On the Use of Dependency Tracking in Theorem Proving

Fiona McNeill

MSc in Artificial Intelligence
Division of Informatics
University of Edinburgh
2000
Abstract

Tracking dependencies within a theory is an important part of theorem proving and theory maintenance. If changes are made to a theory, dependency information will facilitate the tracking of the effects of this change throughout the theory, thereby enabling it to be updated accordingly.

There is currently little automated support in theorem provers for dependency tracking. The aim of this project is to implement a system to return dependency information during theorem proving and provide the user with editing assistance to act on this information. This is implemented for the theorem prover Isabelle through the generic user interface Proof General.

Such a system has successfully been implemented and feedback from members of the theorem proving community indicate that it may be of much use. The implementation of this project has also suggested many ideas for further work based on this system.
Acknowledgements

I would like to thank my supervisors, Professor Alan Bundy and Dr Louise Dennis for all their help and advice during the course of the research reported herein. I would also like to thank Dr Jacques Fleuriot and Dr David Aspinall for their time and patience in helping me learn about Isabelle and Proof General. I thank the EPSRC who provided financial support during my year of study.
Contents

1 Introduction .................................................. 1
  1.1 Motives and Objectives .................................. 1
  1.2 Implemented and Future Projects ..................... 3
  1.3 Systems Used ............................................ 4
  1.4 Summary of Results .................................... 4
  1.5 Format of Dissertation ................................ 5

2 Literature Survey ........................................... 7
  2.1 Introduction ............................................. 7
  2.2 Altering the Theory .................................... 8
    2.2.1 Introduction .................................... 8
    2.2.2 Theory Revision and Belief Revision ............ 9
    2.2.3 Non-monotonic and Paraconsistent Logic ........ 13
    2.2.4 Abstraction ..................................... 14
  2.3 Utilising Proofs ...................................... 15
    2.3.1 Reusing Proofs .................................. 15
    2.3.2 Analogy ......................................... 16
2.3.3 Proof Engineering ........................................... 17
2.4 Dealing with Conflict in a Knowledge Base ................. 18
2.5 Dependency Tracking in Theorem Provers .................... 19
2.6 Conclusion ................................................... 20

3 Systems Used ................................................... 21
3.1 Choice of Platform ........................................... 21
3.2 Proof General ................................................ 23
3.2.1 Background ............................................. 23
3.2.2 Extents and Spans ...................................... 24
3.3 Isabelle ..................................................... 26
3.4 Emacs Lisp ................................................... 28

4 Development of System ......................................... 29
4.1 Specification of System ..................................... 29
4.2 Design Decisions .......................................... 31
4.3 Examples .................................................... 33
4.4 Summary ..................................................... 36

5 Implementation .................................................. 37
5.1 Overview of System ........................................ 37
5.2 Retrieving the Information from Isabelle .................. 38
5.3 Setting Variables .......................................... 41
5.4 Setting the Span Properties ................................ 44
5.5 Updating the Theorem and Definition Lists ............... 44
5.6 Defining the Menus ........................................ 46
5.7 Defining the Functions .................................... 48
5.8 Summary ..................................................... 50
6 Evaluation and Testing

6.1 Evaluation of Correctness of System ........................................... 53
6.2 Evaluation of Usefulness and Relevance of System ....................... 56
   6.2.1 Aims of Evaluation ...................................................... 56
   6.2.2 Results of Evaluation .................................................. 57
6.3 Evaluation of Efficiency of System ............................................ 59

7 Related Work

7.1 Cynthia ................................................................................. 62
7.2 KIV ...................................................................................... 65
7.3 Dependence Tracking in Argumentation ...................................... 65
7.4 Isabelle .................................................................................. 67
7.5 Proof General ................................................................. 68
7.6 Conclusion ............................................................................. 68

8 Further Work and Conclusions

8.1 Improvements to DPG .......................................................... 70
   8.1.1 Correcting Existing Problems ........................................... 70
   8.1.2 Adding New Functionality ............................................... 72
8.2 Implementing the Future Project ............................................. 82
8.3 Concluding Remarks ............................................................ 85

Appendices

A Program Code

iv
B  Worked Examples - Example 1  92

C  Worked Examples - Example 2  96

D  Evaluation Questionnaire  99

E  Glossary  101
List of Figures

4.1 Original menus in Proof General .......................... 31

5.1 Interaction of the systems ............................... 39
5.2 Theorem context-sensitive menu ......................... 40
5.3 General menu in theorem file ........................... 41
5.4 Menu in definition file .................................... 42
5.5 Flow of control in Proof General and DPG .............. 48
5.6 Theorem context-sensitive menu .......................... 52
5.7 Theorem file menu ......................................... 52
5.8 Definition file menu ........................................ 52

B.1 Screenshot 1.1 ........................................... 92
B.2 Screenshot 1.2 ........................................... 93
B.3 Screenshot 1.3 ........................................... 93
B.4 Screenshot 1.4 ........................................... 94
B.5  Screenshot 1.5 ................................. 94
B.6  Screenshot 1.6 ................................. 95
C.1  Screenshot 2.1 ................................. 96
C.2  Screenshot 2.2 ................................. 97
C.3  Screenshot 2.3 ................................. 97
C.4  Screenshot 2.4 ................................. 98
C.5  Screenshot 2.5 ................................. 98
List of Tables

6.1 Table of times ................................................. 60
Chapter 1

Introduction

1.1 Motives and Objectives

Automated theorem proving is an important and expanding area within Artificial Intelligence. The majority of automated theorem proving is performed interactively, with systems to build and guide the proof but with high-level decisions being made by human users. The aim of automated theorem provers is thus to make guiding an automated proof as easy as possible for the user. One issue that is a concern for the user is the tracking of dependencies within a theory. A theory consists of definitions, lemmas, axioms, theorems, and so on, all generally interacting in some way to build the overall theory. Thus within a theory there will be lemmas that depend on definitions or axioms, theorems that depend on lemmas, and so on. The aim of dependency tracking is to keep some kind of record of these relationships between different elements of a theory.

Theorem proving is concerned not only with creating and developing proofs,
but also with altering, maintaining and updating proofs. It is during alteration and maintenance that dependency information is particularly important. The most obvious situation where dependency information will be necessary is when a change is made to an object in the theory, such as a definition or a theorem. This will have a reverberating effect throughout the whole theory, and in order to examine which parts of the theory will be affected the user must consider what is dependent on the changed object. In a small theory this may not be hard, but in a large theory it will be very difficult for the user to keep track of all the dependencies within the theory, and some kind of automated system to perform this role is therefore desirable.

The aim of this project is to create a system that will track dependencies throughout a theory as it is being proved, and to relate this information back to the user in a helpful manner as and when it is required. The system will also facilitate editing the theory in the light of this dependency information.

This project is part of a wider project, which involves considering the tracking of ‘theories’ in common sense reasoning. A theory formed in common sense reasoning requires continual updating as new information becomes available since, unlike in mathematics or logic, it is rarely feasible to consider all possible factors in advance. Hence in everyday life people often have to informally update their theories by accounting for new information, propagating this new information through their other beliefs, and considering how this change affects the validity of their theory. The idea is to create a formal system that mirrors this process. The formal system has to allow a change to be made, propagate this change through the system by using dependency information and then rerun and patch the proof.
This project is a first step towards the end goal. An area of particular interest and relevance has been chosen for an initial focus. The main motivation for choosing dependency tracking was that this is an area which is not only very useful in practical theorem proving but is also something that is not widely implemented in other systems. Therefore, it was hoped that this project could create a system of real value to the theorem proving community.

This area is a good part of the wider project to develop in isolation, since it is important in conventional interactive theorem proving as well as in a system such as the one proposed above, where the system is self-contained and fully automatic. No attempt was made in this project to automatically propagate changes through the theory as would be necessary for the wider project. This is not necessary in interactive theorem proving because users of interactive systems generally wish to retain some autonomy over their theories. Hence implementing automatic propagation would not be so compatible with the aim to provide a useful tool for existing theorem provers. The aim is rather to provide a purely passive system whose purpose is to provide information to the user to improve their theory maintenance experience. Therefore, automatic propagation was omitted until work could be started on the larger project.

1.2 Implemented and Future Projects

Although most of this project is naturally concerned with the system actually implemented, there is some further discussion on the wider system of the original idea, mostly concerning the theory behind it and the possibility of going on to implement it at a later date. The implemented system has been named DPG to stand for Dependency Proof General, and is referred
to as such throughout the project. The future project is referred to as the common-sense theory maintenance project.

1.3 Systems Used

DPG was integrated into Proof General, which is a generic user interface to various theorem provers written as an extension to Emacs. Thus much of the functionality for editing the theory came from already existing functionality within Proof General, and DPG concentrated on tracking dependence, which is not currently available in Proof General. The theorem prover that was used during this project is Isabelle.

1.4 Summary of Results

A system was implemented that tracked dependency information for theories proved through the Proof General system using the theorem prover Isabelle. Information about this dependence was provided for the user by drop-down and popup menus. These menus gave the user options to highlight all the children or parents of their chosen theorem or definition, or to move to any given child or parent. They also provided the user the option of viewing a list of parent theorems theorems of definitions, or child theorems for any theorem.

A theorem or definition P is defined as a dependency or a parent of a theorem T if P is used in the proof of T. A theorem or definition C is defined as a dependent or child of a theorem or definition T if T is used in the proof of C. The words parent and child are used in preference to the words
dependency and dependent since their meaning is more immediately comprehensible. Definitions do not have parents, since they are taken as facts and depend on nothing.

1.5 Format of Dissertation

All chapters in this document contain information relevant DPG. In addition, chapters 2 and 8 provide some discussion and information concerning the common-sense theory maintenance project.

Chapter 2 presents a literature survey containing a discussion of the theory behind both DPG and the common-sense theory manipulation project.

Chapter 3 contains a discussion of Proof General and Isabelle, the systems used during this project.

Chapter 4 contains a specification for DPG.

Chapter 5 describes how DPG was implemented.

Chapter 6 contains details of the testing and evaluation that was carried out for DPG.

Chapter 7 discusses other systems and work related to DPG.
Chapter 8 contains a discussion of the further work suggested by the development of DPG and a discussion about how the future project may be implemented, and presents some conclusions drawn from the project.
Chapter 2

Literature Survey

2.1 Introduction

This section is intended to provide background information for the whole project; both for the part of it that was implemented and the envisaged common-sense theory manipulation project. Some of the theory behind the latter project has been included so that the project that has been implemented can be understood in its position as a part of a wider project; this consideration has very much driven the development and direction of DPG, and also gives the reader some idea as to the purpose and relevance of the wider project.
2.2 Altering the Theory

2.2.1 Introduction

One of the central ideas behind the project is altering and manipulating theories in the face of new information. This new information may come from a flaw detected in the system, or from outside factors that were not considered, or from increased awareness of the task at hand, and may require the addition, change or removal of elements of the original theory. The main focus is incorporating these changes into the theory whilst changing the original theory as little as possible. In practice it is often not possible to simply incorporate the new information into the old theory without invalidating some of the original theory; in this case the theory must be altered, but in a way that creates a valid theory with as little change as possible. Methods for doing this kind of theory change are considered in the section below on theory and belief revision.

Another consideration is the logic behind altering theories. Since altering the theory may lead to inconsistencies within the theory, and theorems that were originally true may come to be considered as false, the underlying logic of the system cannot be classical. Some alternative logics are discussed.

The problem of altering the theory in the face of new information can also be considered as analogous to abstraction, although this analogy is in reverse. This is because abstraction consists of removing detail from a theory until a basic ‘skeletal’ theory is reached, whereas this project is concerned with incorporating more and more detail as further information is discovered until the theory is much richer than it was originally; if new information is con-
tinually being revealed there may never come a point of completion. This is discussed in the section on abstraction below.

2.2.2 Theory Revision and Belief Revision

Theory revision and belief revision are essentially the same concept; theory revision sometimes refers to a broader subject than belief revision, but in the context of the theory behind this project the two terms are interchangeable, and for convenience will be referred to throughout this section as theory revision. Theory revision is generally concerned with altering a knowledge base. Although theorem proving does not usually include anything that is referred to as the knowledge base, this should simply be taken as referring to the information contained in the theory; in particular the definitions, axioms, lemmas and so on.

Some of the most influential work done in theory revision is by Alchourron, Gardenfors and Makinson (Ourston and Mooney, 1990) in what is generally referred to as the AGM postulate. Here, they state that the three aspects of theory revision are as follows:

1. **expansion** - information is added to the knowledge base, hence expanding it. This is comparable to Tarski’s consequence operations (Schlechta, 1991).

