MASTER IN COMPUTER SCIENCE

Theory Development in an Automated Theorem Prover

Ricour, Stéphane

Award date:
1991

Awarding institution:
University of Namur

Link to publication

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Theory Development in an Automated Theorem Prover

par Stéphane RICOUR

promoteur :

Professeur B. Le Charlier

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Année académique 1990-1991

Acknowledgements

First of all I would like to express my gratefulness to Alan Smaill and David Basin from the Edinburgh university department of artificial intelligence, who have devoted a lot of their precious time to my education in Type Theory and in the Oyster/Clam theorem prover. Without their advice and encouragements it should have been impossible for me to achieve this work.

I wish also to thank Baudouin Le Charlier from the FUNDP, Namur, the promoter of my work. He gave me the opportunity to spend five months in Edinburgh and helped me in the writing of this essay.

Next, we would like to thank our parents for their generous and constant support shown throughout all these years spent in college.

Finally, I want to thank all the people who have been a great help to us, although they have not been directly involved in the completion of this work. So, we owe a great deal of gratitude to the Edinburgh DReaMers who helped to make my stay in Edinburgh so delightful.

Abstract

English

Oyster/Clam is an automated theorem prover based on Martin-Löf's constructive theory of types. Recent work has centred round the notion of *proof plan* to guide the search for proofs. Experiments with proof plans for finding proofs by induction, using heuristics adapted from the work of Boyer and Moore, have shown encouraging results.

This work apply this approach to the data structure of finite sets. A systematic approach is taken to building up the machinery needed to use such a theory, in the form of suitable tactics and a library of theorem.

Français

Oyster/Clam est un démonstrateur automatique de théorèmes basé sur la théorie constructive des types développée par Martin-Löf'. Les travaux récents ont été effectués autour de la notion de *plan de preuve* pour guider la recherche de démonstrations. Les expérimentations avec les plans de preuves dans la recherche de démonstrations par induction en utilisant une heuristique adaptée des travaux de Boyer et Moore ont donné des résultats encourageants.

Ce travail applique cette approche à la structure de donnée des ensembles finis. Une approche systématique est prise pour fournir les mécanismes nécessaires à l'emploi d'une telle théorie sous forme de tactiques appropriées et d'une librairie de théorèmes.

Contents

1 **Introduction** 1

2 **The type theory** 3

2.1 Introduction 3

2.2 Types 4

2.3 Universes 7

2.4 Term evaluation 7

2.5 Proposition as types 8

3 **Oyster** 10

3.1 Introduction 10

3.2 Theorems and proofs 11

3.3 Term extraction 12

3.4 The rule base 14

3.4.1 Constructors 14

3.4.2 selectors 16

3.4.3 Type formation 17

3.5 Tactics 18

3.6 The definition mechanism 19

4 **Clam** 21

4.1 introduction 21

4.2 Methods 22

4.3 Planners and Proof plans 23

4.4 Automatization of proofs by induction 23

5 Representation of the finite set theory **29**

5.1 Introduction 29

5.2 Finite set as a new basic type 30

5.3 Building the finite set type as an abstraction of the existing types . 31

5.4 The language of the finite set theory 34

6 The power set as a Boolean algebra **39**

6.1 Introduction 39

6.2 What is a Boolean algebra ? 40

6.3 An induction principle over the finite sets 42

6.4 Rippling out the wave fronts 43

6.4.1 Set union 46

6.4.2 Set intersection 47

6.4.3 Set difference 48

6.5 Analysis of the results 49

6.5.1 Explanation of a plan generated by Clam 49

6.6 Analysis of the failures 50

7 Conclusion **55**

A Proof plans generated by Clam **57**

List of Figures

2.1 propositions-as-types 9

5.1 proof of def_union 36
5.2 basic definitions of the finite set theory 37
5.3 other functions over the finite sets 38

6.1 scheme/5 for the set induction 44
6.2 Proof plan for the theorem 1.1 50
6.3 Proof of the theorem 1.1 51

A.1 Proof plan for the theorem 1.1 58
A.2 Proof plan for the theorem 1.2 58
A.3 Proof plan for the theorem 2.1 59
A.4 Proof plan for the theorem 4.2' 59
A.5 Proof plan for the theorem 5.1 60

Chapter 1

Introduction

Since the advent of Artificial Intelligence, huge efforts have been done for the automatization of mathematical verification. Lot of systems exists today with variable performance.

Our work is based on the Oyster/Clam system, developed at the University of Edinburgh by Pr. Alan Bundy and the group of mathematical reasoning [Horn 88], [van Harmelen 89]. This system is based on Martin-Löf's constructive theory of types developed as a formalisation of constructive mathematics [Martin-löf 79], [Martin-löf 73] .

The first chapter will introduce intuitionistic theory of types which was originally developed as a symbolism for the codification of constructive mathematics but which may be viewed also as a programming language.

The second chapter describes the Oyster proof checker. Oyster is an interactive proof editor closely based on the Nuprl system [Constable et al. 86]. Proofs are constructed in a top-down fashion by application of the rules of inference. The object-level logic is a version of Martin-Löf's type theory while Prolog provides a language to write tactics. Since the object-level logic is constructive, terms of an enlarged λ-calculus can be computed from complete proofs, and these so-called extract terms can then be executed by application on appropriate inputs.

Clam is a meta-level system built on top of Oyster in an attempt of turning the interactive proof editor into a fully automatic theorem proving system. To guide the search of proof, Oyster tactics are specified into methods. Methods are specifications in term of pre- and post-conditions. Giving a theorem, a search is done at the meta-level using the methods for finding the tactics that should make up the proof. The result of that search is a tree of methods called proof-plan.

The next chapters, reflect my work amid the mathematical reasoning group. The project consists to apply the approach of proof-plans to the data structure of finite sets using a systematic approach to build the machinery needed to use such a theory in the form of suitable tactics and a library of theorems.

After a discussion about the representation we will give to the finite sets, we make the development of the theory culminate in proving the theorems needed to show that the power set with the inclusion relation makes up a Boolean algebra. We will try to give the first lines of a methodology for systematically providing the system the tools it needs to generate the proofs of theorems.

Chapter 2

The type theory

2.1 Introduction

This chapter is intended to give an overview of constructive type theory based on typed λ-calculus.

The elements of the theory are types and the members of those types. A type is a collection of objects having similar structure. Henceforth in this chapter we will use capital letters to design a type and lower case letters to design the members.

The proof of a theorem in constructive logic will implicitly give the way to construct an inhabitant of the corresponding type. We will show that this feature allows to interpret the constructive logic as a high-level programming language.

The basic objects are called terms, they are built using variables and operators. The terms of the type theory underlying Oyster are those of the λ-calculus (variables, abstractions and applications) and terms generated by extensions of the typed λ-calculus (type constructors, data constructors, constants, primitive recursion, list recursion and arithmetic functions).

In our presentation, we assume that the reader is familiar with common syntactic notions such as substitution and variable-binding. We denote the simultaneous substitutions of the terms t_i for the free occurrences of the variable x_i in s by the term :

$s[t_1, t_2, ...t_i/x_1, x_2, ..., x_i]$.

2.2 Types

A type is defined by designing a constant as type, and choosing a notation for its elements. These are the canonical expressions that are the canonical members of the type. The members of a type are the terms that have as value the canonical members of the type. To complete the definition of a type an equivalence relation on its members called the *equality in* that type is associated to the type. This equality is a three place relation : $t \; = \; s \; in \; T$.

There is also an equivalence relation, $T \; = \; S$, on types called *type equality* and it is defined as follow : two types are equal if and only if they evaluate to equal canonical types.

Here are introduced the types of the theory. We use `typewriter` font to signify actual Oyster syntax. The types are listed with their canonical and non canonical constants.

The type theory underlying the Oyster system has the following basic types :

- `atom` as in Prolog or Lisp, its canonical members are the denumerably many character strings written `atom('...')` ;

- `pnat` providing the natural numbers with Peano arithmetic. The canonical elements are `0` and terms of the form `s(...)` where s is the successor function on natural members. Finally we define the non canonical constant `p_ind(x,a,[u,v,t])` where x is of type pnat, a and t are terms of type T and u,v are free variables in t, by the computation rules :

  $p\_ind(0, a, [u, v, t]) = a$

$p\_ind(s(x), a, [u, v, t]) = t[x, p\_ind(x, a, [u, v, t])/u, v]$

- `void` is an empty type, i.e. there are no members of `void`;
- `int` is the type of integers.The canonical members are the integers in the common sense (`... -2, -1, 0, 1, 2, ...`) There are non canonical constructors for integers : the usual arithmetic operations.

Oyster also embodies the following derived type concepts and set constructors; considering A and B as types:

- `A list` is a type

  - its members are the empty list `nil` and non empty lists `x::y` constructed from an element x of type A and a list y of type *A list.* The non canonical constant `list_ind(x,y,[u,v,w,z])`, where x is of type A list, y and z of type T and u,v,w are free variables in z, is defined by the computation rules :

  $list\_ind(nil, y, [u, v, w, z]) = y$

  $list\_ind(a :: b, y, [u, v, w, z]) = z[a, b, list\_ind(b, y, [u, v, w, z])/u, v, w]$

- `A` # `B` is a type

  - the cartesian product of types A and B has as members the ordered pairs a&b where a is of type A and b of type B. We defined the non canonical constant `spread(s,[u,v,t])` where s is of type `A` # `B`, t is of type T and u,v are free variables in t, by means of the computation rule :

  $spread(a\&b, [u, v, t]) = t[a, b/u, v]$

- `A` \ `B` is a type

  - the disjoint union of types A and B. Its members have the form `inl(a)` where a is in A and `inr(b)`, where b is in B. Given an element of `A` \ `B`, it must be possible to decide which component it is in. The decision operator has the form : `decide(z, [u,f], [v,g])` and the computation rules are :

$$decide(inl(a),[u,f],[v,g]) = f[a/u]$$

$$decide(inr(a),[u,f],[v,g]) = g[a/v]$$

- `A -> B` is a type

  - the function types. Members are λ-terms of the form `lambda(x,`t_x`)`where x is of type A and t_x of type A $\rightarrow$ B. The application of a function will be noted `f of a` where f is any element of A $\rightarrow$ B and a is an element of A.

