scheme)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: induction,v 1.8 1998/09/15 16:00:39 img Exp $
 *
 * $Log: induction,v $
 * Revision 1.8  1998/09/15 16:00:39  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.7  1997/11/08 12:18:06  img
 * typo fix
 *
 * Revision 1.6  1997/10/17 14:25:58  rjb
 * Added source of induction lemma to method arguments.
 *
 * Revision 1.5  1996/12/11 14:06:16  img
 * Merge mthd and smthd libraries.
 *
 * Revision 1.4  1995/10/03  12:55:38  img
 * arity changed to induction/1;  new scheme mechanism
 *
 * Revision 1.3  1995/03/01  02:37:25  img
 *  * Checking induction scheme is a precondition
 *
 * Revision 1.2  1994/12/07  18:45:46  dream
 *  * dynamic version --- uses induction_pre
 *
 * Revision 1.1  1994/09/16  09:33:29  dream
 * Initial revision
 *
 */

method(induction(Lemma-Scheme),
```

```
       H==>G,
       [induction_suggestion(H,G,Scheme),
scheme(Lemma,Scheme,H==>G,BaseSeqs,StepSeqs)],
       [append(BaseSeqs,StepSeqs,Seqs)],
       Seqs,
       induction(Lemma-Scheme)).
```

Do an induction defined by `Scheme` (see `schemes/5` for a description of a scheme). For example, `induction(lemma(plusind)-[(x:pnat)-plus(v0,v1)])` is `plus` induction on `x`.

`ind-strat(Submethods)`

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: ind_strat,v 1.8 1998/09/15 15:16:19 img Exp $
 *
 * $Log: ind_strat,v $
 * Revision 1.8  1998/09/15 15:16:19  img
 * prefer casesplits to wholly flawed inductions
 *
 * Revision 1.7  1998/08/26 12:55:52  img
 * Single call to scheme_suggestion is used to decide between flawed and
 * unflawed induction.  casesplit is tried if it is unflawed and there
 * are no unflawed inductions.
 *
 * Revision 1.6  1998/07/30 16:03:21  img
 * Allow base- and step-case methods to fail on any of the subgoals
 * resulting from induction (this patch just brings the step-case into
 * line with the base case).  This patch is a reponse to the situation in
 * which some progress is made on some base-cases, and yet one of the
 * step-cases fails.  There is an argument that if this situation does
 * arise then the correct thing to do is fail anyway.
 *
 * Revision 1.5  1998/06/10 08:34:06  img
 * allow possibility that mthd(base_case) may not be applicable to base-case
 *
 * Revision 1.4  1997/10/17 17:18:42  rjb
 * Fixed comment.
 *
 * Revision 1.3  1997/10/17 14:25:58  rjb
 * Added source of induction lemma to method arguments.
 *
 * Revision 1.2  1995/10/03 12:54:10  img
 * Induciton preconditions present verbatim to avoid duplication of call
 * to scheme.  Use induction/1 rather than induction/2.
 *
 * Revision 1.1  1994/09/16  09:33:29  dream
 * Initial revision
 *
 */
method(ind_strat(Method),
       H==>G,
       [scheme_suggestion(H,G,induction,ripple,AllInductionInfo),
/* try an unflawed induction */
((selection_heuristic(AllInductionInfo, Scheme,0-_-_),
  scheme(Lemma,Scheme,H==>G,BSeqs,SSeqs));
 applicable_submethod(H==>G, casesplit(datatype(unflawed,P)),Plan,AllSeqs);
 (/* using same analysis, attempt flawed induction if there was no unflawed one */
  \+ (selection_heuristic(AllInductionInfo, Scheme,0-_-_),
      scheme(Lemma,Scheme,H==>G,BSeqs,SSeqs)),
  selection_heuristic(AllInductionInfo, Scheme,_-_-M),
  M > 0,%pointless to try wholly flawed induction (?)
  scheme(Lemma,Scheme,H==>G,BSeqs,SSeqs));
  applicable_submethod(H==>G, casesplit(datatype(_,P)),Plan,AllSeqs))],

       [(var(Scheme) -> /* Doing a casesplit */
    (Method = cases(datatype(P)));
    /* INDUCTION */
    (Method = (induction(Lemma-Scheme) then CasesTactics),
     map_list(BSeqs,BSeq:=>NBSeq,
      ((applicable_submethod(BSeq,base_case(Ms),_,BSeq1),
NBSeq=BSeq1-base_case(Ms))
   orelse NBSeq=BSeq-idtac),
      BSeq1sBTs),
     zip(BSeq1sBTs,BSeq1s,BaseTactics1),
     flatten(BaseTactics1,BaseTactics),
     flatten(BSeq1s,FBSeq1s),
     map_list(SSeqs,SSeq:=>NSSeq,
      ((applicable_submethod(SSeq,step_case(Ms),_,SSeq1),
NSSeq=SSeq1-step_case(Ms))
   orelse (erase_sequent(SSeq,SSeqErase),
   NSSeq=SSeqErase-idtac)),
      SSeq1sSTs),
     /* ensure at least one stepcase has made progress */
        !,
     member(_-step_case(_),SSeq1sSTs),
     zip(SSeq1sSTs,SSeq1s,StepTactics),
     flatten(SSeq1s,FSSeq1s),
     append(BaseTactics, StepTactics, CasesTactics),
     append(FBSeq1s,FSSeq1s,AllSeqs)))],
       AllSeqs,
       ind_strat(Method)).
```

This method has played an important role in the development of the idea of proof-plans, since it was the first large scale method of any significance to be developed ([4]). Note that the preconditions are exactly those of the `induction/1` method. The method constructs its postconditions by explicitly following the way the method is constructed out of smaller methods: After applying the `scheme/5` predicate to determine the step- and base-cases after induction, we apply the `base-case/1` method to the base-cases and we apply the `step-case/1` method to the step-cases. The single argument of the `ind-strat/1` method will be bound to the structure representing the chain of constituent methods built up during the application of the larger method. Since this structure is usually very long-winded, Clam's pretty-printer treats it specially, and suppresses it to indicate just the induction scheme and variable (making it look very much like the induction method).

`normalize(Normalisation)`

The `normalize/1` method is a compound method, which is built as an iterator over a number of `normal/1` methods. Rather than listing the rather straightforward code of all of these methods, we just summarise their actions:

`normal(univ-intro)` Removes a universal quantifier from the front of the current goal (corresponds to the Oyster `intro`-rule for the dependent function type).

`normal(imply-intro)` Removes an implication from the front of the current goal (corresponds to the Oyster `intro`-rule for the function type).

`normal(conjunct-elim(HName,[New1,New2]))` Replaces a conjunctive hypothesis by two new hypotheses for each of the separate conjuncts.

At various points we also experimented with other normalisation operations, which are at the moment not included in the code. These are:

`normal(univ-imply-intro)` This is as `imply-intro`, but works for universally quantified goals.

`normal(exist-elim(H))` Removes an existentially quantified hypothesis `H:x:T#P` (and adds `P[x0/x]` to the hypothesis list) by picking a witness `x0` (corresponds to the Oyster `elim`-rule for the dependent product type).

`normal(imply-elim(H,Lemma))` Removes an implication `H:A=>B` from the hypothesis list (and adds `B` to the hypothesis list) if `A` is provable using only a lemma or is trivially true (corresponds to the Oyster `elim`-rule for the function type).

This set of normalisation operations is by no means exhaustive, and can (should) be extended in the future when the need arises.

`identity`

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: identity,v 1.2 1998/09/15 16:00:38 img Exp $
 *
 * $Log: identity,v $
 * Revision 1.2  1998/09/15 16:00:38  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 *
 */

% IDENTITY METHOD: too simple for words.
method(identity,
       _ ==> G,
       [matrix(_,MM,G),
        annotations(_,_,X = X in _,MM)],
       [],
       [],
       identity).
```

This terminating method simply checks if the goal is of the form `X=X in Type`, and terminates the plan. Almost no preconditions, no postconditions, no output-formula.

This method illustrates that some operations we would like to do via matching (such as ignoring the universal quantifiers) cannot in fact be done through matching, and must be explicitly encoded in the preconditions. It would maybe be nice if we had a more powerful pattern matching language which would allow us operations like this.

`apply-lemma(Lemma)`

```
/*
 * @(#)$Id: apply_lemma,v 1.2 1998/06/10 08:31:09 img Exp $
 *
 * $Log: apply_lemma,v $
 * Revision 1.2  1998/06/10 08:31:09  img
 * extend context
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 */

% LEMMA METHOD: optionally loaded. Is useful in places, but
% expensive in general.
% Bit of a hack to deal with commutativity of =.
method(apply_lemma(Lemma),
       H==>G,
       [strip_meta_annotations(G,GG),
        precon_matrix(_,AllPre=>Concl,GG),
append( _, Pre, AllPre ),
precon_matrix( [], Pre=>Concl, Matrix ),
        theorem(Lemma,LemmaGoal),
        (instantiate(H,[],LemmaGoal,Matrix,_)
         v
         (Matrix=(L=R in T),instantiate(H,[],LemmaGoal,R=L in T,_)))],
       [],
       [],
       apply_lemma(Lemma)).
```

This terminating method checks if there is a lemma which is a universally quantified version of the current goal (i.e., a lemma that can be instantiated to the current goal). Notice the particular hack to spot lemmas which happen to be of the right form, except that they have the right- and left-hand side of an equality swapped. Not very nice…

`backchain-lemma(Lemma)`

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: backchain_lemma,v 1.3 1998/09/15 16:00:31 img Exp $
 *
 * $Log: backchain_lemma,v $
 * Revision 1.3  1998/09/15 16:00:31  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.2  1998/06/10 08:33:00  img
 * extend context
 *
 * Revision 1.1  1994/09/16 09:33:29  dream
 * Initial revision
 *
 */

% BACKCHAIN_LEMMA METHOD: as lemma, but this time looks
% for lemma that instantiates to C=>G, with G goal and C a
% condition which is trivially true.
%
% Same hack with commutativity of = as above with lemma.
% I realise that lemma above is only a special case of
% backchain_lemma (namely with 0 conjuncts), but it's too tricky
% to formulate it all in one method, so we use two.
method(backchain_lemma(Lemma),
       H==>G,
       [strip_meta_annotations(G,GG),
        matrix(_,Matrix,GG),
        theorem(Lemma,LemmaGoal),
        (   instantiate(H,[],LemmaGoal,Cond=>Matrix,_)
v
(   Matrix=(L=R in T), instantiate(H,[],LemmaGoal,Cond=>R=L in T,_))),
immediate(H==>Cond)],
       [],
       [],
       backchain_lemma(Lemma)).
```

This terminating method looks for universally quantified lemmas that instantiate to `Cond=>Matrix` where `Matrix` is the matrix of the current goal `G` and `Cond` a formula that is trivially true. Thus, this method applies one step of *backward-chaining*. Same hack with commutativity of `=` as with the `apply-lemma/1` method above.

pwf-then-fertilize(Type,Plan)

```
/* -*- Mode: Prolog -*-
 */

 /* Type should be instantiated to either weak or strong */

method(pwf_then_fertilize(strong,PWFertPlan),
H==>G,
        [((PWFertilize = pw_fertilize(_),
   applicable_submethod(H==>G,PWFertilize,_,Seqs)) orelse
  Seqs = [H==>G]),
 map_list(Seqs, Seq :=> (Plan-NewSeq),
          (   ((Plan = fertilization_strong(_),
        applicable_submethod(Seq,Plan,_,NewSeq))
      orelse
      (   NewSeq = Seq, Plan = idtac))),PlanNewSeqs),
         zip(PlanNewSeqs,Plans,NewSeqsF),
 flatten(NewSeqsF,NewSeqs),
 (var(PWFertilize) ->
     (Plans = [PWFertPlan],\+ PWFertPlan=idtac);
     PWFertPlan = (PWFertilize then Plans))],
        [],
        NewSeqs,
        pwf_then_fertilize(strong,PWFertPlan)).


method(pwf_then_fertilize(weak,PWFertPlan),
H==>G,
        [((PWFertilize = pw_fertilize(_),
   applicable_submethod(H==>G,PWFertilize,_,Seqs)) orelse
  Seqs = [H==>G]),
 map_list(Seqs, Seq :=> (Plan-NewSeq),
          (FLR = fertilize_left_or_right(_),
   ((applicable_submethod(Seq,FLR,_,[UH==>UG]),
     Plan = FLR,
     notraw_to_used(UH,HSeq),
     (NewSeq = (HSeq==>UG)))
            orelse
    (NewSeq = Seq, Plan = idtac))),PlanNewSeqs),
         zip(PlanNewSeqs,Plans,NewSeqs),
 (var(PWFertilize) ->
     (Plans = [PWFertPlan],\+ PWFertPlan=idtac);
     PWFertPlan = (PWFertilize then Plans))],
        [],
        NewSeqs,
        pwf_then_fertilize(weak,PWFertPlan)).
```

This method implements piecewise-fertilization (see Blue Book note 1286 for a description of piecewise fertilization).

pwf(Rule)

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: pwf,v 1.6 2008/05/21 14:14:13 smaill Exp $
 *
 * $Log: pwf,v $
 * Revision 1.6  2008/05/21 14:14:13  smaill
 * update for sicstus4
 *
 * Revision 1.5  2005/05/06 19:35:27  smaill
 * fix object vars to coincide with Oyster implementation
 *
 * Revision 1.4  1999/02/02 09:51:01  img
 * revisions for tactic support
 *
 * Revision 1.3  1999/01/11 11:03:18  img
 * *** empty log message ***
 *
 * Revision 1.2  1999/01/11 11:02:47  img
 * Added headers.  Spurious clause deleted.  Additional parameters added
 * in concert with tactic support.
 */

/* these rules should be improved: casesplits may have removed
quantification from goal, and so the skeleton of and ind. hyp may
differ from goal BUT only in the prefix.  ie. need to take an instance
of the hyp.  See BB note 1286 for more information */

method(pwf(i_fert(imp,VV,V,W)),
       H==>Ap=>Bp,
       [contains_wave_fronts(Ap=>Bp),
inductive_hypothesis(raw,VV:A=>B,H,used(pw),NewH),
share_skeleton(Ap=>Bp,A=>B)],
       [hfree([V,W],NewH),erase(Ap,Ape),
 inductive_hypotheses(raw,[V:Ap],[IHa]),
 inductive_hypotheses(raw,[W:B],[IHb]),
 append(NewH,[IHa],H1), append(NewH,[IHb,V:Ape], H2)],
[H1==>A, H2==>Bp],
```

```
pwf(i_fert(imp,VV,V,W))).

method(pwf(i_fert(Op,VV,V1,V2)), /* And/Or */
       OH==>ApBp,
       [contains_wave_fronts(ApBp),
ApBp =..[Op,Ap,Bp], (Op = '#'; Op = '\\'), AB =.. [Op,A,B],
        inductive_hypothesis(raw,VV:AB,OH,used(pw),H),
        share_skeleton(ApBp,AB)],
[hfree([V1,V2],H),
 inductive_hypotheses(raw,[V1:A],[IHa]),
 inductive_hypotheses(raw,[V2:B],[IHb]),
 append(H,[IHa],H1), append(H,[IHb],H2)],
[H1==>Ap, H2==>Bp],
pwf(i_fert(Op,VV,V1,V2))).

method(pwf(and_i_cfert),
OOH==>G,
[is_annotated(G,#,[TagA:Ap,TagB:Bp]),% two holes
 /* this clause would otherwise overlap with strong
    fertilization; we exclude that possibility here */
 (TagA = hole ->
     (contains_wave_fronts(Ap),
      inductive_hypothesis(raw,Va:A,OOH,used(pw),BH),
              share_skeleton(Ap,A));
     BH = OOH),
 (TagB = hole ->
     (contains_wave_fronts(Bp),
      inductive_hypothesis(raw,Vb:B,OOH,used(pw),AH),
      share_skeleton(Bp,B));
     AH = OOH)],
[],
[AH==>Ap, BH==>Bp],
pwf(and_i_cfert)).

method(pwf(or_i_fert(Side)),
OH==>G,
[is_annotated(G,\,[S1:AAp,S2:BBp]),
 (S1 = hole -> (Ap = AAp, Side = left);
     (Ap = BBp, Side = right)),
 inductive_hypothesis(raw,Va:A,OH,used(pw),H),
 share_skeleton(Ap,A)],
[],
[H==>Ap],
pwf(or_i_fert(Side))).

method(pwf(and_e_fert(Side,Va,V)),
OH==>Ap,
[inductive_hypothesis(raw,Va:AB,OH,used(pw),H),
 is_annotated(AB,#,[S1:AA,S2:BB]),
 (S1 = hole -> (A = AA, Side = left);
     (A = BB, Side = right)),
 share_skeleton(Ap,A)],
[hfree([V],H),
 inductive_hypotheses(raw,[V:A],[IHa]),
 append(H,[IHa],H1)],
[H1==>Ap],
pwf(and_e_fert(Side,Va,V))).

method(pwf(imp_ir_fert),
OH==>G,
[is_annotated(G,=>,[front:B,hole:Ap]),
 inductive_hypothesis(raw,V:A,OH,used(pw),H),
 share_skeleton(Ap,A)],
[],
[H==>Ap],
pwf(imp_ir_fert)).

method(pwf(imp_e_fert(VV,V)),
OH==>Ap,
[inductive_hypothesis(raw,VV:AB,OH,used(pw),H),
 is_annotated(AB,=>,[front:B,hole:A]),
 share_skeleton(Ap,A)],
[hfree([V],H),
 inductive_hypotheses(raw,[V:A],[IHa]),
 append(H,[IHa],H1)],
[H==>B,H1==>Ap],
pwf(imp_e_fert(VV,V))).
```

This is part of piecewise-fertilization.

2.3.2 Default configuration of methods and submethods

Although methods can be loaded into and deleted from the system (see §3.3 on page 92), the system starts up with a default configuration of methods and submethods, as follows. Recall that only methods are stored in the library—submethods are simply methods that have been loaded as such. (See `lib-load/1` for more information on load methods and submethods.)

The contents of the methods database can be found with the command `list-methods/0`; on normal Clam startup this is

`base-case/1`	`generalise/2`
`normalize/1`	`ind-strat/1`

This configuration is chosen in such a way that, when using the depth-first planner (see §2.4.2 on page 74), the systems combines methods as prescribed in the induction strategy encoded in the `ind-strat/1` method.

A number of methods are also loaded as submethods by default, since they are needed by the methods above. Since these submethods are not directly used by planners (but only called from within methods), the order of the submethods database is largely irrelevant. They can be found via `list-submethods/0`:

`equal/2`	`normalize-term/1`	`casesplit/1`
`existential/2`	`sym-eval/1`	`apply-lemma/1`
`backchain-lemma/1`	`normal/1`	`induction/1`
`base-case/1`	`wave/4`	`unblock/3`
`unblock-lazy/1`	`unblock-then-wave/2`	`ripple/2`
`unblock-fertilize-lazy/1`	`fertilization-strong/1`	`weak-fertilize/4`
`weak-fertilize-left/1`	`weak-fertilize-right/1`	`fertilize-left-or-right/2`
`cancellation/2`	`ripple-and-cancel/1`	`fertilize-then-ripple/1`
`fertilization-weak/1`	`fertilize/2`	`unblock-then-fertilize/2`
`step-case/1`	`elementary/1`	

2.4 The basic planners

This section will discuss the planners that are part of Clam and which can be used to construct proof-plans for a given theorem, using the available methods. The first subsection will discuss the general mechanism of the planners, and the essential predicates which are common to all planners. The other subsections each discuss one type of planner in more detail. This is of course not meant to suggest that the current set of planners is in any way final or optimal. Together with the formulation of more and better methods, the formulation of more and better planners is a major research topic in Clam.

2.4.1 The basic planning mechanism

All planners currently employed in Clam are forward chaining planners: every planner starts by looking at the top sequent of the theorem to be proved, and then tries to find out which methods are applicable (i.e., which methods have a matching input-slot and a succeeding preconditions-slot). After picking one of these applicable methods the planner computes the output sequent by evaluating the postconditions-slot of the chosen method. This output sequent will then serve as the input sequent for the next recursive cycle of the planner, until a method has been found which terminates the plan (in other words: until a terminating method (a method with an empty output-slot) has been found).

In this description of the planning process, a number of choice points occur: often more than one method will be applicable to the input sequent, and one method may apply in more than one way (i.e., its preconditions may be satisfied in more than one way). The planners described below differ in the way they behave at these choice points (in other words: they differ in the way they traverse the search space generated by the applicability of the methods). This search space for the planners is also called the *planning space*. Some of them will make a rather uninformed but cheap choice, some will try to make a more informed choice; some of them will make sure that choice gets equal treatment, others will favour one choice over others, etc.

The plans that are produced by Clam's planners are not just sequences of methods. Remember that the output-slot of a method is a list of sequents. Thus, the application of one method can generate a number of output-sequents, and further methods will have to be applied to each of these sequents. These methods are chained together using Oyster's `then/2` connective. An expression of the form

M_1 `then` $[M_{1,1} \ldots M_{1,n}]$ denotes the application of method M_1, followed by the application of methods $M_{1,1} \ldots M_{1,n}$ to the resulting n subgoals. As a result, a plan produced by Clam is a tree-structured object, with as the elements of the tree the methods that are applied as part of the plan. Figure 2.4 shows an example of a simple plan produced by Clam. A tree-structured plan is called a *branching plan*.

The Clam pretty-printer treats the `then/2` connective specially: Since the expression M_1`then`$[M_{1,1}]$ is equivalent to M_1 `then` $M_{1,1}$ if M_1 only produces one subgoal, the unbracketed version will be produced by the pretty-printer in that case. This explains why only the subgoals of the `induction/1` method in figure 2.4 appear to be bracketed, since it is the only method in the plan shown there which produces more than one subgoal.

The set of planners described below is not in any way complete. Only planners with very simple search strategies have been built (depth-first, breadth-first, iterative deepening), and so far this has proved sufficient because the search space at the planning level has been fairly small. However, in the future it might be necessary to add more sophisticated planners. An obvious possibility is for instance a planner that has access to "the plan so far". Such a planner could choose steps on the basis of steps chosen earlier in the plan. This can for instance be used as an anti-looping device.

All of the planners described below conform to a common interface, and can all be called in a similar way. For planner called '`planner`' there will be predicates `planner/[0;1;2;3]`, as follows:

`planner(+Sequent,?Plan,?Output)`

Succeeds when `Plan` is a plan (a tree of methods) which, when applied to `Sequent`, will result in the output sequents `Output`. Although it is possible to use this predicate to check the correctness of a given plan (mode `planner(+,+,+)`), or to compute the output sequents of a given plan (mode `planner(+,+,-)`), it is most often used to generate a plan with desired output sequents for a given `Sequent` (mode `planner(+,-,+)`). More often than not, the desired out sequents will be the empty list (i.e., we will be interested in the generation of *complete plans*). This is why we have the predicate:

`planner(?Plan,?Output)`

This predicate is as `planner/3`, except that the input sequent to the planner is taken to be the current Oyster sequent. The predicate can be used to check the correctness/completeness of a given plan (mode `planner(+,+)`), or (more likely) to generate a plan with given outputs for the current sequent (mode `planner(-,+)`). Often, we will want to construct a complete plan for the current sequent, which is why we have the predicate:

`planner(?Plan)`

This predicate is as `planner/2`, except that it forces the output list to be the empty list. In other words, `planner/1` only produces complete plans. Can be used to check correctness/completeness of plans, or to generate plans.

The final member of the family of predicates that exists for each planner is `planner/0`:

`planner`

This predicate is as `planner/1`, except that it pretty-prints the generated plan on the output stream.

As explained above, a crucial step in the planning process is to find out which methods are to a given input sequent. For this purpose, all planners use the same predicate, namely the predicate `applicable/[1;2;3;4]`. The different versions of this predicate will be described below, before we continue with the description of each of the planners.

`applicable(+Sequent, ?Method)`

Succeeds if `Method` is applicable to `Sequent`. The `Method`'s applicability is tested by matching its input-slot against the `Sequent` followed by evaluating its preconditions which must succeed. This predicate can be used either to test the applicability of a given `Method`, or to generate all applicable `Methods`.

A special case is when `Method` is of the form `try M`. In this case `applicable/2` succeeds even when `M` is not applicable to `Sequent`.

`applicable(?Method)`

A version of `applicable/2` with the first argument (the `Sequent`) defaulting to the *current sequent*. The current sequent is the sequent at the current position in an Oyster proof tree (as specified by the predicates `select/[0;1]`, `slct/[0;1]` and `pos/[0;1]`).

`applicable(+Sequent, ?Method, ?PostConds, ?Outputs)`

This predicate succeeds if `Method` is applicable to `Sequent` with output-slot `Outputs`, while the postconditions-slot of `Method` evaluates to `PostConds`. Whereas `applicable/[1;2]` only evaluates a method's preconditions-slot and matches it against the input-slot, `applicable/4` also evaluates the postconditions-slot, and matches it against the output-slot. Possible usage of this predicate includes mode `applicable(+,+,-,-)` to compute the postconditions and output-slot of a given method, `applicable(+,-,-,-)` to search for applicable methods, and `applicable(+,-,-,+)` or `applicable(+,-,+,-)` to search for methods that will give certain desired postconditions or output-slot.

A special case is when `Method` is of the form `try M`. `try M` behaves exactly as `M` when `M` is applicable to `Sequent`. If `M` is not applicable to `Sequent`, then `applicable/4` will still succeed with `PostConds=[] Outputs=[Sequent]`.

The convention in that a the postconditions of methods are not allowed to fail if the input-slot matches and the preconditions succeed can be expressed by stating that `applicable/[3;4]` always succeed if `applicable/[1;2]` succeed. It is considered an error if in some situation `applicable/[3;4]` fail but `applicable/[1;2]` succeed. As a result, `applicable/[3;4]` subsume `applicable/[1;2]`. However, `applicable/[1;2]` avoids the computation of the `Method`'s postconditions-slot, and is therefore much cheaper, so we keep both versions of the predicate around.

`applicable(?Method, ?PostConds, ?Outputs)`

This predicate is to `applicable/4` what `applicable/1` is to `applicable/2`: It is the same as `applicable/4`, but with the `Sequent` argument defaulting to the current sequent

---

`applicable-submethod/[1;2;3;4]`

All these predicates are exactly as their `applicable/[1;2;3;4]` counterparts, except that they test for applicability of submethods, instead of methods.

`applicable-anymethod[1;2;3;4]`

The predicates `applicable-anymethod/[1;2;3;4]` is the disjunction of the predicates `applicable/[1;2;3;4]` and `applicable-submethod/[1;2;3;4]`

After all these general predicates, we will now turn to the discussion of each of the planners.

2.4.2 The depth-first planner

`dplan/[0;1;2;3]`

The depth-first planner is the simplest of Clam's family of planners. Whenever it comes to a choice point in the planning process, it just pursues all choices in a chronological order. Thus, methods are tried in the order in which they are generated by the `applicable/[1;2;3;4]` predicate, that is, the order in which they occur in the methods database, and choice points in the evaluation of pre- and postconditions-slots are determined by Prolog's search strategy.

As a result, this planner is the fastest of all in the sense that it does not spend much time considering what choice to make next. On the other hand, it is very prone to getting trapped into infinite branches in the planning space, or to making very uninformed and obviously wrong choices. The only control that the user has over the behaviour of the depth-first planner is by reordering the (sub)methods in the database, or by re-coding the pre- and postconditions of the methods. By carefully ordering the (sub)methods database, a large number of theorems can be proved even with a brain damaged planner such as the depth-first planner (using suitably chosen methods from §2.3.1 on page 42, all theorems mentioned in [7] can be proved using the depth-first planner).

A number of optimisations have been made in the code of the depth-first planner which make it slightly less brain damaged. Both of these optimisations are for the case when we are searching for a complete plan, that is: a plan with an empty list of output sequents. The first optimisation applies to all planners currently part of Clam, and I believe it should apply to all planners ever part of Clam. When looking for applicable methods, the planners first look for terminating applicable methods, that is: applicable methods whose output-slot is an empty list of sequents. If any such methods can be found, the chronologically first one of these is chosen, and the planner terminates.

The second optimisation is a more debatable one, and applies when the planner produces a branching plan. Due to the depth-first nature of the planner, it first tries to fully complete one branch of a plan before starting the construction of the next branch. The optimisation consists of freezing the computation for a branch once the planner has found a complete plan for a branch. This means that failure in the construction of the n-th branch of a plan will never lead to re-computation of any of the $n-1$st branches of the plan. This optimisation relies essentially on the *linearity assumption* for proof-plans, which says that subplans for conjunctive branches can always be combined without interference. This assumption justifies not re-doing any previously completed branches after failure in a later branch. It is not entirely clear whether this linearity assumption holds for proof-plans. It does not hold for plans in general (see numerous articles in the planning literature on this, or [30] for the case of proof-plans in particular).

`plan(+Thm)`

The `plan/1` predicate composes the loading of the definitions relating to the conjecture `Thm` (see §3.3 on page 92) with the search for a depth-first plan.

`dplanTeX/[0;1]`

This behaves as `dplan/[0;1]`, only the file clamtrace.tex is automatically created in the startup directory. This file is a complete source of the proof-plan attempt. Meta-level annotations are drawn in the 'box-and-underline' style; sinks and other annotations are also depicted. `portray-level/3` affects the TEX output as it does in the non-TEX case. The style files require to run LATEX on the file clamtrace.tex are supplied in the Clam distribution directory info-for-users.

These annotations are produced via a special collection of portray predicates, given in proof-planning/portrayTeX.pl.

NB. If a planning attempt is interrupted for some reason during `dplanTeX`, Clam may be left in a state in which it continues writing to the trace file. Terminate LATEX tracing by calling `stopoutputTeX/0`.

2.4.3 The breadth-first planner

`bplan/[0;1;2;3]`

A second planner which follows an uninformed search strategy is the breadth-first planner. It traverses the planning space in a breadth-first way, that is: it first tries to construct a plan of size n in all possible ways, before it goes on to investigate any plans of size $n+1$. The *size of a plan* is defined as the *depth of a plan*: it is the length of the longest branch in the plan, measured from the root-node. For example, the plan shown in figure2.4 on page 85 has size 6. It would be possible (and useful and interesting) to develop breadth-first planners that would use different metrics for measuring the size of a plan. Another possible metric to investigate would be the *weight of a plan*, which is defined as the total number of nodes in a plan. Under this metric, the plan from figure2.4 on page 85 would be size 8.

The major advantages of breadth-first planning are that firstly it will always find a plan if there is one, and secondly that it will always find the shortest possible plan. The combination of these properties is generally called *admissibility*

However, the breadth-first planner is very slow to generate plans for two reasons. The first reason is inherent to breadth-first planners in general: they exhaustively traverse the planning space (which typically grows exponentially at each deeper level), and consequently take ages to reach any significant depth. The second reason is more specific to Clam: Clam, and all its planners, are implemented in Prolog, which is naturally more suited for depth-first than breadth-first search strategies. Consequently, the second of the two optimisations that have been applied to the depth-first planner (see previous section), could not be applied to the breadth-first planner, which will therefore spend much more time backtracking through rather useless branches in the planning space. The result of this is that the breadth-first planner is too slow to generate any but the simplest plans. In fact, the only realistic plan ever generated by the breadth-first planner is the one shown in figure2.4 on page 85.

2.4.4 The iterative-deepening planner

`idplan/[0;1;2;3]`

A good compromise between the efficiency of the depth-first planner and the exhaustive nature of the breadth-first planner is the last of Clam's uninformed planners, the iterative-deepening planner. This planner performs a depth-first search similar to the depth-first planner, but only searches through plans up to a maximum length n. If no plans can be found up to length n, the iterative-deepening planner increases the maximum length to $n + 1$ and starts again. This strategy ensures that the iterative-deepening planner has the admissibility property of the breadth-first planner, but that it can be implemented as an efficient depth-first planner (enhanced with a cut-off depth).

It might look at first sight that the iterative-deepening planner must be really inefficient, since after increasing the cut-off depth from n to $n+1$, it re-does all the work up to level n in order to investigate the plans of level $n + 1$. However, since the planning space grows exponentially with n, there are as many plans of length $< n$ as there are of length n (namely $O(b^n)$ in both cases, where b is the branching factor of the planning space). In fact, it can be shown (e.g., see [19]) that among all uninformed search strategies which are admissible, iterative deepening has the lowest asymptotic complexity in both time ($O(b^n)$) and space ($O(n)$). Breadth-first search on the other hand is only asymptotically optimal in time and is really bad (exponential) in space. The actual complexity of breadth-first search is of course lower than that for iterative-deepening (namely by the small constant factor $b/b{-}1$), but this is easily off-set by the difference in space-complexity in favour of iterative-deepening. Thus, iterative-deepening is asymptotically optimal in both time and space, whereas breadth-first is asymptotically optimal only in time and really bad in space, and the actual complexities of iterative-deepening and breadth-first are very close.

`idplanTeX/[0;1]`

This is to `idplan/[0;1]` what `dplanTeX/[0;1]` is to `dplan/[0;1]`.

Two generalisations of the iterative-deepening planner are possible. As with the breadth-first planner, it would be interesting to investigate the use of other metrics than depth to compute the size of a plan. Secondly, there is no reason why we should increase the cut-off depth by 1 every time. We can in general increase the cut-off depth from n to $n + \delta$, where δ can be any fixed number, or even a function of n. The behaviour of δ for the iterative-deepening planner is under control of the user via the predicate:

`bound(-B)`

On successive backtracking, `B` should be bound to increasing values to be used as the cut-off depth for the iterative-deepening planner. A possible (and the default) implementation for `bound/1` is:

```
bound(B) :- genint(B).
```

which increases the cut-off depth by 1 each time. Alternatively, we could define:

```
bound(B) :- genint(B,n)
```

with `n` any positive integer. This would increase the cut-off depth by steps of n each time. An even more flexible definition of `bound/1` would be:

```
bound(B) :- genint(N), bound(N,B).
bound(N,B) :- N>0, N1 is N-1, bound(N1,B1), delta(N,D), B is B1+D.
delta(1,8).
delta(2,4).
delta(3,2).
delta(N,D) :- N>3,D=1.
```

which increases the cut-off depth by a varying amount, computed by the predicate `delta/1`. In this example, the cut-off depth would go through the sequence $8, 12, 14, 15, 16, \ldots$.

`viplan/[0;1;2;3]`

---

viplan/[0;1;2;3] is exactly as idplan/[0;1;2;3], but produces *vi*sually attractive output which enables the user to follow the planner's path through the search space of applicable methods. Works only on VT100 look-a-like terminals.

2.4.5 The best-first planner

`gdplan/[0;1;2;3]`

The only planner in Clam that employs a heuristic search strategy (that is: a search strategy that is informed by properties of the planning space) is the best-first planner. This planner is very similar to the depth-first planner, except that its behaviour on choice points can be programmed by the user, through the predicate `select-method/3`.

`select-method(+Sequent,?Method,?Output)`

This predicate takes a `Sequent`, and should return the `Method` that should be applied at this point in the planning process, and the `Output` sequents that this `Method` should produce. On backtracking, this predicate should produce further choices for the method to be applied to `Sequent` during the planning process. In general, this predicate will investigate which methods are applicable to the given `Sequent`, and then select one of these `Method` for application by the planner. At first sight it would not appear necessary to return the list of `Output` sequents (i.e., the instantiated output-slot) as well as the chosen method. However, a chosen method might be applicable in more than one way (through choice-points in the preconditions-slot). By specifying the `Output` slot, the user can control not only which `Method` will be applied, but also how it will be applied. When writing a particular version of the `select-method/3` predicate, care should be taken to not just blindly generate first all applicable methods, and then perform some selection procedure. Firstly, generating all applicable methods can in general be very expensive, and most of these methods will then be ignored by the planning process anyway. Secondly, an infinite number of methods might be applicable (or: a method might be applicable in an infinite number of ways). This situation, corresponding to an infinite branching factor in the planning space, would lead to non-termination of the `select-method/3` predicate, and therefore of the best-first planner. An example implementation of `select-method/3` is the following trivial version which just mimics the chronological behaviour of the `applicable/4` predicate, making the best-first planner behave as a depth-first planner:

```
select_method(Sequent, Method, Output) :-
    applicable(Sequent, Method, _, Output)
```

For the reasons discussed above, this implementation would be much better than the following, equivalent, code:

```
select_method(Sequent, Method, Output) :-
    findall([Method,Output],
            applicable(Sequent, Method, _, Output), L),
    member([Method,Output], L).
```

The search spaces encountered at the planning level have so far been so small that we have had no real need for the heuristic planner, and as a result, no coherent heuristic strategy is implemented at the moment.

2.5 The hint planners

The hint mechanism in Clam 2.8.4is unsupported.

This section describes Clam's Hint Mechanism (HM). This mechanism provides the user with a means of helping Clam build plans for proofs by giving it hints like those found in mathematical proofs.

Some proofs require the use of techniques for which we don't have the general knowledge required to write a method. Clam would be unable to find a proof-plan for a theorem whose proof requires the use of such techniques just like a student of mathematics would find hard to prove some theorems if he or she had not been given a hint to solve a particular hard step of the proof.

Ideally, Clam should only use constrained methods and a good heuristic function for the Best-first planner which, combined, would tell Clam the appropriate choices to make at each node of the search space. This way, search would be minimal and Clam would find a plan quickly for every provable sequent. Unfortunately we don't have yet a good uniform heuristic function and all the methods we require to prove all theorems and eliminate search.

What we often have though, is an insight, coming rather from experience than from some kind of theory, that tells us what proof techniques to follow at certain stages. The central idea behind the HM is that this insight can be formalised in a language and incorporated to Clam in the form of hints. This enables Clam to use the knowledge contained in the hints to prove harder theorems. By doing so, it also enables us to use a more versatile environment in which we might discover, by experimenting, the underlying theories to develop the methods and heuristics required to achieve a fully automatic theorem prover.

Giving hints to Clam then, consists of telling it what technique it should use to solve a particular sub-problem. The technique can be a regular method known to Clam or a special kind of method called *hint-method* predefined by the user. When giving regular methods, the user simply alters the order in which the methods are tried in the search, thus saving Clam some work. When giving hint-methods on the other hand, the user is introducing a special proof procedure that is only applicable to a reduced number of cases. These cases are specified outside the hint-methods in pieces of code called *hint-contexts*. Hint-methods can only be used via the hint mechanism.

The Hint Mechanism for Clam consists of the following parts:

1. A language to express hints.
2. An extension of the library mechanism to handle a database of hints in the same way it handles methods and submethods.
3. A set of planners very similar to the planners described earlier but with the facility of using a given set of hints to build the plan.

Using the hint mechanism we can give Clam hints in two ways: in *batch* mode or *interactive* mode. In the batch mode, the user provides the planner, from the start, with a list of all the hints he or she considers appropriate and then the planner carries out the standard planning process and tries to use the given hints to build the plan. In interactive mode, an interactive session with the planner enables the user to examine selected parts of the planning process and provide the relevant hints "on the spot".

The full description of the development of this mechanism can be found in [21].

2.5.1 The hint-methods and hint-contexts

Hint-methods are very similar to methods. They live in a separate data base but they are handled in a similar way (see §3.3 on page 92). The main difference is that they are parameterized by a predicate called `hint-context`[6]. This predicate appears as the first precondition of the hint-methods and has two uses. The first one is to define different cases (one in each clause) where the hint-method is applicable, that is, specific theorems or families of theorems. The second use, is to provide the hint-method the instantiation of variables required by the rest of the preconditions, postconditions and output. For instance, if the hint-method generalises subexpressions, the hint-context will indicate what theorems need a generalisation, and what the subexpressions to be generalised are in each case.

When the user wants to prove a theorem using a hint, he must first decide what hint will be needed and then design a hint-context clause for the theorem (family of theorems) before running the planner. `hint-context` clauses must be loaded just like regular Prolog code. When the planner is run, the `hint-context` clause must already be present in memory for the planner to use it. The `hint-context` clause is defined as follows:

```
hint_context(<hint-method>, <label>, <Input>, <Parameters> ) :-
    <body>.
```

Hint-method is the name of the hint-method to which the context is linked. *Label* is a constant to distinguish this context clause from the rest. *Input* is the input sequent and *parameters* is a list of parameters to be instantiated in the context. *Body* may be any Prolog code.

Hint-methods are defined in separate files in the "hint" directory of the library using the following pattern:

```
hint( name( label, ...  ),
      input,
      [hint_context(name,label,input,[term1,,...,termn]),...],
      postconditions,
      output,
      tactic  ).
```

Figure 2.5 shows an example of a hint-method and figure 2.6 shows some hint-contexts defined for it.

2.5.2 The hint planners

The hint mechanism currently has extensions of the Clam planners described above, to handle hints. The extensions are: dhtplan for dplan, idhtplan for idplan and

[6] This is a dynamic predicate, so it can be defined in various files and consulted when necessary. Currently, there is a file called `hint_contexts` in `meta-level-support/` where all the `hint-context` clauses are defined.

gdhtplan for gdplan. They all work with the same arguments as their non-hint cousins except that the first argument is now an extra argument where a list of hints is to be passed.

`dhtplan/[1;2;3;4]`

This is the hint version of dplan/4. The first argument must be a list of hints and the rest of the arguments work exactly as in dplan/4. The planner will do a depth-first search to build a plan but, before selecting the next applicable method at each decision point, it will try to use any of the hints given in the first argument. If the list of hints is empty, dhtplan will perform exactly as dplan/4.

`idhtplan/[1;2;3;4]`

This is the hint version of itplan/4. The first argument must be a list of hints and the rest of the arguments work exactly as in itplan/4. The planner will do a iterative-deepening search to build a plan but, before selecting the next applicable method at each decision point, it will try to use any of the hints given in the first argument. If the list of hints is empty, dhtplan will perform exactly as itplan/4.

`gdhtplan/[1;2;3;4]`

This is the hint version of gdplan/4. The first argument must be a list of hints and the rest of the arguments work exactly as in gdplan/4. The planner will do a best-first search to build a plan but, before using the next method provided by the heuristic function at each decision point, it will try to use any of the hints given in the first argument. If the list of hints is empty, dhtplan will perform exactly as gdplan/4.

2.5.3 The definition of hints

A hint for Clam is a specification of a position in the plan tree and an action to perform at that point. There are *regular* hints and *always-hints*. Regular hints are used only once in a plan and, after they have been used, they are removed from the list of hints. The always-hints on the contrary, are used as many times as possible and remain in the list of hints. In all, there are four possible kinds of hint:

`after(<position>, <action>)`

This is a regular hint that specifies an *action* to be taken when the current node of the partial plan is a descendant of the *position* given in the first argument.

`imm-after(<position>, <action>)`

This is a regular hint that specifies an action to be taken when the current node of the partial plan is a daughter node of the position given in the first argument.

`alw-after(<position>, <action>)`

This hint is the "always" version of the *after* hint above. It has the same effect but it won't be removed from the hint list after it has been used.

`alw-imm-after(<position>, <action>)`

This hint is the "always" version of the *imm-after* hint above. It has the same effect but it won't be removed from the hint list after it has been used.

A position in a partial plan is given by a *path section*. This is a sequent of methods with their arguments separated by "then". For example:

```
induction(_) then ..... induction(_) then tautology(_)
```

Methods in a path section may also specify what branch to follow after its application (branch extension). For example:

```
after( casesplit(_)-2, <action> )
```

indicates that *action* should be performed on the second branch of induction (i.e., step case). We can even use an anonymous Prolog variable in place of any method where we do not care about what method is used. For example:

```
after(_, <action>)
```

indicates that the action is to be taken immediately.

This mechanism is in general enough to specify a position in the plan tree by giving a path section consisting of a single method (with possibly a branch extension), but the system allows the use of a more general path if it is needed.

Actions may be either a method, a hint-method, a term of the form:

```
no( <Method | Hint-method> )
```

or the constant *askme*. If the action proposed is a method or a hint-method, the hint suggests that the planner should try applying the action when the position in the plan tree has been reached. If the action specified is a *no*-term, the planner will avoid applying its argument when the position is reached. Finally, if the action is the constant "askme", the planner, once the position has been reached, will invoke the interactive hint mechanism (see below).

When using a hint involving "immediately after", if the position indicated is reached and the action is not applicable, the system will start the interactive mode. This will enable the user to check why the action could not be performed interactively. If the hint does not involve "immediately after" then the system will not stop when the action is not applicable. This is because the position where the planner is supposed to apply the action is more approximate and the system would have to stop in too many places before reaching the appropriate position.

2.5.4 The interactive session

When the interactive hint mechanism is triggered, a brief menu as a prompt is displayed as follows.

```
[ t, pro, seq, pla, c, a, e, sel, r, h ] <?>


        The options of the menu are:

          (t)est method/hint
          (pro)log,
          (seq)uent.
          (pla)n.
          (c)ontexts.
          (a)pplicable methods,
          (e)dit hint list.
```

```
(sel)ect method.
(r)esume
(h)elp.
```

The (t) option allows the user to test the applicability of a method or hint-method. It displays the last instantiation of all the succeeding preconditions. That is, the system will try to make all preconditions true and in this process it might backtrack finding different instantiations for the variables in each case. If it is the case that not all the preconditions are satisfied, then the system will display the last instantiation of the variables tried. If all preconditions succeeded then it displays all succeeding postconditions and the output.

This option is helpful for debugging purposes when the user thought a method (hint-method) was applicable at a certain stage, but it was not, and he would like to know what went wrong.

The (pro) option allows the user to send goals directly to Prolog. As it is implemented, the metainterpreter will show the instantiation of variables if the goal succeeded. All variables in the goal will be numbered by order of appearance and will be displayed with the corresponding instantiation. Type the goal "true." to return to the main menu.

The (seq) option displays the current sequent in the planning process.

The (pla) option displays the partial plan constructed so far by the planner and indicates with $< current >$ the section of the plan currently being computed.

The (c) option displays all hint-context clauses currently in memory. The (a) option displays all applicable methods and hint-methods to the current sequent having the specified output.

The (e) option. When a *regular* hint in the list given to the planner is used, it is removed from the list so that it can only apply once. When using the interactive hint mechanism, the user may want to restore the hint into the list to try applying it again or may want to add or delete another hint. This can be done using (e) option. When called, this option shows the list of hints and another menu to edit it.

The (sel) option. The aim of the interactive hint mechanism and of hints in general is to help the planner in deciding what to do during difficult stages of the planning process. Once the user has examined the planning process with the other options of the menu, she may use (sel) option to tell the planner what method or hint-method it should apply.

The (r) option terminates interactive session, leaves the askme hint in the list, undoes all changes done in the session and continues with normal planning. This option is useful when the planner stopped in an undesired stage, or if the user just wants to trace what the planner is doing. In the latter case, a good hint to try would be:

```
after( _, askme )
```

(h) option displays a longer menu to remind the user what the options are.

Before the prompt, the program sometimes gives a notice saying what effects it is looking for. In these cases, the planner is searching for methods or hint-methods that yield this effects (normally []) in their output. When the prompt is not preceded by any notice, it means the system will display or apply any method or hint-method without restrictions on what the output should look like.

2.5.5 Meta-Hints

- It is a good idea to trace the planning process using alw-after(_,askme). It gives an idea of what the sequent looks like at certain stage, what the hypothesis are, etc. This helps designing the contexts for a more automatic proof.
- Remember that almost all output produced by the system is "portrayed". This means that what you see is normally nicer that the real representation. Copying literally or using the mouse will seldom work. It is therefore very important to bear in mind the arity of methods and what the arguments are when trying to make the system apply them.
- Remember that every time an *after* or *imm-after* hint is used, it will be removed from the list of hints (unless you use the resume option in interactive mode). You may use the (e) option to insert more hints, before letting the planner continue if you would like to reuse some hint.
- If you modify the hint list and afterwards you use the (r) (resume) option to continue planning, all changes done in the present interactive session will disappear so the list of hints will be just as it was before the session. If you've modified the hint list and you just decided you want to resume planning but you don't want to loose your changes, there is a trick you can use. Select (sel) option and give it an anonymous Prolog variable. This will keep your changes and will select the next applicable method.
- When using the interactive (askme) hint mechanism, it is worth paying attention to the effects the planner is looking for at each stage because all processes related to applicability of methods/hint-methods will be constrained by this parameter. So, when asking for all applicable methods (option (a)) the system will show all applicable methods to the current sequent with the required effects. If you give the system a hint ((sel) option) and it replies it is not applicable, check the required effects. If the system stops and tells you it is looking for some effects you wouldn't like to deal with, use (r) option, the system will immediately look for unrestricted methods.
- Some example theorems proved using hints can be found in [22].

```
/* -*- Mode: Prolog -*-
 * @(#)$Id: eval_def,v 1.2 1998/09/15 16:00:33 img Exp $
 *
 * $Log: eval_def,v $
 * Revision 1.2  1998/09/15 16:00:33  img
 * use immediate/[1,2]; indentation changes
 *
 * Revision 1.1  1996/12/11 15:09:20  img
 * Merge of mthd and smthd libraries.
 *
 * Revision 1.4  1996/05/23 11:20:36  img
 * incorrect argument order in reduction_rule/6
 *
 * Revision 1.3  1996/05/14  16:00:09  img
 * Use reduction rules
 *
 * Revision 1.2  1995/12/04  14:11:57  img
 * evaluation order documented;  don't eval_def functor positions
 *
 * Revision 1.1  1994/09/16  09:34:27  dream
 * Initial revision
 */

/* Symbolically evaluates a term in the goal by applying one of its
 * defining equations.  In order to prevent interference with rippling
 * it will not apply when waves are present.
 *
 * Evaluation is in a outermost/rightmost reduction strategy.  This
 * ordering is as result of exp-at/3 term traversal.  */
method(eval_def( Pos, [Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),
        wave_fronts(_, [], Matrix),
        exp_at(Matrix,Pos,Exp),
\+ Pos = [0|_],%don't eval functors
        not metavar(Exp),%or meta-variables
reduction_rule(Exp,NewExp,C,Dir,Rule,_),
        polarity_compatible(Matrix, Pos, Dir),
immediate(H==>C)],
       [% Once have applied a base-case ignore wave-fronts
        replace(Pos,NewExp,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG)
       ],
       [H==>NewG],
       eval_def(Pos,[Rule,Dir])).
```

Figure 2.2: The `eval-def/2` method.

```
method(normal(imply_elim(HName,Lemma)),
          H==>G,
          [hyp(HName:A=>B,H),
           (hyp(Lemma:A,H)
            v applicable(H==>A,apply_lemma(Lemma))
            v applicable(H==>A,backchain_lemma(Lemma)))],
          [hfree([NewH],H),
           del_element(HName:A=>B,H,HThin)],
          [[NewH:B|HThin]==>G],
          normal(imply_elim(Lemma))).
```

Figure 2.3: The `normal/1` method.

```
induction(lemma(pnat_primitive)-[(x:pnat)-s(v0)]) then
  [base_case([...]),
   step_case([...])
  ]
```

Figure 2.4: A simple Clam plan.

```
% GEN_HINT METHOD:
%
% Generalisation Hint Method.
%
% Positions is a list of subexpression's positions to generalise.
% Var: Variable to be used.
% Hint_name: Name of context in which the method should be used.

hint(gen_hint(HintName, Positions, Var:pnat ),
       H==>G,
       [hint_context( gen_hint, HintName, H==>G, [ Positions ] ),
        matrix(Vs,M,G),
        % the last 2 conjuncts will always succeed, and are not really
        % needed for applicability test, so they could go in the
        % postconds, but we have them here to get the second arg of the
        % method instantiated even without running the postconds...
        append(Vs,H,VsH),
        free([Var],VsH)],
       [replace_list(Positions, Var, M, NewM),
        matrix(Vs,NewM,NewG)],
       [H==>Var:pnat=>NewG],
       gen_hint(Positions,Var:pnat,_)).
```

Figure 2.5: Hint method gen-hint. It is used to generalise variables apart

```
% This hint contexts is for the hint method gen_hint.

