A Proof-Centric approach to Mathematical Assistants

Lucas Dixon and Jacques Fleuriot

School of Informatics, University of Edinburgh,
Appleton Tower, Crighton Street, Edinburgh, EH8 9LE, UK

Abstract

We present an approach to mathematical assistants which uses readable, executable proof scripts as the central language for interaction. We examine an implementation that combines the Isar language, the Isabelle theorem prover and the IsaPlanner proof planner. We argue that this synergy provides a flexible environment for the exploration, certification, and presentation of mathematical proof.

Key words: proof planning, Isabelle, Isar, IsaPlanner, theorem proving, proof assistant

1 Introduction

The central medium for a mathematician's work is proof. Thus, it seems natural for a tool aimed at helping mathematicians to focus on aiding the presentation, assisting the exploration, and certifying the correctness of proof. Typesetting software, such as Latex, has long been popular for presenting proofs. In this article, we introduce and analyse a proof-centred approach to mathematical assistants which tries to unify these different tasks.

Our approach is centred around a representation of proofs that is human-readable, human-writable, and machine-checkable. We use a proof planning framework to provide automation for proof construction but preserve the ability for users to write the proofs directly. The novel feature of this approach is the use of a high level proof language as the central medium for interaction between the user and the mathematical assistant.

Email address: {lucas.dixon, jacques.fleuriot}@ed.ac.uk
(Lucas Dixon and Jacques Fleuriot).

In this article, we introduce and analyse an embodiment of this proof-centred approach which combines recent developments in interactive and automatic theorem proving. It uses structured proof texts to present the proofs, interactive theorem provers to verify their soundness, and proof planners to automate their generation and aid proof-exploration. These are implemented in the Isar language [1], the Isabelle theorem prover [2], and the IsaPlanner proof planner [3], respectively.

We argue that the central requirement of proof-centred mathematical assistants does not lie in more powerful automation, but in automation that leaves the user with a readable proof state while making some kind of 'progress'. We cite the rippling strategy as one such proof technique. We also argue that it may be better to have weaker proof tools that facilitates exploration of proof rather than more powerful ones that either succeed, or fail without helping the user. This leads us to the main challenges and directions for future work.

We first introduce the proof-centric approach highlighting the key challenges (§2). We present Isabelle, Isar and IsaPlanner (§3) which provide an initial implementation of our approach. We then consider the need for sufficient mathematical vernacular (§4), modular mathematical development (§5), knowledge management (§6), and support for exploration of proof (§7). This leads us to the pragmatic issues of user interface (§10) and proof presentation (§11), that are essential to the adoption of such an assistant by mathematicians. Throughout, we illustrate our ideas using our Isabelle/Isar/IsaPlanner combination, examining its suitability as a mathematical assistant and identifying the areas for future work. Finally, we describe related work (§12) and provide concluding remarks (§13).

2 Overview of the Proof-Centric Approach

This approach aims to support the interaction with and development of declarative, intelligible, and machine checkable proof. Such a representation of proof was initiated by the Mizar project [4] and several similar approaches to expressing proof have since been developed [1,5–10]. The main additions to Mizar's approach have been to make the proof language:

- generic in the sense of being independent of the underlying logic, as implemented in Isar [1] and Declare [7],
- extensible in order to support the theory level additions to the basic proof language, as implemented in Isar [1] and SPL [5],
- support underspecification, such as missing steps in a proof. This has been described by Wiedijk's as a notion of proof-sketches [11], and by Autexier et al. as under specification [6].

