# The Diagonalization Method In Automatic Proof

## AIM4 Project Dissertation

Jeremy Gow

Department of Artificial Intelligence
University of Edinburgh
1997

## Abstract

In proof planning, common patterns of mathematical reasoning are embodied in tactics. Each of these is associated with a *method*, specifying its preconditions and effects. Using methods, a plan for a proof can be constructed from tactics. This plan is then used to guide a theorem prover through the proof. We have implemented diagonalization, a common form of mathematical reasoning, in Clam, a proof planning system. The methods are based on a formalisation of diagonalization inspired by results of Cantor and Turing. Tested against several theorems these methods constructed plans for all of them. We have been successful in using these plans to generate informal, English proofs of the theorems. However, there is still scope to generalise the methods.

**Acknowledgements**

I would like to thank my supervisors Alan Bundy, Toby Walsh and Antony Maciocia for their invaluable guidance during this project. Thanks also to Manfred Kerber and Michael Kohlhase for their help.

Thank you to my friends Colin Cameron, Ben Curry and Robert Hutton-Squire for their support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Diagonalization is a form of mathematical reasoning found in a range of different proofs. The aim of this project is to implement diagonalization reasoning in Clam, a proof planning system, and to test it.

The Clam system constructs plans for proving theorems, which are then used to guide Oyster, a theorem prover, through the proofs (for an overview see [Bundy *et al.* 90], for more detail [vanHarmelen 96]). These plans are made up of tactics, which represent common patterns of reasoning. A tactic's preconditions and effects are specified by a method. Implementing diagonalization in Clam has involved writing a set of methods capable of producing plans for diagonalization proofs.

This project has involved an attempting to formalise the notion of diagonalization, based on proofs by Cantor and Turing, and using this formalisation to design the Clam methods, along with auxiliary methods required to do non-diagonalization reasoning. These were tested against several diagonalization theorems, including Cantor's original diagonalization results and the unsolvability of the halting problem. The methods successfully planned all of the test theorems. These plans were not used to guide the Oyster theorem prover, as developing the domain theories and tactics required for this is a significant task. Instead they were used to produce informal, English proofs of the results in order to assess the methods.

The definition and use of diagonalization arguments is discussed in §2. §3 covers previous work related to proof planning and diagonalization. We then describe the design (§4) and empirical testing (§5) of the Clam methods. §6 describes the use of the diagonalization proof plans to generate informal, English proofs. Our work is compared with related research in §7. §8 discusses possible extensions to the project and our overall conclusions are given in §9.

# Chapter 2

# Proof by Diagonalization

The original diagonalization argument was used by Georg Cantor in 1891 to prove that $\mathbb{R}$, the set of reals numbers, has greater cardinality than $\mathbb{N}$, the set of natural numbers[1]. He used it again to prove the cardinality of a set is less than it's powerset (see [Kleene 52] for both proofs).

Diagonalization has also been used to prove the unsolvability of the halting problem (due to Turing) [Cutland 80], and several other results from recursive function theory, such as the existence of non-computable functions $\mathbb{N} \rightarrow \mathbb{N}$ [Cutland 80]. The central part of Gödel's first incompleteness theorem was also a form of diagonalization argument (see [Boolos & Jeffrey 80] or [Kleene 52]).

## 2.1 Diagonalization Arguments

The notion of proof by diagonalization is not formally defined by mathematicians, but refers to the common pattern of reasoning used in several results, such as those mentioned above. Informally, it obtains a contradiction when there exists a set of functions A$\rightarrow$B which is indexed by A. If $f_k$ is the func-

[1] actually proved for the subset [0,1] of $\mathbb{R}$

tion indexed by k∈A, then figure 2.1 illustrates a diagonalization argument where A is the natural numbers. The rows of the array correspond to the functions A→B, whilst the columns are the values each function takes at a given number. The *diagonal* of the array is the list of values given by $f_k(k)$. We define the *diagonal element* as a function d:A→B which differs from the diagonal at every value. Now d is a function A→B, so it appears on the list. But it also differs from *every* member of the list at the diagonal, hence the required contradiction.

$$\begin{array}{ccccc} & 1 & 2 & 3 & \dots \\ f_1 & f_1(1) & f_1(2) & f_1(3) & \dots \\ f_2 & f_2(1) & f_2(2) & f_2(3) & \dots \\ f_3 & f_3(1) & f_3(2) & f_3(3) & \dots \\ \vdots & \vdots & \vdots & \vdots & \end{array}$$

Figure 2.1: A diagonalization argument, with the diagonal marked.

In an attempt to make this concept more formal, we present a model of a formal generalised diagonalization argument in figure 2.2. This is inspired by Cantor's theorems and Turing's proof of the unsolvability of the halting problem, and it's relationship to these results is shown in table 2.1.

Although this formal argument captures the notion of diagonalization as used by Cantor or Turing, it may be inadequate for more complex examples, which could require a more generalised diagonalization argument. We shall return to this point in §2.2.

Note that in Cantor's theorems, the powerset $\mathbb{P}(a)$ is represented as the

Goal: A contradiction.

Given:

- T, the set of all functions f:A$\rightarrow$B such that C(f) for some condition C.
- a surjective function S:A$\rightarrow$T.
- a function Z:B$\rightarrow$B with no fixed points, ie. $\forall$x$\in$B. Z(x)$\neq$x.

Proof:

1) Define the **diagonal element** d(x) = Z(S(x)(x)).
2) Show C(d), and as d:A$\rightarrow$B then d$\in$T.
3) So for some q$\in$A, d = S(q), as S is surjective.
4) Thus S(q)(q) = d(q) = Z(S(q)(q)), which is a contradiction.

Figure 2.2: A formal generalised diagonalization argument

set of all functions a$\rightarrow${true,false}, and the interval [0,1] as the set of all functions $\mathbb{N}$ $\rightarrow${0,1,. . .,9}, as in [Huang *et al.* 95a]. This representation of [0,1] ignores the problem of periods in the largest digit, e.g. 0.6999. . . = 0.7, so some members have more than one corresponding function.

From the formal argument we can extract the following prerequisites for diagonalization:

1. The current goal is a contradiction.

2. There is a surjective function S:A$\rightarrow$T.

3. All members of T are functions A$\rightarrow$B.

4. There is a function Z:B$\rightarrow$B with no fixed points, or equivalently, there are at least two distinct elements in B.

5. The diagonal element d:A$\rightarrow$B can be shown to satisfy membership conditions of T.

| Object | Powerset | Card. of $\mathbb{R}$ | Halting |
|---|---|---|---|
| A | a | $\mathbb{N}$ | $\mathbb{N}$ |
| B | {true,false} | {0,1,. . .,9} | $\mathbb{N} \cup \{u\}$ |
| T | P(a) | [0,1] | {unary computable fns} |
| Z(x) | $\neg$ | if(x=0, 1, 0) | if(x=0, u, 0) |
| C(x) | none | none | x is computable |

Table 2.1: Relationship between generalised diagonalization argument and some theorems.

These prerequisites will be useful in writing Clam methods to plan when diagonalization can be used in a proof.

## 2.2 Diagonalization Theorems

We now look at two uses for diagonalization in proofs: Cantor's method and Turing's method, following [Hofstadter 79]. Both methods use diagonalization as outlined above to obtain a contradiction, but use it to refute different kinds of assumption. Given the set T (as defined above), Cantor's method proves there is no surjective function S: A$\rightarrow$B by assuming the contrary and using diagonalization to obtain a contradiction. Informally, it assumes T can be indexed by A, then constructs d$\in$T that has no index. It is used to prove such theorems as (all from [Kleene 52]):

- The cardinality of the powerset of A, $\mathbb{P}$(A) is greater than A (Cantor).
- The cardinality of the real numbers $\mathbb{R}$ is greater than the natural numbers $\mathbb{N}$ (Cantor).
- The cardinality of the functions $\mathbb{N} \rightarrow \mathbb{N}$ is greater than $\mathbb{N}$.
- The cardinality of the functions $\mathbb{N} \rightarrow \mathbb{N}$ is greater than the primitive recursive functions $\mathbb{N} \rightarrow \mathbb{N}$ (R. Péter).

With Turing's method it is given that S indexes T. That some $X:A \rightarrow B$ is not in T can be proved by assuming $X \in T$ and using diagonalization to obtain a contradiction. Informally, it assumes $X \in T$, then uses this to construct a $d \in T$ that has no index. It is used to prove such theorems as (all from [Cutland 80]):

- There exists an uncomputable function $\mathbb{N} \rightarrow \mathbb{N}$.
- The undecidability of the self-halting problem (Turing).
- The undecidability of the halting problem (Turing).

The methods differ in the indirect assumption that causes the contradiction: that T can be indexed by A with Cantor's method, and that some X is in T with Turing's method. These may not be the only proof methods that employ diagonalization arguments, but they do represent the most common uses in mathematics.

The above analysis constitutes a rational reconstruction of diagonalization based upon Cantor and Turing's proof methods. We have used this as the basis for our diagonalization methods, as described in the next chapter.

Note that this theory is not general enough to describe all possible arguments that could be considered diagonalization. For instance, Gödel's theorem cannot be captured by this formalisation. This is discussed further in chapter 8.

# Chapter 3

# Background Work

## 3.1 Proof Planning

The Clam system [vanHarmelen 96] was created by the Mathematical Reasoning Group at Edinburgh, where it has been used to do extensive work on proof planning. This work has focused mainly on proof by mathematical induction [Bundy 88]. A key idea in the induction methods was *rippling* [Bundy *et al.* 93], a technique for controlling rewriting which could be used to prove the induction step case. Empirical testing of this work, described in [Bundy *et al.* 91], suggests it is a good technique for reducing the search involved in automated reasoning, as the planning search space is much smaller than the original object-level space.

Successful work has been done in other domains, such as summing series [Walsh *et al.* 92], which has also involved the rippling technique. The philosophy behind this approach, that proof plans provide a higher-level understanding of proofs, is discussed in [Bundy 91], which also suggests criteria by which proof plans may be assessed.

Another proof planning system, Omega, has been developed at Saarbrücken [Huang *et al.* 94]. The Omega system differs significantly from Clam in that its methods are represented declaratively, so that they can easily be modified by the system automatically.

## 3.2 Diagonalization with Omega

The research group at Saarbrücken have already implemented diagonalization reasoning in their Omega proof planning system. Their interest is in automatically adapting methods to cope with novel situations [Huang *et al.* 95b].

In their work involving diagonalization [Huang *et al.* 95a] they extracted an initial method `Diag-1` from a user-guided proof of Cantor's powerset theorem (see §2.2), which was then mechanically reformulated to `Diag-2`, a method dealing with Cantor's theorem on the cardinality of $\mathbb{R}$. The two methods were generalised to `Diag-1-2`, which covered both theorems. This was in turn mechanically generalised to `Diag-1-2-3` to cope with the halting problem. Gödel's theorem has also been looked at in this context [Melis 94].

The difference in method representation between Clam and Omega is such that the Clam methods required for this project cannot be based upon the Omega diagonalization methods. The latter do, however, give useful insights into how diagonalization proofs may be formalised.