2. **contraction** - information is removed from the knowledge base.

3. **revision** - some information in the knowledge base is altered.

The third item, revision, has also been separated out into two different subsections (Langley et al., 1994). This paper considers all facts in the knowledge base to be nodes and considers their position in the knowledge base
in light of their parents and children. Theory revision is considered in the light of changing links between nodes in the database rather than changing the nodes themselves. The idea of nodes and links is a rather useful way of looking at the knowledge base for the purposes of this project, since if the nodes are considered to be proof objects such as definitions or theorems then the links represent the interdependence between them. The idea of changing links rather than the nodes themselves may also be of use, since if the node is a proof object then changing the node may also change the links, or the dependencies. So for this project, the idea of changing both the nodes and the links is interesting.

The four points are listed below:

1. **Add** (or expand) - a link is created between a node P and a node C, making C the child of P.

2. **Delete** (or contract) - a link is removed between a node P and one of its child nodes C so that they are no longer related.

3. **Move** - removes a node C from its parent P1 and creates a link between C and P2 so that it now has different parents.

4. **Switch** - two child nodes C1 and C2, that both have the same parent P, have their positions switched.

The ideas behind points 3 and 4 do not really cover the idea of revision within the knowledge base for a theorem prover. Altering links or dependencies may create change as suggested in points 3 and 4 but other options may be equally valid, for example removing the dependence between a child and many of its parents and replacing it with a link between the child and a single new parent.
Finding the Minimum Change

A central consideration in theory revision is how to preserve the original theory to the greatest possible extent whilst incorporating the necessary changes. A common approach to this problem is to use hill-climbing techniques to find a point where the changes to the theory are minimal within the locale, hence the minimum change found would not necessarily be the smallest change possible. Some work has been done (Greiner, 1999) investigating the practicality of finding a globally optimal solution but this was found to be computationally intractable. This is certainly an issue that concerns the larger project, since it seems important to keep beliefs as constant as possible. However, it is not one which is much addressed in the dependency tracking project.

Types of Error

Ourston and Mooney (Ourston and Mooney, 1990) classify the kind of errors that occur in theories into two types:

1. **Over-generality** - for example, when an incorrect rule is present or when an existing rule is missing a constraint from its premise. Also covered in this could be failure to identify and distinguish terms; hence the theory is generalised to ignore these terms. Over-generality could also apply to individual functions, which could be too general through not having enough arguments.

2. **Over-specificity** - for example, when a rule in the theory has an additional, incorrect, constraint or when a theory is missing a rule which is necessary. Also, functions could be over-specific through having too many arguments.
Many theory revision systems search for only one of these groups, most commonly over-specificity. The general theme of both DPG and the common-sense theory manipulation project is to search for over-generality errors, but there may be occasions when over-specificity occurs and it would be ideal to incorporate all types of error, since all could occur in theory maintenance.

One major drawback of theory revision systems is that they generally deal with only propositional logic, or in some case Horn clauses. The theory that is processed by a theorem prover may not be represented in this fashion; hence in these cases existing theory revision systems are not likely to be directly useful, rather it is helpful to look at the theory behind them and consider how it may be applicable. In some domains the theory may be expressed in logic and in these cases existing theory revision systems may be more applicable.

**Theory Maintenance Systems**

Practical implementations of theory revision can be found in theory maintenance systems. These are systems that were designed primarily by engineers for practical purposes rather than by logicians in a theoretical manner; hence the theory behind them may not be as neat as would be desirable but they provide a good example of theory revision in practice (Gardenfors and Rott, 1995). These are focussed on the justifications for the beliefs in their knowledge bases. Each belief has a corresponding inlist \( I \) and outlist \( O \). For the belief to be considered true, the beliefs in \( I \) should be considered true and the beliefs in \( O \) should be considered false. Thus when a belief is changed, the impact of this can be traced through the in and out lists in which it appears, which will belong to those beliefs that are dependent on it in some way. This is reminiscent of the work by Mandy Haggith, which is discussed in section
2.4. It is also similar to the approach taken in this project, although here the idea of directly labelling nodes is replaced by tracking the dependence relations of the proof objects.

2.2.3 Non-monotonic and Paraconsistent Logic

In classical logic if a fact is true it cannot later become false; classical logic is designed to be ideal reasoning about mathematical objects. However, mathematical objects are of a manifestly different nature to real world objects, which can rarely be given an ideal representation. In real world cases it is simplistic to consider reasoning as a linear process. New information does not simply add to the existing knowledge base as it would do in mathematics; rather it can sometimes challenge or negate facts that exist in that knowledge base, which is what often happens during theory revision. So the logic must be capable of dealing with contradiction. Contradiction is not allowable in classical reasoning because of the existence of the law *ex falso quodlibet*, which states that from contradiction anything is provable. This is obviously hopeless in a situation where we expect contradiction to occur and to refine the knowledge base rather than collapsing it. Non-monotonic logic is useful here because because it allows originally true facts to become false.

Non-monotonic reasoning is mentioned briefly because it constitutes the underlying logic of theory revision. The link can be seen by considering the changes necessary to a theory K when a set of propositions A is added. The revision necessary to the theory is analogous to determining what follows from A under a suitable non-monotonic inference operation that depends on K (Makinson and Gardenfors, 1991).
Another logic that deals with contradiction is paraconsistent logic. The main difference between non-monotonic logic and paraconsistent logic is that non-monotonic logic can shrink the knowledge base when new information is added by revising previously accepted facts, whereas paraconsistent logic does not delete information involved in contradiction. Rather, it considers contradictory statements to be both true and false, hence preserving the principle of excluded middle whilst allowing contradiction.

Paraconsistent logic can be useful in theory revision, and in particular in the tracking of beliefs, since people often have contradictory beliefs and sometimes these contradictions are perfectly rational. For example, a rational person could write a book after much research and claim a certain set of propositions. Hence they believe that this set of propositions are justified and constitute a valid theory. However, since they are rational, they also believe that no book of any complexity contains only truths, hence they also believe that the set of their propositions is not justified. Thus they rationally believe both that a thing is true and that it is not true (Stanford). This kind of contradiction of belief is expressible in paraconsistent logic.

2.2.4 Abstraction

The purpose of the common-sense theory maintenance project could be considered to be the implementation of ‘anti-abstraction’, or the enrichment of a theory by the incremental addition of further information.

Abstraction was originally used in theorem proving as a powerful heuristic and is also useful because it can provide high level explanations, learn
abstract plans and reason by analogy (Giunchiglia and Walsh, 1992). In semi-automated abstraction, detail is abstracted away from a theory, a proof created for the core structure that remains, and the detail then reinserted incrementally, ensuring at each stage that the proof remains valid. This method is used in the GPS system which is used in problem solving. According to Polya (Polya, 1945) this is a common method amongst human mathematicians. The final stage of this procedure is very similar to the intention behind the future project, where in place of the abstracted detail, new information will be added incrementally whilst attempting to maintain as much of the proof as possible.

2.3 Utilising Proofs

Another relevant aspect of the project is the idea of using existing proofs to act as some kind of guide for a new proof, either because the new proof consists of the old proof with some alteration or because the old proof provides a guideline for the new proof. Some work on these ideas is discussed below.

2.3.1 Reusing Proofs

The cost of developing proofs in formal verification is very high; therefore work has been done on the possibility of reusing parts of proofs to create a similar but altered proof. This is of much relevance to the project; in DPG, the cost involved is largely considered in terms of man hours. Since complex proofs usually consist of fairly simple subproofs, it can be relatively easy to manipulate these. One method of reusing the proof would be to identify those subproofs that are no longer valid and alter these. This can either be
done manually or by using some kind of internal analogy to transform the original subproof into an appropriate new subproof. This is relevant to the project, since theories made up of definitions and theorems are analogous to complex proofs made up of smaller sub-proofs.

There are four main stages in reusing the proof outlined by (Melis and Schairer, 1998):

1. Analyse and store the generalisation of the subproof, which must capture relevant subformulae of a goal.

2. When new subproofs need to be done, find the relevant generalisation and find a mapping from the source to the target. This retrieval can either be done automatically or interactively.

3. The source proof is then derivationally replayed. Decisions in the source problem are replayed if their justification holds in the target problem.

4. Any subproofs left unfinished are derived from first principles.

Thus, although there is still work to be done in producing the new proof, the amount of work is much reduced.

2.3.2 Analogy

Another method of drawing information from previous proofs is analogy. The basic purpose of analogy is to find a mapping between an original object (or proof) and a target object (or proof). Analogy is applicable in wide range of situations, and one of its strengths is that, unlike ordinary pattern matching,
it can map between different domains, including mapping from an known domain to an unknown domain. These strengths are not really relevant for the purposes of DPG, since the mapping will be between very similar objects, i.e. the previous proof and the future proof.

The major drawback of using analogy is that it is computationally expensive. Analogy removes some of the original computational cost by reusing information and engineering a situation where it is not necessary to start a proof from scratch. However, the cost of implementing the analogy is so high that it may well prove more expensive than simply creating a new proof.

However, despite these drawbacks analogy may still be of some use in implementing the common-sense theory manipulation project. The ability to map between different domains may be very useful in formalising common sense reasoning, and the fact that analogy is a very powerful method of pattern matching could be essential in such a broad project.

2.3.3 Proof Engineering

Oliver Pons (Pons, 2000) outlines tools that would be of interest to theorem provers. This is of interest to DPG as these are the kind of tools implemented within this project and, since they provide information about the proof, are certainly of value to someone who wished to alter that proof to incorporate some new information.

Several methods of providing information to the user are outlined. One area that he points out as being relevant is presenting dependency information;
this idea is central to the dependency tracking project. The usefulness of presenting this information in a proof-tree format is discussed; however, the practicality of implementing this for a large theory is not discussed, nor is how this information is to be related to editing the theory in light of this information.

2.4 Dealing with Conflict in a Knowledge Base

Mandy Haggith (Haggith, 1995) discusses the possibility of automating a system which could contain conflicting ideas. The system as discussed is an interactive one where the user acts as a chairman, balancing the counter-arguments against each other. There is no discussion of the possibility of automating this procedure. As discussed in section 2.2.3, the *ex falso quod libet* rule does not allow for conflict. In that section this was overcome by using non-monotonic reasoning; in (Haggith, 1995) the possibility of overcoming this by using meta-level reasoning is discussed.

By creating a meta-level at which to observe the object level arguments it is possible to retain monotonic logic at the object-level, since the conflict can be dealt with at the meta-level. There are two approaches to implementing this: either to abstract from the object-level to the meta-level, called mark-up, or to use the meta-level to guide the formation of the object level. Haggith gives a framework that could be used in either approach, which involves defining meta-level objects, relations between these objects, and meta-level rules. Conflict can then be explored by forming a conflict set which is used to represent those propositions the user is currently interested in. This can be focussed or expanded using various rules. Higher-order meta-level relations
can be defined from the basic meta-level relations.

Whilst these ideas could be of some interest to the project, using non-monotonic reasoning seems to be a more practical approach to the domain of theorem proving than using meta-level reasoning, since theorem provers do not generally allow for meta-level reasoning. However, it is an approach that may be useful in the common-sense theory manipulation project.

2.5 Dependency Tracking in Theorem Provers

Although dependency tracking is an important and sometimes difficult part of theory maintenance, most theorem provers do not support much automatic dependency tracking. Isabelle is capable of tracking dependencies and reporting them back to the user in a fixed way, as discussed in section 7.4. However, this information is not reported back to the user in a very helpful manner and does not facilitate editing the theory on the basis of this information.

The theorem prover Coq has similar facilities to Isabelle in that it can track dependencies to a certain extent but not provide the user with much helpful information relating to this information.

In general, it seems that a few theorem provers have some basic ability to track dependencies through proofs, but this is generally not done in a particularly user-friendly manner.
2.6 Conclusion

This chapter has discussed a range of subjects including theory revision, non-monotonic and paraconsistent logics, reusing proofs, analogy and abstraction. There is also a look at dependency tracking in theorem provers.

The full project is of a very broad and non-specific nature and thus there is an enormous amount of literature that might be of some relevance to it. The above survey is an attempt to identify the key areas of interest, in particular keeping the needs of DPG in mind.