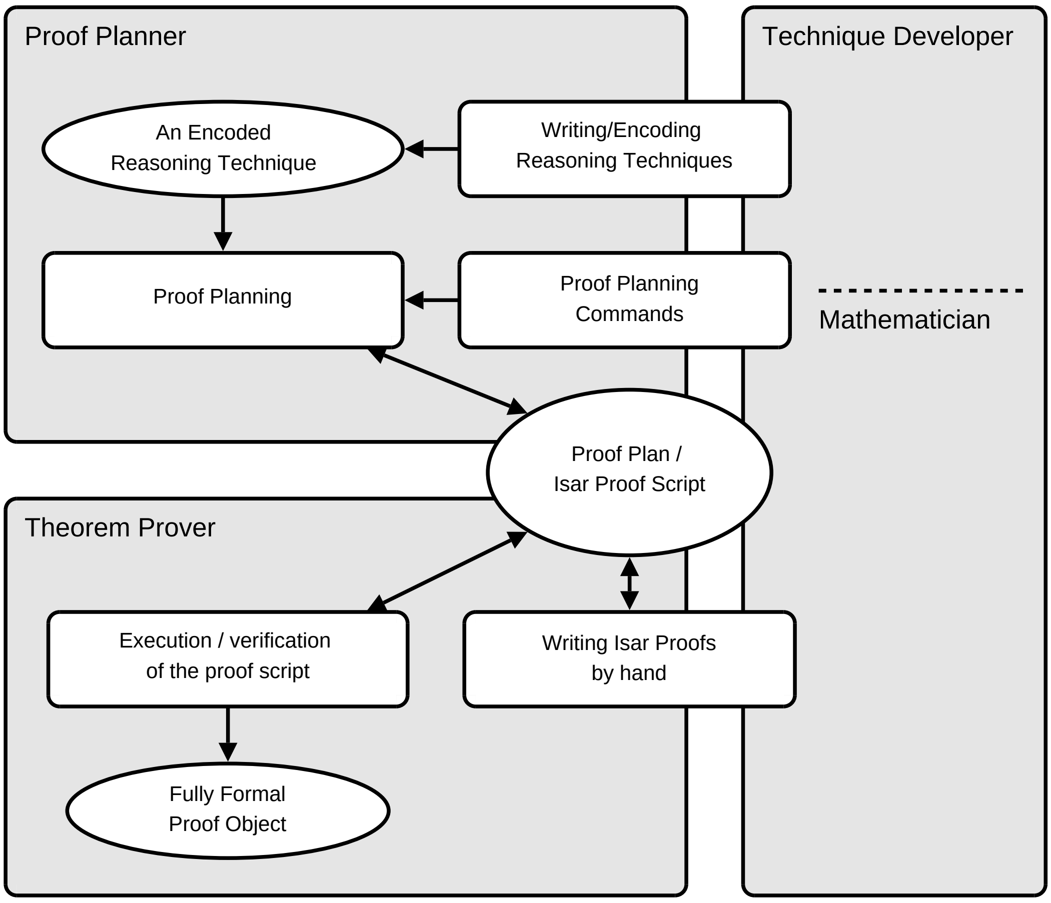
Proof Planner
Technique Developer
An Encoded Reasoning Technique
Writing/Encoding Reasoning Techniques
Proof Planning
Proof Planning Commands
Mathematician
Proof Plan / Isar Proof Script
Theorem Prover
Execution / verification of the proof script
Writing Isar Proofs by hand
Fully Formal Proof Object

Fig. 1. An overview of the interactions within a proof-centred mathematical assistant. The boxes represent processes and the circles data.

In our approach, we express the state of a proof attempt explicitly within the proof script by stating any unproved goals as statements justified by 'gaps'. Our work extends Wiedijk's notion of proof sketches by allowing the gaps to be annotated with references to proof techniques that can later be used in an attempt to complete the proof. We use this representation of proof as the central object of interest in the interaction between between the user, the proof planning machinery, and the underlying theorem proving. An overview that illustrates these interactions is shown in Fig. 1. The mathematician interacts with the proof assistant by selecting proof planning commands and by manually writing parts of the proofs. Writing Isar proof scripts by hand is currently the only means of expressing proofs for users of Isabelle. The combination of automatic and interactive writing of the proof scripts provides the user with a more automated but still flexible approach to guiding the proof system. We believe that the encoding of techniques requires a more developed knowledge of the underlying system, and thus we draw a line between the mathematician and the technique developer.