% This clause is for the theorem x+(x+x)=(x+x)+x in pnat.
% The parameters are the positions of the variables to be generalised.


hint_context(gen_hint,
             plus_assoc,
             _==>G,
             [
              [[1,1,1],
               [1,1,2,1]]
             ]
            ):- matrix(_,plus(X,plus(X,X))=plus(plus(X,X),X) in pnat,G).

% This clause is for the theorem halfpnat.
% The parameters are the positions of the variables to be generalised.

hint_context(gen_hint,
             halfpnat,
             _==>plus(X,s(X))=S in pnat,
             [
              [[1,1,1],
               [1,1,2,1]]
             ]
            ):-
             wave_fronts(s(plus(X,X)),_,S).
```

Figure 2.6: Hint contexts for hint methods gen-hint and gen-thm used for theorems plus-assoc, halfpnat and rot-length

Chapter 3

Tactics, utilities and libraries

3.1 The tactics

After a plan has been constructed by one of the planners, it can be executed to construct an actual Oyster proof. For this purpose, Clam provides a tactic corresponding to each of the methods,[1] which, when executed, will perform the proof steps specified by the method. Plan execution is particularly simple when the names of methods and tactics are identified (as is the case in Clam). Plans can simply be executed by passing them to the Oyster predicate `apply/1`.

In order to minimise the dependency of Clam on different versions of Oyster, the tactics of Clam assume that Oyster's autotactic has been switched off (that is, the value of the autotactic should be `idtac/0`).

A quasi-autotactic is being used in many of Clam's tactics. This tactic, called `wfftac/0`, or its `repeat`-ed form `wfftacs/0`, is assumed to solve any goals of the form `Expression in Type`. The code of `wfftac/0` is somewhat dependent on the type of theorem that is being proved. A mechanism has been implemented which automatically installs the version of `wfftac/0` appropriate to the current theorem. In order to make this mechanism work, the user should always use the Clam predicate `slct/[0;1]` instead of the Oyster predicate `select/[0;1]`.

`wfftacs(+Flag)`

The `wfftacs/1` predicate enables the setting of `wfftacs/0`. `wfftacs(on)` enables `wfftacs/0` and `wfftacs(off)` disables `wfftacs/0`. By default `wfftacs/0` is enabled.

`wfftacs-status(-Flag)`

`Flag` is instantiated to the current status of `wfftacs/0`.

`slct(Thm)`

The predicates `slct/[0;1]` are identical to `select/[0;1]` in Oyster, except that they also manipulate the definition of `wfftac/0` to be the right form for the selected theorem. Thus, in the context of Clam, `slct/[0;1]` should always be used instead of `select/[0;1]`.

Two tactics are not properly implemented, and rely on the Oyster `because/0` inference rule (proof by intimidation) for their execution. These tactics are

[1] With the exception of the decision procedures for Presburger arithmetic.

- the `clam-arith/0` tactic, called from within the tautology checker to compensate for Oyster's abysmal arithmetic, and
- one of the clauses of the `rewrite-at-pos/3` tactic which performs rewrites. See comments there for an explanation.

These improperly implemented tactics print out a "proof by intimidation" warning when executed.

3.2 Utilities

This section describes some of the utilities which are not strictly needed for the functionality of Clam, but which are indispensable for making life with Clam bearable.

3.2.1 Pretty-printing

`print-complementary-sets(+Cs)`

This predicate prints complementary sets in a manner which makes them somewhat legible. `Cs` is a complementary set, as described under `complementary-sets/[1;2]`.

Example Given the definition of membership we have:

```
| ?- complementary_sets([member2,member3],P),
       print_complementary_sets(P).
(member3)       A=B in int=>void -> member(A,B::C) = member(A,C)
(member2)       A=B in int       -> member(A,B::C) = {true}
```

Plans are constructed as Prolog terms (using the `then/2` functor to combine methods into a tree structure). These terms become quickly unreadable, and for this purpose Clam provides a simple pretty-printer. This is controlled using the `portray-type/1`, `portray-level/3` and the contents of the file portrayTeX.pl. See3.2.2 on the facing page.

`print-plan(+Plan)`

This predicate prints terms in the manner shown in figure2.4 on page 85. The behaviour of this pretty-printer is fixed, and cannot be influenced by the user, except by the use of the portray machinery (see3.2.2 on the facing page).

`print-plan`

This predicate prints the proof underneath the current node in the proof tree in the same manner as `print-plan/1` prints plans. It can be seen as a variation on (abbreviation of) Oyster's `display/0` predicate.

`snap`

This predicate is as Oyster's pretty print predicate `snapshot/0`, except that it provides shorter output by suppressing all the hypotheses, and only printing goals and inference rules.

`snap(+File)`

As `snap/0`, but with output redirected to `File`, rather than the current output stream.

3.2.2 Portrayal of terms

There is some degree of user control over the way in which Clam prints terms. It is possible to control the degree of detail shown when terms are printed on a term-by-term basis, as well the overall format of all prints. Currently, there are formats for plain ASCII, TEX, and Emacs.

A *portray level* governs the amount of information displayed by Clam. This is a natural number less than 100. The higher the portray level, the more detail, the lower the number, less detail.

Portray levels can be changed on a term-by-term basis, or for all terms. A portray level is read/written using `portray-level/3`.

When a term is printed, `portray/2` describes the portrayal information for that term. The appropriate portrayal method is selected according to a specific portray level (if there is one), or according to the default (if there is not), and according to the current `portray-type/1`.

The portray type specifies one of plain ASCII, TEXor Emacs. Various default portrayals for each of these types are defined in portrayTeX.pl.

`portray/2(+T,?Fmt)`

Term `T` should be printed according to the format information `Fmt`. `Fmt` is a list of the form `T:[L1-P1, L2-P2, ...]`. `T` is one of the portray types; `Li` is a natural number less than 100; `Pi` is a list of Prolog terms.

Clam chooses the appropriate element of `Fmt` based on `portray-type/1`. Then, the portray level of the term to be printed is computed: the first `Pi` such that `Li` is greater than or equal to the portray level is obtained. Then each element of this `Pi` is printed in sequence.

For example, we have:

```
portray(Front, [tex:[99-['\\wfout{',Term,'}']],
                normal:[99-['``', Term,'''''','<',Dir,'>']],
                emacs:[99-['wfout(', Term,')']]]) :-
    stripfront(Front,hard,Dir,Term).
```

Which shows how wave-fronts are portrayed according to the portray type.

```
portray(ripple(A,P), [tex:[99-[ripple,'(\ldots)']],
                      _  :[50-[ripple,'(...)'],
                           99-['ripple(',A,',',P,')']]]).
```

shows that terms of the form `ripple(A,P)` at portray levels less than 50 are printed.

`portray-level(+T,?O,?N)`

`T` is a template term; all terms that are printed which `T` matches are portrayed at level `O`. If `N` is non-ground, the portray level for those terms is changed from `O` to `N`.

In the special case of `T == default`, the default portray level is manipulated.

See also `idplanTeX/[0;1]`.

`portray-type(?T)`

`T` is the current portray type. There is a stack of such types: the topmost type is said to be 'current'.

`pop-portray-type`

The current portray type is popped from the stack. The new top element on the stack is the current portray type.

If the stack only contains one element, that element cannot be popped and `pop-portray-type/0` fails.

`push-portray-type(+T)`

`T` becomes the new current portray type. `T` must be one of the following supported types:

`normal` Normal ASCII output. This is the default.

`tex` The output is suitable for processing with TeX.

`emacs` The output is suitable for use with the Clam Emacs mode. This mode provides a limited form of colouring and font control to depict annotations etc.

3.2.3 Tracing planners

In addition to the hint mechanism described in §2.5 Clam provides a very simple tracing package that allows the user to monitor the activities of the planners during the planning process. The user can set a tracing level, using the predicate:

`trace-plan(?Current,?New)`

`Current` will be unified with the current tracing level. If `New` is bound to a non-negative integer, the tracing level will be set to `New`. If `New` is unbound, it will be unified with the current tracing level. Notice that this predicate can be used for multiple purposes. Mode `trace-plan(-,-)` can be used to inquire for the current tracing level, mode `trace-plan(+,-)` can be used to test the current tracing level, mode `trace-plan(-,+)` can be used to set the tracing level.

Currently implemented tracing levels are:

0 No tracing.

10 Prints when the iterative-deepening increases cut-off depth.

20 During plan construction, prints which (sub)methods have been selected. During plan execution via `apply-plan/1` (see below), prints which method is being executed

22 Default.

23 During `lib-load/[1;2;3]` and `lib-load-dep/3`, show which definitions, equations, lemmas etc. are loaded.

30 Prints which (sub)methods are being tested for applicability.

40 Prints when preconditions and postconditions of (sub)methods succeed; `lib-load/[1;2;3]` and `lib-load-dep/3` give verbose output of all rewrites and reduction rules added to the database.

Tracing levels are cumulative. If the current tracing level is set to n, then all tracing levels $k \leq n$ are active. The gaps between the tracing levels have been left to facilitate implementation of future levels.

By default, the tracing level is set to 20.

3.2.4 Applying plans & programs

The products of Clam's planners are only plans for proofs, they are not proofs themselves. In order to produce a proof, we have to apply a plan in the object-level logic (in our case Oyster). As described in §3.1 on page 87, plans can be executed simply by passing them as an argument to Oyster's `apply/1` predicate, because tactics and methods have the same name by convention. This produces a single step proof, in which the only proof step is the application of the proof plan as a single refinement.

However, it is often much nicer to have a proof where each constituent method of a plan corresponds to a single proof step. This can be achieved by executing a plan using the predicate `apply-plan/1`:

`apply-plan(+Plan)`

This predicate applies `Plan`, and makes each method in `Plan` a single refinement step in the proof. Progress of the plan execution process can be monitored using the tracing package (tracing level 20). A minor variation is:

`apply-plan-check(+Plan)`

This predicate is `apply-plan/1`, except that this predicate also checks whether the application of each method produces the output sequents that are specified in the method's output-slot. If this check fails, `apply-plan-check/1` gives an error message about the failing method and its position in the proof tree and fails.

`prove(+Thm)`

The `prove/1` predicate composes the loading of the definitions relating to the conjecture `Thm` (see §3.3 on the following page) with the search and execution of a depth-first plan.

`apply-ext(+ArgsList)`

The predicate `apply-ext/1` provides an interface for executing extract terms. An Oyster extract term encodes the computational content of an Oyster proof. `ArgsList` is the list of arguments required by the extracted function. For example, assuming that a synthesis proof for list concatenation has been constructed called `append`. Then using `apply-ext/1` the concatenation of the lists `[1,2,3]` and `[4,5,6]` can be achieved as follows:

```
| ?- apply_ext([[1,2,3],[4,5,6]]).
(append  [1,2,3] [4,5,6]) = [1,2,3,4,5,6]
```

3.3 The library mechanism

One of Oyster's notably lacking features is a decent library mechanism. When proving even moderately complex theorems, it becomes very painful to keep track of the dependencies between theorems, lemmas, definitions, etc. To make life with Clam a bit easier, Clam provides a simple library mechanism which is geared towards the needs of Clam. Clam distinguishes a number of *logical objects*, which play different roles in constructing proofs and proof plans.

3.3.1 Logical objects

Logical objects are divided into number of different *logical object types*. A logical object is always designated by a term `T(N)`, where `T` is the type of the object and `N` the name of the object.

Certain behaviours are associated with the loading and saving of the various kinds of logical object: the possible types of logical objects and these behaviours are described below:

`plan`: A `plan` logical object denotes a proof-plan for some theorem. These logical objects are automatically added to the Prolog environment when a proof-plan for a theorem has been found—they cannot be created by a user or be loaded from a file (this may change in future versions of Clam).

For example, when a proof-plan for a theorem called '`assp`' has been found, it can be saved with

```
lib_save(plan(assp)).
```

The purpose behind saving proof-plans is to better document and record Clam's performance for benchmarking and so on.

`thm`: The `thm` type consists of theorems, corresponding to Oyster conjectures. There is no distinction between a theorem and a conjecture as far as the library mechanism is concerned. Typically, a `thm` is loaded as a conjecture, some proof-planning or other theorem proving is carried out, the the resulting theorem is saved.

`thm` objects can be loaded and saved.

`lemma`: A `lemma` also corresponds to an Oyster theorem. However the idea of a `lemma` is that it is not a theorem which is interesting in its own right, but rather something which is only needed for technical reasons. An example of a lemma would be some boring arithmetic equality that Oyster is too brain damaged to deal with. Other theorems (of type `thm`) can also be used as lemmas by Clam, but `lemma`s should not be used for anything else, whereas `thm`s are expected to be used for other purposes as well (e.g. as input for planning tasks). Clam is expected to be able to produce proof-plans for `thm`s, whereas no such expectation exists for `lemma`s.

`synth`: A `synth` is an Oyster theorem which is only used to synthesize the definition of a particular function. In this sense, `synth`s are close to `def`s.

`synth` objects are not normally loaded directly by the user: the are automatically loaded when a `def` object of the same name is loaded. When loading `def(D)` Clam checks for the presence of `synth(D)` in the current libraries. If such an object is found, it is loaded. Notice that this dependency between `def` and `synth` objects is not reflected in the needs.pl file.

`scheme`: A `scheme` is an Oyster theorem which proves the validity of a particular non-standard induction `scheme`. Thus, a `scheme` is typically a higher order theorem. A `scheme` is expected to be proved by hand; loading a scheme via `lib-load(scheme(S))` loads `S` and attempts to translate it into the meta-level representation of induction schemes used by `scheme/3` and `scheme/5`. For example, to justify *plus* induction the following theorem would be proved:

```
phi:(pnat=>u(2))=>
  phi of 0=>
    phi of s(0)=>
      (x:pnat=>y:pnat=>phi of x=>phi of y=>phi of plus(x,y))=>
        z:pnat=>phi of z
```

See `scheme/3` and `scheme/5` for additional information.

Schemes can be loaded and saved.

`wave`: A `wave` is an Oyster theorem like a `thm` (that is: Clam is expected to be able to construct a proof-plan for it[2]), but the fact that the theorem is marked as a `wave` indicates that it can be used as a wave-rule. Such wave-rules are stored as rewrite rules (see §4.3.3 and `rewrite-rule/5`). See [6, 1] for a description of wave-rules.

Since wave-rules derive from rewrite rules, there is no sense in which the library stores wave-rules, so the idea of loading and saving them is rather anomalous. Loading `wave(W)` object causes Clam to load `thm(W)` and then *process* that `thm` object into a rewrite rule. Notice then that loading a `wave` object introduces a `thm` object (of the same name) and a collection of rewrite rules from which wave-rules may be later extracted.

Saving `wave(W)` has the dual effect: the object `thm(W)` is saved into the library as a theorem.

In the special case that the library mechanism attempts to load a collection of wave objects, described via `wave([W1,W2,...,Wn])`, each of the individual objects `wave(W1)`, through `wave(Wn)` is loaded. Clam then attempts some additional processing to extract complementary-rewrite-rules from the resulting set of rewrite rules. (See `wave/4` for more information.)

`def`: A `def` corresponds to an Oyster definition, using Oyster's `<==>` operator. Clam and Oyster have slightly different ideas about what a definition is: Oyster thinks that definitions are constructed with `<==>`, whereas Clam thinks that definitions are constructed via recursion equations, which are themselves constructed as Oyster theorems of type `eqn`. Thus, for every definition of type `def`, there will be a number of corresponding recursion equations of type `eqn`.

The library mechanism knows of this dependency and so loading and saving `def` objects causes a corresponding loading and saving of the equations associated with that definition. On loading, Clam processes the rewrite rules resulting from the `eqn`s in an attempt to extract complementary-rewrite-rules from them. See discussion above under the `wave` entry.

`eqn`: An `eqn` is an Oyster theorem which is to be interpreted as the recursion equation for a particular definition of type `def` having the same name. Equation objects are not normally loaded and saved directly: they are loaded/saved as a side-effect of loading/saving the corresponding definition.

[2]Though this is not a prerequisite of a `wave` object.

Saving an individual equation is possible. It is not possible to load an individual equation without referring to the name of the definition of which that equation is considered a definition. For example, `lib-save(def(plus1))` saves the first numbered equation making up the definition of the symbol `plus`. `lib-load(eqn(plus1))` will report an error. `lib-load(eqn(plus,plus1))` will load the equation `plus1` and associate it with the definition of `plus`.

Notice that it is a bad idea to load equations in this way since it by-passes Clam's processing for complementary sets, as shown in this example:

```
| ?- lib_load(eqn(plus,plus1)).
Loaded eqn(plus1)
Added (=) equ(pnat,left) rewrite-record for plus1
Added (=) equ(pnat,left) reduction-record for plus1
Clam WARNING: Loading a single equation will not update any
              complementary rewrite sets.
Clam WARNING: You must re-load the entire definition to build these.
```

`eqns`: This is not really a logical object but rather a notational convenience. Call it a pseudo-object. It refers to all the `eqn` logical objects collectively. That is, `eqns(D)` is much the same as `eqn(D1)`, through `eqn(Dn)`. The advantage of using `eqns(D)` is that complementary set processing is carried out on the equations.

`defeqn`: This is not really a logical object but rather a notational convenience. Call it a pseudo-object. It is only to be use in the context of a `lib-save/2`: Using `lib-create/[1;2]` it is possible to create `def` objects and their corresponding `eqn` objects and a `synth` object at the same time; `defeqn` conveniently refers to all of these as a single object for the purpose of saving them and immediately re-loading the definition (and so causing the definition, equations and synth to be processed).

`defeqn` objects cannot be loaded.

`red`: Refers to a `thm` object that has been processed into a reduction rule. Loading `red(R)` causes Clam to load `thm(R)` and then attempt to extract a reduction rule from that theorem. (In this respect is quite similar to the loading of a `wave` object.)

No warning or error is reported if Clam cannot extract a reduction rule—the `thm` object remains loaded. Saving a `red` simply saves the `thm` from which it it was extracted.

A reduction rule is a rule that has been shown to be measure decreasing according to the current registry. These rules are applied as part of the symbolic evaluation method (`sym-eval/1`) and unblocking (`unblock/3`). See §A.4 on page 135 for more information.

`redwave`: Used to refer to a reduction rule and a wave-rule simultaneously (again, a pseudo object). These can be loaded but not saved.

`mthd`: A `mthd` is a Clam method, represented either as a `method/6` clause, or as an `iterator/4` (or `iterator-lazy/4`) clause of the form `iterator(method,...,...,...)` (or `iterator-lazy(submethod,...,...,...)`).

`smthd`: A `smthd` is a Clam method, represented either as a `submethod/6` clause, or as an `iterator/4` (or `iterator-lazy/4`)

Object	*Load?*	*Save?*	*Processing*	*Comment*
plan	N	Y	None	Created by planner
thm	Y	Y	None	
lemma	Y	Y	None	Interesting only to tactics
synth	Y	Y	None	Loaded automatically with `def`
scheme	Y	Y	Induction rules	
wave	Y	y	RR, CS	based on `thm`
eqn	N/R	Y	RR, RedR, CS	based on `thm`
eqns	Y	Y	RR, RedR, CS	
defeqn	N	Y	None	Use only after `lib-create`
red	Y	Y	RedR	based on `thm`
redwave	Y	N	RR, RedR	`red` and `wave`
mthd	Y	N		
smthd	Y	N		
hint	Y	N		
trs	N	N		

Table 3.1: Summary of logical objects. Key: RR – rewrite-rule; RedR – reduction-rule; CS – complementary set

clause of the form `iterator(submethod,...,...,...)` (or `iterator-lazy(submethod,...,...,...)`).

`hint`: A `hint` is a Clam hint-method represented as a `hint/6` clause.

`trs`: `trs` is the name of a terminating rewrite system. Such a logical object is defined by a collection of rules and a collection of registries. Clam currently only supports one `trs`, defined by the rules of `reduction-rule/6` and the two registries `positive` and `negative`. (See `registry/4`.)

Table 3.3.1 shows a summary of the logical objects.

Example logical objects Below are some examples for each of the above types of logical objects.

`thm`: `x:pnat=>y:pnat=>plus(x,y)=plus(y,x) in pnat`

is a theorem of type `thm`.

`def`: `plus(x,y) <==> p_ind(x,y,[~,v,s(v)])`

is a definition of type `def`.

`eqn`: `y:pnat=>plus(0,y)=y in pnat`
`x:pnat=>y:pnat=>plus(s(x),y)=s(plus(x,y)) in pnat`

are both recursion equations of type `eqn`, corresponding to the definition of `plus/2` above.

`synth`: `x:pnat=>y:pnat=>pnat`

together with the corresponding proof which synthesizes addition would be of type `synth` (defining plus). (Notice that such a synthetic definition would still need corresponding recursion equations to be of any use during proof-plan construction.)

```
lemma: n:pnat=>
         m:pnat=>
```

```
(times(n,m)=0 in pnat=>void)=>m=0 in pnat=>void
```

is a simple theorem about arithmetic that Oyster should know about (but doesn't). It is therefore best seen as of type `lemma`, although, when we decided to build proof-plans for this statement, it could be upgraded to type `thm`.

`wave:` `a:pnat=>b:pnat=>c:pnat=> times(plus(b,c),a)=plus(times(b,a),times(c,a))in pnat,` although a `thm` in its own right, could be declared as a wave rule as well.

`red:` A measure decreasing rewrite rule.

```
x:pnat=>y:pnat=>plus(x,s(y))=s(plus(x,y)) in pnat.
```

`mthd:` Any `method/6` term described in §2.3.1 on page 42 is an example of a `mthd`. The other way of making methods is through an `iterator/4` or `iterator-lazy/4` clause:

```
iterator(method,normalize,submethods,[normal(_)]).

iterator-lazy(method,normalize,submethods,[normal(_)]).
```

`smthd:` Any `submethod/6` described in §2.3.1 on page 42 is an example of a `smthd`. The other way of making submethods is through an `iterator/4` or `iterator-lazy/4` clause:

```
iterator(submethod,ripple_out,methods,[wave(_,_)]).

iterator-lazy(submethod,ripple_out,methods,[wave(_,_)]).
```

`hint:` Any `hint/6` term described in §2.5.1 on page 79 is an example of a `hint`.

Associated with each logical object type is a functor which can be wrapped around the name of an object to indicate its type. Such expressions will be called *typed logical objects*. The name of a logical object is always an atom, except for methods, whose name is a functor specification of the form `f/n`. Thus, the expression `def(plus)` indicates that `plus` is a `definition`, and the expression `mthd(base/2)` indicates that `base` is a method of arity 2.

Dependencies between logical objects can be registered in Clam using the needs file which is always defined in the file `needs`. The predicate `needs/2` keeps track of the various dependencies between logical objects.

`needs(+Object,+Needed)`

`Object` is a typed logical object, and `Needed` is a list of typed logical objects. This indicates that Object needs all the objects listed in `Needed`. The `needs/2` clauses will be used by Clam's `lib-load/2` predicate to determine which objects should be loaded in which order. Example:

```
needs(thm(comm),        [def(times)]).
needs(def(times),       [def(plus)]).
```

states that the theorem `comm` (commutativity of multiplication) needs the definition of `times` and that the definition of `times` needs the definition of `plus`. Clam provides a database of predefined `needs/2` clauses, but this database can be altered by the user via assert/retract statements. Warning: loading a set of new `needs/2` clauses from a file will result in the built-in database being overwritten, so explicit calls to assert/retract must be used. (Alternatively, users can take a copy of the built-in database from the file `needs` in Clam's source directory, and add their own `needs/2` clauses). The database of `needs/2` clauses is order independent.

A number of dependency rules are built into Clam, so that they do not have to be stated each time:

- `needs(def(O), [eqn(O)])`. Thus, whenever a definition is loaded, the corresponding recursion equations will also be loaded.
- `needs(eqn(O), [wave(O),red(O)])`. Thus, whenever a recursion equation is loaded, the system will try to regard it as both a wave-rule and as a reduction rule.

`needed(?Needer,?Needed)`

This predicate succeeds if `Needed` is a typed logical object that is needed (directly or indirectly) by the typed logical object `Needer`, according to the `needs/2` database. This predicate can be used to interrogate the `needs/2` database and effectively provides the transitive closure of the `needs/2` predicate. It can be used both ways round, that is: to inquire which logical objects are needed by a given logical object (mode `needed(+,-)`), or to find all logical objects that need a given logical object (mode `needed(-,+)`).

Clam uses the `needs/2` dependency database to automatically load all the required logical objects in the correct order. For this purpose, it makes certain assumption about the way logical objects are stored in files.

- Every logical object `O` of type `T` is stored in a file `T/O`. Where `T` denotes a subdirectory of the current library directory. For example, the definition of the function `plus/2` (the typed logical object `def(plus)`) lives in the file `def/plus`. There are two exceptions to this rule:

  1. The simplest exception is the `wave` type. Objects of type `wave` do **not** live in a `wave`-directory. Instead they live in the `thm`-directory. (The only purpose of assigning a logical object the `wave`-type is to recognise it as a wave-rule.)
  2. The second exception applies to the `eqn` type. As explained above, for every object of type `def` (an Oyster definition), there will be a number of objects of type `eqn` (the corresponding recursion equations). Because there will in general be more than one recursion equation per definition, the equations for a `def` -object called `O` do not live in a file `eqn/O`, but instead may be found as a number of files in the `eqn` directory of the library.

     Currently, there are two filenaming conventions to indicate the numbered equations which belong to some definition:

     – files `eqn/O1`, `eqn/O2`, ... That is, numerals are concatenated to the right-hand of the name.
     – files `eqn/O.1`, `eqn/O.2`, ... That is, numerals are concatenated to the right-hand of the name separated by a period. This second format is to be preferred over the first one.

     In both cases, notice that the equations *must be consecutively numbered.*

Summarising, the file naming conventions of the library mechanism are:

- Possible types for logical objects are `thm`, `lemma`, `synth`, `scheme`, `wave`, `def`, `eqn`, `mthd` and `smthd`.
- Any object `O` of type `T` lives in a file `T/O`, except:
- An object `O` of type `wave` lives in a file `thm/O`, and
- An object `O` of type `eqn` lives in a file `eqn/O`n, with $n = 0, \ldots, 9$.

A number of predicates exists to manipulate these typed logical objects:

- `lib-create/[1;2]` allows the interactive user to create definitions and corresponding equations from the Clam command line. This is not a fully general mechanism in that there are some definitions and equations which cannot be created in this way. However, in most cases the user will be able to put it to good use.
- `lib-load/[1;2;3]` and `lib-load-dep/3` are used to load objects from files into the current Prolog environment.
- `lib-present/1` is used to interrogate the current Prolog environment about the presence of typed logical objects.
- `lib-delete/1` is used to delete typed logical objects from the current Prolog environment.
- `lib-save/[1;2]` is used to save typed logical objects from the current Prolog environment to a file.
- `lib-edit/[1;2]` is used for editing library objects.
- `lib-set/1` is used for setting some global parameters that affect the library mechanism.

These predicates will be discussed below:

`lib-create(defeqn(+O),+Dir)`

Create a `defeqn` pseudo-object (pseudo in that it is really a collection of a `def` and one or more `eqn`s, and a `synth`).

`lib-create` allows the interactive user to create a `def` and then give one or more corresponding equations (`eqn`s). The `synth` object is also created. The equations may be conditional. Once created, these definitions and equations must be saved (using `lib-save(defeqn(O))`) in order to process them ready to be used by Clam during proof-planning.

To create a definition, the user interactively provides a type for `O`: it is assumed that types are in uncurried form: however, they are automatically converted into curried form internally.

Then the user enters a number of equations describing `O`. The enumeration of the equations is terminated by the token '`eod.`', meaning "end of definition". All entry is terminated by a period '`.`'.

Equations have the following general form (note the period):

```
LHS = RHS.
```

or, if they are conditional equations,

```
COND => LHS = RHS.
```

At the end of this process `lib-create` has

- made a `def` object for `O`:

```
O(x1, ..., xn) <==> term_of(synth(O))
```

- created a `synth` object. This is a theorem of the type entered above. This theorem must be proven by the user.
- made a number of `eqn` objects, on per equation entered. Each of these is a theorem that is to be proved. Again, these proofs are left to the user.

The proofs referred to above constitute a (constructive) demonstration that there exists a total, primitive recursive function which satisfies the equations given.

Example. A definition of the function *nat_plus* is given, and the familiar equations for it are then enumerated. Proofs are left as an exercise.