Any plans to implement the common-sense theory manipulation project would need careful study of the literature for clues as to how best to approach this problem.
Chapter 3

Systems Used

3.1 Choice of Platform

The aim of the dependency tracking project is to design a system to provide the user with increased assistance in the experience of editing theorems and their proofs. The idea is to provide functionality so that making changes within developed theories, and using an old proof to form a basis for a new proof, will be easier than it is at the moment. There were therefore decisions to be made as to the environment this additional functionality would be created in. The basic requirement was that there must be some kind of link to one or more theorem provers so that the additional functionality could be used to edit the theory and the proof.

There was initially some discussion regarding the possibility of writing a theorem prover specifically for this project. The advantage of this would be that the theorem prover would be designed around the desired additional functionality, which would ensure that the theorem prover performed the necessary
functions, such as dependency tracking. However, this idea was rejected for three reasons. Firstly, it was desirable that the additional functionality could actually be of use within the theorem proving community upon completion, which was unlikely to be the case if it was based on a specially-developed theorem prover. This is because any theorem prover developed as a brief initial stage in a four month project would of necessity be a toy system; it would not be powerful and robust enough to be of much value to the theorem-proving community. Secondly, if a theorem prover were chosen that was already in common use then it would be less difficult for users to integrate the additional functionality into their regular theorem-proving experience. The third reason was that it seemed an unnecessary diversion in such a short project; it would be preferable to take an existing theorem prover and devote the available coding time to advancing DPG as much as possible.

The next consideration was how the additional functionality would be integrated for use in tandem with the theorem prover. It was desirable for the user to be presented not merely with facts, but with an opportunity to use these facts to update the theory without too much difficulty. A good way of doing this seemed to be to integrate an editor and a theorem prover to make the user experience more dynamic. It was infeasible to create an editor for this task since this would be extremely complicated. Therefore it was necessary to tie an existing editor in with a theorem prover. Dynamic systems that tie editors into theorem provers in order to make theorem alteration easier already exist. It was therefore necessary to decide whether to link an editor to a theorem prover independently or to integrate the additional functionality into an existing intermediary which already makes use of an editor. It was decided that the latter option would be preferable. This is largely because this would allow us to make use of the functionality already existing within this link, and hence save time on coding functionality peripheral to the main
aim of the project. For example, it would not be necessary to build a user interface from scratch, merely to add additional features to an existing one. The other main advantage of this choice was that if the additional functionality was added to a system already in use it would be fairly easy for users of this system to employ the additional functionality during their ordinary theorem proving practice. However, if the additional functionality was in an isolated system it would be harder to persuade users of the theorem prover it was based on to use it.

Much of the discussion above has involved an assumption that the additional functionality would be relevant to only one theorem prover. However, it is possible to make an editor with hooks to several theorem provers, and hence give the editor a greater generality. Another advantage of using an existing intermediary is that this may already be set up to link the editor to several theorem provers.

3.2 Proof General

3.2.1 Background

Proof General is a top-level user interface for the theorem provers Isabelle, Coq and Lego. It is written as an extension to Emacs. Proof General was chosen as it already provided a good editing environment within which the additional functionality could be built. It is in common use within the theorem-proving community so the additional functionality could be accessed by a fairly large group of people. It is already compatible with three theorem provers, Isabelle, Coq and Lego; a fourth, HOL, is in the process of being incorporated and Proof General is designed to be easily customised to other
theorem provers. Hence it is possible to make the additional functionality applicable to a number of theorem provers. In this project the work is concentrated on only one theorem prover, but using Proof General means that the additional functionality may be customised to other theorem provers at a later date without a great deal of additional coding.

The aim of Proof General is to provide an interface for users of theorem provers that will provide extra functionality, for example simplified interaction, script management, toolbars and menus and increased adaptability (Aspinall et al., 2000). Hence it was possible to add DPG as another aspect of the functionality of Proof General. When the file is processed by Proof General, the theory is highlighted in specific colours: blue highlighting indicates that that part of the theory has been processed by Proof General and has been proven by the theorem prover, pink highlighting indicates that that part of the theory is in the queue to be processed and has not yet been passed through the theorem prover.

3.2.2 Extents and Spans

Proof General makes use of objects called spans. XEmacs provides an ‘extent’ first-class data type. GNU Emacs provides similar functionality through a mechanism called ‘overlays’, though the programming interfaces differ somewhat. Proof General provides abstractions called spans to hide the differences between extents and overlays, thereby ensuring that Proof General is compatible with both commonly used forms of Emacs.

A span defines a region of text with associated properties. Emacs is able to handle buffers with multiple spans, so that a buffer can contain any number
of spans, including a single span cover all the text in the buffer. Hence a buffer containing many theorems can be conveniently divided so that each theorem is delineated by a span, and so each theorem can be accessed and referenced independently. Spans are also able to assign certain properties to regions of text. Often one of these properties is a face property, which assigns the span some visual properties such as colour. If the span is assigned a face property that differs from the default face property of the buffer then it will appear visually different to the rest of the buffer. This is useful as it allows highlighting of individual spans. It is possible to create any kind of property at all, but these properties can be divided into two types. One type of property, which could be considered ‘active’, causes actions in other parts of the system. For example, if an input event occurs within a span then the system will automatically check if the span has a ‘keymap’ property. If so, the system will check whether the input event, for example a right mouse click, is bound to a particular function by the keymap property, and if so this function is called. If the span has a property named ‘keymap’ which has been used as an arbitrary name for a non-keymap property, then this could cause confusion in the system. The face property is also an ‘active’ type of property. If a property is given a ‘non-active’ name the system will not expect any particular action to arise from it. Non-active properties are useful for storing information. For example, one could define a property ‘parents’ and set this property to a list of parents. Thus when a user queried the value of this property for a span they would be returned the relevant parents. Any number of properties can be assigned to a span and if the user queries the value of a property within a span where this property does not exist, this property will be returned as nil rather than as a statement that this property does not exist or as an error.

Once Proof General has processed a span, that is its theorem has been proved,
by default that span is set to be read-only and no edits can be made, since once the span’s theorem has been proved there should be no need to alter it. However, the information provided by DPG changes this situation because the dependency information may cause the user to wish to change a theorem, even though that theorem has been proved. Thus the user is likely to want to edit spans that have already been processed. This read-only property of processed spans is controlled by a default setting in Proof General, which is usually set to on. However, when DPG is used it is desirable to change this default so that editing of processed spans is permitted.

3.3 Isabelle

Of the three theorem provers that Proof General is currently adapted to, Isabelle was chosen to concentrate on in this project. The main motivation for this was that Isabelle already has some theory management built in and was capable of tracking dependencies. This is a big advantage as it meant that work could be focussed on the additional functionality rather than on altering the theorem prover. This ability is further discussed in section 7.4. Another influencing factor was that there was Isabelle expertise available within the division should assistance be required.

Isabelle differs from Coq and Lego in that two files are required for each theory. One sets up the background of the theory and contains the definitions and constants on which the theory is based; this has a given name with .thy appended, i.e. theory-name.thy. This file is referred to throughout this document as the definition file. When the definition file is processed by Proof General it contains only one span that covers all the text in the file. The other file contains the theorems and their proofs. This file is referred
to throughout this document as the theorem file and has the same name as
the definition file with .ML appended in place of .thy, i.e. theory-name.ML. The general format of a theorem in Isabelle is as follows: each theorem generally starts with a goal line of the form Goal "statement of goal". The following steps are Isabelle tactics attempting to prove the goal and the final step is a line of the form qed "name". The end of the theorem is signified by the qed line and a span is created around the theorem that starts at the goal line and ends at the qed line. For example:

Goal "inj(Rep_pnat)";
by (rtac inj_inverseI 1);
by (rtac Rep_pnat_inverse 1);
qed "inj_Rep_pnat";

The name that appears in the qed line is set as the name property of the span. The generally uniform layout of theorems within the theorem file mean that it is easy to separate them into individual spans. There are occasions when the layout of the theorem file is not uniform and this can cause problems. This is discussed in sections 6.1 and 8.1.1. Within the theory file there is no such uniform layout such that it is easy to tell where one definition ends and another begins. Hence Proof General does not split the definition file into different spans.

These two files are referred to as the definition and theorem files throughout this document because these names describe their contents in a clear manner to the reader. However, within DPG they are referred to as the thy file and the ML file; for example, the drop-down menus contain the options "Move to thy file" and "Move to ML file". This is because, although these names are less clear to a reader, they will be clearer to a user of the system since the
user will actually be dealing with files named X.thy and X.ML and may not have come across the names definition file and theorem file. Hence to mention definition and theorem files in the menus of DPG could cause confusion.

3.4 Emacs Lisp

Proof General is written in Emacs Lisp as it is an extension to Emacs. Therefore the easiest way of adding extra functionality to it is to write that functionality in Emacs Lisp also.

Emacs Lisp is a dialect of MACLisp. The main difference is that Emacs Lisp has access to many packages which provide Emacs interface components. Another major difference is that whereas recursion is a commonly-used method in Lisp this is not an efficient method in Emacs Lisp and is best replaced by iteration. Other Lisp variants can automatically convert tail recursive functions into their iterative equivalents but Emacs Lisp is interpreted and does not have this ability. Therefore if functions are written tail recursively the stack depth builds up rapidly and for large files the maximum stack depth will be reached before the file is processed.
Chapter 4

Development of System

This chapter contains a discussion of the specification and the aims of DPG, including some details as to how the design decisions were made. It also contains three examples of situations where DPG will be useful.

4.1 Specification of System

Main aims of the system:

- Track the dependency information for each theorem. Every theorem is annotated with three new properties:
  - The full list of dependencies is stored as the `dependencies`, or `parents` property for each theorem. The theorems contained in this list are those on which the given theorem depends, or its `parent` theorems.
- The full list was filtered to return only those dependencies, or parents, within the file; this was stored as the dependencies-within-file or parents within file property for each theorem.

- The information contained in the filtered dependency list is used to build up a list, for every theorem, of those theorems which depend on it, or its child theorems. This information is stored as the dependents-within-file or children within file property for each theorem.

- The span covering the definition file is given a property which contains the names of all the definitions in the file paired with a list of all their children. Definition names are also added to the list of parents for each theorem as appropriate.

- This information about dependence relationships between the theorems and the definitions, and within the theorems, is related to the user:

  - The user is provided with drop-down and popup menus that appear in the menubar and when the right mouse button is clicked in both the theorem and definition files. The user is also provided with context-sensitive menus that appear when the user right-clicks within a theorem’s span.

  - These menus contain items that allow the user to view the dependence relations. The main way this is done is through highlighting. The menu can be used to highlight the children of any definition, or the children or parents of any theorem. The user may also view a list of all the children or parents for any theorem from within the context-sensitive theorem menus.
• The user can move to any theorem or definition in order to edit it. This is done in the context-sensitive menu by providing a list of children or parents that the user may wish to edit. In the general menus in the theorem and definition files the user may simply choose from a list of all the theorems of definitions.

![Figure 4.1: Original menus in Proof General](image)

### 4.2 Design Decisions

It was important to decide how the information was to be relayed back to the user. Since the system was designed to be used on large files it was important that this information should be easy to take in; a long list of children, for example, would be very difficult for the user to digest and use. It seemed that the best way of relating such information is visually, for example, by highlighting the relevant spans. This would mean that users could skim through a large file, seeing at a glance all the theorems that concerned them. It was important also that this highlighting remained whilst they were editing the
file, until such time as they chose to remove or change the highlighting. Other functions that seemed important were functions that allowed them to move through the file in a useful way, e.g. by moving to the start of a theorem child or definition. It would also be necessary to move from the theorem to the definition file and back with ease.

The next design decision was how to allow the user access to these functions. One possibility was to set the span properties and then the user could simply type a command into the mini-buffer, a common method of executing functions in Emacs, to retrieve this information. However, this did not seem a particularly user-friendly method of presentation, especially since the system is running within Proof General which does not generally employ this method. It is hard to make an argument for which approach is better; using menus and buttons can be clearer for the user, but regular users may wish to learn the commands to type into the minibuffer so that they do not need to use the mouse. The user can, of course, use the Eval option in the minibuffer to evaluate any function within DPG, as it can be used to evaluate any function in the system, but the functions of DPG do not currently incorporate the ability to accept input directly from the minibuffer, see section 8.1.2. This is because it seemed ideal to provide some kind of visual tools for the user, such as buttons or menus, so that this system would appeal to a wider range of people. Some of the information related would be relevant to each particular span. An interface that could be activated from within a particular span that gives the user options that were not available from elsewhere (for example, highlight children of this span) was necessary. A good way of doing this is by using context-sensitive menus. The menu could be created by an event within the span, for instance a given button click, and information about what span the event took place in could be used to generate a suitable menu.
It was also desirable to provide the user with some general information out-
with each particular span. In this case, the user would still require access
to the Proof General menu that normally appears when one right clicks, as
shown in diagram 4.1, so it was necessary to add the dependence information
menu to this pre-existing menu. It also seemed desirable to add this gen-
eral dependence information to the menubar menu. The addition of buttons
was contemplated; however, since most menu items contain submenus, and
buttons are designed for single operations it was decided that this was not a
good method of presentation. Since editing of definitions may be done within
the definition file, a menu was also added here so that the user could observe
children of definitions and move to theorems.