- `x: A # B` is a type

  - the dependent product has as members the ordered pairs `a&b` such that a is an element of A and the second component, b, is an element of type $B[a/x]$. The non canonical constant `spread(s,[u,v,t])` acts as for the cartesian product.

- `x: A -> B` is a type

  - the dependent function describes functions whose range type depends on the input. Members are λ-terms of the form `lambda(x,`t_x`)` but the result type of t_x may depend on the value of x. The application of a function will be noted `f of a` where f is any element of $x : A \rightarrow B$ and a is an element of A.

- `{x: A \ B}` is a type

  - the set type has as members all values of x in type A which satisfy B. Strictly speaking, members are ordered pairs $a\&b$ where a is an element of type A and b an element of B_a, i.e. a proof that a satisfies the proposition B. But the set constructor provides a mechanism for hiding information to simplify computation. So members of a set type `{x: A \ B}`, are members of A while the information that those members satisfy B is hidden.

- `A /// [x,y,B]` is a type

- the quotient type builds a new type from the basic type A, using the equivalence relation B over members of A. It can therefore be used to redefine equality in type A.

2.3 Universes

Types are classified in universes forming a hierarchy. `u(1)` is the first universe and contains all the basic types of the theory and types built using the above mentioned type constructors.

In addition to the types in $u(1)$, `u(2)` contains so-called *large types,* namely $u(1)$ and types built from it such as $A \rightarrow u(1), u(1) \rightarrow u(1), A \rightarrow (B \rightarrow u(1))$ and so forth.

The hierarchy of universes is cumulative. By cumulative we mean that the universe $u(i)$ is in universe $u(i+1)$ and that every element of $u(i)$ is also an element of $u(i+1)$.

Universes are themselves types and every type occurs in a universe. In fact A is a type if and only if it belongs to a universe. Conversely all the elements of a universe are types.

2.4 Term evaluation

The computation system is based on a lazy evaluation procedure. The closed terms (this means terms without free variable) are either canonical or non canonical . Each canonical term has itself as value (for instance the natural numbers). The evaluation of non canonical terms cannot always succeed since there are terms which have no canonical form. When a non canonical term evaluates, it is said to be reducible. The evaluator successively chooses a non canonical subterm in appropriate form and replaces it with a term closer to canonical form. This process of replacing such a term with another is called term reduction. A non canonical reducible form is called a redex while the canonical term resulting from the evaluation is called its contractum. As example for the list recursion combinator *list_ind*,

$list\_ind(x, y, [u, v, w, z])$ evaluates to the value of y if x evaluates to the empty list, *nil*;

and evaluates to $z[h, t, list\_ind(t, y, [u, v, w, z])/u, v, w]$ if x evaluates to the non empty list $h :: t$.

2.5 Proposition as types

What is maybe the most interesting feature of the type theory is the consideration of the propositions as types. This consideration can be expressed as an isomorphism between type theory and propositional logic. Proposition correspond to types, proofs to lambda terms and theorems to types with inhabiting elements. In intuitionistic logic, a proposition is true if and only if there is an evidence for it. In terms of the type theory we have the notion of a type being inhabited where this is true if and only if the type has at least one member. Specifically, a proposition is a type inhabited by proofs of the proposition if it is valid, or are empty otherwise. There is only one canonical element in these types : `axiom`, the most trivial proof.

The consideration of propositions as types enables to define higher-order logic by means of the correspondences spelled by figure (2.1). For example to show that the type $A \# B$ is inhabited one needs to find an inhabitant for the type A and one for the type B. Considering propositions as types this is equivalent to show an evidence for the proposition A and one for the proposition B so the cartesian products acts precisely as the conjunction $A \wedge B$.

Propositions	Types
$A \vee B$	$A \mid B$
$A \wedge B$	$A \# B$
$A \Rightarrow B$	$A \rightarrow B$
$\exists x \in A.B$	$x : A \# B$
$\forall x \in A; B$	$x : A \rightarrow B$
$a = b \in A$	$a = b\ in\ A$

Figure 2.1: propositions-as-types

Chapter 3

Oyster

3.1 Introduction

Oyster is a proof development system closely based on the Cornell Nuprl system [Constable et al. 86] of which it is a Prolog reimplementation.

Oyster can be seen as the conjunction of two languages, the object language which is the mathematical language of the system, and the metalanguage which is a programming language to write proof-generating programs on assertion expressed in the object language. The object language is the constructive theory of types based on the work of Martin-Löf outlined in the previous chapter. The metalanguage is Prolog as opposed to ML in the Nuprl system and it enables the user to formalize proof techniques in the shape of Prolog programs called *Tactics.*

Oyster is a primitive kernel on which can be built other tools like a library system, a graphic interface and so on. Further more the almost unlimited complexity that the tactics can reach could transform the system from a simple proof checker into a powerful proof generator.

Since the object level logic is constructive, a proof of a theorem in the Oyster system implicitly provides directions for constructing a witness of the truth of the theorem. These so called *extract terms* can then be executed by application on appropriate inputs.

Subsequently in this chapter we will describe how to formalize theorems and proofs in the system. In that section our aim will be to show the working of the system without entering all the details of the user interface. Readers interested in this point will refer to the Oyster user manual [Horn 88]. Then we will explain the concept of extract term and how Oyster can lead to program synthesis. The presentation of the inference rules contained in the base rule will follow and the last section of the chapter will concern the tactics.

3.2 Theorems and proofs

In Oyster a theorem consists of a goal statement of the form :

$H \;\Rightarrow\; G$

where H is a hypothesis list and G is an assertion in the object-level logic; the $\Rightarrow$ correspond to $\vdash$ or the sequent arrow.

We distinguish three kind of hypothesis :

- *definitions* are hypothesis of the form :

  $d(x, y, ...) \;\; <==> \;\; t_{x,y,...}$

  where d is an arbitrary identifier; $x, y, ...$ form a possibly empty parameter list and $t_{x,y,...}$ is any type theoretic term with free variable $x, y, ...$ Definitions can also be defined globally. The section 3.6 explains more in details the definition mechanism.

- *Assumptions* are hypothesis of the form :

  $v : T$

  meaning that v is an inhabitant of type T.

- The last kind of hypothesis are *references to theorems*. Those have the form :

  $v : H \Rightarrow G$

  or $v : H \Rightarrow G\ ext\ E$

  where H is a list hypothesis and G the goal of the theorem while E is the extract term derived from that proof. v is a local name for the theorem.

The aim of a proof is to show that the goal, G is inhabited by explicitly constructing a member. This construction process might be characterized as stepwise refinement proof yielding the extract term E which is guarantied to be a member of G.

A proof is a finite tree whose nodes are pairs consisting of a sequent and a rule name or a placeholder for a rule name. The sequent part is composed by an hypothesis list an a goal. The sequent part of a child is entirely determined by the sequent part and the rule name of the parent node. The rules specify a finite number of subgoals needed to achieve the proof of the goal.

The general shape of a node is :

sequent by rule name

1. *subsequent* 1
2. *subsequent* 2

...

n. *subsequent n*

3.3 Term extraction

Since Oyster embodies a constructive logic, proving the truth of an assertion in the system is equivalent to showing that the type corresponding to the statement is inhabited, and proving that a type is inhabited in a constructive setting requires that the user specify how an object of the type be built.

Implicitly associated with each proof is a term whose type is specified by the main assertion being proved. That term can be used for synthesizing programs corresponding to proofs. This so called extract term is refined top-down during the proof process.

For each rule of inference there is a constructor which links together the constructions corresponding to the arguments of the major connective, the rule involves.

In any stage of the proof development it is possible to access the extract term of the proof constructed so far. Open subgoals of the proof, if they have any constructive significance, correspond to Prolog variables in the extract term.

There is a built-in evaluator for type theoretic terms, which allows the direct execution of Oyster programs. The evaluation works as explained in the previous chapter. It reduces gradually the non canonical terms so that they reach a canonical form which is the result of the evaluation.

We can think of Oyster as being a program synthesizer since if we consider, in constructive type theory, a programming problem as being a list of specifications, then a proof that the specifications can be met defines an algorithm which solves the problem. In other words, programs are synthesized from their specifications by proving a theorem of the form:

$\forall Input\ \exists Output;\ spec(Input, Output)$

where $spec(Input, Output)$ is a relationship between the input and the output of the desired program.

3.4 The rule base

Each type is associated with a collection of inference rules which can be used to reason about the type. In addition there is a collection of general rules concerned with equality and substitution which apply to all types. For each type, T, there are introductions and elimination rules. Essentially, Introduction rules generate constructor terms as extracts and elimination rules generate destructors of the appropriate type.

Our purpose in this section will not be to enumerate all those rules. For that we refer the reader to the Oyster user manual [Horn 88]. But our objective is to present a classification of the rules. The only rules that we will explicitly exhibit are illustrations presented in the following refinement style.