# Chapter 4

# Design of Methods

For the planning of any diagonalization proof, two types of method are required: those which implement the actual diagonalization reasoning, and those needed to do other parts of the proof – the support methods. These support methods may vary depending on the mathematical domain of the theorem, but the diagonalization methods should not.

Nine methods have been designed in total. These are the diagonalization methods `diagonal/5`, `diagonal_cond/6` and `construct_diag/0`, and the support methods `indirect/1`, `imply_goal/2`, `hyp_goal/1`, `cardinality/3`, `compose_surj/6` and `comp_equiv/2`. Below we describe the design of these Clam methods, and the parts they play in the planning process.

Apart from `known/2`, `gen_symbol/1`, `istype/2`, `in_ind/1` and `set_ind/1` all predicates are from the method language given in [vanHarmelen 96]. `known(A,Hyp)` succeeds if A matches a lemma or appears in the list Hyp. Each time `gen_symbol(X)` is called it instantiates a variable X to another member of the list of atoms `gen1`, `gen2`, `gen3`, ... , whilst `istype(A,T)` returns the type `T` of certain atoms `A`. `in_ind/1` is a flag that indicates the planning process has applied the `indirect/1` method, and `set_ind/1` changes this flag. These predicates are defined in full in appendix C.

## 4.1 Diagonalization Methods

These methods have been based on the description of diagonalization given in §2.1. From that model, the prerequisites for a diagonalization proof are the following:

1. The current goal is a contradiction.
2. There is a surjective function `S:A→T`.
3. All members of `T` are functions `A→B`.
4. There are at least two distinct elements in B.
5. The diagonal element `d:A→B` can be shown to be in `T`.

For diagonalization to take place these prerequisites must be satisfied. (1) to (3) are either given by the theorem, or established by support methods (see §4.2). The diagonalization methods and possibly further planning by support methods then establish (4) and (5).

Given (1) to (3) hold, diagonalization can be planned in two distinct ways, depending on the definition of the set `T`. If this set `T` is simply the set of functions `A→B` then the `diagonal/5` method is applied. As (5) is trivially true no further planning is necessary, and so `diagonal/5` is a terminating method.

If `T` has other conditions of membership[1] then the `diagonal_cond/6` method is applied. This establishes (5) by having the output goal be that `d` satisfies these conditions. As these will depend on the domain of the theorem, this goal must be solved by support methods (see §4.2).

Both these diagonalization methods call the `construct_diag/0` method from their preconditions, which checks prerequisite (4) is satisfied, and constructs a suitable diagonal element. An alternative arrangement that was

[1] For example, `T` is the set of *computable* functions `A→B`.

considered had an extra output goal `Hyp==>diag(D,S,B)` for both the methods, which could be solved by `construct_diag/0`. This would have given `diagonal_cond/6` two output goals:

1. construct a diagonal element `D`.

2. show `D` satisfies condition `Cond`.

However, once (1) had been solved, i.e. `D` instantiated, then (2) might fail due to the particular choice of `D`. Clam's backtracking mechanism will fail the `diagonal_cond/6` method, rather than retrying subgoal (1) to construct another diagonal element. This could lead to a planning attempt failing despite there being a choice of `D` which could satisfy (2).

Having `construct_diag/0` called from the preconditions of the diagonalization methods solves this problem, as this precondition could potentially be satisfied by several different diagonal elements, and so the `diagonal_cond/6` method could be applied using any one of them. Hence the method can be reapplied with a new choice of diagonal element when another fails, allowing more than one choice to be tried during planning.

In general, if one of a set of subgoals fails, Clam will not backtrack on the others. This can lead to a possible plan not being found if two subgoals interact, i.e. the solution to the first affects the solution to the second. This is a problem with Clam's planning mechanism that should be investigated.

```
method(diagonal(A,B,T,S,D),
       Hyp ==> void,
       [(known(S:fun#surj(S,A,T),Hyp);                          (P1)
         (known(surj(S,A,T),Hyp),                               (P2)
          known(S:fun,Hyp))),                                   (P3)
        known(Form,Hyp),                                        (P4)
        instantiate(_,Form,el(x,T)<=>map(x,A,B),Ins),           (P5)
        matrix(Vars,el(X,_)<=>map(X,_,_),Form),                 (P6)
        member(X:fun,Vars),                                     (P7)
        applicable(Hyp==>diag(D,S,B),construct_diag,_,_)],      (P8)
       [],
       [],
       diagonal(A,B,T,S,D)).
```

The `diagonal/5` method is applicable to sequents of the form `Hyp==>void`, ie. the goal is a contradiction, and has six preconditions, (P1) to (P6). These state: (P1 to P3) There is a lemma or a formula in the hypothesis list `Hyp` that states: there is a surjective function `S`: `A`→`B`. (P4) `Form` is a lemma or formula in the hypothesis list `Hyp` that (P5) can be instantiated to a formula which states: (P6 & P7) `x` is an element of set `T` if and only if `x` is a map from set `A` to set `B`, ie. `T` is the set of all functions `A`→`B`. (P8) The method `construct_diag/0` is applicable to the sequent `Hyp==>diag(D,S,B)`, where `D` is a meta-variable. This means '`D` is a diagonal element constructed from the function `S`, mapping onto the set `B`'. If this succeeds `D` is instantiated to the diagonal element.

As the `diagonal/5` method is terminating it has no postconditions or output formulae. The tactic `diagonal/5` is passed the sets `A`,`B` and `T`, the function `S` and the diagonal element `D`, from which it is possible to construct the diagonalization argument.

Figure 4.1: The `diagonal/5` method

```
method(diagonal_cond(A,B,T,S,D,Cond),
       Hyp ==> void,
       [(known(S:fun#surj(S,A,T),Hyp);                       (P1)
         (known(surj(S,A,T),Hyp),                            (P2)
          known(S:fun,Hyp))),                                (P3)
        known(Form,Hyp),                                     (P4)
        instantiate(_,Form,el(d,T)<=>map(d,A,B)#Cond,Ins),   (P5)
        matrix(Vars,el(X,_)<=>map(X,_,_)#_,Form),            (P6)
        member(X:fun,Vars),                                  (P7)
        applicable(Hyp==>diag(D,S,B),construct_diag,_,_)],   (P8)
        [append([d:fun,map(d,A,B),x:nat=>defn(d,D)],
                Hyp,NewHyp)],
        [NewHyp==>Cond],
        diagonal_cond(A,B,T,S,D,Cond)).
```

The `diagonal_cond/6` method's preconditions differ from the `diagonal/5` method's (see figure 4.2) only in that `Form` must now be instantiated to a formula which states: (P5 to P7) `d` is an element of set `T` if and only if `d` is a map from set `A` to set `B` *and* conditions `Cond` are satisfied by `d`. The postconditions add the definition of the diagonal element `d` to `Hyp` to give the new the hypothesis list `NewHyp`. The output sequent is `NewHyp==>Cond`, where `Cond` are the conditions that `d` is in `T`. The tactic `diagonal_cond/6` is passed the same as `diagonal/5`, along with the conditions `Cond`.

Figure 4.2: The `diagonal_cond/6` method

```
method(construct_diag,
       Hyp==>diag(D,S,B),
       [known(el(E1,B),Hyp),                                    (P1)
        known(el(E2,B),Hyp),                                    (P2)
        not E1 = E2,                                            (P3)
        ((istype(E1,E1Type),                                    (P4)
           D = if(ap(ap(S,x),x) = E1 in E1Type, E2, E1))        (P5)
          orelse
          (istype(E2,E2Type),                                   (P6)
           D = if(ap(ap(S,x),x) = E2 in E2Type, E1, E2)))],     (P7)
       [],
       [],
       construct_diag).
```

The method `construct_diag/0` solves sequents of the form `Hyp==> diag(D,S,B)`, where D is a diagonal element constructed from the function `S`, mapping onto the set `B`. The preconditions of `construct_diag/0` are: (P1 to P3) There are lemmas or formulae in the hypothesis list `Hyp` which state: `E1` and `E2` are distinct elements of the set `B`. And either (P4) `E1` is of type `E1Type` and (P5) `D` is `if(ap(ap(S,x),x) = E1 in E1Type, E2, E1)`, or else (P6) `E2` is of type `E2Type` and (P7) `D` is `if(ap(ap(S,x),x) = E2 in E2Type, E1, E2)`. The method sends no arguments to the tactic `construct_diag/0`, and is terminating, so has no postconditions or output formulae.

Figure 4.3: The `construct_diag/0` method

## 4.2 Support Methods

The support methods handle the non-diagonalization planning required for a theorem. This can be divided into pre- and post-diagonalization planning.

The methods `indirect/1` (figure 4.4), `cardinality/3` (figure 4.5) and `compose_surj/6` (figure 4.6) are required for pre-diagonalization planning, i.e. they set up the conditions necessary for a diagonalization proof (namely prerequisites (1) and (2), above). The `indirect/1` method plans proof by contradiction, satisfying prerequisite (1). The `cardinality/3` method implements Cantor's method for proving the cardinality of set A is greater than T by proving the goal 'there is no surjective function from T to A'. In the cardinality proofs this negated goal becomes a hypothesis (by `indirect/1`), satisfying prerequisite (2). `compose_surj/6` takes two surjective functions $s_1 : A \to B$ and $s_2 : B \to C$ and asserts that their composition $s_2 \circ s_1 : A \to C$ is also surjective. In some proofs this is necessary to satisfy prerequisite (2).

The `imply_goal/2` (figure 4.8), `hyp_goal/0` (figure 4.7) and `comp_equiv/2` (figures 4.9 & 4.10) methods are required for post-diagonalization planning, i.e. they solve output goals produced by the `diagonal_cond/5` diagonalization method required to establish prerequisite (5) (see §4.1). Given $A \Rightarrow B$ the method `imply_goal/2` replaces a goal which instantiates to $B$ with a similarly instantiated $A$. The `hyp_goal/0` method finishes off plan branches where the goal can be instantiated to one of the hypotheses. `comp_equiv/2` reasons about computability of functions/expressions, replacing a goal of the form 'X is computable' with an equivalent 'Y is computable', which should be easier to solve.

```
method(indirect(Hyp2),
        Hyp ==> Goal,
        [precon_matrix(Vars,Pre=>void,Goal),  (P1)
         not Pre = [],                        (P2)
         in_ind(0)],                          (P3)
        [append(Vars,Pre,Hyp2),
         append(Hyp2,Hyp,NewHyp),
         set_ind(1)],
        [NewHyp==>void],
        indirect(Hyp2)).

method(indirect([Goal=>void]),
        Hyp ==> Goal,
        [not matrix(_,_=>void,Goal),          (P4)
         not Goal = void,                     (P5)
         in_ind(0)],                          (P6)
        [set_ind(1)],
        [[Goal=>void|Hyp]==>void],
        indirect([Goal=>void])).
```

The `indirect/1` method implements proof by contradiction. It consists of two clauses, both of which take the current sequent, add the negation of it's goal (the indirect assumption) to the hypothesis list and make the new goal `void`. Both clauses are applicable to sequents of the form `Hyp==>Goal`.