4.3 Examples

- Suppose a logical error is noticed in a particular theorem that is caus-
ing problems, for example the word odd has been used in a theorem in
place of the word even. The proof of this theorem is therefore designed
around this mistake and the child theorems of this may be designed
to compensate for this error, in order to attain the result which was
considered to be correct.

The user may correct the original mistake by editing the theorem. How-
ever, if this theorem has children it is likely that logical flaws based on
this mistake will occur in the children and/or descendents of the origi-
nal theorem. It is therefore helpful for the user to be able to identify
where these mistakes may occur so that he may correct all of them be-
fore re-running the proof. Hence the user will right-click on the mouse
when the cursor is in the problem theorem, and the span-sensitive menu
will appear. The user can then choose the option ‘highlight children’. Alternatively the user may bring up the general dependence menu by right-clicking anywhere in the theorem file, but outside a span, and choose ‘highlight children of theorem’ and then select the relevant theorem name from the submenu that appears.

All the theorems within the theorem file that are children of the problem theorem will now be highlighted. The user can move among them, changing at will. The highlighting will not be removed unless the user brings up a menu by right-clicking or by choosing from the menubar and then chooses either ‘unhighlight’ or another highlighting option; in the latter case the old highlighting will be removed before the new highlighting is performed. Once the user has altered one of the child theorems of the problem theorem, he may then wish to look at the children of the altered child-theorem to see if the change has any effect on these. He can do this in the way described above and descend as far as he wishes. An option for returning back up the dependence tree, if, for example, the user has forgotten the name of the theorem at the higher level, is to bring up the span-sensitive menu within the theorem he has just changed and choose the option ‘highlight parents’. All the parents will then be highlighted and he can check through to see which one he wants. Another method is to choose the option ‘Move to parent’ which will present him with a list of parents he may like to move to.

- Suppose a definition is refined:

This is going to have ramifications throughout the file as there are likely to be a lot of theorems within the theorem file that are dependent on the definition. Once the user has edited the definition in the definition
file, it is clear that the proof may break down unless the theorems that depend on that definition are also updated. The system allows the user to choose the option ‘highlight children of definition’ for the relevant definition. This option will take the user to the theorem file, where all the theorems are, and those theorems that are children of the definition will be highlighted in orange, which is the defined colour for child highlighting. Thus it is immediately apparent to him which theorems will be affected.

The user can then go in and change any of the highlighted theorems that he wishes to. He can then proceed as above to alter all the child theorems.

- Suppose the user wishes to copy a theorem to a different theory:

Here, the user can copy and paste the required theorem from the original theory to the new theory. However, the theorem may depend on other theorems and definitions in the original theory, so if it is going to be provable in the new theory, all of these ancestors must also be copied. So the user can choose the option “Highlight parents” and see at a glance what other objects need to be copied to the new theory.

These examples are illustrated with screenshots in appendices B and C. Appendix B shows a combination of examples 1 and 2, and appendix C shows example 3.
4.4 Summary

The main aims of DPG have been outlined. These include the tracking and filtering of dependency information. Context-sensitive, popup and drop-down menus are used to provide options for the user to view these dependencies and to provide facilities to allow easy editing of and navigation through these dependencies.

This chapter then went on to outline the major decisions that were made during the design of the system, including a discussion as to why these decisions were necessary and how they were made.

The chapter concludes with three examples of how and why DPG might be useful.

A summary of the specification for DPG is that it is required to provide dependency tracking functionality above and beyond that which is currently available in theorem provers. This additional information must be related back to the user in a way that is easy to take in and understand, and will be helpful when the users wishes to edit the theory on this basis. This kind of functionality is not currently provided in Isabelle, nor within Proof General. In fact, it is unusual in any theorem prover.
Chapter 5

Implementation

5.1 Overview of System

DPG is designed to provide the user with information about dependence between theorems and definitions within a given theory. The interaction between systems is illustrated in figure 5.1.

There are various stages to this process:

- The ML code for the theorem prover Isabelle has been altered so that Isabelle sends a message informing Proof General as to what the parents are for each theorem. This is done after the completion of each theorem proof within Isabelle.

- The Proof General files have been altered so that Proof General will receive these messages and then record them suitable variables.
• Every time a span is completed, this information is used to set span properties for parents, parents within the file and children within the file. Variables are also created to record which theorems and definitions have been mentioned in this information.

• Context sensitive menus have been defined which use these properties to return dependence information. Three types of context sensitive menus are defined: a theorem menu that appears when the user right clicks within a given theorem in the theorem file, a menu that appears when the user right clicks in the theorem file outwith a theorem and a menu that appears when the user right clicks in the definition file. Each of these menus contains appropriate functions. Screenshots of these menus are shown in figures 5.2, 5.3 and 5.4 respectively.

• The functions that are contained in the menu are defined; these are highlighting functions and functions to move the cursor.

5.2 Retrieving the Information from Isabelle

Isabelle already has the ability to automatically track dependency information and keep a record of the dependencies. This is exhibited in Isabelle by a function \texttt{thm_deps[thm]} which will draw a dependency graph for a theorem or set of theorems. This includes everything that Isabelle has used in the proofs for the theorems, including built-in definitions and theorems.

Unfortunately the way that Isabelle reports dependency information is rather confused. It would be best if Isabelle reported not only all the theorems that were used in the proof, but also all the definitions. This is the case for
most definitions in the definition file but there are certain definitions such as primitive recursive ones that are simply ignored by Isabelle. However, if the proof is run through Isabelle twice then the second reporting of dependency information includes not only everything that was given before but also these definitions that were ignored the first time. Hence it is possible to retrieve all the necessary information, but only by running the proof through Isabelle, or, at a higher level, Proof General, twice. This is just a minor inconvenience for small files but for large files it adds considerably to the running times. This problem is discussed in further detail in section 8.1.1.

A set of functions was written \(^1\) that used the dependency tracking code already present in Isabelle to force Isabelle to inform Proof General what

\(^1\)This file was written with help from David Aspinall and Jacques Fleuriot
Figure 5.2: Theorem context-sensitive menu

these dependencies are. This is done by getting Isabelle to return an urgent message to Proof General. There is a function within the Proof General files that picks up urgent messages when they are received and checks to see what kind of message they are. Proof General will save the information contained in the message in a predefined variable which can be searched for from other parts of the code.
5.3 Setting Variables

It was necessary to alter the Proof General files so as to search for the message from Isabelle and, if such a message is found, to pass it through hooks placed in the Proof General files to DPG code\textsuperscript{2}. Proof General contains several different modules. There is a module for each theorem prover which is loaded only when the particular theorem prover is used, and there is a directory which contains generic files which are always used when Proof General is run, regardless of the particular theorem prover. It was necessary to add code to the Isabelle directory within Proof General to ensure that the message from Isabelle is noted by Proof General. Beyond this, however, it is desirable that the coding be added to the generic files in Proof General so that the system is as general as possible. Hence, although the system at the moment can only be used with Isabelle, a lot of the coding that has been done would be equally valid with Coq, Lego or any other theorem prover provided that it could relay back the dependency information. It will not be possible to use the existing code directly with another theorem prover, since

\textsuperscript{2}The alteration of the Proof General files was done with some help from David Aspinal.
some of the existing code is Isabelle specific, for example the assumption that the definitions are kept in a file appended .thy and the theorems are kept in a file appended .ML. Such assumptions would not be valid for other theorem provers. However, much of the code could be reused and that code which could not be could form a guideline for additional code based on other theorem provers.

Within the generic files for Proof General is a function that receives and processes urgent messages from the theorem provers. Hence code was added to this function to force Proof General to observe this new type of message, and further code was added to instruct Proof General how to process this message if it was found. If such a message were found, Proof General would alter it to the desired form and store it in a global variable that could be accessed elsewhere.
The function proof-done-advancing is called every time a span is completed and takes this span as an argument. It is within this function that hooks to DPG are added. Coding was added to force this function to check for the existence of the variable containing dependency information. If such a variable was not found, for example if another theorem prover was being used, or Isabelle were being used without the dependency tracking activated, then Proof General will just continue as normal with the rest of the code, and the additional functionality of the system will not be called. However, if this variable does exist then DPG is used. The algorithm for this code is given below. The code itself can be viewed in Appendix A.

for each span:

IF a dependency message has been received from Isabelle
THEN

IF a dependency menu exists
THEN delete it
AND set dependency span property
AND set menu-variable as a keymap
AND filter list of parents
AND set filtered list as parents within file property
AND call function to update children lists and lists containing names of theorems and definitions.
AND call function to create general theorem file menu
AND add this new menu to the theorem file menubar
AND call function that creates definition file menu
5.4 Setting the Span Properties

If theorem dependencies are found to exist then various functions are called which update the function of the system. Three new span properties are defined. In the code these are named dependencies, dependencies-within-file and dependents-within-file, but they are referred to in this document as parents, parents within file and children within file since these terms are more immediately understandable. The parents variable is simply set to all the theorem dependencies. Parents within file is set as the result of filtering this parents list for these parents that exist only within the file that the span is in or the corresponding definition file. Each dependency is prepended with the filename of the file it is found in, and the system simply searches for those with the relevant filename prepended. The filtered list that is set as parents within file is then passed to a function that updates the children property. This takes the name of the current span and, for each span named in the parents within file list, will merge their children within file property with the current span name. In this way the updating of the children property is done incrementally rather than as a whole at the end of the proof process. This is important because the user may want to do the proof incrementally; if they are stepping through the proof rather than evaluating it all at once then they will always have access to the most up to date child information for each span rather than having to complete the whole proof before they can retrieve this information.

5.5 Updating the Theorem and Definition Lists

Two lists, one containing names of theorems in every file and the other containing the names of definitions in every file, are updated. During the pro-
cessing of the dependency information, the system becomes more informed about the names of the theorems and definitions in the theory. This happens in two ways: firstly, a lot of names crop up in the dependency information, and secondly the system can discover the name of the span it is currently processing. The list of theorems is updated through the latter way; since the span of every theorem will be processed as the theory is processed, it is possible simply to add the name of the current span’s theorem to the theorem list every time. Thus the theorem list contains all the theorems that have been processed so far. Since there is no such easy method of updating the definition list, this is done through the former way. The dependency information is divided into theorems and definitions during the processing of the span, and all objects that have been designated as definitions are added to the definitions list. This means that only those definitions that have children in the theory are reported. This is sufficient for the purpose of tracking dependence information but unless Isabelle is altered to report to Proof General the names of all definitions within the definition file, there it is not easy to discover the names of those definitions upon which nothing depends.

The way in which the system divides the dependency information into theorems and definitions is based on whether there name in the dependency information matches the name of a span; the span around a theorem generally takes the name of the theorem, whereas the span around a definition, because it incorporates the whole of the definition file, does not. There are some problems inherent in this approach which are discussed further in sections 6.1 and 8.1.1.
5.6 Defining the Menus

Once the dependency information has been determined it is necessary to present this information to the user. For reasons discussed in section 4.2, it was decided to present this information visually, and to allow users to access it through menus.

The first menu that was defined was the theorem context-sensitive menu (see figure 5.2). This needed to be created specifically for each theorem and would contain information that was relevant only to that theorem. This menu is constructed on the fly when the user right-clicks, using information that is updated each time a span was processed during the proof. Since there needed to be some way of recalling the menu information from each span, a keymap property was created to bind the menu to the span. Within this variable the right button of the mouse is defined as the event that will call the menu. This button was chosen as this is the standard method of bringing up popup menus within Emacs. The function that is bound to the right button of the mouse by the keymap is one to create the appropriate context-sensitive menu. The position of the event, i.e. where the user is when he right-clicks, informs the system as to what the appropriate menu is by reference to what span the cursor is in.