In what follows, we will use a running example of the proof that the sum of odd numbers up to n is equal to n^2. This example is sufficiently small to be clear and complex enough to illustrate the various characteristics of our approach. An Isar-style proof script showing an attempted proof of this theorem is shown in Fig. 2. The `gap` statements are the holes in the proof where the text in parenthesis suggests a technique that might be used to fill the missing step. The goal of such a representation is to provide a flexible approach to proof. In particular, proofs can be constructed in a top-down or bottom-up style, as well as explored in a backward or forward direction, while maintaining a consistent and readable presentation of the proof.

The annotation of gaps allows the partial application of proof technique. In particular, they can be unfolded in a step by step manner, where each unfolding results in a sub-proof script which can contain further gaps. The annotations on gaps are the names of techniques with which to continue the technique's application. By expressing the continuation of a technique explicitly, we also allow user interaction with the proof attempt: they can modify the proof script, including the continuation, or they can ask the system to continue. Being executed an the underlying LCF-style theorem prover ensures that the proof scripts are correct modulo the explicitly stated gaps.

A key feature of our approach is that proof scripts are equivalent to proof plans. This allows systems such as IsaPlanner, which manipulate proof plans, to provide the tools needed for manipulation of proof scripts. In particular, it allows techniques to express partial or complete behaviour in terms of readable proof scripts. It also provides the needed notion of techniques with continuations and lazy evaluation.

An interesting extension to traditional proof planning techniques arising from this work is that they can now be used in a manner not designed to just prove a subgoal, but also to explore just part of a possible proof attempt. Such techniques result in a modified proof script rather than just a new subgoal.

Working at the level of proof scripts provides proof planning with more information than is available to Isabelle's tactics. This allows the enncoding of techniques, such as Ireland's induction revision proof critic [12], that cannot be expressed as Isabelle tactics. It also allows the interactive use of proof critics [13] to perform modifications to a proof plan, and opens up the possibility for a notion of proof by analogy to another proof script. In summary, this presents a novel approach to interactive proof planning which allows the automatic manipulation and generation proof scripts to be interleaved with their manual construction.

A possible criticism of this approach is that it asks a lot from the representation of proof. In particular, it requires a careful balance between providing the user

```
theorem sum_of_odds: $(\sum_{i<n}\ 2 * i + 1) = n^2$
proof (induct $n$)
  show $(\sum_{i<0}\ 2 * i + 1) = 0^2$
    gap (simplification)
next
  fix $n$
  assume IH: $(\sum_{i<n}\ 2 * i + 1) = n^2$
  show $(\sum_{i<(Suc\ n)}\ 2 * i + 1) = (Suc\ n)^2$
    gap (rippling)
qed
```

Fig. 2. An example Isar proof script with 'gaps'.

with a fine level of control over the way proofs are presented, and expressing them in a form that can be manipulated and machine checked. The degree to which the Isar language suffices is likely to be a question of personal taste, although its usage among mathematicians will in the end answer this question.

3 The Foundations of an Implementation

In this section we introduce the main components of an implementation of our approach. Firstly, we outline the Isabelle proof assistant which provides the logical underpinnings and basic machinery for formal proof. We then describe the Isar language which allows proofs to be expressed in an intelligible form while allowing them to be machine checked using Isabelle. Lastly, we present proof planning in IsaPlanner, which manipulates and searches for declarative descriptions of Isar proofs.

3.1 Theorem Proving: Isabelle

Isabelle is a proof assistant written in ML which supports formal reasoning in a number of object logics [2,14]. Examples of such object logics include Zermelo-Fraenkel set theory (ZF), first order logic (FOL), higher order logic (HOL), and constructive type theory (CTT). Object logics are formed and manipulated by Isabelle's intuitionistic higher order meta-logic, which supports polymorphic typing and performs type-inference.

Formalisation of mathematics in Isabelle involves defining constants and types about which properties are then proved. Mixfix annotations are used to manage the parsing and printing for the concrete syntax of the underlying lambda calculus. Syntax translations support more complex relationships between the syntax and the underlying terms.