The first clause has the preconditions: (P1) `Goal` is of the form $P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow$ `void`, where `Pre` $= [P_1, \ldots, P_n]$, i.e. the goal is a negated formula; (P2) But `Goal` is not just `void`, i.e. the goal is not already a contradiction. In the second clause (P4) ensures `Goal` is *not* of the form specified in the first clause, and (P5) ensures `Goal` is not `void`. The preconditions (P3) and (P6) check the flag `in_ind/1` is 0, indicating that the method has not been called before in this planning process. This flag is set to 1 in the postconditions of both clauses.

The new hypothesis list in the first clause is the universal variables and preconditions, or the negated `Goal` in the second clause, appended to `Hyp`. In both cases the new goal is `void` and the new hypotheses are passed to the tactic `indirect/1`.

Figure 4.4: The `indirect/1` method

```
method(cardinality(A,T,Sym),
       Hyp ==> Goal,
       [precon_matrix(Vars,Pre=>greater_card(A,T),Goal),  (P1)
        gen_symbol(Sym),                                   (P2)
        not exp_at(Goal,_,Sym),                            (P3)
        not universal_var(Hyp==>Goal,Sym)],                (P4)
       [append(Hyp,Vars,Hyp2),
        append(Hyp2,Pre,NewHyp)],
       [NewHyp ==> (Sym:fun # surj(Sym,T,A))=>void],
       cardinality(A,T,Sym)).
```

The `cardinality/3` method implements Cantor's method of proving the cardinality of set $A$ is greater than $B$ by showing there is no surjective function from $B \to A$. The method can be applied to a sequent of the form `Hyp==>Goal` if the following preconditions are satisfied: (P1) The matrix of `Goal` is of the form `P1=>P2=>...=>Pn=>greater_card(A,B)`, where `Vars` is unified to the quantifiers of `Goal` and `Pre` to the list `[P1,P2,...,Pn]`; (P2) There is a symbol `Sym` that (P3 & P4) does not appear in the input sequent.

The postconditions move the quantifier list `Vars` and the list of preconditions `Pre` to the hypothesis list, unifying it with `NewHyp`. The output sequent states that there is no surjective function `Sym` from set `B` to set `A`. The tactic `cardinality/3` is passed the sets `A` and `B` and the function `Sym`.

Figure 4.5: The `cardinality/3` method

```
method(compose_surj(S1,S2,Comp,A,B,C),
       Hyp ==> Goal,
       [known(S1:fun#surj(S1,A,B),Hyp),     (P1)
        known(S2:fun#surj(S2,B,C),Hyp),     (P2)
        not known(_:fun#surj(_,A,C),Hyp)],  (P3)
       [gen_symbol(Comp),
        append([compose(S2,S1,Comp),Comp:fun#surj(Comp,A,C)],
               Hyp,NewHyp)],
       [NewHyp ==> Goal],
       compose_surj(S1,S2,Comp,A,B,C)).
```

Method compose_surj/6 is shown in figure 4.6. It is applicable to sequents Hyp==>Goal such that: (P1 & P2)) There are lemmas or formulae in the hypothesis list Hyp which state: there exists surjective functions S1:A$\rightarrow$ B and S2:B$\rightarrow$C; (P3) There are no such lemmas or formulae which state: there exists a surjective functions A$\rightarrow$C.

When it is applied, formulae stating Comp is the composition of S1 and S2, and that Comp is a surjective function A$\rightarrow$C are added to the hypothesis list. The goal remains unchanged. The functions S1, S2 and Comp, and the sets A, B and C are passed to the tactic compose_surj/6.

Figure 4.6: The compose_surj/6 method

```
method(hyp_goal,
       Hyp ==> Goal,
       [member(Form,Hyp),
        instantiate(_,Form,Goal,_)],
       [],
       [],
       hyp_goal).
```

The hyp_goal/0 method is a terminating method which applies to sequents of the form Hyp==>Goal if Goal instantiates to some member of Hyp.

Figure 4.7: The hyp_goal/0 method

```
method(imply_goal(Form,NewGoal),
       Hyp ==> Goal,
       [not precon_matrix(_,_=>void,Goal), (P1)
        known(Form,Hyp),                   (P2)
        precon_matrix(Vars,Pre=>Con,Form), (P3)
        matrix(Vars,Con,Con2),             (P4)
        instantiate(Con2,Goal,Ins)],       (P5)
       [(Pre = [Form2]
         orelse
         (append(Pre3,[Con3],Pre),
          precon_matrix([],Pre3=>Con3,Form2))),
        matrix(Vars,Form2,Form3),
        instantiate(Form3,NewGoal,Ins)],
       [Hyp ==> NewGoal],
       imply_goal(Form,NewGoal).
```

Given $A \Rightarrow B$ the `imply_goal/2` method replaces a goal which instantiates to $B$ with a similarly instantiated $A$. It's preconditions for the sequent `Hyp==>Goal` are the following: (P1) `Goal` is not of the form ($P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow$ `void`), i.e. the goal is not a negated formula. These can cause `instantiate/3` in (P5) to loop infinitely. (P2) There is a lemma or formula `Form` in the hypothesis list `Hyp` (P3) of the form ($V_1 \Rightarrow \ldots \Rightarrow V_n \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow$ `Con`), where $V_i$ are variables, `Vars` $= [V_1, \ldots, V_n]$ and `Pre` $= [P_1, \ldots, P_m]$, (P4) such that $V_1 \Rightarrow \ldots \Rightarrow V_n \Rightarrow$ `Con` (P5) instantiates to `Goal`, with instantiation `Ins`.

The postconditions of `imply_goal/2` instantiate the formula ($V_1 \Rightarrow \ldots \Rightarrow V_n \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_m$) with instantiation `Ins` and makes this the output goal `NewGoal`. The formulas `Form` and `NewGoal` are passed to the tactic `imply_goal/2`.

Figure 4.8: The `imply_goal/2` method

```
method(comp_equiv(Goal,comp(F)),
       Hyp ==> Goal,
       [matrix(_,comp(ExpG),Goal),
        known(Form,Hyp),
        matrix(VarF,defn(F,Exp),Form),
        known(comp(F),Hyp),
        matrix(VarF,Exp,MatrixF),
        instantiate(MatrixF,ExpG,_)],
       [],
       [Hyp ==> comp(F)],
       comp_equiv(Goal,comp(F))).

method(comp_equiv(comp(F),NewGoal),
       Hyp ==> comp(F),
       [known(Form,Hyp),
        matrix(Vars,defn(F,Exp),Form)],
       [matrix(Vars,comp(Exp),NewGoal)],
       [Hyp ==> NewGoal],
       comp_equiv(comp(F),NewGoal)).
```

The `comp_equiv/2` method reasons about computability of functions/ expressions, with each clause replacing a goal of the form 'X is computable' with an equivalent 'Y is computable', which should be easier to solve. The action of each clause is shown in figure 4.11. Clause 1 replaces `comp(Exp)` with `comp(F)` if there are lemmas or formulae in the hypotheis list which state: `Exp` defines function `F` and `F` is computable. Thus it identifies an expression that defines a computable function. Clause 2 replaces `comp(F)` with `comp(Exp)` if there are lemmas or formulae in the hypotheis list which state: `Exp` defines function `F`. Thus it replaces a function with it's definition. Clause 3 is given in figure 4.10.

Figure 4.9: The `comp_equiv/2` method: clause 1 and 2

```
method(comp_equiv(Goal,NewGoal),
       Hyp ==> Goal,
       [matrix(Vars,comp(if(Cond,A,B)),Goal),
        nonvar(A),
        nonvar(B)],
       [matrix(Vars,comp(if(Cond,C,D)),NewGoal)],
       [Hyp ==> NewGoal],
       comp_equiv(Goal,NewGoal)).
```

Clause 3 of the `comp_equiv/2` method replaces `comp(if(Cond,A,B))` with `comp(if(Cond,C,D))`, where `A` and `B` are not variables, but `C` and `D` are. Thus it replaces a if-expression with an abstracted version. Clauses 1 and 2 are given in figure 4.9

Figure 4.10: The `comp_equiv/2` method: clause 3

| Clause | X | Y | Conditions |
|---|---|---|---|
| 1 | Exp | F | defn(F,Exp), F is computable |
| 2 | F | Exp | defn(F,Exp) |
| 3 | if(Cond,A,B) | if(Cond,B,C) | A,B not variables,<br>C,D are variables |

Figure 4.11: The clauses of `comp_equiv/2` replace the goal 'X is computable' with 'Y is computable'. defn(A,B) means 'function A is defined as expression B'.

When these methods were initially implemented, it was found that for certain theorems the `indirect/1` method caused the planner to loop. This happens when the method has been applied to give the goal `void`, which is followed by a method that has some other type of output goal. Then `indirect/1` can be applied again, causing the planning process to loop indefinitely. `diagonal_cond/6` is such a method.

To prevent this we created a predicate `in_ind/1` to act as a flag indicating that `indirect/1` had already been applied. At the start of the planning process this is set to 0, and must be 0 for `indirect/1` to be applicable. After it is applied the flag is set to 1. Although this is a rather simplistic solution, as there will be plans which require more than one application of `indirect/1`, we found it was adequate for the purposes of this project[2].

## 4.3 Method Order

The order the methods are stored in the Clam database is an important design consideration, as this decides the order in which the depth-first planner will attempt to apply them. This could significantly affect the efficiency of the planning process. For instance, terminating methods or those with easily tested preconditions should be given higher priority [vanHarmelen 96].

Following this reasoning, `hyp_goal/0` has the highest priority, as it a very simple terminating method. Next are `cardinality/3`, `imply_goal` and `comp_equiv/2` as they demand very specific kinds of input goal, so their preconditions fail easily. The diagonalization methods are also quite specific, so next are `diagonal/5` and `diagonal_cond/6`, the terminating method having higher priority. The methods with quite low priority are: `compose_surj/3`, which is only worth using if nothing else is applicable, as it only affects the hypothesis list; `indirect/1`, because it is the most general method;

[2]note that all of the theorems described in §2.2 need only one indirect argument.

`construct_diag/0` as it only needs to be called from the preconditions of the other diagonalization methods. This gives the following method order:

```
hyp_goal/0
cardinality/3
imply_goal/2
comp_equiv/2
diagonal/5
diagonal_cond/6
compose_surj/3
indirect/1
construct_diag/0
```

## 4.4 Evaluating the Methods

How might we assess these methods? [Bundy 91] suggest several criteria for evaluating methods. These are:

**Correctness** the associated tactic will give the required proof when executed.

**Psychological Validity** the method corresponds to how human mathematicians reason.

**Expectancy** there should be some expectancy of success for the method.

**Generality** the method should account for a wide range of proofs.

**Simplicity** the method should not be too complex.

**Efficiency** the associated tactic should be efficient.

**Parsimony** the fewer methods that account for a collection of proofs the better.

The diagonalization methods described in §4.1 satisfy several of these criteria. They are based on the formalisation of diagonalization, a well-known proof method, and so are psychologically valid and carry an expectancy of success, i.e. if the preconditions are satisfied we can say why the proof will succeed (as described in §2.1). Also, the theory they are based can account for several diagonalization theorems, so this suggests the methods are fairly general.

Of the support methods, `indirect/1` probably satisfies these criteria the best: proof by contradiction is a well-known, simple, general proof method guaranteed to succeed. Although `cardinality/3` and `imply_goal/2` are not as general, they too are fairly simple and based on a standard technique, and so have intuitive appeal. Both have high expectancy - they will succeed if applied.