```
| ?- lib_create(defeqn(nat_plus)).
Enter type for nat_plus: (pnat # pnat)=>pnat.
Enter equations for nat_plus ("eod." to finish)
nat_plus1: nat_plus(0,x) = x.
nat_plus2: nat_plus(s(x),y) = s(nat_plus(x,y)).
nat_plus3: eod.
Definition of nat_plus completed.
Use lib_save(defeqn(nat_plus)) to save and register your definition.
```

NB. Clam does not automatically save your definitions, nor does it register the equations. This means that they will be ignored by Clam during proof-planning. To register them, you must use `lib-save(defeqn(nat-plus))`, which saves the objects associated with the `defeqn` object and then immediately reloads them.

```
| ?- lib_save(defeqn(nat_plus),'lib-save').
Saved def(nat_plus)
Saved synth(nat_plus)
Saved eqn(nat_plus1)
Saved eqn(nat_plus2)
Registering these definitions...
Loaded synth(nat_plus)
Clam WARNING: Theorem nat_plus has status incomplete
Loaded eqn(nat_plus1)
Clam WARNING: Theorem nat_plus1 has status incomplete
Loaded eqn(nat_plus2)
Clam WARNING: Theorem nat_plus2 has status incomplete
Added rewrite-record for nat_plus1
Added rewrite-record for nat_plus2
Added rewrite-record for nat_plus2
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(pnat,left) reduction-record for nat_plus1
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(pnat,left) reduction-record for nat_plus2
Loaded def(nat_plus)
```

`lib-create(defeqn(+O))`

As `lib-create/2`, with `Dir` defaulting to the current directory.

`lib-delete(?T(?O))`

`T(O)` will be unified with a logical object present in the current database, and this object will be deleted from the database. The predicate tries to maintain database consistency by deleting all aspects of the specified object. For instance, if a `def` is deleted, the corresponding `eqn`s are also deleted, and if present, so is the `synth` associated with it. Equations are deleted if they are present and numbered consecutively from 1; consistency may be lost when individual `eqn`s are deleted thus destroying the consecutive numbering.

The simplest use of this predicate is to delete a single fully specified object:

```
:- lib_delete(thm(assp)).
```

However, by partially specifying `T(O)` and backtracking over `lib-delete/1`, it is possible to delete more than one object at once. For instance:

```
:- lib_delete(mthd(M)),fail.
```

will delete all methods from the system.

Deleting reduction rules. When a reduction rule is deleted, the registry is not changed even though the remaining rules may be terminating under more general registry. For example,

```
| ?- lib_delete(red(nat_plus2)).
Deleting reduction record for nat_plus2...done
Clam info:
  Some rewrite rules have been removed from the TRS; However,
Clam info:
  any possible weakenings of the registry have not been made.
```

`lib-delete`

This predicate deletes all logical objects from the current environment.

`lib-load(+T(+O),+Dir)`

This predicate will load a logical object `O` of type `T` from the corresponding file(s) in directory `Dir`, using the file-name conventions described above. Furthermore, it will also (and first) load all logical objects which are needed by `O` (directly and indirectly), according to the `needs/2` database. All these auxiliary objects are also loaded from directory `Dir`. `Dir` can be specified as a relative directory from the current directory or as an absolute pathname.

Instead of a single typed logical object, the first argument can also be a list of typed logical objects, in which case `lib-load/2` will iterate over all elements of the list. Failure to load any of these objects will prevent all subsequent objects from being loaded.

If the typed logical object `T(O)`, or any of the objects it needs directly or indirectly, are already loaded, they will not be loaded again.

For logical objects of type `def(D)` Clam loads as many *consecutively numbered* equations of the form `eqn(D`n`)` as can be found in the library (starting from $n = 1$), and these will be added to the reduction rules database.

Definitions having a type of the form $A => A = u(1)$ (that is, binary predicates) are given special attention. For such definitions Clam attempts to show that the defined symbol is transitive, by setting up and trying to prove a conjecture stating the transitivity of the symbol. Clam uses the predicate `quickly-provable/1` for these proofs. The flag `prove-trans/0` can be used to switch this facility off. If `trans-proving/0` is retracted, Clam does not attempt any automatic processing of transitivity proofs.

Definitions having a type of the form A => A => B (that is, binary functions) can also given special attention if the flag `prove-comm/0` is set. If it is, then for such definitions Clam attempts to show that the defined symbol commutative, i.e. that $f(x, y) = f(y, x)$, by setting up and trying to prove a conjecture stating the commutativity of the symbol. Clam uses the predicate `quickly-provable/1` for these proofs. If the symbol is found to be commutative, then commuted versions of all defining equations for the symbol are loaded, where the original equation is of the form$f(A, B) \Rightarrow g(f(C, D))$ and the added equation is of the form: $f(B, A) \Rightarrow g(f(D, C))$. This could be extended to equations where the LHS only contains f as a subexpression, where the LHS or RHS contains multiple occurrences of f, etc. The rewriting tactics have not been extended to cope with these commuted equations, which is why the facility is by default turned off.

Example The following examples illustrate the behaviour of `lib-load`.

```
| ?- asserta(comm_proving). % set comm_proving flag.
| ?- trace_plan(_,23).   % show what is loaded by lib-load
| ?- lib_load([def(plus),def(geq)]).
Loaded eqn(plus1)
Loaded eqn(plus2)
Added (=) equ(pnat,left) rewrite-record for plus1
Added (=) equ(pnat,left) rewrite-record for plus2
Added (=) equ(pnat,right) rewrite-record for plus2
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(pnat,left) reduction-record for plus1
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(pnat,left) reduction-record for plus2
Loaded def(plus)
Clam INFO: Definition def(plus) has the type of a binary function.
Clam INFO: Trying to show it is commutative.
Clam INFO: Proved def(plus) to be commutative.
Clam INFO: Adding commuted versions of wave rules.
Clam INFO: Note, need to add code for tactics for commuted wave rules soon.
Clam INFO: Added commuted wave rule (equ(pnat,right)) for equation plus2.
Clam INFO: Added commuted wave rule (equ(pnat,left)) for equation plus2.
Clam INFO: Added commuted wave rule (equ(pnat,left)) for equation plus1.
Loaded synth(leq)
Loaded eqn(geq1)
Loaded eqn(geq2)
Loaded eqn(geq3)
Added (=) equ(u(1),left) rewrite-record for geq1
Added (=) equ(u(1),left) rewrite-record for geq2
Added (=) equ(u(1),left) rewrite-record for geq3
Added (=) equ(u(1),right) rewrite-record for geq3
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(u(1),left) reduction-record for geq1
Clam INFO: [Extended registry positive]
```

```
Clam INFO: [Extended registry negative]
Added (=) equ(u(1),left) reduction-record for geq2
Clam INFO: [Extended registry positive]
Clam INFO: [Extended registry negative]
Added (=) equ(u(1),left) reduction-record for geq3
Loaded def(geq)
Clam INFO: Definition def(geq) has the type of a binary function.
Clam INFO: Trying to show it is commutative.
Clam INFO: Failed to prove def(geq) commutative.
Clam INFO: Definition def(geq) has the type of a transitive relation.
Clam INFO: Trying to show it is indeed transitive.
Clam INFO: Proved def(geq) to be transitive.

yes
| ?-
```

First `plus` is loaded—notice how `plus1` and `plus2` are automatically loaded when `def(plus)` is loaded, and shown to be measure decreasing reduction rules. Then `geq` is loaded.

```
?- lib_load([def(plus),thm(assp),thm(comp)])]).
```

first loaded the definition of `plus`, then loads two `thm` objects: `assp` then `comp`.

---

lib-load(+T(+O))

This is as `lib-load/2`, but the `Path` argument will be the default library path, as set by `lib-set/2`.

---

lib-load(wave(+OList))

A specialised version of the `lib-load` predicate which allows us to recognise arbitrary complementary sets of rewrites. `OList` must be a list of theorems which are be treated as a set of complementary set of rewrites. They are handed to `complementary-set/2` for processing.

---

lib-load(+Mthd(+M),+Pos,+Dir)

These forms of the `lib-load` predicate are meant only for logical objects of types `mthd` or `smthd` (in other words, `Mthd` is one of the atoms `mthd` or `smthd`, and `M` is of the form `f/n`). For these objects we want to be able to specify the relative location where they are to be inserted into the database. For this purpose the second argument of `lib-load/[2;3]` can be a position. `Pos` can be one of the following four values, each specifying a position in the database where the (sub)method `M` is to be inserted:

- `first` `M` is inserted as the first item in the database.
- `last` `M` is inserted as the last item in the database.
- `before F/N` `M` is inserted just before the method `F/N` in the database.
- `after F/N` `M` is inserted just after the method `F/N` in the database.

If the `Pos` argument is not specified (as in `lib-load(mthd(M),Dir)` or `lib-load(mthd(M))`), the default value for `Pos` is `last` unless the specified method `M` already occurs in the database, in which case the default value for `Pos` is the current position of `M`.

No more than one copy of any (sub)method ever occurs in the database. Thus, reloading a (sub)method into the database results in removing the old copy of the (sub)method. In this way, `lib-load/[1;2;3]` resembles the Prolog predicate `reconsult/1` and not the predicate `consult/1`. Because of this, the easiest way to move a method from one position in the database to a new position is to reload the method, while specifying its new position. Notice that a (sub)method is allowed to have more than one clause (such as the `wave/4` method), but the above enforces that these clauses must appear consecutively in the (sub)methods database.

Examples of the use of these predicates are:

```
:- lib_load(mthd(induction/1),after(ind_strat/1)).
:- lib_load(mthd(identity/0),first).
:- lib_load(mthd(sym_eval/1),last).
```

`lib-load(+Mthd(+M),+Pos)`

As lib-load(+Mthd(+M),+Pos,+Dir) except that `Dir` is instantiated to be the current library directory.

`lib-load-dep(+Thing,?Dep,+Dir)`

This is a version of `lib-load/2` for loading logical object `Thing` from the library in director `Dir`. `lib-load-dep/3` does not use the `needs/2` database. It automatically analyses the logical object in question (`Thing` may be any of `thm, red, wave, lemma, eqn`) and determines which definitions must be loaded for `Thing` to be loaded. This is does recursively, and hence calculates the dependancies beteween theorems and definitions etc.

`Dep` is a tree showing the objects upon which `Thing` depends.

Notice that there is no sanity check on these dependancies: if a `def` object refers to itself recursively `lib-load-dep/3` is likely to diverge. (This scenario is illegal anyway as far as Oyster is concerned.)

For example,

```
| ?- lib_load_dep(thm(rotlen),D,lib).
D = [def(rotate)-[def(app)-[],def(tl)-[],def(hd)-[]],
     def(length)-[],def(app)-[]]
yes.
```

`lib-present(?T(?O))`

`T(O)` will be unified with a typed logical object in the current environment. This can be used to test for the presence of a specified logical object, or to generate a set of logical objects from the environment on backtracking, by partially specifying `T(O)`.

NB: whilst cancellation (see `cancel-rule/2`) and equality (see `equal-rule/2`) records are not library objects, they will be displayed as such by `lib-present/1`.

`lib-present`

This predicate prints the names of all logical objects in the current environment.

NB: whilst cancellation (see `cancel-rule/2`) and equality (see `equal-rule/2`) records are not library objects, they will be displayed as such by `lib-present/0`.

`lib-save(+T(+O),+Dir)`

This predicate will save a logical object `O` of type `T` in a file in directory `Dir`, using the file-name conventions described above. Notice that it will **not** save any of the objects that are needed by `O`. The only exception to this is when saving a `def` object, when all the corresponding recursion equations will also be saved in directory `Dir`. The only two exceptions to this are when

1. saving a `def` object, when all the corresponding recursion equations will also be saved in directory `Dir`, and,

2. when saving a `defeqn` object, which saves the corresponding `def` object, and the associated equations, and the `synth` object.

3. when saving a `plan` object, the following information is recorded in the library:

   - the name of the theorem for which the plan was constructed;
   - the version number of Clam that produced the plan;
   - the Clam environment—all logical objects present at the time the proof-planning was carried out *with the exception of* `plan` *objects*.

   This information is useful when comparing proof-plans across different versions of Clam and different collections of methods. It allows a user to reproduce precisely the environment in which a plan was found. Future versions of Clam will provide support for storing multiple plans for a single theorem in the library.

As for `lib-load/2`, the first argument can also be a list of typed logical objects, in which case `lib-save/2` will iterate over all elements of the list, and `O` may also be a list of logical objects of type `T`, in which case each is saved in list order.

When saving objects of type `def(D)`, a each *consecutively numbered* theorem called Dn is saved (from $n = 1$). Compare with `lib-load/1`.

Since (sub)methods are not created on-line by Prolog or Oyster programming (unlike `def`s, `thm`s etc.), `lib-save/[1;2]` will not work for (sub)methods. However, if this is felt as a restriction it can easily be lifted.

`lib-save(+T(+O))`

As `lib-save/2`, with `Dir` defaulting to the current directory.

`lib-edit(+Mthd)`

For those users who do not use Emacs-like interfaces, this predicate allows editing of library objects from within Clam. At the moment, it only allows editing of methods. If `Mthd` is a (sub)method specification, calling this predicate will edit the specified (sub)method in the default library directory. After editing ends, `Mthd` is automatically (re)loaded into the system.

The editor that is used for the editing operation is taken from the shell environment variable VISUAL, or, if this is not set, from the shell environment variable EDITOR, or if this is not set either, will default to `vi`.

Since in SICStus Prolog it is impossible to find out the values of environment variables, the editor will always default `vi`. Of course, it is still possible to affect the value of the editor using the `lib-set/1` predicate.

`lib-edit(+Mthd,+Dir)`

As `lib-edit/1`, except that `Mthd` is not taken from the default library directory, but from directory `Dir` instead.

`lib-set(+P)`

This predicate can be used to set various parameters which affect the behaviour of Clam's library. Currently, the value of `P` can be:

`dir(+P)` This will change the value of the library search directory path to `P`. `P` is a list of directories; the special token '`*`' may appear in the list to indicate that Clam should search the system directory at that point. For example,

```
lib_set(dir(['~joseph/clam/lib','*'])).
```

allows searching of user `arthur`'s personal Clam library before the default library is searched. The default system library may be found using `lib-dir-system/1`, but this directory cannot be changed. `lib-set(dir(['*']))` is the default path setting.

Currently, local needs files are not supported, so this means that a single needs file must reflect dependencies across all libraries. This will be improved in a future release.

`sdir(+D)` This will change the value of the default library saving directory to `D`. `D` is a directory. For example,

```
lib_set(sdir('~arthur/clam/lib-new')).
```

Subsequent `lib-save`'s will use that directory by default.

`editor(+E)` This will change the value of the editor to `E`.

Part II

Programmer Manual

Chapter 4

Representations

This second part of this note is intended for readers who want to understand the inner workings of Clam so that they can change the way it works. This can vary from adding new planners to the existing set, to changing the way Clam interfaces with Oyster, to changing the internal organisation of Clam, etc. Note that method-programming is not discussed here. It was discussed in the *User Manual* (section2.1 on page 9), since Clam users should already be able to change methods. You don't have to be (or shouldn't have to be ...) a Clam programmer in order to experiment with different methods.

This *Programmer Manual* discusses

- the representation of induction schemes,
- the mechanics of constructing iterators,
- Clam's storage mechanism for definitions, theorem, lemmas, recursion equations etc, (which is not the same as Oyster's representation mechanism),
- the representation of wave-fronts,
- the representation of the methods- and submethods-databases,
- and a list of utilities to make a programmer's life easier.

We have not discussed many parts of Clam's code, and the interested reader should refer to Clam's source files for these. The ratio of comment to code is quite high at the moment (more than 1:1, better than I've ever produced before), so most of the code should be fairly understandable. The organisation of Clam's source code across the various Prolog files is explained in appendixC.1 on page 143.

4.1 Induction schemes

An important part of Clam's current ability to produce proof-plans relies on an effective representation of induction schemes. This section discusses this representation in some detail.

As discussed earlier, the `scheme/3` and `scheme/5` predicates implement the Clam-level representation and application of induction rules. Some logics (such as Oyster) require justification of non-standard (i.e., non built-in) induction schemes: lemmas justifying these inductions are stored as logical objects of type `scheme`. For each such object loaded from the library, Clam automatically creates a corresponding meta-level induction scheme. This scheme record is stored in the scheme database to be access via `scheme/3`.

Due to restrictions in the implementation, some scheme objects cannot be translated into `scheme/3` objects. This will be improved in the future.

Currently, the Clam library contains a number of different induction schemes. Here we give each of them together with the corresponding higher-order theorem from the `scheme` database which justifies it ($phi(x)$ is a schematic formula, possibly containing x).

primitive induction over `pnat`:

```
scheme([s(_)], _,
[    []          ==> phi(0),
[X:pnat, phi(X)] ==> phi(s(X))]
                 ==> phi(N:pnat)).
```

$$\frac{\vdash \phi(0) \quad x\!:\!pnat, \phi(x) \vdash \phi(s(x))}{\vdash \forall x\!:\!pnat.\phi(x)}$$

(This induction is built into Oyster, so no justification is required.)

two-step induction over `pnat` (twos):

```
scheme([s(s(_))], twos,
[                ==> phi(0),
                 ==> phi(s(0)),
[X:pnat, phi(X)] ==> phi(s(s(X)))]
                 ==> phi(N:pnat)).
```

$$\frac{\vdash \phi(0) \quad \vdash \phi(s(0)) \quad x\!:\!pnat, \phi(x) \vdash \phi(s(s(x)))}{\vdash \forall x\!:\!pnat.\phi(x)}$$

```
phi:(pnat=>u(2))=>
 phi of 0=>
  phi of s(0)=>
   (x:pnat=>phi of x=>phi of s(s(x)))=>
    z:pnat=>phi of z
```

plus induction over `pnat` (plusind):

```
scheme([plus(_,_)], plusind,
[                              ==> phi(0),
                               ==> phi(s(0)),
[X:pnat,Y:pnat,phi(X), phi(Y)] ==> phi(plus(X,Y))]
                               ==> phi(Z:pnat)).
```

$$\frac{\vdash \phi(0) \quad \vdash \phi(s(0)) \quad x\!:\!pnat, y\!:\!pnat, \phi(x), \phi(y) \vdash \phi(x+y)}{\vdash \forall x\!:\!pnat.\phi(x)}$$

```
phi:(pnat=>u(2))=>
 phi of 0=>
  phi of s(0)=>
   (x:pnat=>y:pnat=>phi of x=>phi of y=>phi of plus(x,y))=>
    z:pnat=>phi of z
```

simple prime induction over pnat (primescheme):

```
scheme([times(_,_)], primescheme,
[                                  ==> phi(0),
                                   ==> phi(s(0)),
  [P:{prime}, X:{posint}, phi(X)] ==> phi(times(P,X))]
                                   ==> phi(Z:{posint})).
```

$$\frac{\vdash \phi(0) \quad \vdash \phi(s(0)) \quad p\!:\!prime, x\!:\!posint, \phi(x) \vdash \phi(p \times x)}{\vdash \forall x\!:\!posint.\phi(x)}$$

```
phi:({posint}=>u(2))=>
 phi of s(0)=>
  (p:{prime}=>x:{posint}=>phi of x=>phi of times(p,x))=>
   z:{posint}=>phi of z
```

simple simultaneous induction on two variables over pnat (pairs):

```
scheme([s(_),s(_)], pairs,
[    [Y:pnat]                  ==> phi(0,Y),
     [X:pnat]                  ==> phi(X,0),
     [X:pnat,Y:pnat, phi(X,Y)] ==> phi(s(X),s(Y))]
                               ==> phi(X:pnat,Y:pnat)).
```

$$\frac{y\!:\!pnat \vdash \phi(0,y) \quad x\!:\!pnat \vdash \phi(x,0) \quad x\!:\!pnat, y\!:\!pnat, \phi(x,y) \vdash \phi(s(x),s(y))}{\vdash \forall x\!:\!pnat.\forall y\!:\!pnat.\phi(x,y)}$$

```
phi:(pnat=>pnat=>u(2))=>
 x:pnat=>
  y:pnat=>
   (y:pnat=>phi of 0 of y)=>
    (x:pnat=>phi of x of 0)=>
     (x:pnat=>y:pnat=>phi of x of y=>phi of s(x)of s(y))=>
      phi of x of y
```

simultaneous induction on two variables over t list and pnat (nat-list-pairs):

```
scheme([s(_),_::_], nat_list_pair,
[ [A:pnat]                           ==> phi(A,nil),
  [B:T list]                         ==> phi(0,B),
  [X:pnat, Y:T, Ys:T list, phi(X,Ys)] ==> phi(s(X),Y::Ys)]
                                     ==> phi(P:pnat, Q:T list)).
```

$$\frac{y\!:\!pnat \vdash \phi(nil,y) \quad x\!:\!list(t) \vdash \phi(x,0) \quad h\!:\!t, x\!:\!list(t), y\!:\!pnat, \phi(x,y) \vdash \phi(h :: x, s(y))}{\vdash \forall x\!:\!list(t).\forall y\!:\!pnat.\phi(x,y)}$$

```
t:u(1)=>
 phi:(t list=>t list=>u(2))=>
  x:t list=>
   y:t list=>
    (y:t list=>phi of nil of y)=>
     (x:t list=>phi of x of nil)=>
      (x:t list=>xe:t=>y:t list=>ye:t=>
          phi of x of y=>
           phi of(xe::x)of(ye::y))=>
       phi of x of y
```

primitive recursion on `t list`:

```
scheme([_::_], _,
[                             ==> phi(nil),
  [H:Type, T:Type list, phi(T)] ==> phi(H::T)]
                              ==> phi(L: Type list)).
```

$$\frac{\vdash \phi(nil) \quad h\!:\!t, x\!:\!list(t), \phi(x) \vdash \phi(h :: x)}{\vdash \forall x\!:\!list(t).\phi(x)}$$

(This induction is built into Oyster, so no justification is required.)

structural induction over trees (treeind):

```
scheme([node(_,_)], treeind,
[[Leaf:T]                          ==> phi(leaf(Leaf)),
 [L:T tree,R:T tree,phi(L),phi(R) ]==> phi(node(L,R))]
                                   ==> phi(Tree:T tree)).
```

$$\frac{y : t \vdash \phi(leaf(y)) \quad l : tree(t), r\!:\!tree(t), \phi(l), \phi(r) \vdash \phi(node(l, r))}{\vdash \forall x\!:\!tree(t).\phi(x)}$$