$H \;\Rightarrow\; T \textit{ ext } t \textit{ by rule}$

  $H_1 \;\Rightarrow\; T_1 \textit{ ext } t_1$

  $\ldots$

  $H_k \;\Rightarrow\; T_k \textit{ ext } t_k$

The goal refined is shown at the top and each subgoal is shown indented underneath. Our illustrations will be taken amid the rules concerning the list types.

The rules are classified in two main categories : *Constructors* and *Selectors*. A third category *Type formation* is added for reasons of perspicuity although strictly speaking those rules should be considered as part of the constructors for universes.

3.4.1 Constructors

- **refinement and realisation rules :** they describe the ways for straight forward refinement of the proof. The main result in applying such a rule is the refinement of the extract term of the top level goal, corresponding to the refinement step connected with the rule. We distinguish refinement rules that produce an extract term that needs further refinements from realisation rules that produces a complete extract term .

Examples :

$H \;\Rightarrow\; A\ list\ ext\ nil\ by\ intro(at(i),\ nil)$
$\quad\Rightarrow\; A\ list\ in\ u(I)$

is a realisation rule since nil is a canonical member of any well formed list type.

$H \;\Rightarrow\; A\ list\ ext\ B :: C\ by\ intro$
$\quad\Rightarrow\; A\ ext\ B$
$\quad\Rightarrow\; A\ list\ ext\ C$

the list construction is a partial refinement over any list type. The head and the tail are built in the subgoals.

- **membership rules :** they are applicable to goals of the form : $A\ in\ T$ and give the conditions under which a canonical object may be judged to inhabit a canonical type. The extract term of such a proof will always be *axiom* since such a goal has no computational meaning.

  Example :

  $H \;\Rightarrow\; B :: C\ in\ A\ list\ by\ intro$
  $\quad\Rightarrow\; B\ in\ A$
  $\quad\Rightarrow\; C\ in\ A\ list$

The understanding of this membership rule is easy : to show that $B :: C$ is of type $A\ list$, we need to show that the head is of type A and the tail of type $A\ list$.

- **equality rules :** these rules give the conditions under which objects having the same structural form may be judged to be equal. They thus applies to goals of the form $A = B\ in\ T$.

  Example :

  $H \Rightarrow A :: B = C :: D\ in\ T\ list\ by\ intro$
  $\quad \Rightarrow A = C\ in\ T$
  $\quad \Rightarrow B = D\ in\ T\ list$

  two lists are equal if they have the same head and the same tail.

3.4.2 selectors

- **refinement and realisation rules :** basically the elimination rules, they exploit the properties of the type of a variable in the hypothesis list for generating a selector construct which is able to handle the general case. The other rules falling in this category are the decision rules.

  Examples :

  $H,\ X : A\ list,\ H' \Rightarrow T\ ext\ list\_ind(X, T_b, [U, V, W, T_u])\ by\ elim(X, new[U, V, W])$
  $\quad \Rightarrow T_{[nil/X]}\ ext\ T_b$
  $\quad U : A,\ V : A\ list,\ W : T_{[V/X]} \Rightarrow T_{[U::V/X]}\ ext\ T_u$

  The elimination of a variable of a list type in the hypothesis list generates a list induction term. The two subgoals correspond respectively to the base and the step cases of the induction.

- **membership rules :** these rules give the conditions under which a non canonical object may be judged to inhabit a type.

  Example :

  $H \;\Rightarrow\; list\_ind(E, T_b, [X, Y, Z, T])\ in\ T_{[E/Z]}$
  $by\ intro(using(A\ list),\ over(Z, T_Z),\ new[U, V, W])$
  $\quad\Rightarrow\; E\ in\ A\ list$
  $\quad\Rightarrow\; T_b\ in\ T_{[nil/Z]}$
  $\quad U : A,\ V : A\ list,\ W : T_{[V/Z]} \;\Rightarrow\; T_{[U,V,W/X,Y,Z]}\ in\ T_{[U::V/Z]}$

  A list induction term is of a given type $T_{[E/Z]}$, if you can supply a type scheme, $over(Z, T_Z)$, such that $T_{[E/Z]}$ is an instantiation of that type scheme for E and the subterms of the induction term can be proven to be in the corresponding instantiations, $T_{[nil/Z]}$ and $T_{[U::V/Z]}$ and if you can predict the type of the base term, E, $using(Alist)$.

- **equality rules :** these rules are appropriate when there is a goal of the form :

  $H \;\Rightarrow\; selector(E, ...) \;=\; E'\ in\ T.$

  They have as consequence a reduction of the selector term.

3.4.3 Type formation

As we explained these are constructor rules for universes. For the list types, we have one refinement rule :

$H \;\Rightarrow\; u(I)\ ext\ A\ list\ by\ intro(A\ list)$
$\quad\Rightarrow\; u(I)\ ext\ A$

expressing that *A list* is a refinement for any universe if A is a refinement for the same universe. There is also a membership rule :

$$
\begin{aligned}
&H \;\Rightarrow\; A\ list\ in\ u(I)\ by\ intro\\
&\quad\Rightarrow\; A\ in\ u(I)
\end{aligned}
$$

A list is a type of universe level i, if A is a type of universe level i.

And there is also an equality rule :

$$
\begin{aligned}
&H \;\Rightarrow\; A\ list\ =\ B\ list\ in\ u(I)\ by\ intro\\
&\quad\Rightarrow\; A\ =\ B\ in\ u(I)
\end{aligned}
$$

two list types are equal if the corresponding base types are equals.

3.5 Tactics

Tactics are bits of Prolog code, containing Oyster commands and *tacticals.* Tacticals are special words derived from the original ML version of Oyster for specifying how the combination of the instructions has to take place.

- `repeat` T tries to apply T on the given problem and recursively repeats it on all the subproblems generated by T.
- T `or` S applies T, and if it fails tries S.
- R `then` S applies R then S to all the subproblems generated by R.
- R `then` $[S_1 ... S_n]$ applies R first and then each S_i to the corresponding subproblem.
- `complete` T succeeds only if T applies and generates no new subgoals.
- `try` U always succeeds. If U applies, it is performed, if it does not, the current goal is left unchanged.

3.6 The definition mechanism

There are different ways to define new objects in Oyster. One of those ways is to directly use the Oyster definition facility to associate the words of the theory we want to define with the terms in the theory. This is done using the predicate <==> . For instance we could define the operation of concatenation of two lists by :

$concat(a, b)$ <==> $list\_ind(a, b, [h, t, v, h :: v])$

the left hand side corresponds to the definiendum while the right hand side corresponds to the definiens.

For complex definitions, however, it is possible to achieve a kind of abstraction through a level of indirection. Instead of directly equating a display form with a term t, t is extracted from a theorem that contains type information ([D. Basin 90]).

We illustrate that method using again the example of the concatenation of two lists. As we already said, building a proof of a given theorem in type theory means constructing implicitly a witness of the type of the proposition to be proved and that different proofs could provide different witness.

We will prove a theorem about lists so that the witness build within the proof is the concatenation of the lists given as hypothesis.

The theorem to be proved is :

$a :$ *t list*,

$b :$ *t list*,

==> *t list*

To prove that theorem, it is enough to give whatever member of *t list*. But it is building the right member, the member that corresponds to the concatenation of lists a and b that we will achieve the abstraction that allows us to define the new concept. With that aim in view, we first refined the goal using the elimination rule on a, $elim(a)$.

$a:\ t\ list,$

$b:\ t\ list,$

$==>\ t\ list \qquad ext\ list\_ind(a, V_b, [U, V, W, V_i) \qquad by\ elim(a)$

1. $t\ list \qquad ext\ V_b$
2. $U : t,\ V : t\ list,\ W : t\ list\ ==> t\ list \qquad ext\ V_i$

The application of the rule generates two subgoals corresponding to the base and step cases of a list induction. While the extract term is a list induction term. The values of the term for the base and step cases are build in the subgoals. Now what remains to be done is to introduce the right value to proof the subgoals so that the induction term corresponds to the list concatenation.

The first subgoals corresponding to the base case of the induction is proved using the rule $intro(b)$. The second subgoal, corresponding to the step case is proved with $intro(U :: W)$.

After that we obtain an extract term corresponding to :

$list\_ind(a, b, [U, V, W, U :: W])$

Now the concatenation of any two lists of type t *list* may be found using the evaluation of the extract term. So if we call the theorem proved above *Concatenation*, we define the operation of concatenate two lists as follow :

$concat(x, y) <==> term\_of\ Concatenation\ of\ x\ of\ y$

This mechanism is convenient to inductively define functions. The top goal of the theorem to be proved shows the origin and target domains of the function.

Chapter 4

Clam

4.1 introduction

In this chapter we will describe the Clam system conceived to transform Oyster in an automated theorem prover. The key feature of this meta-level system is an analysis of the tactics in order to guess when it is appropriate to apply a tactic and what are the consequences of the application of that tactic. This goal is reached by specifying the tactics in term of pre-conditions and post-conditions.

Those specifications of the tactics are called *methods.* Once tactics are specified by methods, planners will then scan them in search of the applicable tactics. The work of the planners will be to connect methods so that they make up a proof plan for the theorem intended to be proved. If the tactics specified by the methods chained in the proof plan are applied, there is great chance that it leads to a proof of the theorem. But this is not guarantied in all the cases since methods describe the main features of the tactics but are not a complete specification of them. The reason for often deliberately writing methods with weak specifications is that in this way the method acts as a heuristic operator which can capture the essential specification of a tactic while leaving out the often expensive checks for finer details.

Last but not least we will explain the application of the proof plan mechanism to proof by mathematical induction for which encouraging results have been obtained.