`compose_surj/6` constructs a surjective composition of two surjective functions, so we can expect it to succeed. But it is very specific, and could perhaps be captured by a more general method that also deals with other transitive relations. `comp_equiv/2` is probably even less general, and also quite complex. It is an attempt to capture the way a human could reason about the computability of functions by looking at common patterns in their structure, but it only deals with a limited number of cases. `hyp_goal/0` is too simple to deserve discussion.

Apart from `compose_surj/6` and `comp_equiv/2`, which are not general enough, the methods fair well against these criteria. However, a full analysis of how well they assist and account for diagonalization proofs would require them to be used to plan such theorems and for these plans to be executed. The next chapter describes empirical testing of their ability to plan diagonalization theorems.

# Chapter 5

# Testing the Methods

The diagonalization methods were tested on seven theorems, corresponding to the results mentioned in §2.2. These are presented in table 5.1, with their formal statements given in table 5.2. Three of the theorems have the enumerability of some set[1] as a precondition. Otherwise, it is fairly easy to see the translation from natural to formal statements.

Of these theorems, natreal, powerset and selfhalt were used for the development of the methods. Once complete, the diagonalization methods were tested with the natfun, natprim, noncomp and halting theorems. During this testing the `compose_surj/3` and `imply_goal/2` methods had to be designed. These were required for the natprim and noncomp theorems respectively. No debugging of the other methods was required during the testing.

For the purposes of this discussion we distinguish two groups of theorems: the 'cardinality' theorems state one set has greater cardinality than another, and so are proved by diagonalization using Cantor's method, i.e. powerset, natreal, natfun and natprim. The 'computability' theorems are concerned with uncomputable functions and are proved by diagonalization using Turing's method, i.e. noncomp, selfhalt and halting.

[1]the existence of a surjective function from the set to $\mathbb{N}$.

| Theorem | Statement |
|---|---|
| powerset | $\mathbb{P}(A)$ has greater cardinality than A. |
| natreal | $\mathbb{R}$ has greater cardinality than $\mathbb{N}$. |
| natfun | The functions $\mathbb{N} \to \mathbb{N}$ has greater cardinality than $\mathbb{N}$. |
| natprim | The functions $\mathbb{N} \to \mathbb{N}$ have greater cardinality than<br>the primitive recursive functions $\mathbb{N} \to \mathbb{N}$. |
| noncomp | There exists an uncomputable function $\mathbb{N} \to \mathbb{N}$. |
| selfhalt | The self-halting problem is undecidable.<br>i.e. If e enumerates $\mathbb{N} \cup \{u\}$ and<br>$\mathrm{f(x)} = \begin{cases} 1 & \text{if } \mathrm{e(x)(x)}=u \\ 0 & \text{otherwise} \end{cases}$ then f is uncomputable. |
| halting | The halting problem is undecidable.<br>i.e. If e enumerates $\mathbb{N} \cup \{u\}$ and<br>$\mathrm{f(x,y)} = \begin{cases} 1 & \text{if } \mathrm{e(x)(y)}=u \\ 0 & \text{otherwise} \end{cases}$ then f is uncomputable. |

Table 5.1: The test theorems

| Theorem | Formal Statement |
|---|---|
| `powerset` | `s:set=>greater_card(pset(s),s)` |
| `natreal` | `greater_card(reals, nats)` |
| `natfun` | `greater_card(natfuns,nats)` |
| `natprim` | `(s:fun#surj(s,nats,natprim))`<br>`=>greater_card(natfuns,natprim)` |
| `noncomp` | `(e:fun#surj(e,nats,natcomp))`<br>`=>(x:fun=>map(x,nats,nats)=>comp(x))=>void` |
| `selfhalt` | `e:fun=>surj(e,nats,natucomp)=>f:fun=>map(f,nats,nats)`<br>`=>(x:nat=>defn(f,if(ap(ap(e,x),x) = u in nat,1,0)))`<br>`=>comp(f)=>void` |
| `halting` | `e:fun=>surj(e,nats,natucomp)=>f:fun=>map(f,nats,nats)`<br>`=>(x:nat=>y:nat=>`<br>`   defn(f,if(ap(ap(e,x),y)=u in nat,1,0)))`<br>`=>comp(f)=>void` |

Table 5.2: Formal statements of the test theorems

During testing Clam was supplied with thirteen lemmas, given in appendix A. Eight were of the simple form '0 is an element of ℕ', while the rest defined the `pset` (powerset) function and the following sets:

- `reals`: $\mathbb{R} = \{\text{f} \mid \text{f: } \mathbb{N} \rightarrow \{0,\ldots,9\}\}$.
- `natfuns`: $\{\text{f} \mid \text{f: } \mathbb{N} \rightarrow \mathbb{N}\}$.
- `natcomp`: $\{\text{f} \mid \text{computable f: } \mathbb{N} \rightarrow \mathbb{N}\}$.
- `natucomp`: $\{\text{f} \mid \text{computable f: } \mathbb{N} \rightarrow \mathbb{N} \cup \{u\}\}$.

Using the depth-first planner, Clam produced plans for all seven theorems. These are shown in figures 5.1, 5.2 and 5.3. Clam planning traces for each theorem are given in appendix B.

## 5.1 Cardinality Plans

As expected, the plans obtained for the cardinality theorems (see figure 5.1) shared a common structure, using the `diagonal/5` method, as no extra membership conditions were attached to the set T (see §4.1). They are all linear[2] plans, with the methods applied in the following order:

1. `cardinality/3`. Goal 'T has greater cardinality than A' becomes 'there is a surjective function A→T'.
2. `indirect/1`. Goal becomes a contradiction.
3. `diagonal/5`. Diagonalization obtains the contradiction.

This corresponds to informal proofs of the theorems. The only deviation from this structure is the `natprim` plan, which applies `compose_surj/6` before `diagonal/5`. This composes two surjective functions to make a third required for diagonalization. There was no search required during planning.

[2] i.e. each tactic has at most one output goal.

`powerset` plan:

```
cardinality(pset(s),s,gen1) then
  indirect([gen1:fun#surj(gen1,s,pset(s))]) then
    diagonal(s,bools,pset(s),gen1,
                          if(ap(ap(gen1,x),x)=0 in nat,1,0))
```

`natreal` plan:

```
cardinality(reals,nats,gen2) then
  indirect([gen2:fun#surj(gen2,nats,reals)]) then
    diagonal(nats,digits,reals,gen2,
                     if(ap(ap(gen2,x),x)=1 in nat,0,1))
```

`natfun` plan:

```
cardinality(natfuns,nats,gen3) then
  indirect([gen3:fun#surj(gen3,nats,natfuns)]) then
    diagonal(nats,nats,natfuns,gen3,
                          if(ap(ap(gen3,x),x)=0 in nat,1,0))
```

`natprim` plan:

```
cardinality(natfuns,natprim,gen4) then
  indirect([gen4:fun#surj(gen4,natprim,natfuns)]) then
   compose_surj(s,gen4,gen5,nats,natprim,natfuns) then
     diagonal(nats,nats,natfuns,gen5,
                          if(ap(ap(gen5,x),x)=0 in nat,1,0))
```

Figure 5.1: Plans obtained for the cardinality theorems

```
noncomp plan:

indirect([e:fun#surj(e,nats,natcomp),
          x:fun=>map(x,nats,nats)=>comp(x)]) then
  diagonal_cond(nats,nats,natcomp,e,
                if(ap(ap(e,x),x)=0 in nat,1,0),comp(d)) then
    imply_goal(x:fun=>map(x,nats,nats)=>comp(x),
               map(d,nats,nats)) then
      hyp_goal


selfhalt plan:

indirect([e:fun,f:fun,surj(e,nats,natucomp),map(f,nats,nats),
          x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0)),
         comp(f)]) then
  diagonal_cond(nats,natu,natucomp,e,
                if(ap(ap(e,x),x)=u in nat,0,u),comp(d)) then
    comp_equiv(comp(d),
               x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)))
                                                        then
      comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)),
                 x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)))
                                                        then
        comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)),
                   comp(f)) then
          hyp_goal
```

Figure 5.2: Plans obtained for the computability theorems `noncomp` and `selfhalt`

## 5.2 Computability Plans

The plans for the computability theorems (see figures 5.2 and 5.3) were also all linear, with the following methods:

1. `indirect/1`. Goal becomes a contradiction.
2. `diagonal_cond/5`. Diagonalization, goal becomes 'd satisfies conditions to be in T'.
3. support methods solve this goal.

In each case the output goal of `diagonal_cond/5` is `comp(d)` - that the diagonal element is computable. In the `noncomp` plan this is solved by the `imply_goal/2` method using the hypothesis that all functions $\mathbb{N} \rightarrow \mathbb{N}$ are computable, then the `hyp_goal/0` method. In the `selfhalt` and `halting`

`halting` plan:

```
indirect([e:fun,f:fun,surj(e,nats,natucomp)),map(f,nats,nats),
          x:nat=>y:nat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0)),
         comp(f)]) then
  diagonal_cond(nats,natu,natucomp,e,
                if(ap(ap(e,x),x)=u in nat,0,u),comp(d)) then
    comp_equiv(comp(d),
              x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)))
                                                        then
      comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)),
                x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)))
                                                        then
        comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)),
                  comp(f)) then
          hyp_goal
```

Figure 5.3: Plan obtained for the computability theorem `halting`

plans the `comp_equiv/2` method is applied several times, until the goal is shown equivalent to one of the hypotheses. Informally, these appear to be good proof plans for the theorems.

No methods failed in planning the `noncomp` theorem. For the `selfhalt` and `halting` plans some search was required (see the planning traces, appendix B), in order to find a diagonal element which could be shown to be computable. In both cases `diagonal_cond/5` was first applied with the diagonal element `if(ap(ap(e,x),x)=0,u,0)`, which failed later in the plan. Upon backtracking it was applied again with `if(ap(ap(e,x),x)=u,0,u)`. This succeeded, being shown computable by `comp_equiv/2` using the hypothesis `comp(if(ap(ap(e,x),x)=u,1,0))`. For each of these diagonal elements, the sequence of goals planned from the goal `comp(d)` during the `selfhalt` theorem is shown in figure 5.4.

Note that the only difference between the goals at the point where the first fails is the value of e(x)(x) in the condition of the if-function. It fails because clause 1 of `comp_equiv/2` cannot match `if(e(x)(x)=0 in nat,X,Y)` with the definition of f, which is `if(e(x)(x)=u in nat,1,0)`. However, the second attempt, `if(e(x)(x)=u in nat,X,Y)`, can be, so `comp_equiv` is applied.

These are the only two diagonal elements `construct_diag/0` (see §4.2) could have constructed, given 0 and u are the only elements of `natu` mentioned by lemmas (see appendix A). The order in which the diagonal elements were constructed depends on the order of the lemmas in the Clam database. Swapping the lemmas around would reduce search in the test results, but would essentially be tuning Clam to these particular examples.