```
t:u(1)=>
 phi:(t tree=>u(2))=>
  (n:t=>phi of leaf(n))=>
   (l:t tree=>r:t tree=>phi of l=>phi of r=>phi of node(l,r))=>
    x:t tree=>phi of x
```

For each of these induction schemes, Clam will automatically extract a separate clause of the `scheme/3` predicate. Furthermore, an extra clause for the `induction/1` tactic is needed to apply the scheme during plan execution: this is not extracted automatically. However, such tactics are provided for the schemes shown above.

This induction tactics, when applied to a sequent, should produce exactly the output sequents as specified in the `scheme/3` predicate. Currently, an induction scheme is described by the term(s) that is(are) substituted for the induction variable(s) in the step sequent (known as the *step term(s)*) or *induction term(s)*. This assumption is somewhat problematic, since different induction schemes sometimes correspond to the same step-term (for instance the simple prime and composite prime inductions above). (See `scheme/5` for more detail on this representation.)

The representation of induction schemes should therefore be expanded with a list of the recursion variables of the scheme. This extension should distinguish the simple prime induction (with only one recursive variable) from the composite prime induction (with two recursive variables).

Thus, extending Clam to cope with more induction schemes should be fairly easy:

1. write a higher-order theorem expressing the validity of the induction, and save it into the library as scheme object.
2. Load this scheme object to allow Clam to extract a new clause for the `scheme/3` predicate.
3. write a new clause for the `induction/1` tactic.

change no. [3] should be made in the file tactics.pl and change no. [2] should result in a new file in the `scheme` subdirectory of the library directory.

For schemes for which Clam cannot extract the `scheme/3` clause, the user may choose to edit schemes.pl directly and add new clauses in the style of those above for such an induction.

4.2 Iterating methods

The general concept of iterators is discussed in section2.2.4.2 on page 38. Such an iterated construct over (sub)methods has been called a *methodical*, since it is to a method what a tactical is to a tactic. Currently, the only available methodical is the iterator (corresponding to the tactical `repeat`). However, this could be extended in the future to deal with other methodicals such as `complete/1`, `progress/1`, etc.

A rather arbitrary restriction on the construction of iterators is that the iterated (sub)methods must only produce at most one output sequent. In other words, their output-slot must be either the empty list or a list of length 1. This restriction means that we don't have to deal with branching iterations, but is a rather arbitrary and not very nice hack. This should be changed in a future version of Clam.

Other restrictions on the behaviour of iterators are more reasonable, and can be varied easily by making small changes to the code of the `iterate-methods/4` predicate. These concern the exhaustive-ness of the iterations performed by iterators, and possible permutations of iterations. For a discussion of these choices, we assume an iterator I constructed out of iterating the list of methods M_i,$i = 1, \ldots, n$. We write $M_{i_1}; \ldots; M_{i_k}$ for a sequence of k applications of these methods (i.e., an iteration of length k).

The length of an iteration: Currently, an iterator I in Clam will always produce maximally long iterations. Thus, if after applying $M_{i_1}, \ldots, M_{i_k}$, another method $M_{i_{k+1}}$ is still applicable, this method will be applied. As a result, it is guaranteed that after an application of an iterator I, none of the iterated method M_i will be applicable. This behaviour can be changed by redefining the predicates `iterate-methods/4` in the file methodical.pl. By changing the use of `orelse/2` in the postconditions-slot of the generated method into a `v/2`, the iterator will also generate subsequences of the maximally long chain, with the longest chain generated first. Thus, if the maximal chain is $M_{i_1}; \ldots; M_{i_k}$, it will generate (on backtracking) all sequences $M_{i_1}; \ldots; M_{i_j}$ for $j = k, \ldots, 1$. Another small change, namely swapping the order of the disjuncts in the postconditions-slot, would change the order in which chains are generated, and would generate the shortest chain first, with longer chains only on backtracking.

Iterations of length 1: At the moment, iterations of length 1 are not suppressed. They are simply returned as a possible application of the iterator (if no further applications are possible, see above). This is not very useful if both I and M_i are available as applicable methods, since an application of M_i and an application of I of length 1 are equivalent, thus doubling the search space for applicable methods. Thus, in general it is good programming technique to not have both M_i and I available for application at the same time. This can be achieved by making M_i a submethod, which will allow the construction of I, but will not make M_i available for application. Alternatively, the `iterate-methods/4` predicate could be changed to disallow iterations of length 1.

Permutations of iterations: Currently, no permutations of sequences of applications are generated. Thus, if I is an iterator over the methods $M_1, \ldots, M_n$, with the methods specified in this order when constructing I, then I will try to apply the M_i in ascending order. Thus, a method M_j will only be applied by I if none of the M_i,$i < j$ apply. This rule can be relaxed by changing the predicate `iterate-methods/4`: remove the use of the `thereis/1` predicate in the preconditions-slot of the generated method. This will result in all possible permutations of applicable methods being generated by I. This change is orthogonal to the removal of the `orelse/2` predicate in the postconditions (to not insist on maximally long chains of iterations). Thus, removing only the `thereis/1` from the preconditions-slot and leaving the `orelse/2` in the postconditions-slot will result in I generating all permutations of maximal length, whereas making both changes will result in I generating all permutations of all lengths.

Terminating iterations: If some of the methods can terminate, we have a choice in how to make I behave: should it prefer terminating M_i over non-terminating ones, or should it just iterate them in a fixed sequence, and stop when it happens to hit a terminating M_i, without actually gravitating towards one? The first (preferring terminating M_is) is obviously preferable, but makes I potentially more expensive, since it will first try all M_is to see if there is a terminating one, and if not, it will have to iterate over the M_is in sequence as usual. These two behaviours can be obtained by changing the order of the two conjuncts in the preconditions of the generated method: having the `thereis/1` first will allow termination but not prefer it, while having the `thereis/1` second will prefer termination at the cost of trying all methods first for termination. Currently, the second option (no preference for terminating M_i) is implemented. Of course, even with the second option, iterated methods can be ordered in the sequence so as to have the terminating ones first, but this is not possible in all cases.

4.3 Caching mechanism

Clam has its own mechanisms for internally storing logical objects such as definitions, lemmas, plans, recursion equations, etc. These representations are different from the representations Oyster uses, although for some they are ultimately based on these Oyster representations (there may be no corresponding Oyster representation, as in the case of plans, for instance). The main reason for these separate Clam representations is efficiency. This section describes the Clam representational system.

4.3.1 Theorem records

When a logical object of type `lemma`, `synth`, `thm` or `eqn` is loaded via the `lib-load/[1;2]` predicate, a `theorem` record is stored in Prolog's record database of the form:

```
record(theorem, theorem(Name, Type, Goal, Thm), Ref)
```

where `Name` is the name of the logical object, `Type` the type of the logical object (as specified in the `Type(Name)` argument to the `lib-load/[1;2]` command), and `Goal` the top-level goal of the logical object. In most cases, `Thm` will be equal to `Name`, except for recursion equations (`Type=eqn`), when `Name` will be of the form `name`n, with $n = 1, \ldots 9$, and `Thm` will be `name` (i.e., `Name` stripped of the last digit). `Name` is the name of the Oyster theorem corresponding to the logical object.

These `theorem` records can be accessed using the predicate `theorem/3`.

`theorem(?Thm, ?Goal, ?T)`

`Thm` is the name of a logical object of `T`, with `T` one of `lemma`, `synth`, `thm` or `eqn`, and `Goal` is the top-level goal of the object. Will not succeed if `Thm` unifies with the currently selected Oyster theorem. This predicate is an extension of `theorem/2`, where `T` is restricted to `thm` or `lemma`.

The reason for having these `theorem` records as an extra layer on top of the Oyster theorem representation is efficiency: It takes Oyster 130 milliseconds to select a theorem and pick up the top-level goal, whereas doing the same task using theorem records takes only 6 milliseconds.

4.3.2 Reduction records

Whenever a logical object of type `eqn` or `red` is loaded, Clam tries to add it to the terminating rewrite system. It will try to prove that the rewrite rule is measure decreasing according to RPOS. It may extend either of the two registries, should this be necessary, and will give message to that effect.

If the rule is measure decreasing, it will be stored as a `reduction record` in Prolog's record database of the form

```
record(reduction, reduction(LHS,RHS,Cond,Dir,Thm), Ref)
```

where `LHS` is the left-hand side of the reduction rule `Thm` is the name of the theorem from which this rule was derived. All universally quantified variables in `Exp` have been replaced by meta (Prolog) variables. These reduction records can be accessed with the predicate `reduction-rule/6`.

The registry may be accessed via `registry/4`.

4.3.3 Rewrite records

Whenever a logical object of type `wave` or `eqn` is loaded it is stored as a `rewrite` record. This record database is used by the dynamic wave-rule application code (see `wave/4` and `ripple/6`).

(The dynamic wave-rule parser caches wave-rules during a session. This means that a rewrite rule will not be parsed into a particular wave-rule more than once in a session.)

4.3.4 Proof-plan records

inxxproof-plan records Whenever Clam finds a proof-plan for a particular theorem `T`, the planning mechanism creates a proof-plan record of the form:

```
record(proof_plan, proof_plan(T,Plan),_).
```

where `Plan` is the proof-plan created. Only one proof-plan record is kept per theorem, and it can only be created by the planning mechanism. Proof-plans can be saved into the library in the normal way using `lib-save`.

4.3.5 Rewrite-rule records

Whenever a logical object of type `eqn` or `thm` is loaded, Clam adds a record to Prolog's record database of the form

```
record(rewrite, rewrite(L,R,C,Dir,Name), Ref)
```

where `L` and `R` are the left- and right-hand-sides of the rule, conditional upon `C`; `Dir` specifies in which direction and of what type the rewrite rule is; `Name` is the name of the corresponding theorem.

More than one such record may be added based upon each object loaded: as many rewrites as Clam can extract will be stored in separate records.

If the flag `prove-comm/0` is `true`, and Clam has determined that a function is commutative, then equations and theorems are subjected to additional processing to form commuted versions, which are asserted as rewrite-rule records. No tactics are available for this yet, so if the commuted rules are used in a proof plan, the corresponding object-level proof will fail.

4.4 Wave-fronts, holes and sinks

Here we describe the data structures we have chosen to represent wave-front and sink.

As described in §A.3.1 on page 131, wave-fronts correspond to a subtree of the term, with a subtree inside it corresponding to the wave hole(s). We implement this annotation with special function symbols whose identity is secret. Clam provides a kind of to hide from the user and programmer the internal details of this annotation representation.

The programmer/user can inspect, create and destruct annotated terms via the interface that Clam provides. These predicates are `iswf/4`, `issink/2` and `iswh/2`.

In fact, things are not secret: the actual functors that Clam are given by `wave-front-functor/1`, `wave-hole-functor/1` and `sink-functor/1`.

As described in §A.3.4 on page 134 sinks delimit term structure in an induction hypothesis which corresponds to a universally quantified variable in an induction hypothesis.

Annotated term	*Prolog*	*Portrayal*
$f(g(x), y)$	`f(g(x),y)`	`f(g(x),y)`
$f(g(\lfloor x \rfloor), y)$	`f(g('@sink@'(x)),y)`	`f(g(\x/),y)`
$\boxed{g(\underline{x})}^{\uparrow}$	`'@wave_front@'(hard,out, g('@wave_var@'(x)))`	`` ``g({x})''<out> ``

Table 4.1: Representation and portrayal of annotations. (The central column is exposing 'secret' information that cannot be trusted!)

4.4.1 Well-annotated terms

A term containing annotations must be well-annotated otherwise Clam will produce an error message when it tries to take the term apart, or apply a wave-rule, for example. It is an error to manipulate terms which are not well-annotated.

The predicate `well-annoated/1` decides well-annotation, that is, membership of the set WAT, as defined in §A.3.1 on page 131.

In practice, it can become difficult to read well-annotated terms because of the large number of wave-fronts for certain annotations. For this reason, annotated terms may be depicted in a maximally-joined form; see `maximally-joined/2`.

4.5 Induction hypotheses

Induction hypotheses are annotated in order that their role in a proof can be recorded and exploited by the preconditions of methods. These annotations also serve as a kind of 'user documentation' that can help in debugging proofs.

An induction hypothesis `IHyp` is tagged by an induction marker in the following way:

```
V:[ihmarker(Usage,Mark)|IHyps]
```

where `Usage` indicates how the induction hypotheses have been used so far, if at all. `Mark` is in place for future developments and currently is not exploited.

An induction hypothesis can be in one three states, corresponding to three distinct phases of an induction proof-plan. Notice that these phases are *particular* to the proof-plan implemented in the standard Clam setup. These three states are:

`raw=raw` the hypothesis has not been used in any way; this is the state immediately following an induction.

`notraw(Ds)` the hypothesis is being used during a phase of iterated . `Ds` is a list consisting of the following atoms:

> `left` the hypothesis has been used in a left-to-right direction during weak-fertilization;
>
> `right` the hypothesis has been used in a right-to-left direction during weak-fertilization;
>
> The first element of `Ds` describes the *nearest* (most recent) use of weak-fertilization, the last element describes the *furthest* (least-recent) use of weak-fertilization.

`used(Ds)` the hypothesis has been used, and the weak-fertilization phase completed. `Ds` may be as above, and in addition, in the case of strong fertilization, `Ds` can be the singleton `[strong]`, indicating that has taken place on that hypothesis. Alternatively, `Ds` may reflect that has taken place: this is indicated with `Ds=pw`.

The above three states are pretty-printed as `'RAW'`, `'NOTRAW'` and `'USED'(Ds)` respectively.

4.6 The (sub)methods database

This section describes the way Clam stores the representations for methods and submethods. Clam distinguishes between external and internal representations of (sub)methods. When loading a (sub)method via `lib-load(mthd(F/N),...)`, the system reads in the external format of the specified (sub)method, transforms it to the appropriate internal format, and stores this format in the internal database. All of the code that manages this process lives in the file method-db.pl.

The external format of a (sub)method can take one of the following forms:

1. `method(MethodName(...), Input, PreConds, PostConds, Output, Tactic)`:
   an explicitly specified method.

2. `iterator(method, MethodName, methods, MethodList)`:
   a method constructed by iterating other methods.

3. `iterator(method, MethodName, submethods, SubMethodList)`:
   a method constructed by iterating other submethods.

4. `submethod(SubMethodName(...), Input, PreConds, PostConds, Output, Tactic)`:
   an explicitly specified submethod.

5. `iterator(submethod, SubMethodName, methods, MethodList)`:
   a submethod constructed by iterating other methods.

6. `iterator(submethod, SubMethodName, submethods, SubMethodList)`:
   a submethod constructed by iterating other submethods.

The corresponding internal representations are:

1. `method(MethodName(...),Input,Pre,Post,Output,Tactic)`
2. `method(MethodName([..MethodCalls..]),In,Pre,Post,Out,Tactic)`
3. `method(MethodName([..SubMethodCalls..]),In,Pre,Post,Out,Tactic)`
4. `submethod(SubMethodName(...),Input,Pre,Post,Output,Tactic)`
5. `submethod(SubMethodName([..MethodCalls..]),In,Pre,Post,Out,Tactic)`
6. `submethod(SubMethodName([..SubMethodCalls..]),In,Pre,Post,Out,Tactic)`

Notice that [1]=[2]=[3] and [4]=[5]=[6], so that external `iterator/4` clauses get mapped into the same internal representation as normal methods and submethods (namely `method/6` clauses), thus giving only 2 different internal representations for 6 external representations.

Notice also that the above representations force iterated (sub)methods to be of arity 1, with the single argument representing the sequence of calls to the iterated methods.

The predicates `mthd-int/3`, `mthd-ext/3` and `ext2int/2` provide an interface to the internal and external representations of methods. If any of these two representations needs to be changed, only these predicates should suffer.

After transformation from external to internal format, the (sub)methods get stored in an internal database. Currently, this database has the form of two lists, one for methods and one for submethods, stored in Prolog's record database.

The main predicates for accessing this database are:

- `load-method/[1;2;3]` and `load-submethod/[1;2;3]` for loading a (sub)method,
- `method/6` and `submethod/6` for accessing the database,
- `delete-method/1`, `delete-submethod/1`, `delete-methods/0` and `delete-submethods/0` for removing (sub)methods from the database
- `list-methods/[0;1]` and `list-submethods/[0;1]` for listing the database

The representation of the database as a recorded list is not a particularly good choice of representation, and was mainly motivated by ease of programming; Accessing a (sub)method in this format means ploughing through the list of all (sub)methods, whereas other means of storage could exploit Prolog's indexing mechanisms in various ways. More efficient representations for a future version could be to store methods as separate items in the asserted database. If we would just store them as `method/6` clauses in the clause store, we could use the Prolog indexing to efficiently find methods given their name. Actually, since we don't often look for a method with a given name, the name would not be the best property to be used for indexing (i.e. to live in the first slot of a `method/6` clause). It would possibly be better to index on some other slot of the `method/6` clauses, such as the postconditions, which are often given as either `[]` or `[_|_]`. Disadvantages of using clauses in the the assert database instead of a list in the record database is that the assert database is a pain to handle (after all, we must be able to assert a clause in any specified position among an existing set of clauses). All this depends on how often we actually modify the (sub)method database. If it is the case (which I think it is) that we modify the database much less frequently than we access it, it might well be worth moving to a representation using the asserted clause database, doing indexing on the postconditions slot.

Chapter 5

Implementation

5.1 Induction preconditions

This section to be written.

5.2 Rippling implementation

This section to be written.

5.3 RPOS implementation

5.3.1 Overview

The code implementing the RPOS simplification ordering, and some utilities to orient equations into terminating rewrite systems, is described in this section. See §A.4 on page 135 for more general information.

The primary predicate is `prove/5` whose arguments are the RPOS registry, the term ordering problem to be determined, a proof object, and a set of atoms to be treated as variables in the ordering problem. (Recall that RPOS is lifted to variables as described in §A.4.2.2 on page 137/)

5.3.2 Registry

Most of the code implementing RPOS needs to know the current registry and so most of the predicates are parameterized by `Prec`, which is the quasi-precedence relation, and `Tau` which is the status function.

5.3.2.1 Quasi-precedence: `Prec`

`Prec` is a representation of the quasi-precedence relation $\succeq$; see §A.4.2.1 on page 135 for the definition.

Clam makes explicit the negation present in the definition of the $\succ$, the strict part of $\succeq$. `Prec` is a pair `P-I` of Prolog lists: `P` is the transitive part of the ordering, consisting of function symbols related by `>=`; `I` is the inequality part of the ordering, `=/=`. `=/=` is a symmetric, irreflexive binary relation. The partial order $\succ$ is the intersection of these two relations.

Remark 1 Will will often ignore the fact that `>=` and `=/=` are kept separate, and simply refer to the precedence.

Now there are some consistency checks to impose on the way in which `Prec` can be extended. For example, we cannot have `Prec` containing `a>=b`, `b>=c`, `a=/=b`, `a>=c` and `c>=a`; from this we can obtain $a \succ a$ which is illegal in a quasi-ordering (it must be reflexive). The predicate `consistent/2` decides that a `Prec` really is a quasi-ordering, and furthermore, that it obeys the restriction laid down in §A.4.2.1 on page 135.

5.3.2.2 Status function: `Tau`

The status function `Tau` is represented as a list of symbol/status pairs. The mapping must be total in that all function symbols in the ordering problem must be in the domain of the mapping.

The range of the mapping (say of a symbol f) consists of the following elements:

`_` Uncommitted status. The status of that function symbol is free to be instantiated during the search for a proof.

`undef` Undefined status. $\tau(f) = \odot$. The status of that function symbol is uncommitted but cannot be committed during the proof. That is a proof is in some sense independent from the status of that symbol.

`ms` Multiset. $\tau(f) = \otimes$.

`lex(D)` Lexicographic. If `D` is ground it must be one of:

`lr` Left-to-right: $\tau(f) = \oplus$.

`rl` Right-to-left: $\tau(f) = \ominus$.

If `D` is a variable, the status of f is lexicographic, but the permutation is uncommitted and may be instantiated during a proof.

5.3.3 Lifting

RPOS is defined over ground first-order terms; lifting to terms containing variables is necessary to treat rewrite systems (see above).

The implementation follows this style because it avoids the pain of worrying about variables becoming instantiated during a proof. Hence variables are simply atoms but their special status is recorded by passing them around as a parameter (called `Vars`). Any atom in `Vars` is treated as if it were a variable.

5.3.4 Ordering problems

An ordering problem is a Prolog term of the form:

`S >= T` Iff `S = T` or `S > T`.

`S = T` Iff `S` and `T` are equivalent under EPOS.

`S > T` Iff `S` is greater than `T` under EPOS.

`S < T` Iff `T > S`.

`S =< T` Iff `T >= S`.

In these ordering problems `S` and `T` must be ground Prolog terms. Atoms appearing in `Vars` indicate which of the atoms in `S` and `T` are to be considered variables by RPOS.

See the description of `prove/5`, `extend-registry-prove/4` in the *User Manual*.

Chapter 6

Programmer utilities

This section describes some of the utilities developed for use by Clam programmers. Some of these utilities are general purpose programming utilities (such as the formatted output package), others are more specific to Clam (such as the statistics and debugging packages).

6.1 Making new versions of Clam

A Makefile can be found in the make directory. This provides a mechanism for building new versions of Clam. The following targets are defined:

`make DIA=qui oyster`: Create a Quintus Prolog runnable image for Oyster.

`make DIA=sic oyster`: Create a SICStus Prolog runnable image for Oyster.

`make DIA=qui clam`: Create a Quintus Prolog runnable image for Clam with all the source code compiled.

`make DIA=sic clam`: Create a SICStus Prolog runnable image for Clam with all the source code compiled.

`make DIA=qui clamlib`: Create a runnable image with only the Quintus Prolog loaded but none of the source code.

`make DIA=sic clamlib`: Create a runnable image with only the SICStus Prolog loaded but none of the source code.

`make clean`: The dustman.

The `Makefile` knows about the location of the Oyster executable image and about the collection of source files for Clam. If either of these changes, the `Makefile` should be updated.

If any Quintus Prolog or SICStus Prolog libraries are needed, the required commands and declarations should be made in the file libs.pl, using `ensure-loaded/1` instructions. The libs.pl file is located in the relevant subdirectory of dialect-support.

Some properties of the Clam system will differ between machines. All these properties should be defined in the file sysdep.pl which is generated when Clam is compiled: currently this file contains predicates that determine paths and directories (`lib-dir/1`, `lib-dir-system/1`, `source-dir/1`, `saving-dir/1` and `clam-version/1`).

Finally, the features which direct the conditional compilation, such as `dialect/1` and `os/1` are defined in this file.

There are a couple of predicates reporting Clam version information:

`clam-version(?N)`

`N` will be unified with the current version of Clam. Current value of `N` is 2.8.4. (See also `clam-patchlevel-info/0`.)

`clam-patchlevel-info`

Prints a short summary of changes since the last patchlevel.

`file-version(?RCS)`

`RCS` is an RCS header from one of Clam's source files.

`lib-dir/1(?Path)`

`Path` is the current Clam library search path. It can be changed using `lib-set/1`. (See `lib-set/1` for an explanation of the library search path.)

`lib-sdir/1(?D)`

`D` is the current Clam saving directory. It can be changed using `lib-set/1`.

`lib-dir-system/1(?D)`

`D` is the directory under which the default Clam library is to be found. This is fixed at compile time and cannot be changed. (See also `lib-set/1` for how to change Clam directory search path.)

`lib-fname-exists/5(+P,?Dir,?D,?T,?F)`

`P` is a path (a path is a list of directories: see `lib-set/1` for further details), `Dir` is a directory in this path which contains the logical object `Type(D)`. The special directory name '`*`' may appear in the path—the default Clam directory location is searched at that point.

`source-dir(?Dir)`

`Dir` will be unified with the directory where the sources of Clam currently live in the system.

6.1.1 Make package

Just as the Unix `make` command provides a facility for incremental compilation, so the Prolog `make/[0;1]` predicate allows for incrementally reloading code.

`make +Flag`

This predicate will compare the modification date on all Prolog files loaded into the system with the time the files were loaded. If the modification time is more recent than the load time, the file will be reloaded. The meaning of "reloaded" depends on `Flag`: `make -i` will load the files interpreted (reconsult them), `make -c` will load the files compiled (recompile them), and `make -n` will only say which files will be reloaded, but not actually reload them.

Because SICStus Prolog does not (easily) allow inspection of clock time and modification time, the `make/[0;1]` predicates do not function when Clam runs under this dialect.

`make`

This is as `make -i`.

6.2 Porting code to other Prolog dialects

Clam was developed using Quintus Prolog and SICStus Prolog. The Makefile allows Clam to be built under these dialects, using code from dialect-support. Earlier versions of Clam were compiled under SWI Prolog, but that dialect is now not supported—it may be in future releases.

Three strategies have been used in trying to make Clam as portable as possible. Firstly, all code is written as much as possible in "vanilla flavour" Prolog, relying as little as possible on system dependent features, and trying to stay inside the cross section of all DEC10 based dialects. Secondly, the system and language dependent features have been localised in a small number of files. The files containing system dependent information are sysdep.pl (pathnames, version number), boot.pl (code for executing make scripts), and libs.pl (low-level code for libraries and saving prolog states).

A final mechanism in assisting with porting code is the use of conditional loading of code, described in §6.1.1.

6.3 Statistics package

The simplest way of collecting statistics on the behaviour of Clam is the predicate `runtime/[2;3]`:

`runtime(+Pred, ?Time)`

This will execute `Pred` as a Prolog predicate, and if successful, will unify `Time` with the CPU time spent while executing `Pred`, measured in milliseconds. This measurement is notoriously unreliable on Unix systems (especially when `Time` is small). Therefore, it is often better to use the predicate `runtime/3`.

`runtime(+Pred, +N, ?Time)`

This will execute `Pred` `N` times, and if successful, will unify `Time` with the average CPU time spent while executing a call to `Pred`. The larger `N` and `Time` are, the more reliable the value of `Time` will be.

More sophisticated statistics concerning the size of the planning space can be collected using the statistics package from the file stats.pl. This file needs to be loaded manually: it is not part of the standard Clam system in order to keep things small. The current version of the statistics package can collect data about the branching factor of the planning space, and about the number of nodes visited while constructing a plan. The predicate for operating the statistics package is `stats/[2;3]`. Currently, the following versions of the `stats/[2;3]` predicate exist:

`stats(branchfactor, {on,off}, +Planner/+Arity)`

If the first argument of `stats/3` is the atom `branchfactor`, then collection of the average branching factor of the planning space for `Planner/Arity` will be switched `on` or `off`, depending on the second argument. `Planner/Arity` must be the specification of the recursive predicate of a planner, for instance `dplan/3`, or `idplan/6`. Since the `stats/3` predicate works by dynamically modifying the code of the specified planning predicate, the particular predicate needs to be declared `dynamic` using the `dynamic/1` declaration in Quintus Prolog. This typically means the appropriate source file for the `Planner/Arity` predicate must be reloaded or recompiled.

`stats(branchfactor, collect, ?N)`

If the first argument of `stats/3` is the atom `branchfactor`, and the second argument is `collect`, then `N` will be unified with the average branching factor as encountered by the planners for which the `branchfactor` statistic has been switched on. It will also remove all data concerning the `branchfactor` statistics, so as to clean up for a new batch of statistics-taking. As a result, this predicate will succeed only once.

`stats(nodesvisited, {on,off})`

If the first argument of `stats/2` is the atom `nodesvisited`, then the collection of the number of nodes visited (by any planner) will be switched `on` or `off`, depending on the second argument. Notice that the `nodesvisited` statistic is not dependent on any particular planner used, contrariwise to the `branchfactor` statistics, which are collected per planner. Since the `stats/2` predicate works by dynamically modifying the code for the `applicable/4` predicate, this predicate needs to be declared `dynamic` using the `dynamic/1` declaration in Quintus Prolog. This typically means that the source file for the `applicable/4` predicate (the file applicable.pl) must be reloaded or recompiled.

`stats(nodesvisited, collect, ?N)`

If the first argument of `stats/3` is the atom `nodesvisited`, and the second argument is the atom `collect`, then `N` will be unified with the number of nodes visited by any planner since the last time the statistics were collected (or switched on). It will also remove all data concerning the `nodesvisited` statistics, so as to clean up for a new batch of statistics-taking. As a result, this predicate will succeed only once.

`stats(rules, {on,off})`

If the first argument of `stats/2` is the atom `rules`, then the counting the number of object-level Oyster rules of inference applied during execution of any tactics will be switched `on` or `off`. The problem with this is that in Quintus, the Oyster's `rule/3` needs to be dynamic. This can only be done by explicitly reloading or recompiling by hand a new version of Oyster which contains the appropriate `:- dynamic rule/3` declaration.

`stats(rules, collect [?Rules,?Wffs])`

If the first argument of `stats/2` is the atom `rules`, and the second argument is the atom `collect`, then `Rules` and `Wffs` will be unified with the number of well-formedness rules (`Wffs`) and non well-formedness rules (`Rules`) applied by Oyster since the most recent collection of the `rules` statistics, or (if not collected before), since the collection of `rules` statistics was started. Well-formedness rules are all those Oyster rules which apply to goals of the form `G in T`, with `G` not of the form `_ = _`.

6.3.1 Debugging utilities

A simple tracing package has been implemented to help debugging and using Clam. This package is described in section3.2.3 on page 90. The predicate that should be used to introduce more trace points in newly constructed code is the predicate `plantraced/2`.

`plantraced(+N, +Pred)`

`N` is compared with the current tracing level, and if this level is at least `N`, `Pred` will be executed. In order to avoid interference with the embedding code, `plantraced/2` never fails or backtracks, neither when `N` is larger then the current tracing level, nor when `Pred` fails or leaves backtrack points.

6.3.2 Benchmarking

After making changes to the code of planners, methods or tactics, we often want to test out the new version of the code on a set of theorems for which the old version was known to work. This process is made easier by the predicates `plan-all/[0;1]`, `plan-from/[1;2]`, `plan-to/[1;2]`, `prove-all/[0;1]`, `prove-from/[1;2]` and `prove-to/[1;2]`. These predicates access examples.pl file which contains clauses of the form:

```
example(Type,Thm,Status).
```

where `Type` is `arith` or `lists` and `Thm` is the name of a theorem. If `Thm` is marked as being provable if `Status` is a variable. The examples.pl resides in the default library directory and is reconsulted every time one of the above `prove-` or `plan-` predicates is invoked.

`plan-all`

---

attempts to construct plans for all theorems recorded in examples.pl.

`plan-all(?Type)`

---

attempts to construct plans for all theorems recorded in examples.pl of type `Type`.

`plan-from(?Thm)`

---

attempts to construct plans for all theorems recorded in examples.pl from `Thm`.

`plan-from(?Type,?Thm)`

---

attempts to construct plans for all theorems recorded in examples.pl of type `Type` starting with `Thm`.

`plan-to(?Thm)`

---

attempts to construct plans for all theorems recorded in examples.pl up to and including `Thm`.

`plan-to(?Type,?Thm)`

---

attempts to construct plans for all theorems recorded in examples.pl of type `Type` up to and including `Thm`.

`prove-all`

attempts to construct and execute plans for all theorems recorded in examples.pl.

`prove-all(?Type)`

attempts to construct and execute plans for all theorems recorded in examples.pl of type `Type`.

`prove-from(?Thm)`

attempts to construct and execute plans for all theorems recorded in examples.pl from `Thm`.

`prove-from(?Type,?Thm)`

attempts to construct and execute plans for all theorems recorded in examples.pl of type `Type` starting with `Thm`.

`prove-to(?Thm)`

attempts to construct and execute plans for all theorems recorded in examples.pl up to and including `Thm`.

`prove-to(?Type,?Thm)`

attempts to construct and execute plans for all theorems recorded in examples.pl of type `Type` up to and including `Thm`.

6.3.3 Pretty printing

Apart from the pretty-printer for plans, described in section3.2.1 on page 88, Clam also makes use of Prolog's `portray/1` pretty-printing hook to make output look somewhat more readable. The current uses of the `portray/1` hook are as follows:

- Terms of the form `elementary(I)`, will be printed as `elementary(...)` if `I` is bound, to suppress the long chain of elementary inference rules usually bound to `I`.
- If `M/1` is an iterated method, than terms of the Terms of the form `M(L)` will be printed as `M([...])` if `L` is bound, to suppress the long chain of method applications usually bound to `L`, with the number of dots in `...` indicating the number of iterations encoded in `L`.

6.3.4 Writef package

Rather then using the Quintus Prolog `format/[2;3]` predicate for doing formatted output, I have been using an old workhorse from the DEC10 library, the `writef/[1;2;3]` predicate:

`writef(+Format, +List)`

`writef(+Format)`

`writef(+File, +Format, +List)`

`writef(+File, +Format)`

All these predicates are documented in the file `writef.doc`.

6.4 Clam should be theory free

This section explains a particular constraint that I (Frank van Harmelen) claim Clam should always satisfy. It also shows how Clam almost satisfies this constraint, and how it can be easily fixed to completely satisfy it. I think it is useful for future programmers on Clam to be aware of these issues, which is why I include this section in the programmer's manual.

Ideally, we would like Clam (or any other theorem prover, for that matter), to be *theory free*: it shouldn't have any particular knowledge about the function and predicate symbols that appear in the theory. For instance, if we are dealing with an arithmetic theory, then the theorem prover should not be "told" that + is associative, since that would amount to cheating. Clam satisfies this theory free requirement quite well: nowhere in the code of either the planner or the methods does it know about special properties of particular function or predicate symbols of the object-level logic. (It does know about the logical constants of the object-level theory, but nobody said that theorem provers should be *logic free*. We only require them to be *theory free*).

Above, I said: Clam satisfies the theory free requirement *quite* well, but unfortunately, not completely. There are two major places where Clam violates the theory free requirement, and uses specialised knowledge about object-level function and predicate symbols:

1. In the formulation of induction schemes (in schemes.pl).
2. In the formulation of the weak fertilization method.

How does Clam violate the theory free requirement in these two places?

1. The induction schemes use specialised knowledge about object-level types and function and predicate symbols for their formulation: the `scheme/3` clauses in schemes.pl are contain them so some extent, and so does the code for `scheme/5`.
2. The weak fertilization method uses two types of specialised knowledge about a number of function and predicate symbols: first that some functions are transitive, and second that functions are symmetric or have a positive polarity under some order (see code in the file `lib/mthd/weak-fertilize`).

How can these violations be removed from Clam? Below I sketch a solution for each of the two violations. They both rely on the introduction of extra families of theorems. Just as current theorems are divided into families such as recursion equations, wave rules, etc., we introduce some more families (for 2) or use an existing family in a new way (for 1), in order to remove all occurrences of specific object-level function and predicate symbols from the code of Clam:

1. Currently, we have a family of (typically 2nd order) theorems called induction schemes. For each member of this family we also require a `scheme/3` clause which tells Clam how the induction specified in the theorem is to be done.

The correspondence between the second-order scheme lemma and the corresponding `scheme/3` clause is close; It should not be too hard to write some code that would automatically produce the code for the `scheme/3` clauses on the basis of the induction scheme theorem (especially if we would formulate the 2nd order induction theorems in a more or less standard way). Then, when the user would load an induction scheme theorem, the system would automatically produce the `scheme/3` clause, and proceed as before. This situation is analogous to the treatment of wave-rules and of recursion equations, where some internal data-structure is produced when a theorem of a particular family is loaded.

2. We should introduce two new families of theorems: Firstly, the family of theorems that show that a particular function is transitive. Members of this family would be easy to recognise automatically (they would all be of the form $f(x, y) \rightarrow f(y, z) \rightarrow f(x, z)$), and they would be used in the first few conjuncts of the preconditions of the weak fertilization method to recognise known transitive function symbols.

   Secondly, we should introduce the family of polarity theorems. Members of this family would be theorems that show that a particular function symbol is positive (or non-decreasing, or monotonic) under some appropriate orderings. Again, they would be to recognise. They would all be of the form $x_1 \preceq_1 x_2 \rightarrow f(x_1) \preceq_2 f(x_2)$ where $\preceq_1$ is a partial ordering on the domain of f and $\preceq_2$ is a partial ordering on the codomain of f. These theorems would also be used in the preconditions of the weak fertilization method, to determine the polarity of function symbols. They would effectively replace the `plrty/5` table in method-pre.pl. In both cases I would imagine that a specialised data-structure is created when the theorem is loaded (for instance, a `plrty/5` entry for polarity theorems).

Rather than actually implementing the above, I have only described how this could be done, convincing myself that, although Clam is not entirely theory free, it could be easily made to be, thus justifying the claim that it genuinely proves theorems for itself, without any prompting from the user.

The ideological position outlined above is also discussed in [31].

A border line position in this debate about "what Clam is allowed to know" are the definitions of object-level types in Oyster. Do these types fall under the category "logical constants"? Strictly speaking no (ask your local logician), and thus, Clam should not be told. However, they do seem integral part of the object-level system Clam is reasoning about, so we would except Clam to have to know. As a result, there are a few places where Clam does get information about the existence and structure of object-level types of Oyster:

1. The predicate `oyster-type/3` in util.pl enumerates Oyster types and corresponding constants and constructors.

2. The predicate `constant/2` in method-pre.pl gives more info about the recursive structure of Oyster types.

3. Four `clam-arith` clauses of `prule/2` in elementary.pl know about the structure of `pnat`.

Currently, I don't regard these three points as "cheating", or in conflict with the position outlined above (and in [31]), until somebody manages to convince me otherwise (or, plainly speaking: until somebody can show me how to get rid of these three bits of code...).

Appendix A

Rippling and Reduction

This chapter provides general background material on the two basic forms of rewriting provided by Clam.

A.1 Introduction

One of the basic logical manipulations that Clam uses is term rewriting. Where possible, Clam ensures that the rewriting is terminating, and to this end, Clam supports two different types are termination argument: *rippling* and *reduction*. Rippling is outlined in sectionA.3 on page 131 and reduction inA.4 on page 135. Both of these are based on the standard notion of rewrite rule, which is treated in sectionA.2

Notation We use the following notation when describing rewriting.

$\Rightarrow$ is Clam's meta-level implication; $\supset$ is object-level implication; $\equiv$ is object-level biimplication; $=$ is object-level equality.[1] We write $\overline{t_n}$ rather than $t_1, \ldots, t_n$, for $n \geq 0$.

A set R of rewrite rules consists of conditional rules of the form $c \Rightarrow l \rightarrow r$ where the union of the free variables in the condition c and the right-hand side r is a subset of the free variables on the left-hand side, l, and finally, l is not a variable. When the condition on a rule is vacuous that rule will be written with the condition elided, as $l \rightarrow r$.

A.2 Rewriting

A.2.1 Polarity

Clam's various rewrite rules are extracted automatically from lemmas (equalities, equivalences and implications). Since Clam rewrites both propositions and terms it is necessary to account for polarity—rewrite rules derived from implication ($\supset$) are distinguished from those derived from equality ($=$) and equivalence ($\equiv$) since they may only be used at positions of certain (logical) polarity.

The user has a fair degree of control as to exactly which lemmas are to be used as rewrites, but it is convenient to define a set of *general rewrites*, REWRITE, from which Clam's reduction and wave-rules are extracted. Any rewrite to be used as a reduction rule or wave-rule must belong to REWRITE.

[1] Equality in Oyster is typed but these types are elided in this chapter.

Definition 1 (Polarity) A proposition p appearing in the conclusion q of a sequent $\Gamma \vdash q$ has a *polarity* that is either positive (+), negative (−) or both (±). In a sequent $\Gamma \vdash G$, G has positive polarity, written G^{+}.

The complement of a polarity p is written $\bar{p}$, defined to be $\bar{+} = -$, $\bar{-} = +$ and $\bar{\pm} = \pm$.

Polarity is defined inductively over the structure of propositions: $(A^{\bar{p}} \supset B^{p})^{p}$, $(A^{p} \wedge B^{p})^{p}$, $(A^{p} \vee B^{p})^{p}$ and $(\neg A^{\bar{p}})^{p}$; the polarity of non-propositional terms is $\pm$. Propositions beneath an equivalence can have either polarity: $(A^{\pm} \equiv B^{\pm})^{p}$.

Clam uses polarity to ensure that rewriting withing propositional structure is sound: when rewriting with respect to equality (=) or biimplication ($\equiv$) the polarity of the term being rewritten is immaterial to soundness. When rewriting with respect to implications, the polarity of the term being rewritten must be either + or −, depending on the direction of the implication. These notions are made precise by *polarized TRS*s.

Definition 2 (Polarized TRS) A polarized TRS T consists of rewrite rules $l \rightarrow_{p} r$ where p is a polarity annotation, one of +, −.

Definition 3 (Polarized term rewriting) Given a polarized TRS T, a term u rewrites in one step to σr, written $u \rightarrow_{T} \sigma r$, iff there is a subterm t of u at polarity p, and one of the following two (not exclusive) conditions holds:

1. $u \rightarrow_{p} v \in T$, or,

2. p is $\pm$ and rules for both + and − are available, that is, *both* of the following hold:
$$u \rightarrow_{+} v \in T \qquad u \rightarrow_{-} v \in T$$

A.2.2 Clam's rewrite rules

In the current Clam implementation, rewrites based on equivalences are in fact stored separately rather than being stored separately as + and − parts.

Rewrites are collected from formulae as and when they are loaded into the environment. Rewrite rules are all stored in the Prolog database, and can be examined using `rewrite-rule/5`.

variables become the variables of the rewrite rule; conditions of conditional rules derived from propositional structure.

Definition 4 (REWRITE) We define the following polarized TRSs:

$$\begin{aligned}
\mathsf{REWRITE}_{+} &\subseteq \left\{ c \Rightarrow l \rightarrow_{+} r \;\middle|\; \text{any of } \begin{array}{ll} c \supset l \supset r & \\ c \supset l = r & c \supset r = l \\ c \supset l \equiv r & c \supset r \equiv l \end{array} \right\} \\
\mathsf{REWRITE}_{-} &\subseteq \left\{ c \Rightarrow l \rightarrow_{-} r \;\middle|\; \text{any of } \begin{array}{ll} c \supset r \supset l & \\ c \supset l = r & c \supset r = l \\ c \supset l \equiv r & c \supset r \equiv l \end{array} \right\} \\
\mathsf{REWRITE} &\stackrel{\text{def}}{=} \mathsf{REWRITE}_{+} \cup \mathsf{REWRITE}_{-}
\end{aligned}$$

where the set comprehension is taken over all provable universally quantified object-level formulae. (The left- and right-hand sides of the rules must satisfy the usual restrictions that l is not a variable and that the free variables in r appear free in l.)

The intention is that $\mathsf{REWRITE}_+$ are the rewrites that are sound at positions of positive polarity, $\mathsf{REWRITE}_-$ sound at negative positions, and $\mathsf{REWRITE}_\pm$ sound at either. $\mathsf{REWRITE}$ is the union of all of these. As we shall see below, both wave-rules and reduction rules are chosen from these sets.

For example, the formula $\forall x.\forall y.x \neq h \supset x \in h :: t \equiv x \in t$ yields the following rewrite-rules (using uppercase symbols to denote variables):

$$\begin{aligned} X \neq H \Rightarrow X \in H :: T &\quad \rightarrow_+ \quad X \in T \\ X \neq H \Rightarrow X \in H :: T &\quad \rightarrow_- \quad X \in T \\ X \in H :: T \equiv X \in T &\quad \rightarrow_- \quad X \neq H \end{aligned}$$

A.3 Annotations and rippling

A.3.1 Syntax of well-annotated terms

Annotations provide a mechanism for controlling the search among rewrite operations in inductive proofs. [6] gives motivation and outlines basic properties of annotated terms.

Here we give a formal definitions of: syntax of annotated terms, skeletons, erasure, annotated rewriting, well-founded measures on terms, weakening, and touch on some aspects of the implementation. Much of this material is taken from [2].

Definition 5 ($\mathsf{WAT}/\mathsf{WATS}$) We assume a set TERM of unannotated first-order terms over some signature Σ (which does not include the symbols $\{\mathsf{wfout}, \mathsf{wfin}, \mathsf{wh}, \mathsf{sink}\}$), and set of variables V.

- $\mathsf{WAT} \subset \mathsf{WATS}$.
- $u \in \mathsf{WAT}$ if $u \in \mathsf{TERM}$.
- $\mathsf{sink}(u) \in \mathsf{WATS}$ if $u \in \mathsf{TERM}$.
- $\mathsf{wfout}(f(\overline{t_n})) \in \mathsf{WAT}$ iff $f \in \Sigma$ is of arity n, and for some i, $t_i = \mathsf{wh}(s_i)$ and for each i where $t_i = \mathsf{wh}(s_i)$, $s_i \in \mathsf{WAT}$, and for each i where $t_i \neq \mathsf{wh}(s_i)$, $t_i \in \mathsf{TERM}$.
- $\mathsf{wfin}(f(\overline{t_n})) \in \mathsf{WAT}$ under similar conditions to the case above.
- $f(\overline{t_n}) \in \mathsf{WAT}$ if $f \in \Sigma$ and each $t_i \in \mathsf{TERM}$ for all i.

The set WAT and WATS differ only in that the latter contains sinks, whilst former does not.

Remark 2 A sink is not permitted to contain an annotated term in this version of Clam.

Remark 3 In the sequel, $\mathsf{wfout}(\cdot)$ will be depicted as $\boxed{\cdot}^{\uparrow}$, $\mathsf{wfin}(\cdot)$ as $\boxed{\cdot}^{\downarrow}$, $\mathsf{wh}(\cdot)$ as $\underline{\cdot}$, and $\mathsf{sink}(\cdot)$ as $\lfloor \cdot \rfloor$.

Example The following are thus annotated terms:

$$\boxed{s(\underline{plus(x, \lfloor x \rfloor)})}^{\uparrow} \qquad plus(\boxed{s(\underline{\lfloor x \rfloor})}^{\downarrow}, \lfloor x \rfloor)$$

Definition 6 ($\mathsf{skels} : \mathsf{WATS} \to 2^{\mathsf{TERM}}$) is defined recursively over well-annotated terms:

$$\begin{array}{rcl}
\mathsf{skels}(u) & = & \{u\} \qquad \text{for all } u \in V \\
\mathsf{skels}(\boxed{f(\overline{t_n})}^{\uparrow}) & = & \{s \mid \text{for some } i,\ t_i = \underline{t_i'} \wedge s \in \mathsf{skels}(t_i')\} \qquad f \in \Sigma \\
\mathsf{skels}(\boxed{f(\overline{t_n})}^{\downarrow}) & = & \{s \mid t_i = \underline{t_i'} \wedge s \in \mathsf{skels}(t_i')\} \qquad f \in \Sigma \\
\mathsf{skels}(\lfloor t \rfloor) & = & v \quad \text{where } v \text{ is a fresh variable} \\
\mathsf{skels}(f(\overline{t_n})) & = & \{f(\overline{s_n}) \mid \text{for all } i,\ s_i \in \mathsf{skels}(t_i)\}
\end{array}$$

Functions over annotated terms will generally be defined over WATS: the restriction of these functions to WAT is trivial and we shall not be formal about it.

The skeleton of a sink term is defined to be some fresh variable: it stands for a 'wild-card'.

Remark 4 When skels is singleton, we often refer to *the* skeleton.

The following notion of skeleton equality is defined over singleton skeletons. We are quite informal here.

Definition 7 (Equality of skeletons) Let a and b be WATS, such that $skels(a) = \{s_a\}$ and $skels(b) = \{s_b\}$ for some TERMs s_a and s_b. These skeletons are equal, $s_a = s_b$ iff there exists some substitution over the wild-cards appearing in s_a and s_b such that

$$s_a\sigma \text{ is identical to } s_b\sigma$$

The intention is that the skeletons two annotated terms are equal providing the only disagreement between those skeletons occurs at sink positions.

Example The skeleton of the first example above is $\{plus(x, w_1)\}$, the skeleton of the second example is $\{plus(w_2, w_3)\}$.

Notice that these skeletons are identical modulo instantiation of the 'wild-card' variables w_1, w_2 and w_3.

The erasure of a well-annotated term is computed by erase.

Definition 8 ($\mathsf{erase} : \mathsf{WATS} \to \mathsf{TERM}$) is defined recursively over well-annotated terms:

$$\begin{array}{rcl}
\mathsf{erase}(u) & = & u \quad \text{for all } u \in V \\
\mathsf{erase}(\boxed{f(\overline{t_n})}^{\uparrow}) & = & f(\overline{s_n}) \quad \text{where if } t_i = \underline{t_i'},\ s_i = erase(t_i') \text{ else } s_i = t_i \\
\mathsf{erase}(\boxed{f(\overline{t_n})}^{\uparrow}) & = & f(\overline{s_n}) \quad \text{where if } t_i = \underline{t_i'},\ s_i = erase(t_i') \text{ else } s_i = t_i \\
\mathsf{erase}(\lfloor t \rfloor) & = & t \\
\mathsf{erase}(f(\overline{t_n})) & = & f(\overline{s_n}) \quad \text{where } s_i = erase(t_i)
\end{array}$$

A.3.2 Wave-rules and rippling

Here we define $>$ which is a well-founded relation over WATS.

Definition 9 ($\succ^\star$) $\succ^\star$ is an annotated reduction ordering on WATs. See [2].

Wave-rules are rewrite rules defined over annotated terms, as follows:

Definition 10 (Wave-rule) For $c \in \mathsf{TERM}$, $l, r \in \mathsf{WAT}$, $c \Rightarrow l \stackrel{\text{rip}}{\rightarrow}_p r$ is a (polarized) wave-rule iff the following three conditions hold:

Soundness

$$c \Rightarrow erase(l) \rightarrow_p erase(r) \quad \in \quad \mathsf{REWRITE}$$

Skeleton preserving

$$skels(l) = skels(r)$$

Termination

$$l \succ^\star r$$

That is, a wave-rule is an annotated, measure-reducing, skeleton-preserving, sink-free, conditional polarized rewrite-rule.

The definition of annotated substitution can be found in [2], along with a notion of wave-rewriting. We shall make do here with an informal definition:

Definition 11 (Rippling) A term s ripples to a term t if one or more wave-rules rewrites s to t, i.e., $s \stackrel{\text{rip}^+}{\rightarrow} t$ where $\stackrel{\text{rip}^+}{\rightarrow}$ is the irreflexive transitive closure of the congruence induced by $\stackrel{\text{rip}}{\rightarrow}$.

A.3.3 Rippling in Clam

Rules which rewrite WATs are called *wave-rules*, they are computed *rewrite rules* according to the definition above (see also §4.3.3 on page 115) as needed during proof-planning. The rewrite database provides the stock of rewrite rules from which these wave-rules can be dynamically constructed—hence the term *dynamic rippling*.

As stated above, rippling is the repeated application of wave-rules: normally in Clam wave-rules are applied to an annotated term until no more wave-rules apply.

There are two basic types of rippling: static and dynamic. Static rippling is what is defined in the previous section. The distinction concerns the manner in which the various conditions on rippling are enforced. Clam supports only static rippling but we describe dynamic rippling here too for completeness.

The important point is that static and dynamic rippling are *different* rewriting relations: in fact, the dynamic rippling relation is strictly larger than the static rippling relation. [2]

[2] Interested readers may like to know that at the time of writing, λClam [25] supports dynamic rippling via embeddings [26]. But, I digress.

A.3.3.1 Static rippling

In static rippling, the annotated rewrite relation is determined by the available wave-rules: these rules may be computed in advance of being needed or they may be computed only when required.

Eager Static wave-rule parsing Here a set of rewrite rules is complied into a set of wave-rules. Each rewrite rule will be compiled into zero or more wave-rules, so as to exhaust all possible ways of extracting a wave-rule from a rewrite rule. Typically a single rewrite rule can be parsed as a wave-rule in many different ways. In many proofs, this is wasteful of both space and time since some of these wave-rules may not be used during proof search.

Lazy Static wave-rule parsing This differs from eager static rippling only in that wave-rules are not compiled in advance of their use. The idea of lazy parsing of rewrite rules is to avoid over-generation of rules that are not used during proof search.

It is important to note that both of these approaches compute the *same* ripple relation: that is, if s lazy static ripples to t then so does it eager static ripple to t, and vice versa. The difference is a practical one: lazy parsing is much more efficient.

(It is worth pointing out that some authors, notably Basin and Walsh, refer to the eager/lazy distinction as static/dynamic.)

A.3.3.2 Dynamic rippling

Dynamic rippling prime characteristic is that it is not easily characterized as a rewrite relation, and that it is certainly different from static rippling.

I will not say anything more on this for the moment.

A.3.4 Role of sinks