4.2 Methods

Methods are specifications of tactics. As describe in [Bundy 88] a method is a data structure with 6 slots :

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

A method and its corresponding tactic are said to be applicable if the goal to be proved matches the input formula of the method and if the method's pre-conditions hold for this sequent. When an applicable methods is found, the output-slot give a schematic description of the resulting sequent of the tactic application and the post-conditions specify further syntactic properties of those sequent.

Although it is possible to use arbitrary Prolog code in the formulation of the pre- and post-conditions slots, a designated language for this purpose has been created. This method language consists of a set of predicates and a set of logical connectives. they are discussed in [van Harmelen 89].

4.3 Planners and Proof plans

The planners employs the methods in the search for a proof of a given Oyster sequent. They firstly find an applicable method (i.e. a method with matching input formula and true pre-conditions); then the planner compute the schematic output formulae and post-conditions of this method, and finally find methods applicable to these output formulae. This process is repeated until no unproven formulae remains.

The process of proof plan construction as described above is not entirely free from search: for a given sequent, more than one method may be applicable, and the system must choose one of them. For the moment four planners are part of Clam. They use different strategy for selecting an applicable method to a sequent. The three first strategies are : *depth first*, *breadth first* and *iterative deepening.* While the fourth strategy, *best first* is the only one employing a heuristic search strategy. In practice, the search at the meta-level is however small enough for the depth-first planner to succeed without very much backtracking.

4.4 Automatization of proofs by induction

The proof plan technique is currently used for the automatic guidance of proofs by mathematical induction.

The heuristic is based on the reconstruction of Boyer and Moore technique for constructing induction formulas, the state of the art in inductive theorem proving [Boyer & Moore 79].

By observing their own proofs and those of others, Robert Boyer and J. Moore noticed that, when proving theorems about recursive functions, two basic methods are available : a simple one in which rewrite rules derived from the recursive definitions are used to symbolically evaluate the theorem to be proved and a more complex one in which induction is used to divide the theorem into two simpler theorems.

The problem when trying to prove a theorem by induction, is that we are faced with a choice. There are often several induction schemata available, and each of them can be used to induct on a different parameter.

Recursion analysis is the name given to the process, embedded in the Boyer and Moore theorem prover of analysing the recursive structure of a conjecture to decide what form of induction to use to prove it.

When an induction schema is applied, it has for consequence that the induction conclusion differ from the hypothesis by the insertion of induction terms in place of the induction variable. Those expressions which appear in the induction conclusion but not in the induction hypothesis, are called *wave fronts.*

The objective of an induction schema choice is well described by the characterisation of a "good" induction quoted in [Stevens 88]:

> Thus we can characterise a "good" induction as one that we can deduce in advance will allow the maximum number of recursive terms in the conclusion to be eliminated.
>
> As a refinement we can additionally stipulate that it leaves a minimum number of recursive terms that we can deduce will be difficult to eliminate.

By recursive terms, one means instances of recursively defined terms.

That leads to a central idea of the proof plan, the tactic called *rippling-out*. This tactic manipulates the induction conclusion to enable the induction hypothesis to be used in its proof.

rippling-out applies *wave rules* that will remove the wave fronts from the innermost position further out.

Let us first define what are wave rules and then exhibit their utilities by example. Wave rules are rewrite rules of the form :

$$F(\ \boxed{S_1(}\ U_1\ \boxed{)}\ ,...,\ \boxed{S_n(}\ U_n\ \boxed{)}\ ) => \boxed{T(}\ F(U_1,...,U_n)\ \boxed{)}$$

where F,T and the S_i are terms with distinguished arguments and T may be empty, but F and the S_i must not be. The S_i are old wave fronts and T is the new wave front. A wave front in a formula can be seen as a designated subterm of that formula. This subterm itself contains a hole not part of the wavefront, called the *wave variable*, the U_i in the general form.

We adopt the convention that wave fronts are noted by boxes , as above.

Application of a wave rule ripples some wave fronts out by one stage. Step formulae of recursive definitions are always wave rules. In addition each theorem that Oyster proves can be tested to see if it has the right form and, if so, it can be stored as a wave rule for future use.

Let now illustrate that concept by the example picked in [Bundy et al 89b], the proof of the associativity of + :

$$\forall\ x, y, z;\quad x + (y + z)\ =\ (x + y) + z$$

by successor induction on x. The recursive definition of + is :

$$\forall u : pnat;\ \ \{0 + u = u\}$$
$$\forall v : pnat, \forall w : pnat;\ \ \{s(v) + w = s(v + w)\}$$

The induction hypothesis is :

$$x + (y + z) = (x + y) + z \qquad (1)$$

where x, y, z represent skolem constants. While the induction conclusion is :

$$\boxed{s(}\ x\ \boxed{)} + (y + z) = (\ \boxed{s(}\ x\ \boxed{)} + y) + z \qquad (2)$$

The induction term is $s(x)$ and the $\boxed{s(}...\boxed{)}$ constructor function is the wave front. The step case of the recursive definition of +, namely

$$\boxed{s(}v\boxed{)} + v = \boxed{s(}v + w\boxed{)}$$

provide a wave rule to which *rippling-out* will apply. Repeated applications of this rule to (2) ripples the two wave fronts to the outside of the left and right terms of the induction conclusion.

$$\begin{array}{l} \boxed{s(}x\boxed{)} + (y + z) = (\boxed{s(}x\boxed{)} + y) + z \\ \boxed{s(}x + (y + z)\boxed{)} = \boxed{s(}x + y\boxed{)} + z \\ \boxed{s(}x + (y + z)\boxed{)} = \boxed{s(}(x + y) + z\boxed{)} \qquad (3) \end{array}$$

When no further rippling out is possible then the induction hypothesis can often be used as a rewrite rule to produce an equation between two identical terms (this is achieved by the tactic *fertilization*). Using it left to right on the left hand side of the induction conclusion (3), it produces the equation :

$$\boxed{s(}(x + y) + z\boxed{)} = \boxed{s(}(x + y) + z\boxed{)}$$

which is readily proved .

The problem is thus to apply an induction schema that will allow the *rippling-out* tactic to succeed. Therefore, recursion analysis locates the recursive functions in the conjecture. Each occurrence of a recursive function, F, with a variable, X, in its recursive argument position, gives rise to a raw induction suggestion. The induction variable is X. The induction schema suggested is the one dual to the form of recursion used to define F.

In the previous example, the proof of :

$x + (y + z) = (x + y) + z,$

the first three occurrence of the recursive defined + give rise to an induction suggestion. The recursive argument is twice x and once$y$.

The dual induction schema corresponding to one step recursion is :

$$\frac{\Gamma \vdash P(0) \qquad \Gamma, x' : pnat, P(x') \vdash P(s(x'))}{\Gamma, x : pnat \vdash P(x)}$$

How to choose which one should become the final induction suggestion ? The application of the one step induction on x will have for consequence the substitution of the two occurrences of x by $s(x)$ in the induction conclusion that will look like :

$$\boxed{s(}\ x\ \boxed{)}\ +(y+z) = (\ \boxed{s(}\ x\ \boxed{)}\ +y) + z$$

In both cases, the recursive definition of + will match the term dominating the occurrence of $s(x)$. Allowing the rippling-out tactic to be used with the step case of the definition as wave rule as we did above.

This is not the case when applying the one step induction on y. In the induction conclusion :

$$x + (\boxed{s(}y\boxed{)}+z) = (x+\boxed{s(}y\boxed{)}) + z\ ,$$

the term dominating the second occurrence of $s(y)$, namely $x + s(y)$ does not match with the step case of the definition of +. The replacement of y by $s(y)$ in this case is said to be unsuitable and the induction suggestion is classified by Boyer and Moore as *flawed.* If unflawed inductions suggestions remain, the flawed ones are rejected. So, in our example, the one step induction on x is finally chosen.

As explained above, recursion analysis is restricted to using rewrite rules based on the step formula of recursive definitions to "ripple out" the wave fronts occurring in the induction conclusion.

This technique has been extended in various ways. Both, the *rippling-out* tactic and the notion of wave rule have been expanded.

The most important development was the extension of the technique so that the *rippling-out* tactic is able to use other rewrite rules than the only step formulae of recursive definitions : all the wave rules known by Clam can be used.

Then the notion of wave rule was extended by the introduction of : multi-wave rules, conditional wave rules, rippling-sideways and rippling on hypothesis.

Those extensions to the rippling-out tactic for guiding inductive proofs are explained in [Bundy et al 89b]. Here we will only give a short overview of the conditional wave rules since they will be used in our search of proofs for some finite set theory propositions.

Conditional wave rules are rules of the form :

$< condition > \rightarrow \ LHS \ \Rightarrow \ RHS$,

where $LHS \ \Rightarrow \ RHS$ is a wave rule.

Note the use of $\rightarrow$ for implication and $\Rightarrow$ for rewriting.

The conditional wave rules should be grouped so that the conditions are complementary. So it will be possible to use the rules after dividing the proof in two parts using the condition and its negation.

Chapter 5

Representation of the finite set theory

5.1 Introduction

After we described in the first three chapters the Oyster/Clam environment for automated proof guidance, our intent will now be the application of the machinery of proof plans to a new data structure, that of finite sets.

Development of a new theory in the Oyster/Clam system can be handled along different lines.

We will consider two different approaches : On one hand we will envisage the addition of a new basic type and the extension of the rule base, This approach will only be introduced; the reader will find a complete discussion about the addition of a basic type representing the finite sets in the work of Paul Chisholm [Chisholm]. On the other hand we will try to build the finite set theory as an abstraction of the existing types by the mean of the definition mechanism.