A better way to reduce this search is suggested by the fact that the first diagonal element could have been used to finish the proof, because if the `if(e(x)(x)=u in nat,1,0)` is computable, then so is `if(e(x)(x)=0 in nat,X,Y)`. Adapting the `comp_equiv/2` method so that it could reason that

| | |
|---|---|
| comp(d) | comp(d) |
| $\Downarrow$ | $\Downarrow$ |
| ∀x.comp(if(e(x)(x)=0 in nat,u,0)) | ∀x.comp(if(e(x)(x)=u in nat,0,u)) |
| $\Downarrow$ | $\Downarrow$ |
| ∀x.comp(if(e(x)(x)=0 in nat,X,Y)) | ∀x.comp(if(e(x)(x)=u in nat,X,Y)) |
| $\Downarrow$ | $\Downarrow$ |
| Failed | comp(f) |
| | $\Downarrow$ `hyp_goal/0` |
| | Success |

Figure 5.4: Planning `selfhalt` with diagonal element d as if(e(x)(x)=0 in nat,u,0) *(left)* or if(e(x)(x)=u in nat,0,u) *(right)*. Unmarked steps are planned by `comp_equiv/2`.

like this would eliminate the backtracking from the `selfhalt` and `halting` theorems.

## 5.3 Test Conclusions

The diagonalization methods were used to plan proofs successfully for all the test theorems, and the resulting plans corresponded to informal proofs of the theorems. Testing revealed that the `comp_equiv/2` method could be improved, and that this could reduce search in the planning process. There were no problems with any of the other support methods.

Four theorems, natfun, natprim, noncomp and halting, that were not involved in the development of the methods were successfully planned using the diagonalization methods. Also, there were no test theorems that the methods failed to produce plans for. We have included all diagonalization theorems that we are currently aware of, except Gödel's theorem. Therefore we expect the diagonalization methods to be fairly robust with respect to new theorems. Note that although Gödel's theorem was not included in the test set, we know it cannot be captured by the current diagonalization methods. This is discussed further in chapter 8.

During the testing the `compose_surj/3` and `imply_goal/2` support methods had to be introduced to plan the theorems natprim and noncomp. The need for a wider variety of support methods is to be expected as new diagonalization theorems are attempted, as the reasoning they are required to do depends so much on the nature of the mathematical domain and specific characteristics of the theorem. It is hoped that as the methods are tried against more theorems, fewer support methods will need to be added. Ideally, we would attain a general purpose set of methods which would support any kind of non-diagonalization reasoning that could be involved in diagonalization proofs. Therefore it is no surprise that currently new theorems often require new support methods, and we can anticipate this being the case until the set of support methods is more comprehensive.

As shown in figure 5.1, the cardinality theorems all have a very similar structure. This suggests that a 'super-method' could be used to guide the use of the methods in theorems of this type. Such a method is unnecessary with the current set of methods, because these plans can already be obtained without any search. But with an extended set of methods search could be required for these plans, and so a 'super-method' for the cardinality theorems would be useful in reducing this search.

That the methods produced reasonable plans and are quite robust is a validation of their design. However, it remains to be seen if these can plans can be successfully used to guide proofs of the theorems. An informal assessment of this is the subject of the next chapter.

# Chapter 6

# Informal Proofs

The ultimate test of the diagonalization methods would be to use them to guide the Oyster theorem prover to produce formal proofs of the test theorems. Unfortunately, configuring Oyster for this task is beyond the scope of this project. This would involve developing suitable theories about each of the theorem domains, e.g. the real numbers, powersets, computable functions, and proving basic lemmas about them. Also, Oyster tactics would have to be developed for each method. Altogether, this quite a considerable task.

Instead we have written a simple PROLOG program `prove` that generates informal, English proofs from Clam proof plans. The proofs this produced from the plans of chapter 5 was then used to assess the quality of these plans, and so the design of the methods. This approach has been used before in [Free 92].

## 6.1 The Translation Program

The `prove` program translates a Clam plan file into a LaTeX file containing an informal proof based on the plan. The program is summarised below.

Every element of Oyster syntax has a translation to LaTeX code defined by the `translate/2` predicate, e.g. `translate(nats,''$\mathbb{N}$'').`, which means `nats` translates to $\mathbb{N}$. More complex terms are defined recursively. After translating the theorem using `translate_thm` and writing it to an output file, the plan is parsed recursively using `parse_plan/1`. For each method a 'LaTeX tactic' is executed, which translates the arguments and writes some relevant LaTeX code to the output file.

The top-level predicate `prove/2` takes two filenames, reads in the plan from the first (using `extract_plan/2`) then translates to the LaTeX proof, writing it to the second. The translation is done by `latex_plan/2` using the following predicates:

`preamble/1` writes standard start of file LaTeX to the output, with the title given by the only argument.

`theorem/1` retracts all instances of `mention/1`, then translates the argument to LaTeX code using `translate_thm/2`, writing it to the output under the heading 'Theorem'. Finally it calls `check_mentions/0`.

`translate_thm/2` each clause translates part of a theorem to LaTeX code. `translate/1` is used if no clause is specified.

`check_mentions/0` for each instance of `mention(X)` asserted, the definition of object X is written to the output, using `defn/2`.

`proof/1` translates the argument to LaTeX code using `parse_plan/1`, writing it to the output under the heading 'Proof'.

`parse_plan/1` writes the relevant LaTeX code for the first tactic of the argument to the output stream, then recurses on the rest. Each clause deals with a different tactic. Formulae passed to the tactic are translated to using `translate/2`.

`translate/2` each clause translates part of a formulae in Clam syntax to LaTeX code. For certain objects X, `mention(X)` is asserted, so that `check_mentions/0` can include the definition of X in the LaTeX file.

`endfile/1` writes standard end of file LaTeX to the output.

New clauses can be added to the predicates `translate/2`, `translate_thm/2` and `parse_plan/1` to handle new expressions and tactics.

## 6.2 Proofs of Test Theorems

Using the `prove` program informal proofs were generated for all the test theorems in §5, and they are presented in appendix D. We believe they are all correct, well-structured diagonalization arguments. This suggests thats the methods capture the informal notion of diagonalization well.

These proofs also show that the informal 'LaTeX tactics' that were written for the methods successfully performed fragments of informal reasoning. This suggests that the corresponding formal Oyster tactics could be successfully written, and hence that the diagonalization methods are correct. Therefore the informal proofs provide good evidence that the proof plans could guide Oyster through the formal proofs.

The success of these informal proofs also raises the possibility of extending the `prove` program to cope with other domains. The program can easily be extended to handle new methods and syntax by adding new clause to `parse_plan/1` and `translate/2`. This suggests a general purpose program could be written to translate any Clam plan into an informal LaTeX proof, where the user wrote their own 'informal tactics'. Such a program would be a valuable addition to the Clam system, enabling the user to evaluate the success of their own methods without having to develop tactics and domain theories for Oyster.

# Chapter 7

# Related Work

In this chapter, we compare our diagonalization methods with those developed for Omega by the Saarbrücken group (see §3.2). Direct comparison is made difficult by differences in method representation between the two systems: declaratively in Omega, procedurally in Clam. Another difficulty is there has been no reports of empirical testing of their methods to date, so they cannot be compared experimentally.

[Huang *et al.* 95a] describes the development of four diagonalization methods: `Diag-1` and `Diag-2` deal with Cantor's powerset and natreal theorems respectively. These are generalised to `Diag-1-2` to cover both proofs, and `Diag-1-2-3` to deal with the halting problem as well. Our diagonalization methods are clearly more general than the first three, in that these Omega methods cannot account for the halting problem.

Therefore our methods are comparable to the `Diag-1-2-3` method. The main difference between them is that `Diag-1-2-3` is more general than `diagonal/5` and `diagonal_cond/6`. For example, the Omega method makes the hypothesis '∃f.F', as opposed to the '∃S.surj(S,A,T)' required for Clam. A lot of information about the proofs has been lost in the Omega approach due to this generalisation. This makes the problem of correctly instantiating the meta-variables during planning a lot harder, although the possibility of storing the bindings for these meta-variables from previous proofs was suggested ([Huang *et al.* 95a], p.16). In fact the `Diag-1-2-3` method was not used in any proof attempts because of this problem [Kerber 97][1]. Our methods can be used to construct proof plans without the need to supply these instantiations, so this currently makes them a more practical approach to planning diagonalization proofs.

However, this generality probably makes `Diag-1-2-3` more flexible. The lack of experimental evidence makes this differcult to assess, but it is expected that a more general notion of diagonalization would make the method more widely applicable. For example, we know our methods are not applicable to Gödel's theorem (see chapter 8). Although the Omega methods were not applied to this result [Kerber 97], their generality suggests it might be possible.

Overall, our diagonalization methods are less general than those developed by the Saarbrücken group, but because of this our methods can actually be used to plan diagonalization proofs. Hence this currently makes them a more practical approach to planning diagonalization.

[1]the Saarbrücken were primarily interested in mechanically adapting methods, rather than developing practical diagonalization proof plans.

# Chapter 8

# Further Work

The work in this project could be extended by improving either the diagonalization or support methods. In this chapter we look at how the diagonalization methods could be generalised to Gödel's theorem, controlling indirect proof and reimplementing some of the support methods as object-level rules.

## 8.1 Gödel's Theorem

Currently the diagonalization methods are unable to produce a plan for Gödel's first incompleteness theorem. This is because they are based on a formalisation of diagonalization (see chapter 2.1) which fails to account for the use of diagonalization required in Gödel's theorem.

To see why this is so, compare our diagonalization argument (see figure 8.1) to the simplified version of Gödel's theorem[1] given in figure 8.2. The diagonalization arguments are numbered (1) to (4) in each figure. They have very similar structures, and we can map certain parts of each argument to each other, the most obvious being the surjective function S and the Gödel numbering g. From this it can be seen that the set A maps to the natural

[1] This proof is adapted from [Kleene 52] and [Boolos & Jeffrey 80]

Goal: A contradiction.

Given:

- T, the set of all functions f:A$\rightarrow$B such that C(f) for some condition C.
- a surjective function S:A$\rightarrow$T.
- a function Z:B$\rightarrow$B with no fixed points, i.e. $\forall$x$\in$B. Z(x)$\neq$x.

Proof:

1) Define the **diagonal element** d(x) = Z(S(x)(x)).
2) Show C(d), and as d:A$\rightarrow$B then d$\in$T.
3) So for some q$\in$A, d = S(q), as S is surjective.
4) Thus S(q)(q) = d(q) = Z(S(q)(q)), which is a contradiction.

Figure 8.1: A diagonalization argument

numbers $\mathbb{N}$, and the set T onto $\mathcal{E}_1$, the set of sentences with one free variable. However, there are differences between the two arguments:

- T is a set of functions, but $\mathcal{E}_1$ is not.
- S(q) is a function which is then evaluated at q, but g(q) is an expression which has q substituted into it.
- the relationship between the diagonal element d and 'g(x) at x' is more complex in Gödel's theorem than our formalisation.
- there is no counterpart to the predicates prov and P in our formalisation.

Because of these differences, our formalisation of diagonalization is unable to account for Gödel's theorem. In future work this formalisation could be generalised to include this result, and the methods adapted accordingly.

Another respect in which Gödel's theorem differs from the other diagonalization proofs is that it is not traditionally proved by contradiction. This

Theorem: $\exists G \in \mathcal{E}. \neg prov(G)$ and $\neg prov(\neg G)$.