Next the menu that appears within the theorem file outwith a span was defined. This menu was not designed to be dependent on any particular theorem but to contain information general to all theorems processed so far. Since a menu already exists for the theorem buffer which provides access to Proof General and Isabelle, it was desirable only to add the dependence information to this menu rather than create a new menu (see figure 5.3). Hence the dependence information was added as a sub-menu to the pre-existing menu. However, unlike the information already contained in this pre-existing menu,
it was not sufficient to create the dependency information submenu once. Since the user may not wish to prove the whole definition file at once it is important that he may access all the existing dependence information at any point rather than waiting until the whole file has been processed. Therefore, every time a span is updated this new information must be added to the dependence information sub-menu. It was clear therefore that this menu needed to be built-up within a function that is called every time a span is updated, so that it was updated incrementally as each span was processed. Once the rest of the dependency related code has been run, the function that creates the menu is run and then this new menu is added to the pre-existing theorem file menu. In this way it is ensured that the dependence information submenu always contains the most up to date theorem information.

The third menu to be defined was the definition file menu. See figure 5.4. This is similar to the theorem file menu and had to be added to the pre-existing definition menu. This was done in a similar way to the adding and updating of the theorem file menu. Hooks needed to be added to call the function that created the definition file menu, and within this function code was added to update the new definition file dependency information menu.

Menu items are evaluated when the menu is required. For the theorem and definition file menus this means whenever the file is processed through Proof General, since these menus appear automatically on the menubar, and are already pre-existing when a user right-clicks. However, the theorem context-sensitive menus are created on the fly whenever the information is required within a particular theorem. At this stage the system works out which theorem the cursor is in and creates a menu specific to that theorem. Menu items are either single items, such as ‘unhighlight’, or, more usually, sub-menus, such as ‘highlight children of theorem’ followed by a list of theorems. See figure 5.5 for an illustration of how all these functions call each other.
Figure 5.5: Flow of control in Proof General and DPG.

The arrows represent the calling of functions. The arrows pointing from the bottom three boxes indicate the fact that these bottom functions call many other functions.

5.7 Defining the Functions

There are two types of functions defined in the menus: those to move the cursor either within a file or to a point in another file, and those to highlight or unhighlight particular theorems.

Two additional faces were defined within the file that sets up the configuration of Proof General. A face is a named collection of graphical properties: font, foreground colour, background colour, and so on, and faces control the
display of text on the screen (Wing, 1999). Here, faces are just used for highlighting. Two colours were chosen, bright orange and pale pink, that would be easily distinguishable from the existing Proof General highlighting colours. These colours can be seen in appendices B and C. Orange was designated as the ‘child’ colour and pale pink as the ‘parent’ colour. There are two main methods of performing the highlighting of dependences: one is from within the theorem context-sensitive menu where information is related to a particular theorem, and the other is within the general menus which are applicable to all theorems. The basic highlighting functions work by finding the span property for either the parents or children and iterating over this list. If the dependence is a theorem then the face of this theorem’s span is highlighted accordingly. If the dependence is a definition then no action is taken. This is because, since the definition file contains only one span which covers all the definition it is possible only to highlight the entire definition file, which would not be very helpful, or to do nothing. It is very hard to separate the definition file into separate spans for each definition. This is because the rules for parsing definitions are complex, and although it is easy to search through the definition file for the first appearance of the name of the definition, it is very hard to find the end of the definition so as to isolate it from the rest of the definition file. Also, in some cases the name of the definition appears in a list of constants at the start of the definition file and the actual definition is later in the file; this makes isolating each definition an even more complicated task. This problem is further discussed in section 8.1.2. If a theorem has definition parents but no theorem parents then a warning message is given to this effect when ‘highlight parents’ is chosen, since to make this text unselectable would be misleading, appearing to indicate that there were no parents.

The menu items that move the cursor to a particular theorem do so by finding
the start of the theorem-span and moving the cursor to there. Menu items that move the cursor to a definition do so by searching through the span in the definition file to find the first mention of the definition and moving the cursor there. However, due to the layout of the definition file the move to the start of a definition can be problematical. This is discussed further in section 8.1.2.

Many auxiliary functions are defined to facilitate these functions. One of these is worth describing due to its central role in many of the above mentioned functions: this a function named `find-span-with-prop`. This is designed to find, within any given file, the span that has the given value for a particular property, in practice always name. This is important because dependence information is stored as a list of span names rather than actual spans, and hence the child information also contains the names of the relevant theorems, but the spans connected to these theorems need to be quoted as spans rather than span names in order to perform any span functions on them. This function works by creating a list of all the spans within the file, which is based on a built-in function, and iterating over this list until either it is empty or the correct span is found.

### 5.8 Summary

The points below summarise the functionality implemented in DPG:

- Theorem context-sensitive menu, shown in figure 5.6.
  - Moving to any child theorems or parent theorems or definitions. Since these moving options give a list of all possible children or
parents, with the parent options divided into two categories: theorems and definitions, they can be useful not just to implement moving but to provide a place for the user to view a list of parents and children.

- Highlighting all child or all parent theorems; these options automatically unhighlight any previous highlighting.
- Unhighlighting all the spans in the buffer.
- Moving to the definition file; this gives the user the choice of opening the definition file in the current window or opening it in another window.

- The theorem file general menu, shown in figure 5.7.
  - Highlighting all the children or all the parents of any given theorem; this option provides a list of the theorems in the file for the user to choose from.
  - Highlighting all the children of any definition; a list of definitions is provided as above.
  - Unhighlighting all the spans in the buffer.
  - Moving to any theorem in the file; a list of theorems is provided for the user to choose from.
  - Moving to the definition file, either in the current window or a new window.

- The definition file menu, shown in figure 5.8.
  - Highlighting all children of a given definition; a list of definitions is provided for the user to choose from.
  - Unhighlighting all the spans in the theorem buffer.
- Moving to any theorem in the theorem file.
- Moving to the theorem file, either in the current window or a new window.

![Theorem context-sensitive menu](image1)

**Figure 5.6: Theorem context-sensitive menu**

![Theorem file menu](image2)

**Figure 5.7: Theorem file menu**

![Definition file menu](image3)

**Figure 5.8: Definition file menu**
Chapter 6

Evaluation and Testing

6.1 Evaluation of Correctness of System

All of intended functionality as described in the specification for DPG in section 4.1 has been implemented. Testing of this functionality has been carried out over a variety of theories. In most cases the testing on the given Isabelle files did not show any errors in the functions. There is great variety in the layout of Isabelle files and many different ways of creating definitions, and so it can be hard to ensure that additional functionality works in every conceivable situation. However, a wide variety of files were used for this testing and all of the functions were tested on each of these files; therefore it seems reasonable to assume that those functions that passed on every occasion are, probably, correctly applicable to any situation.

However, there is one particular area where the functions are not giving correct results, which is the listing of definitions. A list of all the definitions is required in the functions in the general theorem file menu and the definition
menu which allow the user the option of highlighting all the children of a chosen definition. A list of relevant definitions is required within the menu attached to each theorem which has parent definitions and should contain all those parent definitions. The problem with these functions is twofold; firstly there are some definitions that are not included in the lists and secondly there are items contained in these lists that are not definitions.

Both of these problems are based on the way the names of the definitions are recorded. Theorem names are recorded by making a list of the name of every theorem as it is processed. However, there is no such simple way of recording definitions, as definition names do not appear in the Isabelle output in a uniform manner. Instead, a list of definition names is built up from the dependency information output. Every time some dependency information is received, each item in the dependency list is observed and if it is deemed to be a definition it is merged with the existing list of definitions within that file. The first problem lies in the fact that definitions will only be recorded if they appear in the dependency information of at least one theorem. This is not a particularly great problem as we would expect all the definitions in a theory to be used at some point, and if they are not then we cannot gain useful information from them. However, it would be neater if they were to appear in the ‘highlight children of definition list’ greyed out, so that user could see immediately that they have no children. As it is users must surmise that they have no children from the fact that they do not appear in the list.

The second problem is more serious and stems from the way it is decided whether a list item is a definition or not. This is done by searching for a theorem whose span name matches the list item, since it is usual for theorem spans to have names. If no matching span is found then the object is deemed a definition. However, there are situations where this method breaks down. It is possible to prove theorems in the theorem file without laying
them out in the common format. This is a problem because it can mean they do not have a qed line, and, since Proof General scans for the qed line in order to set the span name, this means that the span has no name. The theorem will still have a name, and hence can appear in lists, but this name is not also attached to a span. It is also possible to add commands within the theorem file, such as a command to add a theorem to the simplification information within Isabelle, and the span surrounding such a command will not have a name either. In both these situations the object will falsely be labelled as a definition. This second error will not cause the system to break down, nor will it provide any false information about dependence. It is also a reasonable assumption that when the user chooses to highlight the children of a particular definition they will know what definition they are interested in and not trawl through the list looking at their options, and hence not be misled by this incorrect information. However, this error will nevertheless indicate to the user that objects are definitions when they are not and this must be considered as a serious flaw. A discussion as to how this problem might be addressed is contained in section 8.1.1.

Another problem is the indirect referencing of theories. It is common practice in theorem proving to create a top-level theory that depends on several other theories. The files for these theories are parsed when the top-level theory is processed so that all the necessary information is available. If the user switches to the buffer containing a sub-level theory once the top-level theory has been proven it will be highlighted in blue, indicating that it has been processed in Proof General. However, in these cases no dependence information is available, and the dependence menus do not appear in these files. This is not a general problem of multiple theories; the system is designed to deal with multiple theories and if several theories are processed in the same session the user can access any of these at any time and have access to full
dependence information for any of them. It seems that the function within
the Proof General files that contains hooks to the additional functionality
code is not being called when theories are parsed indirectly. This is further
discussed in section 8.1.1.

6.2 Evaluation of Usefulness and Relevance
of System

The above section is a discussion of whether the aims of the project were
correctly implemented. However, just as relevant to the success of the project
is the question of whether the correctly implemented aims of the project are
of value or use in the theorem proving community. When the aims of the
project were being designed, members of the theorem-proving community
were consulted about what kind of functionality what be useful. Therefore,
an attempt was made to persuade regular users of theorem provers to use
DPG, and to record their comments and feedback so that it could be seen
whether these aims had been usefully implemented.

6.2.1 Aims of Evaluation

The best way to do evaluation of the system is to install the additional code
in the systems of regular Proof General users and give them an introduction
to what DPG does and how to operate it. These users could then continue
with their ordinary work over a period of time, using DPG whenever they
wished to and making records of how often they used it. After this set period
they would be subjected to some kind of interrogation, probably an interview
or a questionnaire, and the outcome of these, in addition to an evaluation
of how regularly they used the system, could be used to assess how useful and relevant DPG is. The main drawback with this approach is that it involves long time scales which are not really available in a four month project. Therefore the aims of the evaluation were scaled down.

A more realistic option for evaluating the system was to find as many volunteers as possible, all of whom should have some experience of theorem proving. They would be provided with details of what the system does and allowed some time to consider how this would be useful. Ideally they would be able to find some theories they had been working on that may benefit from the additional functionality. They would then have a chance to work with the system for a short period either remotely or, preferably, under observation so that their reactions to the systems could be noted and problems recorded as they occurred. They would later be asked to fill in a questionnaire recording their experience of the system and their ideas for improvements and additional or different features they would like to see.

This latter option is the course that was pursued, although unfortunately not in nearly so thorough a manner as would have been desirable. Due to the unforeseen unavailability of several potential evaluators during the short period available, only two volunteers were identified who had the necessary experience and were willing and able to carry out the evaluation at the appropriate time. The results of this evaluation are discussed below.

6.2.2 Results of Evaluation

After an interactive demonstration, the two volunteers were asked to complete questionnaires concerning their opinions of the system. This question-
naire can be viewed in appendix D. A summary of points that were made follows:

- The functionality provided by the system could be very useful in theorem proving. People are often required to maintain theories that they have not written and so they are not aware of how the files are built up; in this case dependence information can be particularly useful. Even for maintenance of their own theories, this information can be useful as it is hard to recall the interdependence of theorems. One comment was that it ‘would save tedious trial and error or hand searching when changes have been made’, another that it would ‘make proof maintenance less ad hoc’.

- Another benefit that was pointed out was that a common method currently employed for finding child theorems and so on is performing a search on the file to see where the given definition or theorem is mentioned. However, theorems and definitions are often used in proofs without being directly mentioned, e.g. by being used in simplification, and hence a simple search would not identify them. Here, the system would provide a deeper level of insight into the proof than would be possible by naive search methods. Also, a more global view of the file is being obtained, rather than the sequential view obtainable by merely looking at the file.

- Both evaluators agreed that the functionality of the system would be useful quite frequently, especially when maintaining large files.

- Some improvements were suggested, most of which are included in the discussion in section 8.1.2.