Soundness is treated by following the LCF design principle of having a fixed logical kernel containing the primitive inference rules. Additional *tactics* to perform higher-level proof steps are written in terms of these rules and previously proved theorems. The ML type system is then used to force theorems to be constructed only in this manner, thus reducing concerns about the soundness of new tools to the consistency of the logical kernel. This provides a disciplined approach to ensuring soundness while providing flexibility for the development of richer proof tools.

To ease and speed the proof process, Isabelle provides the user with a number of generic, as well as logic-specific proof tools. These range from simple mechanisms for combining theorems to fully automatic theorem provers. One of these is the generic simplification package which supports higher order conditional rewriting using previously proved theorems. The user can customise its behaviour by temporarily or permanently adding theorems to the simplification set. Other generic automatic tactics provided by Isabelle include a classical reasoner [15,16] and the automatic tactic which attempts to prove all subgoals by a combination of simplification and classical reasoning.

Another important requirement for practical theory development is the need for tools to support new definitions. In the methodology of conservative extensions, adopted by Isabelle/HOL, these mechanisms should not assert new axioms. Isabelle/HOL hosts an array of such conservative definitional mechanisms including support for inductively defined sets, inductive datatypes, types as sets, extensible records as well as the usual mechanisms for defining functions and types.

One important feature of proof planners such as Omega [17] is their use of external tools to provide additional calculational and proof support. This raises the question of how to integrate external systems into a proof-centred mathematical assistant. This is an especially pertinent point if we wish to take advantage of the significant developments in computer algebra systems. In Isabelle, interaction with other systems is supported through an 'oracle' mechanism which allows Isabelle to treat conjectures as tagged theorems. The dependencies on the oracle can then be tracked automatically. This provides a pragmatic yet disciplined approach to soundness while supporting the use of external systems. The main problem then becomes managing a translation between systems, which is dependent on the exact systems being integrated.

To provide a further degree of belief in the correctness of a proof, Isabelle produces proof terms that describe the theorem's derivation in terms of the primitive logical inferences. This supports the validation of proof using a small proof checker independent of Isabelle. However, such proof terms provide far too much detail to be humanly checked, let alone easily readable. Furthermore, as well as failing to capture the 'idea' behind a proof, these do not provide a

```
Goal "$\sum_{i<n} 2 * i + 1 = n^2$";
by (induct_tac "n" 1);
by (Simp_tac 1);
by (Simp_tac 1);
by (simp_tac (simpset() addsimps [power2_eq_square]) 1);
qed "sum_of_odds";
```

Fig. 3. An example ML procedural proof for the sum of n odd numbers in Peano arithmetic.

useful way of storing proofs. This is due to their verbose nature, the difficulty of modifying them, and the difficulty in reusing them. It is thus normal for users of Isabelle to store *proof scripts* in a file that contains the tactic commands to re-derive the proofs.

Before the development of Isar, proof scripts were typically expressed 'procedurally' as a sequence of ML proof commands. For example, a procedural-style proof that the sum of odd numbers up to n is equal to n^2, is shown Fig. 3. In this proof script, the `by` function applies a tactic to the current goal and `qed` stores a proved theorem for later use. The tactic `induct_tac` selects and applies an induction scheme, and `Simp_tac` and `simp_tac` are tactics that simply the goal. The latter simplification tactic is given an explicit simplification set, which in the above proof includes the lemma (`power2_eq_square`: $n*n = n^2$). Although these proofs support reuse of tactics, they are generally not readable off-line, that is without tracing through the goals resulting from each proof step.

3.2 Readable, Executable Proof Scripts: Isar

Isar aims to provide a language which is both human-readable and machine-checkable [1], following the style used by the Mizar system [4]. It provides a natural deduction style of writing proofs for the Isabelle theorem prover and allows abbreviations using higher order pattern matching. It is independent of the object logic and has been instantiated for Isabelle's HOL, ZF, and FOL, for instance. Furthermore, it has been designed in an extensible fashion which supports defining additional domain specific elements.