This last approach, as the other one, should allow us to experiment the machinery of proof plans applied to the new data structure but this approach will also allow us to eval the power of expressiveness of Oyster.

5.2 Finite set as a new basic type

Building a new basic type representing the finite sets could be done in a similar way as the list types. The first thing to do is to define the primitive constants required : $fset$, $\emptyset$, $\bullet$ and $fset\_ind$. $fset\_ind$ is a non-canonical constant and has the computation rules :

$$\begin{aligned} fset\_ind(\emptyset, y, [u,v,w,z]) &= y \\ fset\_ind(a \bullet b, y, [u,v,w,z]) &= z[a, b, list\_ind(b, y, [u,v,w,z])/u, v, w] \end{aligned}$$

meaning that $fset\_ind(s, y, [u,v,w,z])$ evaluates to y if s evaluates to the empty set. And if s evaluates to a non-empty set,$a \bullet b$, $fset\_ind(s, y, [u,v,w,z])$ has the value$z[a, b, list_ind(b, y, [u,v,w,z])/u, v, w]$.

In order to define a type whose objects are finite sets, we require a type from which the members of the sets are drawn. Once we have such a type, let say A, we can define the type of finite sets of objects of type A.

We will denote that type :

$fset(A)$

The canonical objects of the finite set type $fset(A)$ are :

the empty set , $\emptyset$,

and the non empty sets : $x \bullet y$

($\bullet$ denoting the set insertion) provided that x is an element of type A and y, a set of type A $fset$.

It is now necessary to prescribe under what conditions two canonical objects are equal :

1. $\emptyset$ and $\emptyset$ are equal canonical objects of $fset(A)$.

2. $a \bullet s$ and $b \bullet t$ are equal canonical objects of fset(A) provided $a = b\ in\ A$ and $s = t\ in\ fset(A)$.

3. $a \bullet s$ and $a \bullet a \bullet s$ are equal canonical objects of fset(A) provided $a\ in\ A$ and $s\ in\ fset(A)$.

4. $a \bullet b \bullet s$ and $b \bullet a \bullet s$ are equal canonical objects of fset(A) provided $a\ in\ A$, $b\ in\ A$ and $s\ in\ fset(A)$.

We showed above how to form canonical objects of $fset(A)$, and when two objects are equal. See [Chisholm] for a complete listing of the formal rules of the type and their justifications.

5.3 Building the finite set type as an abstraction of the existing types

In this section our aim will be to build our theory on the top of the existing ones. We used the built in types and types constructors of Oyster as well as the definition mechanism to create a level of abstraction in the language that will allow us to speak in the terms of the finite set theory.

The abstraction is achieved by linking the specific vocabulary of the finite set theory language with definitions in term of the built-in types and type constructors. The verification of the finite set theory axioms will then allow us to state if the built representation is a representation of the finite sets theory.

It is worthwhile to note that even in the case of adding a new basic type, it is necessary to make a choice of representation. But the choice does not take place at the same level of abstraction. In the case of a new basic type, we need to decide on a Prolog representation of the new structure (Prolog, since Oyster is written in Prolog), while in this case we need to set up the theory on the higher level of abstraction offered by Oyster/Clam.

There is a large choice of possible representations to build finite set theory within the type theory. In related works, lists without redundancy and strictly ordered lists are used. But it is also possible to imagine a representation based on simple lists without any other restriction.

The characteristic of the representation that will allow us to decide which one seems to be the most appropriate, is how will be expressed the equality between two finite sets so that the rules for reasoning about equality (the general *equality rule*, the *rewrite* rule, *introduction* rules for equality of non canonical members) can be applied to it.

The first thing to decide is either if we want to keep one of the basic equalities or if want to define a new one. To keep one of the basic equalities means to choose a representation such that the equality between two sets has the same truth values as the equality between the underlying type used to define the finite set type. On the other hand define a new equality means the use of quotient types.

Let see what are the consequences of that choice. At the first glance, it seems to be more comfortable to use quotient types and so be able to use the most natural way to formulate the equality between two sets. We mean that two sets are equal if and only if they have the same members. Moreover this should allow a considerable latitude in the choice of the representation. The definition :

$$fset(T) \;<==>\; T\;list///[x,y,(\forall\, e:T;\; (e \in x \;\rightarrow\; e \in y) \wedge (e \in y \;\rightarrow\; e \in x))]$$

represents the finite sets by list without any restriction and defined the equality as explained above. But this mechanism is, in fact, not as easy of use as it seems. Indeed the use of a quotient type to define finite sets will oblige us, each time we want to define a function on them, to show that the operation respects the new equality relation. If we have the quotient type :

$A///[x, y, E]$,

in order to define a function :

$f : A///[x, y, E] \rightarrow B$

one must show that the operation respects E, that is, $E(x, y)$ implies $f(x) = f(y)\ in\ B$.

The alternative to the use of a quotient type is to keep a basic equality. In that case, the choice is confined to the representation of the finite sets by strictly ordered lists without redundancy. Indeed it is necessary that each set has one and only one list representation. For instance the representation of the finite sets by ordered lists with redundancy is unsuitable : the sequent $1 \bullet 2 \bullet 3 \bullet nil = 1 \bullet 2 \bullet 2 \bullet 3 \bullet nil$ is clearly false in term of lists even if both lists stands for the same set $\{1, 2, 3\}$.

So only the representation of finite sets by strictly ordered lists answers the purpose that two sets are equal if and only if the lists they hide are also equal. But such a representation must also exist for all the sets, i.e. we need to find distinct ordered list to represent each set. This implies that for all the types of elements on which we want to build sets, we can define a strict order. Is this possible ? A negative answer would mean that the representation is too restrictive and only characterize a part of the finite sets theory. Happily this is not the case since whatever the type of the element is, the lexicographic order on the representation of the elements will provide the relation we are looking for (given the finiteness of the representation).

We will adopt this last representation to develop our theory. The price to pay will be a heaviness in the representation .

5.4 The language of the finite set theory

In order to form a type whose objects are finite sets, we require a type from which the members of the sets are drawn. But since we chose to represent the finite sets by strictly ordered lists without redundancy, this type from which the members are drawn must be defined with such a relation. Remember that a strict order is a relation, $\prec$, satisfying the following properties :

irreflexivity: $\forall x;\ \neg x \prec x$

transitivity : $\forall x\, \forall y\, \forall z;\ (x \prec y) \wedge (y \prec z) \rightarrow (x \prec z)$

And if we add that the order must be total, we obtain the definition of the type with its relation :

$$
\begin{aligned}
\mathit{ordered\_type} \;<==>\; & t : u(1) \wedge \\
& \exists\, r : t \rightarrow t \rightarrow u(1) \wedge \\
& \quad \forall\, x,y \in t;\ r(x,y) \vee r(y,x) \vee x = y \wedge \\
& \quad \forall\, x,y,z \in t;\ r(x,y) \wedge r(y,z) \rightarrow r(x,z) \wedge \\
& \quad \forall\, x \in t;\ \neg r(x,x)
\end{aligned}
$$

This definition is made up of three components : a type, t, belonging to the first universe, then a relation on the members of that type and the properties of the relation so that it is a strictly order on the elements of t. All the sets we will deal with, must have their members drawn from such a type accompanied by a strictly ordered relation.

Now we can define the type of the elements from which we will draw the members of our sets. Since the logical conjunction corresponds to the cartesian product in Type Theory, members of *ordered_type* will be ordered pairs. The type of the members of the finite set will thus be the first component of a t of type *ordered_type*.

$elem(t)\ <==>\ spread(t, [a, \_, a])$

where _ is equivalent to the Prolog anonymous variable.

The type of the finite sets on the type elem(t), will be a subset of the lists of elem(t), the strictly ordered lists. That property of the lists, we mean to be strictly ordered is verified by means of a theorem 'is_ordered' so that we obtain the definition using the subset constructor :

$fset(t)\ <==>\ \{s : elem(t)\ list \mid ((term\ of\ is\_ordered)\ of\ t)\ of\ s\}$

The figures 5.2 and 5.3 show the definitions that make up the language. In order to simplify the notation, from now on we will omit when unnecessary the reference type and use a more common notation. Thus we will write $a \cup b$ instead of set_union(t,a,b), $a \cap b$ instead of set_inter(t,a,b), and so on using the classical notations of the set theory.

To define the concepts we quite often employed the definition mechanism explain in 3.6. For instance, we define the union of two sets by :

$A \cup B <==> term\_of\ def\_union$

where def_union is a theorem whose statement is :

$A : fset \ \rightarrow\ B : fset\ \rightarrow\ fset$

This statement does not contain much information. It only tells us that the union is a function that, given two sets, gives another set. The works consists of building the proof that produces the witness corresponding to the union of A and B.

So all the computational information will come from the proof construction. The figure 5.1 shows the complete proof of the theorem def_union.

We will use different proofs of the same theorem, giving different witness to define functions that have the same origin and target domains, like the sets intersection, difference and so on.