- Both evaluators agreed that the system was easy to use, one comment being ‘as easy to use as Proof General’, so DPG did not introduce any
additional user problems. However, neither of the evaluators used the system for a long period of time, hence there may be problems of which they were not aware.

The above discussion is obviously of limited value since the sample of evaluators was so small. However, the personal experience of the two evaluators indicates that the system may have much to offer.

6.3 Evaluation of Efficiency of System

Another consideration in evaluating the usability of the system is how the run time of Proof General is affected. Table 6.1 shows a comparison of run times for processing various theories in Proof General. Three times are given; firstly, run time without any dependence functionality included (time 1), secondly, run time with dependency tracking in Isabelle but no processing of this information (time 2), and thirdly, run time with DPG (time 3). The three timings for each file were all taken consecutively in an attempt to make the conditions the same for each. The percentage increase listed in the final column is the increase from time 1 to time 3, i.e. the difference between Proof General with no dependency tracking and Proof General with DPG fully implemented.

The results of these time trials are extremely varied. Three files of the five tested showed very little change in the timings for the three runs; as expected, timings with dependency tracking was slower than without and timings with full functionality implemented was slower still, but the percentage increase from the first to the third was less than 8%. However, the other two show a marked increase in the timings, with the percentage increase being 150%
and 180%. The third file, PRat, takes much longer to process because it calls other files during its processing. This is where we would expect to see a marked increase in time between standard Proof General and DPG, since the amount of dependence information would be getting extremely large. However, although one would be prepared for a reasonable increase in time between the two, an increase of 180% is much larger than would desirable. The second file, PNat, also shows a large increase in time. This is of some concern, since this is not a particularly large file. The reason for this seems to be that the dependence information is particularly complex for this file.

It seems that using the system on large files may well cause Proof General to run considerably slower. The change seems to be negligible for small files, but this is little consolation since the system is of most use for large files. This slowing down does not negate the value of the system, since this is just the initial run time. Once the theory has been proved and DPG is actually being used, there will be no problem and these times will have no effect. However, it is tedious for users to wait for too long for their theories to be processed before they can actually do any work with them. One limiting factor to the optimisation of the processing speed is the fact that storage demands in Isabelle increase up to seven times when dependencies are being

<table>
<thead>
<tr>
<th>Theory</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>0.40.94</td>
<td>0.40.83</td>
<td>0.44.06</td>
<td>7.6</td>
</tr>
<tr>
<td>PNat</td>
<td>0.55.22</td>
<td>1.00.92</td>
<td>1.38.26</td>
<td>150</td>
</tr>
<tr>
<td>PRat</td>
<td>2.22.60</td>
<td>2.43.63</td>
<td>6.33.85</td>
<td>180</td>
</tr>
<tr>
<td>Fun</td>
<td>0.48.02</td>
<td>0.58.12</td>
<td>1.01.34</td>
<td>7.9</td>
</tr>
<tr>
<td>Ord</td>
<td>0.20.77</td>
<td>0.20.93</td>
<td>0.21.30</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Table 6.1: Table of times
tracked, with attendant slowing of the system, as discussed in section 7.4. Possible solutions to this problem are discussed in section 8.1.2.
Chapter 7

Related Work

This chapter contains a comparison with other systems that are carrying out related work. This is designed to apply particularly to what has actually been implemented within this project rather than the future project.

7.1 Cynthia

Cynthia is an editor designed by Jon Whittle to assist novice programmers to learn ML, which is a functional programming language (Whittle et al., 1997). The main idea behind the system is that of using analogy to create new programs. It is very common for novice programmers to write a new program by finding a program they have already written or have found in the literature that is in some way similar to the new program, and using it as a basis for their new program. Cynthia is an interactive system that guides students in this method of drawing analogies between existing and proposed code.
Although the top level of Cynthia is dealing with programming, the underlying structure of Cynthia is based in theorem proving. This is because theorem proving can sometimes be considered as analogous to program verification, and so checking the validity of a theorem or theory can be seen as analogous to program verification.

From the point of view of this project, the most interesting part of Cynthia is the way in which it allows users to build up their proofs. An example is given in (Whittle et al., 1997) of a user altering a program for finding the length of a list into a program for counting the number of nodes in a tree. The necessary steps are summarised below:

1. The RENAME command can be used, for example to change each occurrence of the word length to count. This command does nothing but a search and replace on the name.

2. The type of the parameter needs to be changed from a list to a tree. So, for example, if the user spots an occurrence of the list nil, they may use the CHANGE TYPE command to change the type to tree. Cynthia will then propagate this change by changing terms referring to lists to terms referring to trees throughout the program.

3. The results for each pattern now need to be altered. For example, the result of the base case for length of a list (i.e. length of nil) will be 0, whereas for the number of nodes in a tree this result will be 1.

This is one approach to the problem that is faced by DPG; namely the problem of making changes but keeping the underlying proof as intact as possible.
In particular, this project has been dealing with one aspect of the propagation stage, which must first involve identifying where the change must occur.

It is very important that the new programs created using Cynthia are correct, since the point of Cynthia is to help novices learn how to program correctly, and hence the underlying proofs must be valid. Programs written in Cynthia are guaranteed to be well-defined and terminating.

Some of the high-level commands in Cynthia include adding an argument and adding a component, which is similar to changes that could be made within the system implemented in this project. Other Cynthia commands, such as those that deal with making patterns and checking recursion, are not so relevant to DPG.

The main difference in focus between Cynthia and DPG is that Cynthia is more in control. This is because Cynthia is a teaching device and so it is acceptable, and even advantageous, for Cynthia to have a clear built-in specification for how changes should be made and what outcome is acceptable. The situation is not so cut and dried for DPG. Since DPG is designed not as a teaching aid but as a tool to aid theorem provers, it needs to allow more autonomy to the users. Like in programming, when errors occur in theorem proving there are often many ways to correct them, and users are likely wish to retain control over how changes are made to their theories. Cynthia is able to get round these problems in programming by restricting its users, which is acceptable because they are learners. Hence, although Cynthia has many similarities with DPG, and since it implements something similar to what is proposed in the common-sense theory maintenance project, it may be a good inspiration for further work, it is operating in a rather different domain and
thus has different requirements.

7.2 KIV

KIV is a tool for the development of correct software, including formal specification and verification. It is based on a theorem prover and uses heuristics to maintain a high level of automation. The user is only forced to intervene if all tactics fail. KIV also has a well developed graphical interface.

Much of KIV is not of relevance to the project at hand; however, some parts of KIV are of interest. One aspect of KIV involves the intelligent reuse and replay of proofs. KIV offers a number of proof engineering facilities to be employed when a proof attempt fails. One method that KIV uses that has not been attempted in this project is the automatic finding of counter-examples to detect unprovable subgoals. This seems a very useful facility but is well outside the range of this project. KIV’s strategy for proof reuse is more relevant. Within KIV, both successful and failed proof attempts are reused automatically to guide the verification after corrections. This is reflected in the editor where the old invalidated proofs are used to create a new, valid proof.

7.3 Dependence Tracking in Argumentation

Mandy Haggith, in her work on argumentation (Haggith, 1995), (Haggith, 1996), has a method for tracking dependencies within arguments. She does not express this relationship as dependencies and dependents, or as parents and children, but rather as justifications and elaborations so as to fit in with
the argumentation model. A parallel can be drawn between her domain and
the domain of theorem proving: the facts of an arguments are mapped to
definitions or axioms within a theory, the propositions are mapped to theo-
rems and lemmas, and the argument itself is mapped to the theory. A set
of propositions S is said to be the justification for a proposition P if they
fully support the proposition P; in other words S is the full set of parents of
P and hence P is provable using S. A set of propositions S is said to be the
elaboration for P if they elaborate or embellish on P; in other words they are
the children of P.

These concepts are manipulated manually. The whole argumentation pro-
cedure, although implemented on a computer, is governed by a human user
who acts in the role of a chairman, arbitrating over arguments. It is the
user who is responsible for ‘marking up’ the text, which involves deciding
which propositions are part of the justification or elaboration set for another
proposition. Hence this dependency tracking is not done automatically as in
DPG. However, once this mark up has been done, the system will manipulate
the propositions accordingly, hence forming coherent arguments.

Mandy Haggith’s system is designed for human users to decide the depen-
dency information in the theory (or argument) and for the system to use the
dependencies automatically to develop the argument. This contrasts with
DPG, where the system is designed to track the dependencies automatically
and then allow the users to make use of this information themselves.
7.4 Isabelle

Isabelle already has some ability to track dependencies. The aim is to record all the logical inferences in detail, while omitting bookkeeping steps that would provide no useful information to the user. One of the problems with using this facility is that proof objects are large and multiply storage requirements by about seven. These objects are not generally part of standard use of Isabelle. The aim of these objects, as stated in the Isabelle manual (Paulson, 1999), is to increase user confidence in Isabelle. The manual also states that proof objects will seldom be given whole to an automatic proof-checker, and that it is up to the user to scrutinise the information for dependence on some object. This is in contrast to DPG, which is designed to provide a more user-friendly experience so that the user does not have to exert particular energy scrutinising information.

Isabelle does provide some functions to make viewing this information easier. It provides options for how much information will be stored: minimum, theorem or full. DPG uses Isabelle set to the theorem level. Other functions inform the user of the size and depth of the derivation or provide a derivation tree. Isabelle will also generate a graph of the derivation as discussed in section 5.2. However, any information returned only gives the names of theorems and definitions involved rather than directly involving the theorems and definitions themselves, e.g. by highlighting them. Nor does this facility enhance editing the theory on the basis of this information.
7.5 Proof General

Proof General already provides much functionality that is made use of in DPG. Proof General is also able to track file dependencies. However, there is no dependency tracking within files and without the alterations made during this project there is no ability to make use of the dependency tracking ability within the theorem provers. Useful manipulation of the theories using dependency information was made rather difficult due to the fact that the definition file was contained in only span. This indicates that Proof General was written without any future implementation of dependency tracking in mind.

7.6 Conclusion

This survey of related work indicates that the kind of functionality created by DPG is not commonly implemented in other systems. There are some systems that have dependency tracking ability but this ability is not often used and not designed to make using this information to edit the theory very easy. Other systems are designed to make editing the theory easy and to visually present information to the user to ease manipulation of the theory, but without tracking dependencies. Most of the systems reviewed in this chapter did not combine the dependency tracking with an informative, visual representation of these dependencies and an environment designed to help editing of theorems on the basis of this information.

A system that seems to incorporate many of the aims of DPG is the KIV system. However, this differs from DPG in that it is aiming for a high level of automation. DPG is designed to be a fully interactive experience, providing
information for users rather than directly manipulating the theory. The domain for KIV is also a little more constrained.
Chapter 8

Further Work and Conclusions

8.1 Improvements to DPG

Although most of the intended additional functionality has been correctly implemented there is still much further work to be done.

8.1.1 Correcting Existing Problems

The first, and most important, task would be to correct those aspects of the system that are not working properly; namely the method of building up the list of definitions, building up the list of dependencies returned the first time Isabelle is run and the problem of parsing indirect files.

The solution of the first problem would involve altering the way DPG decides whether an object refers to a definition or theorem. One way of achieving this
may be to retrieve information about definitions from the theorem prover. It seems clear that the theorem prover must have an awareness of what definitions are contained in the theory because it is able to use these definitions in proofs. From observation of Isabelle output, it appears that this process happens, as one might expect, at the beginning of the processing of each theory. The first thing Isabelle does when processing a theory is to process the definition file. At this point Isabelle builds up a record of the contents of the definition file. It should be possible to write some ML code that would get Isabelle to return a message to Proof General of the form, "the definitions are "defl def2 ...\"", for example, in a similar manner to the way that Isabelle is made to return the dependency information as a message to Proof General. This returned information could then be set as a property for the span within the definition file. When it was necessary for the system to decide whether an object was a definition or a theorem, it could be checked to see if it was listed within this property. If it failed this test, it would be tested to check whether it was a theorem, and if it failed both tests to label it as miscellaneous. However, this would also be problematic as the easiest way of identifying theorems, which is by searching for spans with the theorem name, only works for theorems laid out in the standard format, as described in section 3.3. Hence a better test for identifying theorems would need to be established. Another method of solving this problem would be to do a search through the definition file every time an object had to be designated as a theorem or a definition. If the object was found in the definition file then it must be a definition. This method would be easier to implement but it would be more time consuming than the above method.