A small example Isar script, proving that the sum of odd numbers up to n is equal to n^2, is shown in Fig. 4. This script shows a feature of Isabelle that allows "$\lambda\ n.\ \sum_{i<n} 2*i+1$" to be abbreviated to $?sumto$ by unifying the higher order pattern "$?sumto\ n = \_$" with the main goal, where $?sumto$ is a variable and "_" is a wildcard.

Another feature of Isar shown in this proof script is the support for a calcula-

theorem `sum_of_odds:` $\sum_{i<n} 2 * i + 1 = n^2$ `(is` $?sumto\ n = \_$`)`
proof `(induct` n`)`
  show $?sumto\ 0 = 0^2$ **by** `simp`
next
  fix n
  assume `IH:` $?sumto\ n = n^2$
  have $?sumto\ (Suc\ n) = ?sumto\ n + Suc(2 * n)$ **by** `simp`
  also have $\ldots = n^2 + Suc(2 * n)$ **using** `IH` **by** `(simp)`
  also have $\ldots = (Suc\ n)^2$ **by** `(simp add: power2_eq_square)`
  finally show $?sumto\ (Suc\ n) = (Suc\ n)^2$ `.`
qed

Fig. 4. An example Isar proof for the sum of n odd numbers in Peano arithmetic.

theorem `sum_of_odds:` $\sum_{i<n} 2 * i + 1 = n^2$ `(is` $?sumto\ n = \_$`)`
proof `(induct` n`, simp, simp)`
  fix `n`
  show `Suc (2 * n + n) = (Suc n)` **by** `(simp add: power2_eq_square)`
qed

Fig. 5. An example Isar proof where the backward proof step is so large that it obscures the proof.

tional style of proof, in the sense of iterated chains of transitive reasoning. In Fig. 4, this is indicated by the sequence of commands "**have**", "**also have**" and "**finally show**". The ability to name assumptions, for example by the calling the induction hypothesis "`IH`" in Fig. 4, further supports calculational and other forward styles of proof.

We remark that although backward proof is supported within the language, if backward steps are too large or numerous the proofs are once more unreadable and procedural in style. For example, the proof shown in Fig. 4 can be expressed in a briefer, but more procedural form, as shown in Fig. 5. In this example, we show a proof script in which the backward proof step includes induction, simplification to solve the base case, and a simplification that applies the induction hypothesis to the step case. The resulting goal is then made explicit and proved by adding the lemma `power2_eq_square` to the simplifier. However, because of the large backward proof step, it becomes unclear why showing this subgoal proved the main theorem. Furthermore, the combination of proof steps in the **proof** command, are essentially procedural as they hide the structure of the inductive proof. This also shows that some discipline is needed to write Isar proof script that are readable.

Internally, Isar operates as a state machine with transitions that incrementally parse elements in the proof language. This machinery has two main modes, one of which supports forward proof by allowing the user to express statements and one of which supports backward proof by allowing the user to apply tactics.

Fig. 6 shows basic elements of the language and how they effect Isar's mode. The transitions take arguments which are omitted for the sake of clarity. As mentioned, the generic Isar machinery is designed in an extensible fashion that supports domain specific additions. These include new transitions, and extensions to the notion of context. Such additions allow new notations for proof, such as the calculational style described earlier, to be added to the basic Isar language.