Sinks provide a mechanism for controlling sideways rippling, and for allowing a more liberal notion of skeleton preservation. A sink marks the occurrence of a term within the induction conclusion whose position is the same as the position of a universally quantified variable in the induction hypothesis. A precondition of a sideways ripple is that a sink occurs at or below the position to which a wave-front is moved.

Since the sink corresponds to a universal variable in the hypothesis, it is permissible, indeed, useful, for the skeleton to be corrupted below the sink position.

Example. The following wave-rule helps to illustrate the need for skeleton preservation modulo sinks. The rewrite rule

$$split\_list(A :: X, W) \rightarrow W :: split\_list(X, A),$$

cannot be applied to the annotated goal

$$\forall w.split\_list(\boxed{h :: \underline{t}}\, , \lfloor w \rfloor)$$

unless the skeleton is allowed to change at the sink position. With skeleton preservation modulo sinks, we can ripple to

$$\forall w. \boxed{w :: \underline{split\_list(t, \lfloor a \rfloor)}}\ .$$

Notice that the contents of the sink has changed, yet the skeletons are equal.

A.4 Reduction

Outside of inductive branches, where there is no requirement for skeleton preservation, a different kind of terminating rewriting may be desirable.

This section describes the termination ordering used in Clam for *reduction rules*. Reduction rules are a subset of rewrite rules (i.e., taken from the set REWRITE) which can be oriented into a terminating reduction ordering (a simplification ordering, as we shall see below). See §4.3.2 on page 115 for more information on the reduction rule database.

A.4.1 Simplification orderings

A partial ordering $>$ is a *simplification ordering* iff

$$s > t \quad \text{implies} \quad \begin{array}{l} f(\cdots s \cdots) > f(\cdots t \cdots) \\ f(\cdots t \cdots) > t \end{array}$$

for any terms s and t and function symbol f. We assume stability under substitution.

We can show that a rewrite system is terminating under a stable simplification ordering $>$ by showing that for each rule $s \rightarrow t \in R$ that $s > t$.

A.4.2 Recursive path ordering with status (RPOS)

Recursive path ordering (RPO) is a simplification ordering (due to Dershowitz) that is parametrized by a *quasi precedence* relation $\succeq$ on function symbols. We can instantiate the precedence relation to make a particular instance of the RPO, and thus obtain a simplification ordering. RPO with status (RPOS), due to Kamin and Levy, is additionally parametrized by a status function τ. Together, these two parameters are called a *registry*, denoted $\rho = \langle \succeq, \tau \rangle$. The ordering is thus written $>_\rho$ (the strict part) and $\geq_\rho$, when we want to include equivalence. (This are defined formally later.)

As is well-known in the rewriting community (the idea was pioneered by Lescanne from what I can gather; the references I used are Forgaard [16] and Steinbach [28]), registries can be computed incrementally. This means that it is not necessary to work out the registry in advance: a new reduction rule can be added to a reduction system and the registry extended as and when necessary (if this is possible) to maintain termination.

Clam's library mechanism (see §3.3 on page 92) ensures that the registry is extended (if possible) as and when new reduction rules are loaded.

A.4.2.1 Precedence, status and registry

A *precedence* $\succeq$is a transitive, irreflexive binary relation on terms. $s \sim t$ means $s \succeq t$ and $t \succeq s$. The induced partial ordering $s \succ t$ is $s \succeq t$ and $s \not\sim t$.

The following are also used:

$$s \sim t \quad \text{means} \quad s \succeq t \text{ and } t \succeq s \tag{A.1}$$

$$s \succ t \quad \text{means} \quad s \succeq t \text{ and } t \not\sim ts \tag{A.2}$$

A *status function* is a mapping from function symbols to one of two[3] status indicators: $\otimes$ or δ. These indicators are used to flag how the arguments of that function symbol are to be compared. $\otimes$ means use the multiset extension. δ means use a lexicographic extension—δ is a permutation function on the arguments.

[3]In fact this can be generalized significantly.

Additionally, we allow an undefined status, $\odot$, to allow us to express that the status of a particular function is undecided, and can be set as required. Typically we can make do with only two permutations: from left to right and from right to left. We will adopt this restriction and denote them by $\oplus$ and $\ominus$ respectively. So we think of τ mapping into $\{\otimes, \oplus, \ominus, \odot\}$.

The following functions are used in connection with status (where $\overrightarrow{t_n}$ abbreviates $t_1, \ldots, t_n$).

Definition 12 (Status functions)

$$\begin{array}{rcl} \langle \overrightarrow{t_n} \rangle^{\oplus} & = & \langle \overrightarrow{t_n} \rangle \\ \langle \overrightarrow{t_n} \rangle^{\ominus} & = & \langle t_n, \ldots, t_1 \rangle \\ \langle \overrightarrow{t_n} \rangle^{\otimes} & = & \{ \overrightarrow{t_n} \} \end{array}$$

where the set on the last line is a multiset.

Definition 13 (Consistency) A registry $\rho = \langle \succeq, \tau \rangle$ is *consistent* iff

1. $f \sim g$ implies $\tau(f) = \tau(g)$, when f and g have a defined status, and,

2. if $f \succeq g$ and $g \succeq h$ and $f \not\sim g$ or $g \not\sim h$ then $f \not\sim h$.

Definition 14 ($>_\rho$, $\geq_\rho$) Given a consistent registry $\rho = \langle \succeq, \tau \rangle$ we define $>_\rho$ over TERM by four disjunctive cases as follows.

$$s = f(\overrightarrow{s_n}) \geq_\rho t = g(\overrightarrow{t_m}) \quad \text{iff}$$

$$\begin{array}{ll} s_i \geq_\rho t & \text{for some } s_i \\ s >_\rho t_i & \text{if } f \succ g \\ s >^*_\rho t & \text{if } f \sim g \\ s >^*_\rho t & \text{if } f \succeq g \text{ and } s >_\rho t_i \text{ for all } t_i \end{array}$$

Where

$$s \geq_\rho t \quad \text{iff} \quad s >_\rho t \text{ or } s \sim_\rho t.$$

(For details of the congruence $\sim_\rho$, readers are referred to [16, 28]: roughly it is the smallest relation extending $\succeq$ to a congruence on terms, accounting for the status function.)

Note that $s >^*_\rho t$ is common to the third and forth clauses, and that $s >_\rho t_i$ is common to the second and forth.

The reader familiar with (the original) RPOS may spot that the last clause is not normally present. It is part of the extension to allow $\succeq$ to be computed incrementally. It simply says that we can proceed on the basis of partial information $f \succeq g$, rather than making a commitment to $f \sim g$ or $f \succ g$, providing that *both* of these are viable. In the case of the $\sim$ extension, we can see that we are reduced to the case dealt with by the third clause; in the case of $\succ$, the second clause. These are conjoined in the last clause.

I have introduced $>^*$ here (and defined it below) to try to make the presentation slightly clearer, since it is defined by cases, according to the status of s and t. To compare s and t according to the multiset extension, the root function symbol of s and t must have status $\otimes$. To compare lexicographically, the status must be $\oplus$ or $\ominus$. Such statuses are *compatible*.

Definition 15 ($>^*_\rho$ **(extension))** We define the multiset and lexicographic extension of $>_\rho$ (for consistent ρ) by two cases, depending on the status of the heads of the terms under comparison.

$$s = f(\overrightarrow{s_n}) >^*_\rho t = g(\overrightarrow{t_m}) \quad \text{iff}$$

$$\begin{array}{rl} \{\overrightarrow{s_n}\} >_\rho \{\overrightarrow{t_m}\} & \text{if } \tau(f) = \otimes \\ \langle\overrightarrow{s_n}\rangle^{\tau(f)} \geq_\rho \langle\overrightarrow{t_m}\rangle^{\tau(g)} & \text{if } \tau(f), \tau(g) \in \{\oplus, \ominus\} \text{ and } s >_\rho t_i \text{ for all } t_i \end{array}$$

which cover the multiset extension and lexicographic extension respectively.

Remark. In the case of the lexicographic comparison, it might seem strange to insist upon the condition $s >_\rho t_i$, namely that s is greater than all the arguments of t. This is necessary since s might otherwise be a subterm of t. For example, we do not want that $f(h(x), x) > f(x, f(h(x), x))$ simply because $h(x) > x$, in a left-to-right lexicographic comparison.

A.4.2.2 Lifting RPOS

$>_\rho$ etc. are lifted to a stable ordering on non-ground terms by treating all variables x appearing as distinguished constants that are unrelated under ρ. That is, $x \sim x$, $\tau(x) = \otimes$ and x and y are incomparable under $\succeq$, for distinct variables x and y.

A.4.3 Computing the registry dynamically

We start with some initial registry and dynamically extend it with assignments of status to function symbols where no status is present, and/or with extensions to $\succeq$. Initially, τ is set to $\odot$ for all function symbols (excepting the nullary functions which represent variables) and $\succeq$ is empty. The registry may only be extended in such a way as to preserve consistency.

The choice points in proof search arise when (i) we can choose either $f \sim g$ or $f \succ g$, or (ii) assigning some status to f and g. Clearly, there may be more than one possible extension. There is a notion of minimality here which can be used to bias the search. An extension e_1 of the registry is smaller than e_2 if e_1 can be extended further to e_2. Computing the minimal extension is expensive, so in practice, the bias is something cruder—try to extend $\succeq$ before τ.

The rules above treat the partial information case $\succeq$ as a conjunction of the two cases for $\sim$ and $\succ$. Similarly, the treatment for $\odot$ status is a conjunction of $\{\otimes, \oplus, \ominus\}$. In either case if the conjunction cannot be established, a commitment is needed for the proof to proceed.

A.4.3.1 Rewriting, polarity and reduction rules

In Clam there are two TRSs, one for positive polarity, and one for negative, with a registry for the ordering in each case. These sets are called $\mathsf{REDUCTION}_+$, $\mathsf{REDUCTION}_-$; the termination of each is justified by a registry, ρ_+ and ρ_-, respectively. These two TRSs collectively define Clam's reduction TRS, $\mathsf{REDUCTION}$.

We take subsets of $\mathsf{REWRITE}$ that satisfy the termination ordering appropriate to reduction:

Definition 16 ($\mathsf{REDUCTION}$)

$$\begin{array}{rcl} \mathsf{REDUCTION}_+ & = & \mathsf{REWRITE}_+ \cap >_+ \\ \mathsf{REDUCTION}_- & = & \mathsf{REWRITE}_- \cap >_- \\ \mathsf{REDUCTION} & = & \mathsf{REWRITE}_+ \cup \mathsf{REDUCTION}_- \end{array}$$

The polarized reduction relation is defined analogous to the polarized rewrite relation (definition3 on page 130).

Remark 5 As in the case of REWRITE, Clam does record explicitly those reduction rules which are derived from equality and biimplication.

A.5 Labelled term rewriting

The rewriting engine in Clam attempts to improve efficiency of reduction (the repeated replacement of a redex by a reduct) by manipulating labelled-terms rather than regular terms. The idea is very simple: labelled terms implement a memo-table that improves efficiency of rewriting.

Labelled terms are terms decorated by markers: each node in the term tree is marked with the token 'nf' or with *label-variables* l_1, l_2. The intended meaning is that a term whose root is labelled with the token nf is in normal form, and all a variable labelling indicates that it is not known if that term is in normal form. When the name of a label-variable doesn't matter, it will be written anonymously, -

Definition 17 (Well-labelled) A labelled term t is *well-labelled* iff for every subterm s of t that is labelled with nf, all subterms of s are labelled with nf.

Since a term is either labelled with nf or with a label-variable, it follows that for a well-labelled term t, all superterms of some subterm of t labelled with a label-variable will be labelled with a label-variable.

An example well-labelled term is $plus^{l_1}(s^{l_2}(x^{l_3}), 0^{\mathsf{nf}})$. Notice that 0 is labelled as being in normal form and the other subterms are labelled with label-variables, meaning 'not known to be in normal form'. Substitutions over labelled variables are as one expects.

Labelled terms are a convenient representation of a memo-table for computing normal forms: terms labelled by nf need not be searched (traversed) when looking for a redex.

We make some simple definitions:

Definition 18 (Unlabel) The function Unlabel from labelled terms to terms yields the term in which all labelling is deleted:

$$\mathsf{Unlabel}(f^X(t_1, \ldots, t_n)) =_{\mathrm{def}} f(\mathsf{Unlabel}(t_1), \ldots, \mathsf{Unlabel}(t_n))$$

for $0 \leq n$.

Definition 19 (V-L) The function V-L from terms to labelled terms yields the labelled term in which all nodes are labelled with a distinct label-variable.

$$\mathsf{V\text{-}L}(f(t_1, \ldots, t_n)) =_{\mathrm{def}} f^{-}(\mathsf{V\text{-}L}(t_1), \ldots, \mathsf{V\text{-}L}(t_n))$$

for $0 \leq n$.

(The term $\mathsf{V\text{-}L}(t)$ is the representation of the term t with an 'empty' memo-table.)

A.5.1 Labelled rewrite system

A labelled rewrite system is a rewrite system over labelled terms. There is no restriction that label-variables of the RHS are a subset of the label-variables on the LHS (and so labelled rewrite systems and rewrite systems are not equivalent).

To propagate labellings through reduction, we label the rules in a set R of rewrite rules to yield a labelled system LR. $l \rightarrow r \in R$ iff $l' \rightarrow r' \in LR$, where $l = \mathsf{Unlabel}(l')$ and $r = \mathsf{Unlabel}(r')$, and:

1. All distinct, non-identical subterms of l' are assigned a fresh label-variable; all occurrences of identical subterms are assigned the *same* label-variable. Notice in particular that all occurrences of some variable V in l' are assigned the same label-variable. (Typically, rules do not normally share non-variable subterms, but sometimes they do.)

2. Subterms of r identical to subterms of l are labelled similarly in r' and l'. (In particular, variables in r' are labelled with the same label-variable as similar variables in l'.)

(The current Clam implementation does not meet this specification: only *variable* subterms are considered: non-identical subterms are labelled with distinct label-variables.)

Notice in particular that l' will be labelled with a label-variable.

For example the rewrite derived from the definition of $plus$,

$$plus(s(X), Y) \rightarrow s(plus(X, Y))$$

becomes the labelled rewrite rule

$$plus^{l_3}(s^{l_4}(X^{l_1}), Y^{l_2}) \rightarrow s^{l_5}(plus^{l_6}(X^{l_1}, Y^{l_2}))$$

Notice that the sharing of label-variable for occurrences of a variable in the rule means that the labelling of the term to which a variable is instantiated is propagated (if necessary) from the redex to the reduct. The memo-table update corresponding to the reduct is computed simply by applying the labelled rewrite.

A.5.2 Labelled term rewriting (LTR)

This section is incomplete, and it is more than likely to be incorrect.

Rewriting with labelled terms is much as before, with the following additional proviso on the labelling of the term to be reduced:

Definition 20 (Labelled term rewriting) The labelled rewrite relation $\rightarrow_{LR}$ is defined over well-labelled terms as follows:

$$s^{\alpha}[u^{\beta}] \rightarrow_{LR} s^{\alpha}[b^{l_2}\sigma] \text{ iff } a^{l_1} \rightarrow b^{l_2} \in LR \text{ and } a^{l_1}\sigma = u^{\beta}\sigma$$

for some mgu σ.

Notice from the above that all redexes in LTR are labelled with a label-variable and that σ is a unifier (it may instantiate label-variables appearing in both u and a).

From the definition of well-labelled, and definition of LTR, one can see that each superterm of a redex is labelled with a label-variable (hence s itself must be labelled with a label-variable). Therefore, when the reduction is made the labelling on the rest of the term need not be altered.

A.5.3 Reduction strategy

The term traversal algorithm used by the rewriting checks to see if the term is a labelled term. If it is, and the node is labelled with nf, that term and its subterms are not searched. If a term is labelled with a variable, then it is searched. If no redex is found, the label-variable at its root is set to 'nf'.

Thus label-variable is instantiated to nf when all subterms are shown to be irreducible (the reduction mechanism must ensure that well-labelling is preserved) or by unification during rewriting.

Soundness is trivial since labelled rewriting is a restriction of normal rewriting.

Completeness The relations $\rightarrow_{LR}$ and $\rightarrow_R$ are different since labellings (even well-labellings) can be added arbitrarily. We need a more general statement.

The claim is that for the terms t and $\mathsf{V\text{-}L}(t)$ we have:

$$t \rightarrow_R^* s \text{ iff } \mathsf{V\text{-}L}(t) \rightarrow_{LR}^* s'$$

where s' is some labelling of s, and $*$ means reflexive transitive closure. We can even make a stronger statement that the reduction sequence in each case is the same. **This section is incomplete! Need to formalize and do the proofs.**

Of course soundness and completeness says nothing of efficiency, but empirical evidence suggests that LTR is faster.

Clam uses labelled term rewriting in the implementation of some of the reduction rule code. The main advantage is that for conditional rewriting it may be expensive to determine that a term is not a redex because of the effort expended in trying to establish the condition.

Appendix B

Decision procedures

Clam contains two decision procedures.

B.1 Intuitionistic propositional logic

The predicate `propositional/2` is a decider for intuitionistic propositional sequent calculus. The algorithm implemented is that due to Dyckhoff [15].

This decider builds tactics when the goal is provable which can be applied to give an object-level proof.

B.2 Presburger arithmetic

The predicate `cooper/1` is a decision procedure for Presburger integer arithmetic [23]. The algorithm implemented is that due to Cooper [14].

The argument to `cooper/1` is a sentence of Presburger arithmetic, as defined by the following grammatical elements:

- universal quantification over integers and natural numbers (`x:int=>...` and `x:pnat=>...`).
- existential quantification over integers and natural numbers (`x:int#...` and `x:pnat#...`).
- propositional connectives (`#`, `\`, `=>`, `<=>`).
- propositions: `true`, `void`.
- the following term constructors: `0`, `s`, `plus`, `times(a,b)` (where at least one of a or b is a ground term), and the integers, `-1`, `1`, `-2`, `2`, `-3`, `3`, etc.
- the following predicates: `leq`, `geq`, `greater`, `less`, `_ = _ in pnat`, `_ = _ in int`.

This grammar is hard-wired. There is an implicit assumption that this grammar agrees with the definitions of appearing in the Clam library. Even worse, the same symbols are used for both integer and natural numbers. Quantification over the natural numbers is internally translated into restricted quantification over the integers.

The algorithm does not as yet build object-level proofs.

Appendix C

Appendix

C.1 The organisation of the source files

NEWS	Information on the latest release.
README	A file containing information about the current version of Clam, (lists of things to do, known bugs), etc.
dialect-support/	Directory containing the boot-strap file subdirectories for the various dialects of Prolog supported by the Makefile. Currently only `sic`, `qui` and `swi` (SICStus Prolog, Quintus Prolog and SWI Prolog) are available. Each sub-directory contains a boot.pl, libs.pl and sysdep.pl.
info-for-users/	This directory contains various information of use to users including the Clam manual and a short introduction to theorem proving using Oyster and Clam. It also contains some auxiliary style files for use with the LaTeX=LaTeX tracing facility (see `dplanTeX[0;1]` and `idplanTeX[0;1]`).
lib/	Library directory with logical objects (constructor).
lib-buffer/	The lib-buffer provides a directory into which Clam -users can copy definitions, theorems, lemmas etc for validation by the current keeper of Clam before being installed in the official lib directory.
lib-save/	The default library directory for saving objects.
low-level-code/	Low-level support routines.
make/Makefile	Commands and dependencies for installing new versions of Clam.
make/clam.v2.8.4.DIA	Executable image for the entire Clam system.
make/clamlib.v2.8.4.DIA	Executable image with all necessary libraries pre-loaded.
make/oyster.DIA	Executable image for Oyster.
config/methods.pl	Code for loading a standard set of methods.
config/tactics.pl	Code for loading standard lemmas for arithmetic tactics.

config/hints.pl	Code for loading a standard set of hints. Note that these configuration files are not consulted — they are goal clauses.
proof-planning/applicable.pl	Code for tests for method-applicability.
proof-planning/library.pl	Code for simple library mechanism.
proof-planning/method-db.pl	Code for maintaining the (sub)method databases.
proof-planning/plan-bf.pl	Code for breadth-first planner.
proof-planning/plan-df.pl	Code for depth-first planner.
proof-planning/plan-dht.pl	Code for depth-first (hint) planner.
proof-planning/plan-gdf.pl	Code for best-first planner.
proof-planning/plan-gdht.pl	Code for best-first (hint) planner.
proof-planning/plan-id.pl	Code for iterative-deepening planner.
proof-planning/plan-idht.pl	Code for iterative-deepening (hint) planner.
proof-planning/plan-toy.pl	Code for toy versions of all planners to use for experiments in artificially constructed search spaces.
proof-planning/plan-vi.pl	Code for visual iterative deepening planner.
proof-planning/util.pl	Code for utilities: tracers, printers, etc., plus generally useful Prolog stuff.
proof-planning/stats.pl	Code for taking statistics.
meta-level-support/cancellation.pl	Code needed to deal with cancellation rules.
meta-level-support/dp	This directory contains the code for the Presburger decision procedure.
meta-level-support/elementary.pl	This file contains a decision procedure for a subset of the propositional part of Oyster logic together with additional datatype properties, e.g. uniqueness.
meta-level-support/hint-context.pl	Contexts for the hint mechanism.
meta-level-support/hint-pre.pl	Code for the hint mechanism.
meta-level-support/method-con.pl	Definition of all the connectives of the method language.
meta-level-support/method-pre.pl	Definition of all the predicates of the method language.
meta-level-support/methodical.pl	Code for constructing methodicals, currently only the iterator.
meta-level-support/propositional.pl	This file contains a decision procedure for the propositional part of Oyster logic.
meta-level-support/recursive.pl	Code for analysing recursive definitions.
meta-level-support/reduction.pl	Code for analysing reduction rules.
meta-level-support/schemes.pl	Representation of induction schemes.
meta-level-support/so.pl	RPOS and miscellaneous predicates for reduction rule machinery..
meta-level-support/tactics.pl	Code for tactics corresponding to methods.
meta-level-support/tactics-wf.pl	Code for tactics for well-formedness goals.
meta-level-support/wave-rules.pl	Code for analysing wave-rules and handling wave-fronts.
object-level-support/oyster-theory.pl	Things particular to the Oyster logic and its background theory.
writef.pl	Writef formatted output package.

C.2 Release Notes

This section describes the changes in each subsequent release of Clam, starting from release 1.1 onwards. We only list changes to the functionality of the system and leave out fixed (and introduced...) bugs.

C.2.1 CVS and Clam

From version 2.2, Clam is under the CVS revision control system. The CVS tags associated with each of the releases is show below in teletype font.

Edinburgh researchers can retreive the latest Clam version for development, by issuing the CVS command