$x \ : \ fset$

$y \ : \ fset$

$\vdash \ fset \qquad by\ elim(x) \qquad ext\ list\_ind(x, T_b, [U, V, W, T_u])$

$\quad 1. \vdash \ fset \qquad by\ intro(y) \qquad ext\ y$

$\qquad \vdash \ y\ in\ fset \qquad by\ intro$

$\qquad\quad \vdash \ true$

$\quad 2.\ v1 \ : \ fset$

$\quad\ \ e \ : \ elem$

$\quad\ \ \vdash \ fset \qquad by\ intro(e \bullet v1) \qquad ext\ e \bullet v1$

$\qquad \vdash \ e \bullet v1\ in\ fset$

Figure 5.1: proof of def_union

- $ordered\_type \;<==>\; t : u(1) \wedge$
  $$\begin{aligned} & r : t \rightarrow t \rightarrow u(1) \wedge \\ & \quad \forall\, x, y \in t;\; r(x,y) \vee r(y,x) \vee x = y \wedge \\ & \quad \forall\, x, y, z \in t;\; r(x,y) \wedge r(y,z) \rightarrow r(x,z) \wedge \\ & \quad \forall\, x \in t;\; \neg r(x,x) \end{aligned}$$

- $elem(t) \;<==>\; spread(t, [a, \_, a])$
- $fset(t) \;<==>\; \{s : elem(t)\ list \mid term\ of\ is\_ordered\ of\ t\ of\ s\}$
- $emptyset <==> nil$
- $a \bullet s$:

  $set\_insert(t, a, s) \;<==>\; term\_of(def\_insertion) of\ t\ of\ a\ of\ s$

Figure 5.2: basic definitions of the finite set theory

- $choose(s)$ $<==>$ $list\_ind(s, void, [x, \_, \_, x])$
- $rest(s)$ $<==>$ $list\_ind(s, nil, [\_, x, \_, x])$
- $card(s)$ $<==>$ $list\_ind(s, 0, [\_, \_, v, v+1])$
- $a - s$:

  $set\_delete(t, a, s)$ $<==>$ $term\_of(def\_deletion) of\ t\ of\ a\ of\ s$
- $s1 \cup s2$:

  $set\_union(t, s1, s2)$ $<==>$ $term\_of(def\_union) of\ t\ of\ s1\ of\ s2$
- $s1 \cap s2$:

  $set\_inter(t, s1, s2)$ $<==>$ $term\_of(def\_intersection) of\ t\ of\ s1\ of\ s2$
- $s1 \setminus s2$:

  $set\_dif(t, s1, s2)$ $<==>$ $term\_of(def\_difference) of\ t\ of\ s1\ of\ s2$
- $s1 \subset s2$:

  $set\_include(t, s1, s2)$ $<==>$ $term\_of(def\_inclusion) of\ t\ of\ s1\ of\ s2$

Figure 5.3: other functions over the finite sets

Chapter 6

The power set as a Boolean algebra

6.1 Introduction

In the previous chapter we discussed a formalism for reasoning about finite set theory. What we want now to do, is to apply the proof plan approach for inductive proofs of finite sets properties. With this aim in view, we make the development of the theory culminate in a proof that the partially ordered set $< PS(X), \subseteq >$, where $PS(...)$ stands for the power set, for all non empty finite sets X, is a Boolean algebra where the maximum element is the set X, the minimum element is the empty set $\emptyset$ and complementation corresponds to set theoretic complementation.

What we will try to do is use a systematic approach giving Clam a library of theorems needed to prove the Boolean algebra properties of $< PS(X), \subseteq>$. An interesting point of our work will be our attempt to foresee the needs for proving a whole class of theorems.

This point is crucial in the development of Clam, since in case of success it will be possible to draw the first lines of a methodology of the development of theories more systematic than the extension of the libraries incremently, theorem by theorem.

But firstly we will introduce the notion of Boolean algebra and the theorems that will make up our objective.

6.2 What is a Boolean algebra ?

In this section we will briefly introduce Boolean algebras. The point is the presentation of the problem, i.e. the theorems we want Clam to prove. For a complete discussion about that topic, we refer to [Bell & Slomson, 69].

We define a Boolean algebra as follow. A Boolean algebra β is an algebraic structure, say $\beta = < B, \vee, \wedge, \neg, 0, 1 >$ satisfying the propositions :

- $\forall x, y \in B;$
$$x \vee y = y \vee x$$
$$x \wedge y = y \wedge x$$
- $\forall x, y, z \in B;$
$$x \vee (y \vee z) = (x \vee y) \vee z$$
$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$
- $\forall x, y \in B;$
$$(x \vee y) \wedge y) = y$$
$$(x \wedge y) \vee y) = y$$
- $\forall x \in B;$
$$x \vee \neg x = 1$$
$$x \wedge \neg x = 0$$

- $\forall x, y, z \in B$;

$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$$
$$(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$$

Applying the theoretical definition to the power set, we obtain the algebraic structure :

$< PS(s), \cup, \cap, \neg, \emptyset, s >$ where s is a non empty set and $\neg$ is the set theoretic complementation with respect to s. It is a Boolean algebra only if we can prove the properties :

- $\forall x, y \in PS(s)$;

$$x \cup y = y \cup x \qquad 1.1$$
$$x \cap y = y \cap x \qquad 1.2$$

- $\forall x, y, z \in PS(s)$;

$$x \cup (y \cup z) = (x \cup y) \cup z \qquad 2.1$$
$$x \cap (y \cap z) = (x \cap y) \cap z \qquad 2.2$$

- $\forall x, y \in PS(s)$;

$$(x \cup y) \cap y) = y \qquad 3.1$$
$$(x \cap y) \cup y) = y \qquad 3.2$$

- $\forall x \in PS(s)$;

$$x \cup \neg x = s \qquad 4.1$$
$$x \cap \neg x = \emptyset \qquad 4.2$$

- $\forall x, y, z \in PS(s)$;

$$(x \cup y) \cap z = (x \cap z) \cup (y \cap z) \qquad 5.1$$
$$(x \cap y) \cup z = (x \cup z) \cap (y \cup z) \qquad 5.2$$

Those ten properties will make up the set of theorems of which we want Clam to find a proof. However for reasons of simplification in the representation, we will generalise those properties.

Properties 1.1 and 1.2 are in fact particularisation of the commutativity of the set's union and intersection while properties 2.1 and 2.2 are particularisation of the associativity of those operations on the sets. So we will prove them in the most general case without burden oneself with extra hypothesis the same reasoning holds for the four properties 3.1, 3.2, 5.1 and 5.2.

For the set complementation, we also simplify both properties by firstly replacing the complement of x by the difference : s/x (by applying the definition of complementation). Then keeping only one extra hypothesis, that x is a subset of s. So 4.1 and 4.2 become :

$\forall s, x \in fset;$

$$x \subset s \rightarrow x \cup (s \setminus x) = s \qquad 4.1'$$

$$x \subset s \rightarrow x \cap (s \setminus x) = \emptyset \qquad 4.2'$$

6.3 An induction principle over the finite sets

The section *What is a Boolean algebra ?* outlined some theorems, properties of the finite sets. We want now Clam to construct proof plans to proof those theorems. The proof we want Clam to construct are proof by induction following the proof strategy explained in 3.4 *Automatization of proofs by induction.* With this aim in view, we will try in this section and in the next one to provide the system with the tools it needs to build those proof plans.

First of all of course, we must provide Clam with an induction schema for the finite sets. To ensure the soundness of this new schema, the theorem justifying the scheme has to be proved. We build this proof interactively in Oyster. So we first proved the theorem :

$$\begin{array}{l}\forall\, \phi \;:\; (fset \;\rightarrow\; u(1)),\\ \phi(\emptyset),\\ \forall\, s \;:\; fset,\; \forall\, e \;:\; elem,\; \phi(s) \rightarrow\; \phi(e \bullet s);\\ \qquad\qquad \rightarrow\; \forall\, t \;:\; fset;\; \phi(t)\end{array}$$

To make Clam able to deal with that new induction schema, we need now to write a method describing that schema.

This method is shown in the figure 6.1. This is written in the method language and needs some explanations.

The description of an induction schema is given in occurrences of the predicate `scheme/5` . The first argument is the schema name, `set_insert(T,A,X)`, the second is the variable to induce on,`Var:fset`.

The schema maps the sequent `H==>G` into a list of base cases `[HBase==>GB]` and a list of step cases `[[A:elem,X:fset,H1:IndHyp|HBase]==>GS]` that will be produced by induction on `Var` .

6.4 Rippling out the wave fronts

The application of the induction schema will makes appear wave fronts in the induction conclusion.

Recursion analysis will then looks for wave rules able to ripple out those wave fronts as previously explained (cfr 4.4).

Most of the wave fronts generated will have one of the forms :

Given s_1 and s_2 of type $fset$ and e of type $elem$,

$\boxed{(e \bullet} s_1 \boxed{)} \;\Delta\; s_2$

or

```
scheme(set_insert(T,A,X),
       Var:fset,
       H==>G,
       [HBase==>GB],
       [[A:elem,X:fset,H1:IndHyp|HBase]==>GS]) :-
           add_to_hyp(Var:fset,H,HBase),
           matrix(Vs,Gm,G), append(Vs,HBase,VsHBase),
           free([A,X,H1],VsHBase),
           del_element(Var:fset,Vs,OVs),
           wave_fronts(set_insert(T,A,X),[[]-[[3]]],StepTerm),
           replace_all(Var,emptyset,Gm,GB1),
                   matrix(OVs,GB1,GB),
           replace_all(Var,X,Gm,IndHyp1),
                   matrix(OVs,IndHyp1,IndHyp),
           replace_all(Var,StepTerm,Gm,GS1),
                   matrix(OVs,GS1,GS).
```

Figure 6.1: scheme/5 for the set induction

$s_1 \; \Delta \; \boxed{\boxed{(e\bullet} \; s_2 \; \boxed{)}}$

where Δ stands for a functions over sets, thus $\cup$, $\cap$, $\setminus$, ...

So we will now check those functions, trying in each case to find wave rules able to deal with such wave fronts.