Given:

- $\mathcal{E}$, the set of sentences of the formal system $\mathcal{S}$.
- $\mathcal{E}_1 \subset \mathcal{E}$, the set of sentences with one free variable.
- $\mathcal{S}$ is consistent, so $\forall x \in \mathcal{E}. \neg(prov(x)$ and $prov(\neg x))$.
- a Gödel numbering of $\mathcal{E}_1$, a surjective function $g{:}\mathbb{N} \to \mathcal{E}_1$.
- prov(sub(g(x),x)) is expressed in $\mathcal{S}$ by predicate P, so $\forall x \in \mathbb{N}$

$$prov(sub(g(x),x)) \Leftrightarrow prov(P(x)).$$

Proof:

Assume $\forall G \in \mathcal{E}. prov(G)$ or $prov(\neg G)$.

1) Define $d = \neg P(x)$.

2) Now $d \in \mathcal{E}_1$.

3) So for some $q \in \mathbb{N}$, $d = g(q)$, as g is surjective.

4) prov(sub(d,q)) or not prov(sub(d,q)), so:

$$\begin{aligned} prov(sub(d,q)) &\Leftrightarrow prov(sub(g(q),q)) \\ &\Leftrightarrow prov(P(q)) \\ &\Leftrightarrow prov(\neg\neg P(q)) \\ &\Leftrightarrow prov(\neg sub(d,q)), \text{ a contradiction.} \end{aligned}$$

$$\begin{aligned} \text{not } prov(sub(d,q)) &\Leftrightarrow \text{not } prov(sub(g(q),q)) \\ &\Leftrightarrow \text{not } prov(P(q)) \\ &\Leftrightarrow prov(\neg P(q)) \text{ by assumption,} \\ &\Leftrightarrow prov(sub(d,q)), \text{ a contradiction.} \end{aligned}$$

Figure 8.2: A simplified version of Gödel's theorem

is the only example of diagonalization we have encountered that does not take place within the context of an indirect proof[2]. This suggests the diagonalization methods could be generalised to be applicable within 'direct' proofs.

A possible problem with generalising the methods is indicated by the methods developed by the Saarbrücken group. Generalising loses information about the proofs, by replacing objects with generalised variables. Instantiations have to found for these varibales during planning, which makes the process harder. A solution to this might be to have a single very general diagonalization method, and have it's application guided by a set of more specific super-methods that represent the various ways that diagonalization reasoning can be employed.

## 8.2 Indirect Proof

From the support methods, `indirect/1` requires a more sophisticated mechanism to prevent looping than the one implemented in this project. At the moment this limits the planner to a single application of `indirect/1` per planning process, including backtracking. A slightly better mechanism would be to place a formula `ind(X)` in the hypothesis list to indicate that `indirect/1` had been applied X times, with some upper limit placed on this value. In this way an each branch of the search could go up to the limit independently of the others, although this is still a rather ad hoc solution. Perhaps implementing a better method would require an investigation into suitable heuristics for indirect proof.

[2]although indirect reasoning is traditionally used to finish the proof, corresponding to step (4) in figure 8.2

## 8.3 Object-Level Rules

Further improvements could also be made to the methods `comp_equiv/2` and `construct_surj/6`, as they are both too specific to be useful in a more general context than this project.

A possible solution to this which could be explored is to replace some of the support methods with object-level rules. These rules could then be applied using some general purpose methods. Given a rule:

$$\forall \vec{x}.(P \Rightarrow C)$$

then a *backwards-chaining* method could check if $C$ instantiated to the current goal, and if so replace it with a similarly instantiated $P$. Another *forward-chaining* method could check if an instantiated $P$ is true given the current hypothesis list and lemmas, and if so add a similarly instantiated $C$ to the hypothesis list.

For example, the `cardinality/3` method could be replaced a by backwards-chaining method and the object-level rule:

$$\forall a : set.\forall b : set.\Big(\neg\exists s : fun.surj(s, a, b) \Rightarrow greater\_card(b, a)\Big)$$

However, the `cardinality/3` method is also applicable to goals of the form $P_1 \Rightarrow P_2 \Rightarrow \ldots P_n \Rightarrow greater\_card(a, b)$, where it moves the preconditions into the hypothesis list. Therefore another method, *implication introduction*, would be required to change goals of the form $P \Rightarrow C$ to $C$, with $P$ added to the hypothesis list. Using these methods, the `comp_equiv/2` and `construct_surj/6` methods could also potentially be replaced by such rules.

How would replacing the methods `cardinality/3`, `comp_equiv/2` and `construct_surj/6` with the backward-chaining, forward-chaining and implication introduction methods and equivalent object-level rules affect the planning process? As there is already a backwards-chaining support method, `imply_goal/2`, this would reduce the number of support methods by one. In a larger method database the new methods could further reduce the number of methods required as they are considerably more general than the current support methods. Therefore this approach would lead to fewer, more general methods, and solve the problem of lack of generality identified in §4.4.

This approach would increase the number of lemmas in the Clam database, which would make planning less efficient, as some methods have preconditions that check all the lemmas in the database for some property. As the methods are more general, and so more widely applicable, the search involved in planning would also increase. Hence replacing support methods with object-level rules would probably make planning less efficient. We do not know if the effect would be significant, but this could be investigated experimentally.

# Chapter 9

# Conclusions

The diagonalization methods designed and implemented in this project performed very well in experimental testing. They produced reasonable plans and were robust with respect to new theorems. The methods have therefore satisfied the primary aim of this project. As further validation of their design, these methods produced plans that could be used to generate good informal proofs of diagonalization theorems.

The project also required some support methods to be developed, with varying success. Replacing some of these with object-level rules would improve the generality of the methods. Our work has also raised the problem of how indirect proof could be controlled in proof planning.

Currently our methods are unable to handle more general diagonalization arguments, like Gödel's theorem. This is because of the limitations of the formalisation of diagonalization the methods were based on. By generalising this conception of diagonalization, the methods could be extended to such results. The Omega implementation of diagonalization has this greater generality, but this currently makes it a less practical approach. A solution to this might be to have a very general diagonalization method, guided by super-methods that control it's use.

This project has demonstrated that diagonalization reasoning can be implemented successfully in a proof planning system. These methods are also the first application of Clam to area of mathematics that has not involved the use of rippling. This suggests that the proof planning approach that has worked well with inductive proof can be extended successfully to new mathematical domains which are significantly different from those explored so far.

# Bibliography


[Boolos & Jeffrey 80] G. Boolos and R. Jeffrey. *Computability and Logic.* Cambridge University Press, 1980.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[Bundy 91] A. Bundy. A science of reasoning. In J.L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[Bundy *et al.* 90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990.

[Bundy *et al.* 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7, 1991.

[Bundy *et al.* 93] 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.

[Cutland 80] N.J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[Free 92] N. Free. Summing series using proof plans. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1992.

[Hofstadter 79] D. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Harvester Press, 1979.

[Huang *et al.* 94] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. Omega–MKRP: A proof development environment. In A. Bundy, editor, *Proceedings of the 12th CADE*, pages 788–792. Springer-Verlag, 1994.

[Huang *et al.* 95a] X. Huang, M. Kerber, and L. Cheikhrouhou. Adapting the diagonalization method by reformulations. In A. Levy and P. Nayak, editors, *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation*. Ville d'Esterel, Canada, 1995.

[Huang *et al.* 95b] X. Huang, M. Kerber, and L. Cheikhrouhou. Adaption of declaratively represented methods in proof planning. SEKI report SR-95-12, Universitat des Saarlandes, Saarbrucken, Germany, 1995.

[Kerber 97] M. Kerber, May 1997. Personal communication.

[Kleene 52] S.C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

[Melis 94] E. Melis. Representing and reformulating diagonalization methods. Technical report CMU-CS-94-174, Carnegie Mellon University, Pittsburgh, USA, 1994.

[vanHarmelen 96] F. van Harmelen. The Clam proof-planner, user manual and programmer manual (version 2.5). Technical report, Mathematical Reasoning Group, Dept. of Artificial Intelligence, Edinburgh, 1996.

[Walsh *et al.* 92] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

# Appendix A

# Lemmas

Table A.1 shows the lemmas that were provided for planning the test theorems.

| Lemma | Formal Statement |
|---|---|
| nat1 | el(0,nats) |
| nat2 | el(1,nats) |
| bool1 | el(0,bools) |
| bool2 | el(1,bools) |
| digit1 | el(0,digits) |
| digit2 | el(1,digits) |
| natu1 | el(u,natu) |
| natu2 | el(0,natu) |
| pset_def | y:set=>x:fun=>el(x,pset(y))<=>map(x,y,bools) |
| reals_def | x:fun=>el(x,reals)<=>map(x,nats,digits) |
| natfuns_def | x:fun=>el(x,natfuns)<=>map(x,nats,nats) |
| natcomp_def | x:fun=>el(x,natcomp)<=>(map(x,nats,nats)#comp(x)) |
| natucomp_def | x:fun=>el(x,natucomp)<=>(map(x,nats,natu)#comp(x)) |

Table A.1: Lemmas provided for planning

# Appendix B

# Clam Planning Traces

## B.1 natreal

```
DEPTH: 0
==>greater_card(reals,nats)
SELECTED METHOD at depth 0: cardinality(reals,nats,gen1)
|DEPTH: 1
|==>(gen1:fun#surj(gen1,nats,reals))=>void
|SELECTED METHOD at depth 1: indirect(...)
||DEPTH: 2
||gen1:fun#surj(gen1,nats,reals)
||==>void
||TERMINATING METHOD at depth 2: diagonal(nats,digits,reals,
gen1,if(ap(ap(gen1,x),x)=0 in nat,1,0))
Clam INFO: Time taken for natreal is 100 milliseconds.
cardinality(reals,nats,gen1) then
  indirect(...) then
    diagonal(nats,digits,reals,gen1,
             if(ap(ap(gen1,x),x)=0 in nat,1,0))
```

## B.2 powerset

```
DEPTH: 0
==>s:set=>greater_card(pset(s),s)
SELECTED METHOD at depth 0: cardinality(pset(s),s,gen2)
|DEPTH: 1
|s:set
|==>(gen2:fun#surj(gen2,s,pset(s)))=>void
```

```
|SELECTED METHOD at depth 1: indirect(...)
||DEPTH: 2
||gen2:fun#surj(gen2,s,pset(s))
||s:set
||==>void
||TERMINATING METHOD at depth 2: diagonal(s,bools,pset(s),gen2,
if(ap(ap(gen2,x),x)=0 in nat,1,0))
Clam INFO: Time taken for powerset is 133 milliseconds.
cardinality(pset(s),s,gen2) then
  indirect(...) then
    diagonal(s,bools,pset(s),gen2,
             if(ap(ap(gen2,x),x)=0 in nat,1,0))
```