The second problem that needs to be addressed is the problem of Isabelle not returning all the dependency information the first time it is processed but only after it has been processed more than once. The method of overcoming
this problem in this project has been to run each theory through Proof General twice. However, this is a poor solution because a great deal of time can be wasted, as discussed in section 6.3. Another approach would be to rebuild Isabelle's higher order logic files with a flag set to get Isabelle to record all the dependencies. This, however, may cause problems within Isabelle.

A third problem concerns the calling of indirect files, that is those that are called through the processing of another file. As discussed in section 6.1, these files show all the expected Proof General annotation but not the DPG annotation. The code within existing Proof General files needs to be examined here to see when the additional functionality is called. This is a pressing problem and should not be very hard to fix. This was not done because the fault was discovered too late on in the project.

8.1.2 Adding New Functionality

With the exception of the above examples, the aims of the project have been successfully implemented. However, as the project progressed new ideas were continually suggesting themselves. Most of these ideas were not implemented since a major addition to the original aims was not possible in the given time. A discussion of these ideas is given below for the benefit of the reader, and in the hope that they will be included in the system at some point.

Loading the Isabelle Code

One of the most obvious flaws of the existing system is that it is necessary to include an instruction to Isabelle to load the file `depends.ML` in the first file.
that is processed by Proof General. The file ‘depends.ML’ includes all the additional ML code to get Isabelle to record the theorem dependencies and report them back to Proof General. Once this file has been called, Isabelle will always record the theorem dependencies; however, if this file is not called directly at the beginning of the procedure then all of the additional functionality will be ignored by the system. The main advantage of this approach is that it is easier to ignore the additional functionality if the user does not wish to make use of it. However, it would be preferable by far to add this line somewhere in the system so that the file is loaded automatically by default, and then have changing that default as a user option. Alternatively, the default could be to avoid using the extra functionality and the user would have the option to include it if they wished. Since no research was done into creating user options and defaults during the project, it seemed diversionary to begin work on it at a late stage. However, if further work were to be done on the system, this should be one of the first problems tackled.

Providing Proof Maintenance History

Another addition to the system that could improve the user’s editing experience would be to make the history of past dependency information available. The most obvious use for this would be in following situation:

A user wishes to change a definition and so highlights all the children of that definition. After working through the child theorems of that definition, and perhaps looking at the children of those theorems and so on, the user feels that they have made all the necessary alterations. However, on rerunning the theory through Proof General, the proof fails at some point. Now they may wish to examine the theorem where failures occur, to look at the children and parents of that theorem, and so on. Information concerning everything
'above' that theorem, i.e. that has already been processed before that theorem, will be available to the user, but information concerning theorems that have not yet been processed will not be. For example, the user will not be able to retrieve information that the next theorem in the theorem file is a child of the broken theorem. However, the system had this information before any alterations were made and the proof was retracted. Therefore it could be advantageous for the system to record this information and allow the user to access it at a later point.

Implementing this does not seem to be insurmountable task. Storing the information would require a different approach to that previously employed in the project, namely storing information as span properties, and this is one of the reasons that this was not implemented during the project. The reason the approach would need to be different is that when the proof is retracted, all the spans are unbound and so any information stored as properties of spans is lost. A different method of storing information could be to store it to files. An item would then be added to the menus to give the user the option of retrieving the history. This would need to be added to the in-theorem menus, where the option would be to view the parent and child lists for the theorem, and in the definition file menu, where the option would be to view the child list of a chosen definition. When the option was chosen it would be possible to make the list appear in another window, or perhaps to appear as a list within the menu. The former option may be preferable as the list could then remain as long as necessary whilst the user visited all items on the list. However, adding this kind of history could also cause problems. It is possible that the changes made before the proof was retracted, as well as altering the content of individual theorems, also altered the dependence relations between theorems. Thus the history information could not be relied on to be correct for the current theory. Therefore, it is questionable whether
it would be to introduce this idea, even though it is clear that there would be many situations in which it could be useful.

Providing Highlighting History

In addition to recording the history of past runs, it would also be useful to record the history of past highlighting within a run. Thus if the user altered a definition they may wish to look at the child theorems of that definition. They then may wish to look at the children of a particular child definition and so change the highlighting. However, when they have finished observing the children at this level they may wish to return to the original level. There is a discussion in section 4.3 as to how the system facilitates this. However, providing a history of previous highlighting would make this even easier. There could be an item in the menu named “Highlight history” and then a list of the most recent highlighting items, for example “highlight children of definition “defl”” of “highlight parents of theorem “thm1””. This idea moves into an area not covered by this project, namely the idea of recording what options the user has chosen. Therefore, on the experience gained in this project, it is hard to say how difficult this would be to implement. It would be necessary for the system to observe what menu item the user chose and then record this in some suitable manner, most likely as a list, which could be stored in a variable. This list would initially be set to nil. When the user chose a highlighting command this would be noted and appended to the list in the correct form. It would also probably be wise to limit the number of alterations stored in this list to a maximum, perhaps of four or five, as it would not be helpful for the user to be presented with every change they have ever done. Thus the number of objects in this list would be evaluated and if it exceeded a certain number, the last element would be removed and
the new element appended to the front. This all seems fairly straightforward and similar to methods already employed in the project, with the exception of tracking the changes the user made, on the difficulty of which no comment can be made.

Providing Recursive Highlighting

Another highlighting function that may be worth introducing is a function that highlights recursively. Thus it would create an option that would highlight all ancestors, or all descendents, since it would not only highlight the first layer of ancestors or descendents but continue until all direct relatives had been highlighted. This would be useful if, for example, the user wished to verify that they had checked every potentially affected theorem before they retracted a proof or, as discussed in the third example in section 4.3, the user wished to copy a particular theorem and all its ancestors from a theory. It would not be particularly hard to implement this function as it would just be a recursive version of functions already written. However, a potential drawback is that it may slow the process down a lot, since for a large file working out all the descendents and all the ancestors of a theorem may be a large task.

Increasing Efficiency

An aspect of the system that could benefit from improvement is the run time. Decreasing the run time for the system could increase its usability considerably, especially for the larger files for which the system is particularly suited. One the major factors behind the slowing down that occurs towards the end
of large files is the way that the menus are compiled. The coding at the moment removes the existing menu, runs the menu creating function to build the menu up again, this time in its up to date state, and then adds this new menu to the menu bar. This is rather inefficient, as a lot of work is done repeatedly, especially in large file. A better approach would be to retain the existing menu and add the new information to this. This could be a little complicated, as new information would need to be added to various lists that form submenus of the menu itself, rather than just tacking an item on to the end of the menu. However, although it may be time consuming and problematical to recode the menu creation in this way, it would certainly be worthwhile if further work is to be done.

**Keyboard Shortcuts and Using the Minibuffer**

One aspect of DPG that may not appeal to certain users is that the options are provided through menus which require the use of a mouse. Some users may prefer to use all the functionality through the keyboard. It is possible to evaluate any function by using the `Eval` function in the minibuffer. However, this is not a very user friendly option. For a start it is requires more keystrokes than simply typing the commands into the minibuffer; this is going to be a rather small difference, but if users are resorting to this often it may become annoying. Secondly, many of the functions take several arguments that are passed to them by the system; it would certainly be annoying for the user to have to type in all these arguments each time he wished to evaluate anything. The way that the menus operate, with the user simply choosing an option rather than giving more detailed information, means that most of the functions called by the menu do not take arguments. The exceptions to this are context-sensitive menus, where some kind of context is clearly necessary.
For the context-sensitive menus in this project, this is provided by an input event; the system finds out in which span this event took place and can then use this span as an argument without requiring the user to put input any further information.

The kind of functions that are designed for the menu are not immediately suitable for keyboard shortcuts, since their role is to put the required functionality into menu items or submenus in case the user wishes to call them. The lower level functions that are called by these menu functions are not currently interactive and hence could not take input directly from the minibuffer. However, it should be simple to make them interactive. If the cursor was in a span in the theorem file when the function was called it would be clear what information would be relevant, just as it is clear what context-sensitive menu to bring up. If the cursor was not in a span or was in the definition file then it would be necessary to prompt the user, after the function name had been entered, as to which theorem or definition they were interested in.

Another addition that would be useful is keyboard shortcuts where a key would be bound to calling a particular function; for example, F1 may be bound to ‘highlight children’. This would also be possible if the functions within DPG are made interactive. Again, the system would need to either be aware of the context or else, if the context was not relevant, would need to prompt the user as to what theorem or definition they were interested in.

It does not seem that implementing this functionality would be particularly hard. Much of the groundwork has already been done, and the two tasks are rather similar. This may increase the appeal of the system to many users.
Providing Dependency Information for Parent Files

A decision was made early on in the project to deal only with dependencies within a file. Given the way theories are usually generated and the time limitations of the project, this was a reasonable decision. However, if the project were to be extended, it may be worth including dependency information for parent files and perhaps all ancestor files.

Proof General already tracks dependence between files so it would not be hard to find out what the parent files were. All that would then be necessary would be a slightly different filtering process, where the system searched not only for dependencies within the file, but also those from the parent files, which the function could be informed of by Proof General. It then seems preferable to sort the dependencies into their various file groups. Those from the file at hand could be reported back to the user in the same way as is currently done. Another menu item could then be added to some or all the menus to allow the user options to view the dependencies within other files, and perhaps to move to these parent files.

Since this functionality is merely a simple extension of the functionality of DPG, this is likely to be easy to implement. A more serious question is how this additional functionality would affect the efficiency of the system. When DPG is only tracking the dependencies of one file there are already serious questions about its efficiency. If it were also expected to deal with several other files, this may lead to the system taking an unreasonable amount of time to process files. However, the section above on increased efficiency contains some suggestions as to how the efficiency of the system may be improved, and if this is achieved then perhaps this problem may not be so serious.
Dividing the Definition File

Some of the drawbacks of the system lie in the fact that it is based on Proof General, which, although extremely useful in many circumstances, also has some disadvantages. One of the most significant of these disadvantages is the fact that the definition file only contains one span. This means that it is not possible to attach properties to individual definitions, as is possible with theorems. Hence all information must be given in a general menu. This is much less convenient and clear for the user, and also means that commands such as ‘move to definition’ are executed in a rather haphazard manner. However, improving this would be a very complicated task indeed. It would require serious reworking of the Proof General Isabelle files, since the files are written on the basis that the definition file will contain only one span. A more serious problem is that it can be hard to determine how the file would be split up, since the format of the definition file is by no means standard. For example, it is possible to state the constants contained in a definition in a different location from the actual definition, e.g. at the beginning of the file. Therefore, it would not be possible to contain both of these within a single span unless the application of spans was rethought. Also, there are many different kinds of definitions that are given different layouts, and identifying all of these may be hard.

Another problem is that it would be hard to name the definition spans. Theorems have the convenient habit of generally ending in a qed line containing their name, so for these theorems it is easy to give a name to their span. However, this is not so in definitions. Without a name attached to the span, the existence of the span would not be so useful, since the attachment of items in the dependency information to particular span is done using names. Therefore, the problems in overcoming this drawback are great and this is
the reason that it was not attempted during this project. One possible way of improving this situation is by restricting the way in which users write their definition files. Although this would make work for the system much easier it would negate the system's aim to be an often used tool, since users are likely to be extremely unwilling to change their regular method of writing theories. However, although it is hard to see how this would be done, if extensive further work were carried out to improve this system then this is one aspect that should be considered.

This discussion concerning the definition file is Isabelle-specific and of no relevance if the theorem prover being used is not Isabelle. Other theorem provers have different ways of storing definitions, with many theorem provers, such as Coq and Lego, storing them in the same file as the theorems. Determining individual definitions may not be such a problem in these kinds of theorem provers, whilst mechanisms for determining whether an object is a theorem or a definition may have to be worked out individually for each theorem prover.

**Retracting Proof Steps**

Another drawback of the Proof General system is that proof lines cannot be retracted one by one if a change is made. This is only possible if the first line retracted is the last one proved, and so on. It is not possible to retract a couple of lines in the middle of the proof to check some alterations that have been made here. The reasoning behind this is probably that other parts of the proof may be dependent on whatever has been changed, and hence allowing this kind of editing would allow the possibility of the proof becoming inconsistent. However, this is rather inconvenient for the kind of editing intended within the system, since it is desirable to alter and reprove
any theorem in the proof whenever the dependence information indicates that this may be necessary. The functionality of DPG could be used to allow this kind of mid-proof retraction by retracting not only the theorem itself but also all of its descendents. In this way it could be ensured that no inconsistency arose in the proof. Implementing this would be easier once recursive functions have been written to track ancestors and descendents. However, a large reworking of Proof General would still be necessary in order to use the dependency information to automatically retract parts of a proof. This is not currently done in Proof General; automatic retraction only occurs if an option is chosen to retract the whole proof.