As previously explained, the wave rules maybe step formula of a recursive definition or theorems having the right form recognized by the system.

In the explanation of the definition mechanism in section 3.6 we did not speak about recursive definitions. In fact what was described there is only how to make correspond a new syntactic form to a sequent in type theory.

The recursive definitions are made by giving the base and step formulae of the definitions as theorems (independently of the syntactic definition given as explain in 3.6). Clam contains a mechanism of recognition that help it to identify recursive definitions.

For instance to give a recursive definition of the set union we just need to provide Clam with both theorems :

$$\begin{aligned} &\forall x \; : \; fset; \\ &\qquad \vdash \; \emptyset \cup x \; = \; x \end{aligned}$$

and

$$\begin{aligned} &\forall x, y \; : \; fset, \\ &\forall e \; : \; elem; \\ &\qquad \vdash \boxed{\boxed{(e\,\bullet} \; x \; \boxed{)}} \cup y \; = \; \boxed{\boxed{e \bullet (} \; x \; \cup \; y \boxed{)}} \end{aligned}$$

As for the induction schema, to preserve the soundness of the system the theorems justifying the recursive definitions and the other wave rules have to be prove. In fact we first prove the theorem and then , when loading it in the system, Clam recognize it as a wave rule.

The following subsections will checked the different functions on sets we are concerned with in ordered to provide recursive definitions and other wave rules that can be guessed if we assume an induction on their arguments.

6.4.1 Set union

A recursive definition of union is obtained by means of the theorems :

$\forall x \ : \ fset;$

$$\vdash \ \emptyset \cup x = x \qquad 6.1$$

and

$\forall x,y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \boxed{(e \bullet} \, x \, \boxed{)} \cup y = \boxed{e \bullet (} \, x \cup y \, \boxed{)} \qquad 6.2$$

An induction over the first argument suggests the wave rule :

$\forall x,y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \boxed{(e \bullet} \, x \boxed{)} \cup y = \boxed{e \bullet (} \, x \cup y \, \boxed{)}$$

But this corresponds to the step formula of the recursive definition, so we do not need to add it. While an induction over the second argument suggests the wave rule :

$\forall x,y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash x \cup \boxed{(e \bullet} \, y \, \boxed{)} = \boxed{e \bullet \ (} \, x \cup y \boxed{)} \qquad 6.3$$

6.4.2 Set intersection

In this case we need an extension of the original wave rules, the conditional wave rules as we described them in section 4.4. Also the steps formulae of the recursive definition will make up a complementary set of conditional wave rules.

So the recursive definition of the set's intersection will be made up of the three theorems :

$\forall x \ : \ fset :$

$$\vdash \ \emptyset \cap x = \emptyset \qquad 7.1$$

$\forall x, y \ : \ fset,$
$\forall e \ : \ elem;$

$$\vdash \ e \in y \ \rightarrow \boxed{\boxed{(e\bullet}\ x\boxed{)}} \cap y \ = \boxed{e\bullet(}\ x \ \cap \ y\boxed{)} \qquad 7.2$$

$\forall x, y \ : \ fset,$
$\forall e \ : \ elem;$

$$\vdash \ \neg \ e \in y \ \rightarrow \boxed{\boxed{(e\bullet}\ x\boxed{)}} \cap y \ = \boxed{(}\ x \ \cap \ y\boxed{)} \qquad 7.3$$

Again the step formulae of the recursive definition, give the wave rules suggested by an induction over the first argument. Another couple of wave rules are necessary to deal with an induction over the second argument :

$\forall x, y \ : \ fset,$
$\forall e \ : \ elem;$

$$\vdash \ e \in x \ \rightarrow \ x \cap \boxed{\boxed{(e\bullet}\ y\boxed{)}} = \boxed{e\bullet(}\ x \ \cap \ y\boxed{)} \qquad 7.4$$

$\forall x, y \ : \ fset,$
$\forall e \ : \ elem;$

$$\vdash \ \neg \ e \in x \ \rightarrow \ x \cap \boxed{\boxed{(e\bullet}\ y\boxed{)}} = \boxed{(}\ x \ \cap \ y\boxed{)} \qquad 7.5$$

6.4.3 Set difference

Again we need conditional wave rules to recursively define the notion of set's difference The recursive definition of the set's difference will be made up of the three theorems :

$\forall x \ : \ fset;$

$$\vdash \ \emptyset \setminus x \ = \ \emptyset \qquad 8.1$$

$\forall x, y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \ e \in y \ \rightarrow \boxed{(e\bullet} \, x \, \boxed{)} \setminus y \ = \boxed{(} \, x \ \setminus \ y \, \boxed{)} \qquad 8.2$$

$\forall x, y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \ \neg \ e \in y \ \rightarrow \boxed{(e\bullet} \, x \, \boxed{)} \setminus y \ = \boxed{e \bullet (} \, x \ \setminus \ y \, \boxed{)} \qquad 8.3$$

Again the step formulae of the recursive definition, give the wave rules suggested by an induction over the first argument. Another couple of wave rules are necessary to deal with an induction over the second argument :

$\forall x, y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \ e \in x \ \rightarrow \ x \setminus \boxed{(e\bullet} \, y \boxed{)} = \boxed{e \ - \ (} \, x \ \setminus \ y \, \boxed{)} \qquad 8.4$$

$\forall x, y \ : \ fset,$

$\forall e \ : \ elem;$

$$\vdash \ \neg e \in x \ \rightarrow \ x \setminus \ (e \bullet y) \ = \ (x \ \setminus \ y) \qquad 8.5$$

6.5 Analysis of the results

By means of the induction schema and the wave rules introduced in the previous sections we directly obtain Clam to prove the theorems : 1.1, 1.2, 2.1, 4.2' and 5.1. An explanation of the proof plan obtained for theorem 1.1 is given here after. The reader will find the other proof plans generated by Clam in the appendix A.

6.5.1 Explanation of a plan generated by Clam

The first theorem we wanted to proof was the commutativity of the set's union (theorem 1.1) :

$$\forall x, y \in fset;\quad x \cup y = y \cup x \qquad 1.1$$

The figure 6.2 shows the proof plan found by Clam for this theorem. The figure 6.3 is an outline of the proof conjecture.

Formulae are sequent of the form $H \vdash G$, where $\vdash$ separates the list of hypothesis, H, from the goal G. The first sequent is the statement of the theorem to be prove.

The (sub)methods selected to rewrite the goal are indicated in parenthesis . Only newly introduced hypothesis are written in successive sequent; so each new goal is inheriting all the hypothesis of the parents goals in the proof.

Ind_strat_I (for induction strategy,) is a super-method for guiding almost the whole of the proof. It is defined by combining the submethods induction, base ,ripple _out and fertilize. Ripple_out itself is a super-methods which is responsible for the repeated application of wave rules to ripple out the wave fronts to the outermost position.

```
ind_strat_I(set_insert(t,v0,v1),x:fset(t)) then
  [ind_strat_I(set_insert(t,v0,v1),y:fset(t)) then
     [tautology(...),
      tautology(...)
     ],
   tautology(...)
  ]
```

Figure 6.2: Proof plan for the theorem 1.1

The ind_strat_I method applies the heuristic of recursion analyses to select the induction schema and then to ripple out the wave fronts. The proof of the commutativity of set's union is obtained by induction on x. A second induction, on y, is necessary to proof the base case while the proof of the step case is obtained by rippling out the wave fronts with the help of the wave rules.

6.6 Analysis of the failures

This section is dedicated to the explanation of the proof 2.2. We will show how it is possible to provide Clam with more tools that will help it to solve the problems it reaches when trying to prove some of the theorems.

The global vision of the theory allows to find properties useful in the search of proofs but often theorems need hints that can not be deduced from that approach. That is why at the end it is still necessary to look each case and find what we have to add to finally obtain the proof.

Why did Clam fail in the search for a proof of the set's intersection associativity ? The theorem to prove is :

$\forall\, x \,:\, fset,$

$\forall\, y \,:\, fset;$

$\vdash\ x \cup y = y \cup x$ (*by induction on* x)

1. $\vdash\ \emptyset \cup y = y \cup \emptyset$ (*by base*)

   $\vdash\ y = y \cup \emptyset$ (*by induction*)

   1.1 $\vdash\ \emptyset = \emptyset \cup \emptyset$ (*by base*)

   $\vdash\ \emptyset = \emptyset$ (*by tautology*)

   $\vdash\ true$

   1.2. $y = y \cup \emptyset,$

   $e \,:\, elem,$

   $\vdash\ e \bullet y = e \bullet y \cup \emptyset$ (*by wave*)

   $\vdash\ e \bullet y = e \bullet\ (y \cup \emptyset)$ (*byfertilize*)

   $\vdash\ e \bullet y = e \bullet y$ (*by tautology*)

   $\vdash\ true$

2. $x \cup y = y \cup x,$

   $e \,:\, elem,$

   $\vdash\ e \bullet x \cup y = y \cup e \bullet x$ (*by* $2 * wave$)

   $\vdash\ e \bullet\ (x \cup y) = e \bullet\ (y \cup x)$ (*by fertilize*)

   $\vdash\ e \bullet\ (x \cup y) = e \bullet\ (x \cup y)$ (*by tautology*)

   $\vdash\ true$

Figure 6.3: Proof of the theorem 1.1

$\forall x, y, z \in PS(s);$

$$x \cap (y \cap z) = (x \cap y) \cap z$$

An induction on x will generate two subgoals :