## B.3 natfun

```
DEPTH: 0
==>greater_card(natfuns,nats)
SELECTED METHOD at depth 0: cardinality(natfuns,nats,gen3)
|DEPTH: 1
|==>(gen3:fun#surj(gen3,nats,natfuns))=>void
|SELECTED METHOD at depth 1: indirect(...)
||DEPTH: 2
||gen3:fun#surj(gen3,nats,natfuns)
||==>void
||TERMINATING METHOD at depth 2: diagonal(nats,nats,natfuns,
gen3,if(ap(ap(gen3,x),x)=0 in nat,1,0))
Clam INFO: Time taken for natfun is 100 milliseconds.
cardinality(natfuns,nats,gen3) then
  indirect(...) then
    diagonal(nats,nats,natfuns,gen3,
             if(ap(ap(gen3,x),x)=0 in nat,1,0))
```

## B.4 natprim

```
DEPTH: 0
==>(s:fun#surj(s,nats,natprim))=>greater_card(natfuns,natprim)
SELECTED METHOD at depth 0: cardinality(natfuns,natprim,gen4)
|DEPTH: 1
|s:fun#surj(s,nats,natprim)
|==>(gen4:fun#surj(gen4,natprim,natfuns))=>void
|SELECTED METHOD at depth 1: indirect(...)
||DEPTH: 2
```

```
||gen4:fun#surj(gen4,natprim,natfuns)
||s:fun#surj(s,nats,natprim)
||==>void
||SELECTED METHOD at depth 2: compose_surj(s,gen4,gen5,nats,
natprim,natfuns)
|||DEPTH: 3
|||compose(gen4,s,gen5)
|||gen5:fun#surj(gen5,nats,natfuns)
|||gen4:fun#surj(gen4,natprim,natfuns)
|||s:fun#surj(s,nats,natprim)
|||==>void
|||TERMINATING METHOD at depth 3: diagonal(nats,nats,natfuns,
gen5,if(ap(ap(gen5,x),x)=0 in nat,1,0))
Clam INFO: Time taken for natprim is 400 milliseconds.
cardinality(natfuns,natprim,gen4) then
  indirect(...) then
    compose_surj(s,gen4,gen5,nats,natprim,natfuns) then
      diagonal(nats,nats,natfuns,gen5,
               if(ap(ap(gen5,x),x)=0 in nat,1,0))
```

## B.5 noncomp

```
DEPTH: 0
==>(e:fun#surj(e,nats,natcomp))=> (x:fun=>map(x,nats,nats)
=>comp(x))=>void
SELECTED METHOD at depth 0: indirect(...)
|DEPTH: 1
|e:fun#surj(e,nats,natcomp)
|x:fun=>map(x,nats,nats)=>comp(x)
|==>void
|SELECTED METHOD at depth 1: diagonal_cond(nats,nats,natcomp,e,
if(ap(ap(e,x),x)=0 in nat,1,0),comp(d))
||DEPTH: 2
||d:fun
||map(d,nats,nats)
||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,1,0))
||e:fun#surj(e,nats,natcomp)
||x:fun=>map(x,nats,nats)=>comp(x)
||==>comp(d)
||SELECTED METHOD at depth 2: imply_goal(x:fun=>map(x,nats,nats)
=>comp(x),map(d,nats,nats))
|||DEPTH: 3
|||d:fun
```

```
|||map(d,nats,nats)
|||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,1,0))
|||e:fun#surj(e,nats,natcomp)
|||x:fun=>map(x,nats,nats)=>comp(x)
|||==>map(d,nats,nats)
|||TERMINATING METHOD at depth 3: hyp_goal
Clam INFO: Time taken for noncomp is 284 milliseconds.
indirect(...) then
  diagonal_cond(nats,nats,natcomp,e,
                if(ap(ap(e,x),x)=0 in nat,1,0),comp(d)) then
    imply_goal(x:fun=>map(x,nats,nats)=>comp(x),
               map(d,nats,nats)) then
      hyp_goal
```

## B.6 selfhalt

```
DEPTH: 0
==>e:fun=>surj(e,nats,natucomp)=>f:fun=>map(f,nats,nats)=>
(x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0)))=>comp(f)=>void
SELECTED METHOD at depth 0: indirect(...)
|DEPTH: 1
|e:fun
|f:fun
|surj(e,nats,natucomp)
|map(f,nats,nats)
|x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
|comp(f)
|==>void
|SELECTED METHOD at depth 1: diagonal_cond(nats,natu,natucomp,e,
if(ap(ap(e,x),x)=0 in nat,u,0),comp(d))
||DEPTH: 2
||d:fun
||map(d,nats,natu)
||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
||e:fun
||f:fun
||surj(e,nats,natucomp)
||map(f,nats,nats)
||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
||comp(f)
||==>comp(d)
||SELECTED METHOD at depth 2: comp_equiv(comp(d),x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,u,0)))
```

```
|||DEPTH: 3
|||d:fun
|||map(d,nats,natu)
|||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
|||e:fun
|||f:fun
|||surj(e,nats,natucomp)
|||map(f,nats,nats)
|||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
|||comp(f)
|||==>x:nat=>comp(if(ap(ap(e,x),x)=0 in nat,u,0))
|||SELECTED METHOD at depth 3: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,u,0)),x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,_13955,_13956)))
||||DEPTH: 4
||||d:fun
||||map(d,nats,natu)
||||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
||||e:fun
||||f:fun
||||surj(e,nats,natucomp)
||||map(f,nats,nats)
||||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
||||comp(f)
||||==>x:nat=>comp(if(ap(ap(e,x),x)=0 in nat,_13955,_13956))
||||FAILED at depth 4
|||FAILED at depth 3
||FAILED at depth 2
|SELECTED METHOD at depth 1: diagonal_cond(nats,natu,natucomp,e,
if(ap(ap(e,x),x)=u in nat,0,u),comp(d))
||DEPTH: 2
||d:fun
||map(d,nats,natu)
||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
||e:fun
||f:fun
||surj(e,nats,natucomp)
||map(f,nats,nats)
||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
||comp(f)
||==>comp(d)
||SELECTED METHOD at depth 2: comp_equiv(comp(d),x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,0,u)))
```

```
|||DEPTH: 3
|||d:fun
|||map(d,nats,natu)
|||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
|||e:fun
|||f:fun
|||surj(e,nats,natucomp)
|||map(f,nats,nats)
|||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
|||comp(f)
|||==>x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u))
|||SELECTED METHOD at depth 3: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,0,u)),x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,_13955,_13956)))
||||DEPTH: 4
||||d:fun
||||map(d,nats,natu)
||||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
||||e:fun
||||f:fun
||||surj(e,nats,natucomp)
||||map(f,nats,nats)
||||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
||||comp(f)
||||==>x:nat=>comp(if(ap(ap(e,x),x)=u in nat,_13955,_13956))
||||SELECTED METHOD at depth 4: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,1,0)),comp(f))
|||||DEPTH: 5
|||||d:fun
|||||map(d,nats,natu)
|||||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
|||||e:fun
|||||f:fun
|||||surj(e,nats,natucomp)
|||||map(f,nats,nats)
|||||x:nat=>defn(f,if(ap(ap(e,x),x)=u in nat,1,0))
|||||comp(f)
|||||==>comp(f)
|||||TERMINATING METHOD at depth 5: hyp_goal
Clam INFO: Time taken for selfhalt is 1517 milliseconds.
indirect(...) then
  diagonal_cond(nats,natu,natucomp,e,
              if(ap(ap(e,x),x)=u in nat,0,u),comp(d)) then
```

```
    comp_equiv(comp(d),
               x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u))) then
      comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)),
                 x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0))) then
        comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)),
                   comp(f)) then
          hyp_goal
```

## B.7 halting

```
DEPTH: 0
==>e:fun=>surj(e,nats,natucomp)=>f:fun=>map(f,nats,nats)=>
(x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0)))=>
comp(f)=>void
SELECTED METHOD at depth 0: indirect(...)
|DEPTH: 1
|e:fun
|f:fun
|surj(e,nats,natucomp)
|map(f,nats,nats)
|x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
|comp(f)
|==>void
|SELECTED METHOD at depth 1: diagonal_cond(nats,natu,natucomp,e,
if(ap(ap(e,x),x)=0 in nat,u,0),comp(d))
||DEPTH: 2
||d:fun
||map(d,nats,natu)
||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
||e:fun
||f:fun
||surj(e,nats,natucomp)
||map(f,nats,nats)
||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
||comp(f)
||==>comp(d)
||SELECTED METHOD at depth 2: comp_equiv(comp(d),x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,u,0)))
|||DEPTH: 3
|||d:fun
|||map(d,nats,natu)
|||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
|||e:fun
```

```
|||f:fun
|||surj(e,nats,natucomp)
|||map(f,nats,nats)
|||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
|||comp(f)
|||==>x:nat=>comp(if(ap(ap(e,x),x)=0 in nat,u,0))
|||SELECTED METHOD at depth 3: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,u,0)),x:nat=>
comp(if(ap(ap(e,x),x)=0 in nat,_13967,_13968)))
||||DEPTH: 4
||||d:fun
||||map(d,nats,natu)
||||x:nat=>defn(d,if(ap(ap(e,x),x)=0 in nat,u,0))
||||e:fun
||||f:fun
||||surj(e,nats,natucomp)
||||map(f,nats,nats)
||||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
||||comp(f)
||||==>x:nat=>comp(if(ap(ap(e,x),x)=0 in nat,_13967,_13968))
||||FAILED at depth 4
|||FAILED at depth 3
||FAILED at depth 2
|SELECTED METHOD at depth 1: diagonal_cond(nats,natu,natucomp,e,
if(ap(ap(e,x),x)=u in nat,0,u),comp(d))
||DEPTH: 2
||d:fun
||map(d,nats,natu)
||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
||e:fun
||f:fun
||surj(e,nats,natucomp)
||map(f,nats,nats)
||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
||comp(f)
||==>comp(d)
||SELECTED METHOD at depth 2: comp_equiv(comp(d),x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,0,u)))
|||DEPTH: 3
|||d:fun
|||map(d,nats,natu)
|||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
|||e:fun
```

```
|||f:fun
|||surj(e,nats,natucomp)
|||map(f,nats,nats)
|||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
|||comp(f)
|||==>x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u))
|||SELECTED METHOD at depth 3: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,0,u)),x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,_13967,_13968)))
||||DEPTH: 4
||||d:fun
||||map(d,nats,natu)
||||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
||||e:fun
||||f:fun
||||surj(e,nats,natucomp)
||||map(f,nats,nats)
||||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
||||comp(f)
||||==>x:nat=>comp(if(ap(ap(e,x),x)=u in nat,_13967,_13968))
||||SELECTED METHOD at depth 4: comp_equiv(x:nat=>
comp(if(ap(ap(e,x),x)=u in nat,1,0)),comp(f))
|||||DEPTH: 5
|||||d:fun
|||||map(d,nats,natu)
|||||x:nat=>defn(d,if(ap(ap(e,x),x)=u in nat,0,u))
|||||e:fun
|||||f:fun
|||||surj(e,nats,natucomp)
|||||map(f,nats,nats)
|||||x:pnat=>y:pnat=>defn(f,if(ap(ap(e,x),y)=u in nat,1,0))
|||||comp(f)
|||||==>comp(f)
|||||TERMINATING METHOD at depth 5: hyp_goal
Clam INFO: Time taken for halting is 1683 milliseconds.
indirect(...) then
  diagonal_cond(nats,natu,natucomp,e,
                if(ap(ap(e,x),x)=u in nat,0,u),comp(d)) then
    comp_equiv(comp(d),
               x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u))) then
      comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,0,u)),
                 x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0))) then
        comp_equiv(x:nat=>comp(if(ap(ap(e,x),x)=u in nat,1,0)),
```

```
          comp(f)) then
hyp_goal
```