```
cvs checkout -rHEAD clam
```

This will checkout the entire Clam system with all the revision control information ready for development work.

To retrieve Clam version 2.8.4 for compilation, but not development, use the CVS command ‘export’ rather than ‘checkout’:

```
cvs export -rCLAM_2_8_4 oyster-clam
```

This will checkout the entire Clam system without the revision control information.

Please note that the these are for very rough guidance only; Please refer to local Edinburgh documentation for notificatio of current practice etc.

C.2.2 Version 1.1, May 1989

1. The predicates `base-eq/2`, `base-eqs/1`, `step-eq/2` and `step-eqs/1` have all been renamed `base-rule/2`, `base-rules/1`, `step-rule/2` and `step-rules/1` .

2. The methods `fertilize-left/2` and `fertilize-right/2` are now submethods, disjunctively joint together in a new method `fertilize/2`.

3. There is now a version of the induction method which explicitly encodes the minimality condition for subsumption in the preconditions, instead of relying on the procedural implementation of `subsumes/2`.

4. The code for tactics has been distributed over two files: tactics-wff for all the well-formedness tactics, and tactics for all the other (“real”) tactics.

5. The code defining the method language is distributed over two files: method-con for the connectives and method-pre for the predicates. Some material is also in the oyster-theory

6. A new predicate has been introduced to specify the default value for the pathname of the library directory: `lib-dir/1`.

7. The predicate `lib-present/1` has been added to inspect the currently available set of logical objects.

8. The predicate `lib-delete/1` has been added to delete logical objects. (`plan` logical objects may not be deleted.)

9. The behaviour of `lib-load/[1;2]` has been changed. When loading in a logical object that is already present, the old predicate did nothing. The new predicate does (re)load the specified logical object, but does not reload any of the objects needed by the specified object (as recorded in the `needs/2` predicate). This is so that new versions of objects can be loaded without having to reload things that possibly did not change.

10. A mechanism for loading and deleting methods has been introduced:

- The library mechanism (`lib-load/[1;2]`, `lib-present/1` and `lib-delete/1`) has been extended to deal with arguments of the form `mthd` and `smthd` for methods and submethods. This enables individual loading of methods from files.
- `delete-methods/0`, `delete-submethods/0`, `list-methods/[0;1]` and `list-submethods/[0;1]` have been introduced (although they could have been formulated in terms of the library mechanism).

As a result of this change, methods now live in individual files instead of one big file.

11. The representation of iterating methods has been changed. It also now possible to construct every possible combination of (sub)methods iterating (sub)methods. Thus, we can construct a method that iterates submethods, a method that iterates methods, a submethod that iterates submethods and a submethod that iterates methods.
12. A new `try/1` methodical has been added to allow fail-save application of (sub)methods.
13. A new `then/2` methodical has been added to allow sequential combination of submethods.
14. The predicate `exists/1` has been renamed to `thereis/1` (to avoid a clash with a built-in NIP Prolog predicate).
15. Portability code for NIP Prolog has been added in the file `nip`.
16. The meta-linguistic connective `or/2` has been renamed `v/2` (to avoid a clash with the Oyster tactical `or/2`).
17. The scripts to construct runnable images of Clam have been upgraded to run under both NIP Prolog and Quintus Prolog.
18. A Makefile is now present to help with installing new versions.
19. The tactic `lemma/1` has been renamed `apply-lemma/1` (to avoid a name clash with the Oyster rule of inference).

C.2.3 Version 1.2, June 1989

1. The format of the `fertilize/2` method has been changed. It is now written in terms of submethods `fertilize-left/2` and `fertilize-right/2`.
2. A new `or/2` methodical has been added to allow disjunctive combination of submethods.
3. The predicate `matrix/3` has been added to the method-language predicates.
4. Zero arity versions of `lib-present`, `lib-delete`, `lib-present /0` and `lib-delete/0` have been added as utilities.
5. The zero arity version of `print-plan`, `print-plan/0` has been added as a utility.
6. Portability code for SWI Prolog has been added in the file `nip`. The main reason for porting to SWI is that it is the only Prolog with decent profiling facilities.
7. The welcome-banner printing is different, to avoid a bug in Quintus Prolog and to make it more portable.

C.2.4 Version 1.3, October 1989

1. Appendix F (describing the contents of the Clam library of definitions and theorems) has been removed from the manual, because the current library is now far too big. At the moment, it contains 44 definitions (comprising 98 recursion equations) and 95 theorems. Most of the theorems and definitions are indexed with their numbers in the Boyer and Moore book [3] in the subdirectory `BM`.

2. The predicate `canonical/2` has been introduced into the method language.

3. The predicate `recursive/3` can now deal with simultaneous recursions.

4. The predicate `recursive/4` has been introduced to deal with conditional recursion equations.

5. The predicate `universal-var/2` has been added to the method language.

6. The predicate `wave-fronts/3` provides a way of manipulating wave-fronts in formulae.

7. The predicate `wave-rule/3` provides a new representation for wave-rules which allows the implementation of the rippling-out control strategy for conditional multiple-wave-rules.

8. The fertilization method has also been substantially reorganised to deal with wave-fronts, and to distinguish between weak and strong fertilization.

9. Only one coherent version of the induction strategy (previously known as the "basic plan", remains as the method `ind-strat-I/1`). The methods `ind-strat-II/1` and `ind-strat-III/1` are now obsolete.

10. We now have a method for doing motivated casesplits in proofs (based on the notion of complementary sets of preconditions).

11. `listof/3` (an amalgam of the Prolog predicates `setof/3` and `findall/3`) has been added to the method language.

12. All methods have been reformulated so that they can now deal with explicitly quantified formula as well as with skolemised variables.

13. A section describing wave-front representation has been added to the manual.

14. Tracing output for planners in general, and for the depth-first planner in particular, has been improved.

15. The rewrite tactics have substantially changed. The functionality of the available rewrite operations should have increased, but I'm not sure how "upwards compatible" each of the individual predicates is.

16. Some new pretty-print predicates are available:
    - `print-plan/0` which pretty-prints the plan below the current sequent in the usual format.
    - `snap/[0;1]` which form a compromise between the very short `print-plan/0` and the still rather verbose `snapshot/[0;1]` provided by Oyster.

17. The default tracing level is now set to 20 rather than 0.

18. For those not using an Emacs interface, it is now possible to edit (sub)methods from within clam, using the predicates `lib-edit/[1;2]`.

19. Some global parameters of the system can now be set using the predicate `lib-set/1`.

20. A new `make/[0;1]` predicate is now available for incrementally reloading changed source files.

21. Iterated methods are now pretty printed differently, so that the printed form indicates the length of the iteration.

22. The tautology checker has been jazzed up to make it deal with a bit more than just propositional tautologies (however, it remains a decidable predicate free of search).

C.2.5 Version 1.4, December 1989

1. Induction schemes can now have more than one step case, although our way of indexing induction schemes (relying on a single induction term to identify a scheme) should also be upgraded in the future.

2. A new predicate `object-level-term/1` has been added to the method language.

3. Clam knows about the polarity of certain object-level function symbols. This is a temporary fix to allow the implementation of a more general version of weak fertilization, and should eventually be replaced by a theory free solution, described in6.4 on page 127. A `polarity/5` predicate has been added to the method language to make this knowledge available inside methods.

4. Base- and step-rules are now stored with universally quantified variables replaced by meta-(Prolog) variables, allowing faster checks for applicability.

5. A new class of theorems, so called reduction rules, have been implemented to improve the behaviour of and the story behind symbolic evaluation.

6. Clam now also runs under SICStus Prolog

7. Path expression (position specifiers, tree coordinates) for specifying positions in formulae are now transparent to wave-front annotations.

8. The predicate `canonical/2` has been renamed `constant/2` to avoid name clashes.

9. Side-ways wave-rules (transverse wave-rules) have been implemented.

10. The `generalise/2` method has been generalised.

11. A more general version of weak fertilization has been implemented.

12. A predicate `source-dir/1` names the source directory for Clam (useful for auto-loading of sources etc).

13. A new statistics facility allows counting of number of inference rules applied at the Oyster object-level during plan execution.

14. Geraint's visual version of the iterative deepening planner has been incorporated.

15. wave-fronts can now be properly joined and split as and when needed.
16. The `wfftac` has been jazzed up (once more) to deal with wff goals of functions.
17. Structural induction over trees has been added.
18. The preconditions of the `ind-strat-I/1` method now explicitly call upon the preconditions of the `induction/2` method, rather than repeating them verbatim.
19. The manual now has seperate indexes for keywords and for predicates.

C.2.6 Version 2.1, November 1993

1. The `induction/2` method has under gone significant modifications. The main change is the use of a heuristic scoring mechanism to rank induction choices.
2. `scheme/5` has been extended to allow for induction over more than one variable simultaneously. This is not, however, a general mechanism for supporting simultaneous induction.
3. `base/2` and `step/2` methods have been replaced by `eval-def/2`. Consequently, methods `base-rule/2`, `base-rules/1`, `step-rule/2` and `step-rules/1` have been removed.
4. A mechanism for dealing with complementary sets of rewrites has been incorporated. As a consequence new database records have been introduced to record complementary rewrites and condition sets.
5. Induction hypotheses are now annotated to indicate there status within step-case proofs.
6. Rippling is implemented as a single method `ripple/1` which iterates over the submethods `wave/4`, `casesplit/1` and `unblock/3`.
7. The `eval-def/2` and `wave/4` methods now include a polarity check.
8. The submethod `unblock/3` has been introduced to support a variety of meta- and object-level rewriting with the aim of facilitating further wave-rule applications.
9. The wave-rule parser has been generalised to allow for the full generality of rippling [6]. This has led to a new wave-rule representation. The predicate `wave-rule/1` is provided for pretty printing wave-rules.
10. The predicate `wave-rule/1` provides a means of pretty printing wave-rules.
11. The meta-level annotations (wave-fronts and sinks) have been brought into line with the literature [6].
12. strong fertilization and weak fertilization have been packaged up within a new method called `fertilize/2`. Weak fertilization now includes post-fertilization rippling as described in [6].
13. Existential rippling [6] has been implemented and consequently the `existential/2` method has been eliminated. A submethod has been introduced called `existential/2` which is invoked within `sym-eval/1` to deal with synthesis theorems. Eventually this will be replaced by an existential version of `eval-def/2`.

14. An additional argument has been added to the `wave/3` method. This argument is for the substitutions generated by existential rippling.

15. `base-case/1` and `step-case/1` methods have been introduced.

16. The `normalize/1` method is not loaded by default but is required for certain theorems in the corpus.

17. The methods language has been extended significantly.

18. A new set of benchmarking predicates have been incorporated (`plan-` and `prove-`). These are built on top of the existing benchmarking machinery. Instead of accessing the needs.pl file, these predicates access the examples.pl file which provides clearer documentation of the current corpus.

19. `ind-strat/1` replaces `ind-strat-I/1` and it can be applied as both a terminating and a non-terminating method.

20. `ind-strat-II/1`, and `induction-min/2` have both been removed.

21. `tautology/[0;1;2]` has been renamed `elementary/[0;1;2]`.

22. `wfftacs` has been strengthend.

23. Two new method iterators have been introduced: `repeat/7` and `iterate/5`.

24. The library has been restructured to reflect the different kinds of logical objects which inhabit it.

25. A hint mechanism has been introduced.

26. The needs.pl file is reconsulted when Clam is invoked.

27. A tutorial guide to Clam has been added to a subdirectory called info-for-users.

28. Due to problems with the dynamic database Clam is incompatible with Quintus version 3.0.

29. `apply-ext/1` provides an interface to the Oyster extraction mechanism making it easier to execute Oyster programs.

C.2.7 Version 2.2, August 1994 (`CLAM_2_2_0`)

1. Correction to the removal of redundant wave-front annotations after an application of the step-case method/submethod.

2. Generalisation of the condition-set record structure.

3. Modification of the casesplit method/submethod to reflect the generalisation of the condition-set record structure.

4. New `make/` directory organisation, and some changes to the organization of the source files:

   - there is now a `config/` directory, which contains files `hints.pl`, `methods.pl`, `tactics.pl`. These files are used to initialize Clam. These files contain Prolog goals (they are not consulted).
   - `make/` directory is quite different. All of the various driver files have been merged into a single file, `makeclam.pl`; the C pre-processor is used to generate a particular driver each time.

5. The behaviour of the Quintus, Sicstus and SWI versions is much closer. `clamlib` no longer loads the `needs.pl` file, this is done *only* by Clam proper (and is done by all Prolog versions).

6. The symbol CPP in `make/Makefile` should point to the C pre-processor. Normally this is `/usr/lib/cpp`.

7. The `CLAMSRC` symbol in `make/Makefile` should be set as normal, but the default is to compute it based on the current working directory. Thus the only thing that may require editing in that file is the location of Oyster: in the standard Oyster-Clam distribution this is not necessary.

8. The predicate `maplist` (in all arities) has been changed to `map-list`, to avoid a name clash with users wishing to use the Quintus map_list library.

C.2.8 Version 2.3 patchlevel 5, 6 May 1995

First version with dynamic wave-rule parsing.

1. Totally new induction preconditions.

2. New step-case, ripple and wave submethods to deal with dynamic rippling.

3. New rewrite database for dynamic wave-rule parsing.

4. New conditional machinery.

5. New complementary wave-rule submethod.

6. Less dependancy on the old wave-rule parsing code; I think all that requires this now is reduction rule stuff.

7. New object-level-support directory for things specific to Oyster and background theory. (This change is transparent to the user.)

C.2.9 Version 2.3 patchlevel 6, 18 July 1995 `CLAM_2_3_6`

1. Bug in existential smthd fixed.

2. `red(plus1right)` and `red(plus2right)` removed from needs.pl for `thm(binom_one)`; `red(times1right)` removed from `thm(evenm)`.

3. Induction method preconditions now allow holes in induction term wave-fronts to be subterms other than variables. For example, $\boxed{h_1 :: \underline{h_2 :: t}}^{\uparrow}$ was not possible previously, but now is.

4. Removed limit of 20 equations per definition. All equations of the form $name_N$ are loaded from when `def(name)` is loaded, starting with $N = 1$, $N = 2$ and so on. $name_{N+1}$ is loaded only when $name_N$ is present, hence: IMPORTANT: All equations must be *consecutively numbered.*

5. Added biconditional operator `<=>`. Tactics `intro_iff` and `elim_iff`; config/tactics.pl does `lib-load(def(iff))` (operator declaration was added to Oyster by Ian Green on 6 June 1995).

6. `clam-patchlevel-info/0` command added for Clam patchlevel information.

7. Only need for old wave-rule parsing code is to parse reduction rules.

8. Speeded up loading of rewrite rules.

9. `lib-create/[1;2]` added for simple interactive creation of `def`, `eqn` and `synth` objects.

10. Bugs in `lib-save/[1;2]` fixed; (`lib-save(def(0))` now saves equations associated with `def(0)` as intended.

11. Manual source split into more managable parts.

C.2.10 Version 2.4 patchlevel 0, 3 October 1995 CLAM_2_4_0

The version number was increased for the following reasons:

- The arity of the induction method and submethod has been changed from 2 to 1. This is to accommodate the revised induction scheme representation. See `induction/1` and `scheme/[3;5]`.
- The scheme database has been completely rewritten; it should now be easier to add new induction schemes.

Other less significant changes are:

1. Rewrite rules may have multiple conditions.

2. The library mechanism now operates with a list of directories (a *path*) which is searched (in order) for library items. For example,

   ```
   lib_set(dir(['~img/sys/clam/lib','*']))
   ```

   allows searching of user img's personal Clam library before the default library (indicated by the special token '`*`') is searched. The default system library may be found using `lib_dir_system(D)`, but this cannot be changed. `lib_set(dir(['*']))` is the default path setting. Currently, local needs files are not supported, so this means that the single needs file must reflect dependancies across all libraries. The saving directory, `lib_set(sdir(.))` has not been changed. (The predicate `lib-fname-exists/5` may be used to search paths.)

3. `lib-sdir/1` added (same as `saving-dir/1`, which is undocumented).

4. Tricky problem in weak-fertilization tactic has been fixed. The problem was a mis-alignment of variable names caused when the weak-fertilization is a right-to-left rewrite where the LHS has more variables than the RHS. These unbound variables were 'arbitrarily' instantiated by the tactic, whilst the method chooses the (unique) instantiation suggested by the skeleton.

5. `idplan-max/[1;2]` added to impose a maximum depth on the DFID planner. (`idplan-max` is not really suggestive of iterative deepening since it does not increase any search depth iteratively; however, the code is from plan-id.pl, so it was named that way for uniformity.)

6. Revised benchmarking code which parameterizes the benchmark by the planner: e.g., `plan_from(idplan_max(10),comm)` will use the planner `idplan_max` (with a search depth bound of 20) for entries in the corpus from (and including) `comm`. Benchmarking code automatically saves successful plan construction using `lib_save(plan(...))`. The library into which plans are saved defaults to the standard library.

7. New logical object called 'plan' has been added to explicitly record the proof-plan associated with a particular theorem. This can be saved into the library via `lib-save/1`: the name of the theorem, the raw proof-plan, the Clam enviroment (type of planner used, Clam version number methods, submethods, rewrites etc., in effect during plan construction) is saved into the library.

8. Totally new implementation of the scheme database. This is almost plug-in-compatible with the old database (which has been removed), but it is much easier to add induction schemes. Difference matching is used to add annotation.

9. Complementary sets are computed and stored at load-time. Access is via `complementary-set/1`. `complementary-set-dynamic/1` is available for run-time construction of complementary sets, should that be needed.

10. Library mechanism supports loading of multiple things in a single call to `lib-load/1`: for example, `lib_load(scheme([pairs,plusind]))`. If one of the objects in the list fails to load the Clam continues trying to load subsequent objects. A warning message is printed in this case.

11. The idea of 'induction scheme' is less ambiguous: the induction (sub)method now has a singe argument which reflects this important change.

    A 'scheme' now makes explicit the connection between a variable and the induction term which replaces it in an induction conclusion. E.g., `[x:pnat-s(v0), y:pnat list-h::t]` means `nat_list_pair` induction.

12. The induction tactic now avoids a problem encountered when renaming of variables in a goal was required to avoid capture. In some cases renaming of these bound variables in the goal is needed to avoid capture of variables present in the induction scheme lemma. Fix is to rename all variables in the scheme lemma apart from all variables (free and bound) in the hypotheses and goal.

C.2.11 Version 2.5 patchlevel 0, 21 June 1996 `CLAM_2_5_0`

This version is based on Clam 2.4, but differs in a few important ways which are discussed below. As an end user, the only major difference is the interface to `eval-def/2` rules, which no longer exists: it is replaced by reduction rules. Comparing the `eval-def/2` method with the old one will illustrate the change. See "Other important changes" below.

Reduction rules & symbolic evaluation

Reduction rules were not available in Clam 2.4, but, due to popular demand, they are back. The reduction rule machinery has been generalized and is used for any (unannotated) rewriting required to be terminating. §A.4 on page 135 describes reduction rules formally. Clam tries to add the following objects to the TRS when they are loaded via the library mechanism:

- `eqn`'s (loaded automatically as part of a definition)
- `red`'s (`thm`'s that the user explicitly wants to use as a reduction rule)

Notice that there is now no distinction between a rewrite which was loaded as an `eqn` and one loaded as a `red`: they are all added to the same database. That database is accessed via `reduction-rule/6`, and the parameters are in `registry/4` (see reduction.pl for more information). The methods for symbolic evaluation have been changed to use reduction rules.

Labelled term rewriting

This is an improvement to the speed with which terms are normalized by the repeated application of rewrite rules. It is only implemented for unannotated terms at the moment, via the predicates `nf/2` and pnf-plus/4: see reduction.pl again.

The `sym-eval/1` method uses `nf-plus/4` so that symbolic evaluation is faster. This is done via a new method called mnormalize-term/1, used in place of the standard `reduction/2` methods. The `reduction/2` methods are still available. (`extending-registry/0` is a flag that determines if the registry is to be dynamically extended: it is set to false by default.)

Other important changes

The first seven of these are incompatibilities with Clam 2.4

1. Deleting a wave or a red will not remove the associated thm (nor anything else).

2. `lib-load(wave(t))` no longer does `lib-load(red(t))`, and vice-versa. This allows more control over rewriting. Use `needs/2` mechanism to enforce this if required.

3. Cancellation rules are no longer used, although they are still generated. This might have repercussions for `ripple-and-cancel/1`.

4. Equality rules no longer exist. They are superseded by reduction rules. In particular, equivalences are now handled by the reduction rule and wave-rule machinery (see below).

5. Since all `eqn`'s are made into reduction rules, there is no longer any need for `func-defeqn/3`: it is superseded by `reduction-rule/6` (an example of this is in the `eval-def/2` method).

6. The definitions of `leq`, `geq`, `greater` and `less` have been revised (they were not all definitions before). A result of these new definitions is the need for `leqzero`, `geqzero`, `lesszero` and `greaterzero` to be added to the library as theorems, and, for certain theorems, for these to be loaded as reduction rules.

7. `step-case/1` preconditions now insist that the goal contains annotation.

8. There is a new type of library object called `trs`; currently there is only one, called `default`, referring to the combination of positive and negative polarites. Currently there is no way of saving these objects to the library, nor of having more than one. `lib-delete(trs(default))` will empty the reduction rule database, and both ordering parameters.

9. Support is provided for casesplits during symbolic evaluation, although this is not loaded by default. The s/methods `base-case-cs/1` and `sym-eval-cs/1` allow branching.

10. There was a restriction in Clam 2.4 that the left-hand-side of all rewrites (`waves` and `reds`) be non-atomic. This has been dropped (it was a bug).

11. When tracing at level 40, all the reduction rules and rewrite rules are displayed as they are generated.

12. Full support for `<=>` rewriting.

C.2.12 Version 2.6 patchlevel 3, 1 October 1997 CLAM_2_6_3

Clam 2.6, patchlevel 3

1. previous releases of Clam 2.6 did not have branching in base-case and sym-eval, as documented in the release notes of Clam 2.6.0. This has now been fixed.
2. Added decision procedure for Presburger arithmetic
3. Dropped distinction bewteen methods and submethods inside the library. Both are now stored in the library as 'methods'. Such objects can be loaded either as `mthd` or as `smthd`, as normal. This simplifies the maintenance of methods and submethods which are identical but for some irrelevant syntax.
4. improved control over portrayal of terms.

C.2.13 Version 2.7 patchlevel 0 CLAM_2_7_0

1. Automatic parsing of `scheme/[3;5]` logical objects.
2. Simultaneous inductions correctly treated by `induction/1` heuristics.
3. Faster simplification ordering for reduction rules.
4. Library mechanism less verbose. More flexible loading of logical objects.
5. Induction analysis and casesplit analysis uniformly treated.
6. New propositional decider.
7. Methods `elementary/1` and `propositional/1` treat annotations uniformly.
8. Type guessing improved.
9. Compiles under SICStus Prolog version 3 patchlevel 5, and under Quintus Prolog version 3.
10. Annotations abstracted into new file annotations.pl.
11. Socket support (under SICStus) for inter-process communication.
12. Manual up-to-date (chapter on background material is incomplete).
13. "Klutz" user guide for basic introduction to Clam in distribution.

C.2.14 Version 2.7 patchlevel 1 CLAM_2_7_1

This release has not been checked extensively; it performs miserably on transitivity proofs due to limitations in the induction selection heuristics.

1. Support for SWI Prolog. This is known to run on at least one Linux machine, but is not widely used at Edinburgh.
2. Verbosity is decreased by default. Most non-essential messages are only shown if the tracing level is greater than 22.
3. Library now supports equations having a filename of the form `root.1`, `root.2`, ..., `root.N` each of the N equations defining root. The old style in which the separator is empty (`root1`, `root2`, ..., `rootN`) is still supported.

4. Clam attempts to show that binary relations are transitive. The switch `trans-proving/0` (default `true`; see config/tactics.pl) controls this feature.

   The predicate `is-transitive/2` does these proofs by calling the decision procedure and, if that is inapplicable or runs beyond a prespecified time limit, the proof-planner itself is called. (The time limit is currently set to 60s for the decision procedure and 60s for the planner; see library.pl.)

   If a relation can be shown to be transitive, this is recored as a `transitive-pred/1` fact. Weak fertilization examines this database.

5. `trivially-falsifiable/2` has been added. This instantiates a universally quantified formula to a ground formula in which variables have been instantiated to random constants of the appropriate type. This ground formula is then evaluated. False instances reveal that the original formula is not a theorem.

6. `weak-fertilize/4` uses `trivially-falsifiable/2` to reject false subgoals.

7. Method `elementary/1`: individual clauses merged to remove unwanted backtracking. Use of decision procedure controlled by `using-presburger/0` switch (default `true`; see config/methods.pl).

8. Method `step-case/1` subgoals are stripped of all annotation.

9. Manual not up-to-date.

C.2.15 Version 2.7, patchlevel 2 `CLAM_2_7_2`

1. Small change to needs mechanism to support multiple libraries. needs.pl should no longer have the catch-all clause `needs(_,[]).`

   normally found at the end of the file.

2. Some additions to the library.

3. Speed improvements in induction preconditions.

4. Method `ind-strat/1`: prefers unflawed induction over unflawed casesplits over flawed induction.

C.2.16 Version 2.8, patchlevel 0, February 1999 `CLAM_2_8_0`

1. Piecewise fertilization method `pwf/1` incorporated into step-case of induction proof-plan.

2. All tactics for the basic induction proof-plan are present. Tactics for Presburger arithmetic not present.

C.2.17 Version 2.8, patchlevel 1, 7th April 1999 `CLAM_2_8_1`

1. Manual brought up-to-date.

2. `lib-load-dep/3` added.

C.2.18 Version 2.8, patchlevel 2, 18th May 1999 `CLAM_2_8_2`

1. If the switch `comm-proving/0` is true (default `false`; see config/tactics.pl) Clam attempts to show that binary functions are commutative.

   The predicate `is-commutative/2` does these proofs by calling the decision procedure and, if that is inapplicable or runs beyond a prespecified time limit, the proof-planner itself is called. (The time limit is currently set to 60s for the decision procedure and 60s for the planner; see library.pl.)

   If a function can be shown to be commutative, then commuted versions of all defining equations for the function are loaded. The rewriting tactics have not been extended to take this into account yet, which is why the switch is by default off.

2. The timeout code has been fixed, and should now allow an arbitrary number of nested timeouts to be set, ensuring that timeouts cause exceptions at the correct points in the code.

C.2.19 Version 2.8, patchlevel 3, 26th April 2005 `CLAM_2_8_3`

1. Fixes to LaTeXand sicstus to make compatible with later releases.
2. Strip out non-functional support for quintus and swi Prolog dialects.
3. Presburger decision procedure is *off* by default.
4. Tactics revised to allow running of default test suite.
5. Step case method changed to strip out annotations in hypotheses after induction.
6. Timing code uses sicstus time-out library.
7. Propositional method slightly extended to recapture symmetry of equations in strong fertilisation.

C.2.20 Version 2.8, patchlevel 4, July 2006 `CLAM_2_8_4`

Wider release of version working under current sicstus.

Bibliography

[1] David Basin and Toby Walsh. Termination orders for rippling. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 466–83, Nancy, France, 1994. Springer-Verlag.

[2] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[3] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available as DAI Research Paper No. 349.

[5] A. Bundy, van Harmelen F., C. Horn, and A. Smaill. The Oyster-Clam system. Research Paper forthcoming, Dept. of Artificial Intelligence, University of Edinburgh, 1989. Submitted to CADE-10.

[6] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[7] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[8] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Available from Edinburgh as Research Paper 419.

[9] A. Bundy, F. van Harmelen, and A. Smaill. Extensions to the rippling-out tactic for guiding inductive proofs. Research Paper 4nn, Dept. of Artificial Intelligence, University of Edinburgh, 1989. Submitted to CADE10.

[10] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[11] M. Carlsson and J. Widén. SICStus prolog user's manual. Research Report SICS R88007B, Swedish Institute of Computer Science SICS, October 1988. ISSN 0283-3638.

[12] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

[13] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[14] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Elsevier, New York, 1972.

[15] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992.

[16] Randy Forgaard. A program for generating and analyzing term rewriting systems. Master's thesis, Laboratory for Computer Science, MIT, USA, September 1984.

[17] J. Hesketh. Tutorial guide to nurprl proof development system. Blue Book Note 423, Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, 1988. NurPRL is the old name of the system now know as Oyster. Blue Book notes are not distributed outside the DReaM group.

[18] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, University of Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[19] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. Reprinted as "optimal path-finding algorithms" in "Search in Artificial Intelligence", L. Kanal, and V. Kumar (eds.), Symbolic Computation Series, Springer Verlag, 1988, pp. 223-267.

[20] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[21] S. Negrete. Proof plans with hints. Master's thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1991.

[22] S. Negrete. Hint Mechanism for Clam. Technical Paper 8, Dept. of Artificial Intelligence, University of Edinburgh, March 1992.

[23] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101, 395. Warsaw, 1930. Annotated English version also available [27].

[24] Quintus. Quintus prolog user guide and reference manual, sun and vax unix. Technical Report Release 2.2, Quintus Computer Systems, Inc., 1988.

[25] J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with lambdaclam. In Claude Kirchner and Hé;ène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.

[26] Alan Smaill and Ian Green. Higher-order annotated terms for proof search. In Joakim von Wright, Jim Grundy, and John 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.

[27] Ryan Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR 84-639, Department of Computer Science, Cornell University, September 1984.

[28] Joachim Steinbach. *Termination of rewriting.* PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, Germany, Janurary 1994.

[29] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Ma., 1986.

[30] F. van Harmelen. Are proof plans linear? or: what does "linear" mean anyway? Blue Book Note 421, Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, 1988. Blue Book notes are not distributed outside the DReaM group.

[31] F. van Harmelen. Clam should be theory free. Blue Book Note 539, Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, 1989. Blue Book notes are not distributed outside the DReaM group.

[32] F. van Harmelen. Generalising weak fertilization. Blue Book Note 538, Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, 1989. Blue Book notes are not distributed outside the DReaM group.

[33] Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. Technical Report TBA, Dept. of Artificial Intelligence, University of Edinburgh, 1994.

Index of methods and methodicals

Index of predicates

Index of Prolog source files

Index