1. $\vdash \emptyset \cap (y \cap z) = (\emptyset \cap y) \cap z$
2. $x \cap (y \cap z) = (x \cap y) \cap z,$
   $e : elem,$
   $\vdash e \bullet x \cap (y \cap z) = (e \bullet x \cap y) \cap z$

The first subgoal will easily be proved using the base formula of the recursive definition of the set's associativity. In its attempts to prove the second subgoal, Clam will try to apply wave rules to ripple out the wave fronts. Since the wave rules applicable are conditionals, the system divides the proof into two cases corresponding to the condition and its negation. So to apply the wave rules 7.4 and 7.5 to the left-hand side of the goal, the proof is divide into two subparts one corresponding to the hypothesis : $e \in (y \cap z)$ and the other one to the hypothesis $\neg\, e \in (y \cap z)$. After that split, the wave rules 7.4 and 7.5 may be applied giving respectively the goals :

2.1. $e \in (y \cap z),$
   $\vdash e \bullet (x \cap (y \cap z)) = (e \bullet x \cap y) \cap z$

2.2 $\neg\, e \in (y \cap z)$
   $\vdash x \cap (y \cap z) = (e \bullet x \cap y) \cap z$

The same reasoning is now applied to ripple out the wave fronts in the right hand side of the goals. It will generate the goals :

2.1.1 $e \in y,$
   $\vdash e \bullet (x \cap (y \cap z) = e \bullet (x \cap y) \cap z$

2.1.2 $\neg e \in y$,

$\vdash e\bullet(x \cap (y \cap z) = (x \cap y) \cap z$

2.2.1 $e \in y$,

$\vdash x \cap (y \cap z) = e\bullet(x \cap y) \cap z$

2.2.2 $\neg e \in y$,

$\vdash x \cap (y \cap z) = (x \cap y) \cap z$

The goal 2.2.2 is now readily proved using the induction hypothesis. The goal 2.1.2 does not contain any wave front that could be ripple out : in the left hand side, the wave front is in the outermost position while there is no more wave front in the right hand side. But in the goals 2.1.1 and 2.2.1, the left hand side still contains wave fronts that can be ripple out. So once again those goals will be divide in two subgoals :

2.1.1.1 $e \in z$,

$\vdash e\bullet(x \cap (y \cap z) = e\bullet((x \cap y) \cap z)$

2.1.1.2 $\neg e \in z$,

$\vdash e\bullet(x \cap (y \cap z) = (x \cap y) \cap z$

2.2.1.1 $e \in z$,

$\vdash x \cap (y \cap z) = e\bullet((x \cap y) \cap z)$

2.2.1.2 $\neg e \in z$,

$\vdash x \cap (y \cap z) = (x \cap y) \cap z$

There is no difficulty to end the proofs of 2.1.1.1 and 2.2.1.2 using the induction hypothesis. Three goals are still not completely proved : 2.1.2, 2.1.1.2 and 2.2.1.1. But those three goals could be easily be proved by exploiting the contradiction in their hypothesis (the reader will remember that the hypothesis are inherited from the parents goals). To show those contradictions here are the goals with the list of the hypothesis in question.

2.1.2 $e \in (y \cap z)$,
$\neg\, e \in y$,
$\vdash e \bullet (x \cap (y \cap z)) = (x \cap y) \cap z$

2.1.1.2 $e \in (y \cap z)$,
$\neg e \in z$,
$\vdash e \bullet (x \cap (y \cap z)) = (x \cap y) \cap z$

2.2.1.1 $\neg\, e \in (y \cap z)$, $e \in y$,
$e \in z$,
$\vdash x \cap (y \cap z) = e \bullet ((x \cap y) \cap z)$

This is what mis Clam to finish the proof of the set's intersection associativity. To make Clam able to exploit those contradiction, we created a terminating method handling the hypothesis to end the proof. The soundness of the method is warrant after the proof of the theorem :

$\forall x, y : fset$,
$\forall e : elem$;
$$\vdash e \in (x \cap y) <==> (e \in x) \wedge (e \in y)$$

With the help of this method for handling the contradiction in the hypothesis list, there is no more obstacle for Clam to find the proof.

Chapter 7

Conclusion

In this work I tried to exploit the possibilities offered by Oyster/Clam system to develop new theories and generate proofs of theorems about those theories. I especially devoted my work to the application of the proof-plan methodology to inductive proofs, using recursion analysis. The two last chapters show encouraging results : given the induction schema over the sets and a set of wave rules, we obtain proof-plans leading to the complete proof of the theorems. This observation shows the efficiency of recursion analysis to catch the main structure of an inductive proof.

But those results were not obtained effortless. The use of the definition mechanism instead of developing a new basic type is heavy when we need to work on such defined terms. A comfortable notation often hides a complex structure. A good extension to this work should be the development of the finite set theory using a new basic type.

A great part of the difficulty was in the proofs of the wave rules and the induction principle. This raises the question if the main part of the proofs is done by Clam when it assembles those theorems together or, on the contrary, if the main part is already done when those theorems are proved (...by the user). In the second alternative, the advantage of Clam should be to avoid the user to have to rewrite the same part of proof each time he needs it but the "intelligence" of the system should be feeble. I will not answer this question, indeed I consider I don't have the knowledge the give a sensible solution.

Appendix A

Proof plans generated by Clam

```
ind_strat_I(set_insert(t,v0,v1),s1:fset(t)) then
  [ind_strat_I(set_insert(t,v0,v1),s2:fset(t)) then
     [tautology(...),
      tautology(...)
     ],
   tautology(...)
  ]
```

Figure A.1: Proof plan for the theorem 1.1

```
induction(set_insert(t,v0,v1),s1:fset(t)) then
[sym_eval([.]) then
     ind_strat_I(set_insert(t,v0,v1),s2:fset(t)) then
       [tautology(...),
        tautology(...)
       ],
   casesplit([member(t,v0,s2)=>void,member(t,v0,s2)]) then
     [wave([1,1],[set_inter3,left]) then
        wave([2,1],[set_inter6,left]) then
          strong_fertilize(v2),
      wave([1,1],[set_inter2,left]) then
        wave([2,1],[set_inter5,left]) then
          weak_fertilize(right,[.]) then
            tautology(...)
     ]
  ]
```

Figure A.2: Proof plan for the theorem 1.2

```
ind_strat_I(set_insert(t,v0,v1),s1:fset(t)) then
  [tautology(...),
   tautology(...)
  ]
```

Figure A.3: Proof plan for the theorem 2.1

```
induction(set_insert(t,v0,v1),s:fset(t)) then
  [sym_eval([..]) then
     tautology(...),
   casesplit([member(t,v0,s1)=>void,member(t,v0,s1)]) then
     [wave([2,1,1],[set_dif2,left]) then
        wave([1,1],[set_inter3,left]) then
          strong_fertilize(v2),
      wave([2,1,1],[set_dif3,left]) then
        strong_fertilize(v2)
     ]
  ]
```

Figure A.4: Proof plan for the theorem 4.2'

```
ind_strat_I(set_insert(t,v0,v1),s1:fset(t)) then
  [tautology(...),
   casesplit([member(t,v0,s3)=>void,member(t,v0,s3)]) then
     [wave([1,1],[set_inter3,left]) then
        wave([2,2,1],[set_inter3,left]) then
          strong_fertilize(v2),
      wave([1,1],[set_inter2,left]) then
        wave([2,2,1],[set_inter2,left]) then
          wave([2,1],[set_union2,left]) then
            weak_fertilize(right,[.]) then
              tautology(...)
     ]
  ]
```

Figure A.5: Proof plan for the theorem 5.1

Bibliography

[D. Basin 90]	D. Basin. *Building Problem Solving Environments In Constructive Type Theory.* PhD thesis, Cornell University, 1990.
[Bell & Slomson, 69]	J.-L. Bell & A.B. Slomson . *Models and Ultraproducts : An introduction.* North-Holland American Elsevier,1969.
[Boyer & Moore 79]	R.S. Boyer and J.S. Moore.*A computational Logic.* Academic Press, 1979. ACM monograph series.
[Bundy 88]	A. Bundy. The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction,* pages 110-120, Springer Verlag, 1988. Longer version available as DAI Research Paper No.349 .
[Bundy et al 89a]	A. Bundy, F. van Harmelen and A. Smaill. *Extensions to the Rippling-out Tactic for Guiding Inductive Proofs.* Research paper 459, Dept. of Artificial Intelligence, Edinburgh, 1989. Submitted to CADE10.
[Bundy et al 89b]	A. Bundy, F. van Harmelen, J. Hesketh and A. Smaill. Experiments with proof plans for induction. In *Journal*

of Automated Reasoning, 1989. Earlier version available from Edinburgh as Research paper n° 413.

[Chisholm] Paul Chisholm.*A theory of finite Sets in Constructive Type Theory.* Department of computer Science, Heriot-Watt University, Edinburgh.

[Constable et al. 86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[van Harmelen 89] F. van Harmelen. *The Clam Proof Planner, User Manual and Programmer Manual.* Technical paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.

[Horn 88] C. Horn. *The Nurprl Proof Development System.* Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Manna & Waldinger, 85] Zohar Manna & Richard Waldinger. *The logical basis for computer programming.* Addison-Weslay publishing company, 1985.

[Martin-löf 79] 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.

[Martin-löf 73] Per Martin- löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium 73*, H.E. Rose and J.

C. Shepherdson, eds.North Holland, Amsterdam. 1973, pages 73-118.

[Stevens 88] Andrew Stevens. *A rational reconstruction of Boyer and Moore's technique for constructing induction formulas.* Research paper 360, Dept. of Artificial Intelligence, Edinburgh, 1989. Submitted to ECAI-88 CADE9.