Proof General Summary

In general, the fact that the system is based in Proof General introduces limitations to it. In retrospect, it still seems a reasonable decision for the purposes of this project to base the system in Proof General since this allowed the production of a reasonable and workable system within the time limit. However, if a similar but larger and more complete system were to be devised without the time restraints it may be more desirable not to base it on Proof General, but to make it independent. An alternative to this would be to introduce changes to Proof General to incorporate all the desired changes and alter all those parts of it that are causing problems in DPG.

8.2 Implementing the Future Project

In addition to the numerous improvements that are feasible within the implemented system, there is also the possibility of implementing the specification
for the envisaged future project. A brief discussion of the possibility of implementing the larger project in the light of the lessons learned during the implementation of this project is included below.

The aims of the intended larger project as follows:

1. **Initialisation**: in this part of the process the editor should run and record the initial proof and record the names of the constants, theorems and definitions.

2. **Editing**: the user can either edit the file directly or else some kind of user interface would be introduced for this; for example, a button named ‘change name of definition’ could be added that would give the user a list of definitions to choose from and allow him to edit the name.

3. **Propagation**: the system would track the dependencies of the theorem or definition that had been changed and would propagate this change through the file.

4. **Checking the proof**: the proof would be rerun and the places where it broke down would be noted and returned to the user.

The implemented project has focussed on the first part of the third aim, whilst also addressing the first and second parts to some extent, though not necessarily in the originally intended way.

One of the main ways in which the implemented project differed from the future project is that the implemented project is passive and fully interactive. That is, the system does not attempt to alter the theory in any way, merely
to provide information to make it easier for a human user to alter the theory. In contrast, the future project is envisaged as being fully automated, or at least partially automated. Ideally, the system would automatically propagate the changes through the theory, automatically run the proof and then point out to the user the places where the proof failed, providing a patching editor so that the user may change these failure points. It would be ideal to get the proof to explore as many branches as possible before returning failure to the user; i.e. if the proof fails on one branch, this failure point will be noted and then the system will attempt to continue down the next branch until success is achieved or a failure point is reached, and so on. Once the proof has progressed as far as possible down each branch, the system will return all the failure points to the user to be patched, and then attempt to continue.

There are problems inherent in this idea. It may be that it is always more useful for the user to examine and attempt to change the proof before it is run again, as it is a waste of time to attempt to prove a theory that is known to be wrong. Some systems such as Cynthia, discussed in section 7.1, provide similar functionality in a completely automated fashion, but these generally operate in constrained circumstances. If the domain is to remain theorem proving, then the question of how relevant the future project will actually be needs to be addressed more seriously.

Another possibility for the future project is to apply it to a domain other than theorem proving. The domain would need to be subject to the laws of logic, and for any real life domain this may mean creating a slightly artificial interpretation of it. However, there are several domains that may suit, for example, law, politics and Socratic dialogue. It would probably be best to spend more time working on the initial system before converting it to be
suitable for the given domain. It would then be necessary to spend some time translating the facts and theories in that domain into some sort of logic so that it would be processable by the system.

There is much unexplored work within the future project, but it is hoped that the information in this project is sufficient to provide some initial ideas and inspiration.

8.3 Concluding Remarks

The notion of dependence is of prime importance in the theorem proving community. It is used during theorem proving, theory maintenance and theory adaptation. Currently there are not many systems that provide dependency information within theorem provers, and the aim of this project has been to implement a basic system to return dependency information during theorem proving within the context of a popular theorem prover.

This aim has been successfully achieved. There is much discussion in this chapter as to all that was not achieved in this project that would be desirable; however, what has been achieved in the project is the implementation of a useful and functional tool to assist in theorem proving. It was simply not feasible to implement all the possible functionality in a project of such restricted length, and the work done so far has not only implemented an important subsection of this, but also provided much insight into what would be useful should a longer project be attempted and which parts of the system would need to be implemented differently in order to produce a more comprehensive and efficient system.
This project has also provided the groundwork for a project of wider scope, incorporating not only the additional functionality desirable for the existing system, but also many other features to enable formal verification of common sense reasoning. Again, much of what has been implemented would need to be rethought in order to proceed with this idea. However, the act of implementing the existing system has been extremely useful in forming a firmer basis on which to build ideas of how best to implement this future project.

In conclusion, the project has largely achieved its aims. It is hoped that the system as it stands may be of use to the theorem-proving community, and it is hoped that the research contained in this project will assist in the building of other, more advanced, systems to track dependency information in theorem proving.
Bibliography


A. Bundy. Dynamic knowledge representation. Internal note, Mathematical Reasoning Group, Division of Informatics, University of Edinburgh.


Appendix A

Program Code

The following page contains the additional code added to the Proof General file proof-script.el. This code is part of a larger function, predefined in this file, which is called every time a span is processed. Hence, this code is called for each span.

This code is the top-level control loop of DPG. Most of the code written for DPG is contained in a file names proof-depends.el which can be found in the generic Proof General directory. However, every function within proof-depends.el is called either directly or indirectly by the code overleaf.

Some of the names that are used in this code are as follows:

- gspan is the span that is currently being processed.
- proof-last-theorem-dependencies is the variable in which any dependency information returned by Isabelle will be stored.
- proof-deps-menu is the name of the predefined menu in the theorem file.
- buffer-file-name is the name of the buffer which is currently being processed by Isabelle.
- thy-add-menus is the function within the Isabelle files of Proof General that compiles the definition file menus.

All of the other names that appear in this code are either defined within the code, or else are names of functions and variables defined in proof-depends.el.
(if proof-last-theorem-dependencies
  (progn (if (boundp 'proof-deps-menu) (easy-menu-remove proof-deps-menu))
    (set-span-property gspan 'dependencies proof-last-theorem-dependencies)
    (set-span-property gspan 'keymap span-context-menu-keymap)
    (if buffer-file-name
      (let* (((buffer-file-name-sans-path
        (car (last (split-string buffer-file-name "/"))))
          (buffer-file-name-sans-extension
            (car (split-string buffer-file-name-sans-path "\.")))))
        (dependencies-within-file-list
          (dependencies-within-file-list
            (dependencies-within-file-list
              (dependencies-within-file-list
                (dependencies-within-file-list
                  (dependencies-within-file-list
                    (dependencies-within-file-list
                      (dependencies-within-file-list
                        (dependencies-within-file-list
                          (dependencies-within-file-list
                            buffer-file-name-sans-extension)))))
                    (dependencies-within-file-list
                      buffer-file-name-sans-extension)))))
              (dependencies-within-file-list
                buffer-file-name-sans-extension)))))
      (set-span-property gspan 'dependencies-within-file
        depcs-within-file-list)
      (update-dependents depcs-within-file-list
        buffer-file-name-sans-path
        buffer-file-name-sans-extension
        (span-property gspan 'name))
      (proof-menu-define-deps buffer-file-name-sans-path)
      (easy-menu-add proof-deps-menu proof-mode-map)
      (let ((thy-file (concat buffer-file-name-sans-extension ".thy")))
        (find-file-noselect thy-file)
        (save-excursion
          (set-buffer thy-file)
          (thy-add-menus buffer-file-name-sans-path)))))))))

(setq proof-last-theorem-dependencies nil)

(setq proof-thm-names-of-files
  (merge-names-list-it (span-property gspan 'name)
    (buffer-name (span-object gspan)) proof-thm-names-of-files))
Appendix B

Worked Examples - Example 1

![Screenshot 1.1]

Figure B.1: Screenshot 1.1

The user has altered the definition ‘pnat_add_def’. The user then wishes to see which theorems may be affected by this change, so he chooses to highlight the children of this definition.
The command chosen above causes the system to jump to the theorem file and highlight in orange all the children of the chosen definition. The user may then move through the file observing the highlighting.

The user spots something that needs to be altered in the theorem ‘pnat_add_less_monol’ and changes this. They then wish to see how this might affect the children of this theorem, so they bring up the context-sensitive menu and choose the option ‘highlight all children’.
Figure B.4: Screenshot 1.4
The user updates all the children of ‘pnat_add_lessto_monol’ to his satisfaction and, since the children theorem he is in has no children of its own, he wishes to move back to the parent theorem. So he chooses the ‘move to parent’ option and picks ‘pnat_add_lessto_monol’ from the list of parents.

Figure B.5: Screenshot 1.5
The user now feels that he has updated everything, but wishes to take another look at the children of the original definition to be sure. Therefore, he uses the theorem file general menu to choose the option ‘highlight children of definition’ before choosing the appropriate definition.
The user is now confident that all the necessary changes have been made and so wishes to remove the highlighting. He chooses the ‘unhighlight’ option which is present in all of the menus.
Appendix C

Worked Examples - Example 2

Figure C.1: Screenshot 2.1

The user wishes is currently created a new theory and decides that the theorem ‘pnat_add_assoc’ will be useful in this new theory. Therefore he wishes to copy this theorem to the new theory, but must also copy all the parents of this theorem in order to make it provable in the new theory. First of all he uses the context-sensitive menu to view a list of all the relevant definitions.
The user may wish to move to the theory file to observe and copy these definitions. Therefore he chooses the ‘Move to thy file’ option, which is available in the theorem context-sensitive menus and the general theorem file menu. He may wish to choose to open the theory file in another window rather than the same one so that he can observe both files at once.

Once the definitions have been copied, the user may wish to return to the original theorem; he can choose to ‘move to theorem’ option within the definition file menu and pick ‘pnat_add_assoc’. Note: the theorem list is longer than shown, only a small section has been left in the diagram.
Figure C.4: Screenshot 2.4
The cursor is now in the correct position in the theorem file. The user may now wish to view what theorems will need to be copied; he does this by using the ‘highlight parents’ option from the theorem context-sensitive menu.

Figure C.5: Screenshot 2.5
All the parents theorems of ‘pnat_add_assoc’ are now highlighted. The user can copy all of these, and then check whether any of these have parents themselves which must also be copied. At any point he has access to a full list of parents of the theorem he is in.
Appendix D

Evaluation Questionnaire

Evaluation Questionnaire

a. Functionality

(a) What functionality does the system provide that is not currently available?
(b) What advantages does the editor have over existing methods?
(c) If there are advantages, how significant are they?
(d) How useful is providing dependence information during theorem proving?
(e) How frequently would you require this functionality?
(f) What extra features do you think could be incorporated into the editor in order to make it more useful?

b. Usability

(a) Was the editor easy to use?
(b) Was the user interface helpful?
(c) Was the functionality of each command in the drop down menu clear from its label?
(d) Was it easy to find the correct command when you wished to do something?
(e) What additional features to the user interface do you feel would improve usability?
c. Comments

(a) Do you have any further comments about the usefulness of the project?
(b) Suggested improvements to the project?
(c) Any other comments
Appendix E

Glossary

Child A theorem T is called a child of a definition or theorem P if P is found in the proof for T. If T is a child of P then P is a parent of T. A definition cannot be a child. A child is the same as a dependent. The word child is usually used in preference to the word dependent as its meaning is clearer.

Definition A definition is an object in a theory that does not require justification; it is taken as a fact. Hence definitions do not have parents.

Dependency See parent.

Dependent See child.

Drop-down menu A drop-down menu is a menu that appears when the user clicks on an item in the menubar.

Extent An extent is a first class data type in XEmacs that describes a region of text. Extents can have properties attached to them.

Isabelle Isabelle is an interactive theorem prover written in ML.

Menubar A menubar is a bar at the top of the window that contains the names of all accessible drop-down menus. The user can view a menu of their choice by clicking on the appropriate name.

Overlay An overlay is similar, though not identical, to an extent but is found in GNU Emacs.

Parent A theorem or definition T is a parent of a theorem C is T is found in the proof for C. If T is a parent of C then C is a child of T. A parent is the same as a dependency. The word parent is usually used in preference to the word dependency as its meaning is clearer.
Popup menu A popup menu appears when the user right-clicks inside a buffer.

Proof General Proof General is a generic user interface to enhance the theorem proving experience of a user. It is currently compatible with Isabelle, Coq and Lego.

Span A span is an abstraction in Proof General to hide the differences between extents and overlays. Hence spans can be used in both XEmacs and GNU Emacs

Theorem A theorem is an object in a theory that requires justification. Hence theorems cannot be accepted unless they have been proved.

Theory A theory is a set of objects such as definitions, theorems, lemmas and axioms. The objects in a theory are generally connected in some way and there is much interdependence.