# Appendix C

# Extra Method Language

```
%
% known/2
%

known(Form,Hyp):-
                member(Form,Hyp).

known(Form,_):-
                theorem(_,Form).


%
% gen_symbol/1
%

:- dynamic(gen_symbol_count/1).

gen_symbol_count(0).

gen_symbol(X):- gen_symbol_count(N),
                member(Inc,[1]),  % Backtracking hack, lengthen list.
                M is N + Inc,
                numeral(M,Nlist),
                append([103,101,110],Nlist,L),
                name(X, L).
```

```
numeral(M,Nlist):-
                retract(gen_symbol_count(_)),
                assert(gen_symbol_count(M)),
                build_numeral(M,Nlist).

build_numeral(0,[]) :- !.

build_numeral(N,Nlist):-
                U is N mod 10,
                M is U + 48,
                N2 is (N - U)//10,
                build_numeral(N2,N2list),
                append(N2list,[M],Nlist).


%
% istype
%

istype(u,nat).

istype(0,nat).

istype(s(X),nat):-
                istype(X,nat).


%
% set_ind/1
%

:- dynamic(in_ind/1).
:- assert(in_ind(0)).

set_ind(1):-
                in_ind(1).

set_ind(1):-
                retract(in_ind(0)),
                assert(in_ind(1)).
```

```
set_ind(0):-
                in_ind(0).

set_ind(0):-
                retract(in_ind(1)),
                assert(in_ind(0)).
```

# Appendix D

# Informal Proofs

The following pages contain the informal proofs of the test theorems, generated by the `prove` program (see §6).

# The powerset theorem

## Theorem

Given:

- s is a set

then $\mathbb{P}$(s) has greater cardinality than s

## Proof

To prove $\mathbb{P}$(s) has greater cardinality than s we show that there is no surjective function $\beta$ from s to $\mathbb{P}$(s).

Suppose this was not the case.

Define $\mathrm{d(x)} = \begin{cases} 1 & \text{if } \beta(\mathrm{x})(\mathrm{x}) = 0 \\ 0 & \text{otherwise} \end{cases}$

Now d$\in\mathbb{P}$(s) as d: s$\to\{0,1\}$.

So for some q$\in$s, d = $\beta$(q), as $\beta$ is surjective.

$$\text{Thus } \mathrm{d(q)} = \begin{cases} 1 & \text{if } \beta(\mathrm{q})(\mathrm{q}) = 0 \\ 0 & \text{otherwise} \end{cases}$$
$$= \begin{cases} 1 & \text{if } \mathrm{d(q)} = 0 \\ 0 & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.

# The natreal theorem

## Theorem

$\mathbb{R}$ has greater cardinality than $\mathbb{N}$

## Proof

To prove $\mathbb{R}$ has greater cardinality than $\mathbb{N}$ we show that there is no surjective function $\alpha$ from $\mathbb{N}$ to $\mathbb{R}$.

Suppose this was not the case.

Define $\mathrm{d(x)} = \begin{cases} 0 & \text{if } \alpha(\mathrm{x})(\mathrm{x}) = 1 \\ 1 & \text{otherwise} \end{cases}$

Now $\mathrm{d}\in\mathbb{R}$ as d: $\mathbb{N}\rightarrow\{0,\ldots,9\}$.

So for some $\mathrm{q}\in\mathbb{N}$, $\mathrm{d} = \alpha(\mathrm{q})$, as $\alpha$ is surjective.

$$\begin{aligned} \text{Thus } \mathrm{d(q)} &= \begin{cases} 0 & \text{if } \alpha(\mathrm{q})(\mathrm{q}) = 1 \\ 1 & \text{otherwise} \end{cases} \\ &= \begin{cases} 0 & \text{if } \mathrm{d(q)} = 1 \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

This is a contradiction, hence the result.

Q.E.D.

# The natfun theorem

## Theorem

$\mathcal{F}$ has greater cardinality than $\mathbb{N}$

$\mathcal{F} = \{\mathrm{f} \mid \mathrm{f}\colon \mathbb{N} \to \mathbb{N}\}$

## Proof

To prove $\mathcal{F}$ has greater cardinality than $\mathbb{N}$ we show that there is no surjective function $\gamma$ from $\mathbb{N}$ to $\mathcal{F}$.

Suppose this was not the case.

Define $\mathrm{d}(\mathrm{x}) = \begin{cases} 1 & \text{if } \gamma(\mathrm{x})(\mathrm{x}) = 0 \\ 0 & \text{otherwise} \end{cases}$

Now $\mathrm{d} \in \mathcal{F}$ as $\mathrm{d}\colon \mathbb{N} \to \mathbb{N}$.

So for some $\mathrm{q} \in \mathbb{N}$, $\mathrm{d} = \gamma(\mathrm{q})$, as $\gamma$ is surjective.

$$\text{Thus } \mathrm{d}(\mathrm{q}) = \begin{cases} 1 & \text{if } \gamma(\mathrm{q})(\mathrm{q}) = 0 \\ 0 & \text{otherwise} \end{cases}$$
$$= \begin{cases} 1 & \text{if } \mathrm{d}(\mathrm{q}) = 0 \\ 0 & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.

# The natprim theorem

## Theorem

Given:

- s is a function and s: $\mathbb{N} \to \mathcal{P}$ is surjective

then $\mathcal{F}$ has greater cardinality than $\mathcal{P}$

$\mathcal{F} = \{f \mid f\colon \mathbb{N} \to \mathbb{N}\}$
$\mathcal{P} = \{f \mid \text{primitive recursive } f\colon \mathbb{N} \to \mathbb{N}\}$

## Proof

To prove $\mathcal{F}$ has greater cardinality than $\mathcal{P}$ we show that there is no surjective function $\delta$ from $\mathcal{P}$ to $\mathcal{F}$.

Suppose this was not the case.

Both s: $\mathbb{N}\to\mathcal{P}$ and $\delta$: $\mathcal{P}\to\mathcal{F}$ are surjective, so $\varepsilon = \delta \circ s\colon \mathbb{N}\to\mathcal{F}$ is also surjective.

Define $d(x) = \begin{cases} 1 & \text{if } \varepsilon(x)(x) = 0 \\ 0 & \text{otherwise} \end{cases}$

Now $d\in\mathcal{F}$ as d: $\mathbb{N}\to\mathbb{N}$.

So for some $q\in\mathbb{N}$, $d = \varepsilon(q)$, as $\varepsilon$ is surjective.

$$\text{Thus } d(q) = \begin{cases} 1 & \text{if } \varepsilon(q)(q) = 0 \\ 0 & \text{otherwise} \end{cases}$$
$$= \begin{cases} 1 & \text{if } d(q) = 0 \\ 0 & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.

# The noncomp theorem

## Theorem

Given:

- e is a function and e: $\mathbb{N} \to \mathcal{C}$ is surjective

then ($\forall$x if x: $\mathbb{N} \to \mathbb{N}$ then x is computable) is false

$\mathcal{C} = \{\text{f} \mid \text{computable f: } \mathbb{N} \to \mathbb{N}\}$

## Proof

Suppose this was not the case.

Define $\text{d(x)} = \begin{cases} 1 & \text{if e(x)(x)} = 0 \\ 0 & \text{otherwise} \end{cases}$

Now d$\in\mathcal{C}$ as:

1. d: $\mathbb{N}\to\mathbb{N}$ and
2. d is computable,
   because $\forall$x if x: $\mathbb{N} \to \mathbb{N}$ then x is computable and d: $\mathbb{N} \to \mathbb{N}$.

So for some q$\in\mathbb{N}$, d = e(q), as e is surjective.

$$\text{Thus d(q)} = \begin{cases} 1 & \text{if e(q)(q)} = 0 \\ 0 & \text{otherwise} \end{cases}$$
$$= \begin{cases} 1 & \text{if d(q)} = 0 \\ 0 & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.

# The selfhalt theorem

## Theorem

Given:

- e is a function
- f is a function
- e: $\mathbb{N} \to \mathcal{C}_u$ is surjective
- f: $\mathbb{N} \to \mathbb{N}$
- $\forall$x f $= \begin{cases} 1 & \text{if e(x)(x) = u} \\ 0 & \text{otherwise} \end{cases}$

then (f is computable) is false

$\mathcal{C}_u = \{\text{f} \mid \text{computable f: } \mathbb{N} \to \mathbb{N} \cup \{u\}\}$

## Proof

Suppose this was not the case.

Define d(x) $= \begin{cases} 0 & \text{if e(x)(x) = u} \\ u & \text{otherwise} \end{cases}$

Now d$\in\mathcal{C}_u$ as:

1. d: $\mathbb{N}\to\mathbb{N} \cup \{u\}$ and
2. d is computable,
   $\Leftrightarrow \forall$x $\begin{cases} 0 & \text{if e(x)(x) = u} \\ u & \text{otherwise} \end{cases}$ is computable
   $\Leftrightarrow \forall$x $\begin{cases} 1 & \text{if e(x)(x) = u} \\ 0 & \text{otherwise} \end{cases}$ is computable
   $\Leftrightarrow$ f is computable

So for some q$\in\mathbb{N}$, d = e(q), as e is surjective.

Thus d(q) $= \begin{cases} 0 & \text{if e(q)(q) = u} \\ u & \text{otherwise} \end{cases}$

$$= \begin{cases} 0 & \text{if d(q) = u} \\ u & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.

# The halting theorem

## Theorem

Given:

- e is a function
- f is a function
- e: $\mathbb{N} \to \mathcal{C}_u$ is surjective
- f: $\mathbb{N} \to \mathbb{N}$
- $\forall$x $\forall$y f = $\begin{cases} 1 & \text{if e(x)(y) = u} \\ 0 & \text{otherwise} \end{cases}$

then (f is computable) is false

$\mathcal{C}_u$ = {f | computable f: $\mathbb{N} \to \mathbb{N} \cup \{u\}$}

## Proof

Suppose this was not the case.

Define d(x) = $\begin{cases} 0 & \text{if e(x)(x) = u} \\ u & \text{otherwise} \end{cases}$

Now d$\in\mathcal{C}_u$ as:

1. d: $\mathbb{N}\to\mathbb{N} \cup \{u\}$ and
2. d is computable,
   $\Leftrightarrow \forall$x $\begin{cases} 0 & \text{if e(x)(x) = u} \\ u & \text{otherwise} \end{cases}$ is computable
   $\Leftrightarrow \forall$x $\begin{cases} 1 & \text{if e(x)(x) = u} \\ 0 & \text{otherwise} \end{cases}$ is computable
   $\Leftrightarrow$ f is computable

So for some q$\in\mathbb{N}$, d = e(q), as e is surjective.

Thus d(q) = $\begin{cases} 0 & \text{if e(q)(q) = u} \\ u & \text{otherwise} \end{cases}$

$$= \begin{cases} 0 & \text{if d(q) = u} \\ u & \text{otherwise} \end{cases}$$

This is a contradiction, hence the result.

Q.E.D.