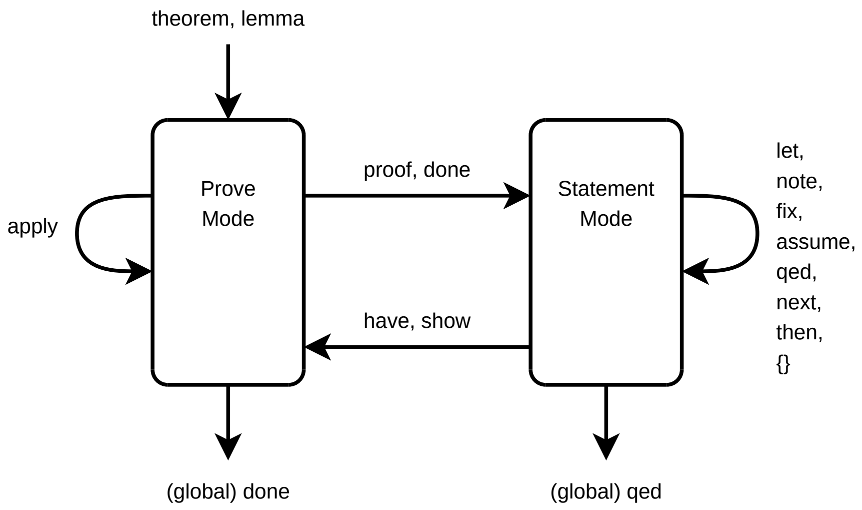

Fig. 6. The basic Isar state machine transitions for parsing a proof.

While the Isar language makes it easier to read proofs, supports abbreviations, and simplifies forward reasoning, the language itself can be difficult to learn. Moreover, the lack of proper support in exploring the application of tactics makes writing proofs slower and more arduous. The reason for this difficulty in exploration is that to examine the effect of applying a tactic, a user must take a backward, procedural proof step. If this solves the goal, then the user will usually replace the backward step with a single tactic justifying the proved statement. However, if the tactic fails to solve the goal, to maintain readability, the user either needs to modify the tactic and try again, or remove the tactic application then state and prove an intermediate result in a forward manner. Only if the user is able to second guess the level of automation available can they directly express the intermediate steps. As a result, it is common for users to explore and find a proof using a procedural style, working backward from the goal, and then rewrite the proof in Isar's structured forward style. Solving these problems involves helping the user with the syntax of the proof language and supporting their knowledge as to the coverage of existing proof-automation. This lack of proper support for exploring Isar style proof is one of central issues that we try to address using proof planning.

3.3 Proof Planning: IsaPlanner

Proof planning is a paradigm for proof automation that focuses on providing mechanisms for encoding heuristic and meta-level guidance [18,19]. It tries to capture common patterns of reasoning for families of similar proofs in terms of objects that we shall call *reasoning techniques*. Proof planning involves searching through the ways that these encoded techniques can be applied to a conjecture. This derives an abstract description of the proof known as a *proof plan* which is typically a declarative representation of a tactic tree that can be executed in a theorem prover to derive a fully formal proof.

IsaPlanner is a generic proof planner for Isabelle that expresses proof plans as Isar proof scripts [3,20]. This allows the proof planning process to be interleaved with the proof plan's execution and supports use of the powerful tactics already available in Isabelle. This has been used to develop an efficient implementation of the rippling technique [21,22]. This version of rippling has been combined with induction, lemma calculation, and a generalisation critic. This combination provides Isabelle with a powerful tool for inductive theorem proving that supports the automatic conjecturing and proof of lemmas.

The underlying approach in IsaPlanner involves breaking proof planning into steps that can be viewed as 'snapshots' of the process. Each snapshot is referred to as a *reasoning state* and is a triple containing the current proof plan, contextual information such as the annotations used during rippling, and an optional next reasoning technique. Steps in proof planning are lazily unfolded by applying the continuation technique to the state in which it is contained. Reasoning techniques are thus functions from a reasoning state to a collection of new reasoning states, where the resulting states represent the possible ways that the technique can be applied. This produces a space of possible ways of unfolding a technique and to which a search strategy, such as depth-or best-first search, can be applied. One feature of IsaPlanner's approach to search is that strategies can be stacked, allowing for example a best-first search to be applied within the context of depth-first one. Such stacking of search allows additional heuristic knowledge to be used for parts of the search, where applicable, without having to derive a more complex heuristic function that encompasses multiple search strategies. One example where this can be used is the rippling technique, where the rippling measure provides a natural heuristic for best-first search.

The basic language for encoding techniques in IsaPlanner resembles a tactic language. It has operations such as `THEN`, `OR` and `REPEAT`, which can be used to combine existing techniques. However, it also supports constructs that cannot be expressed within most tactic languages, including that of Isabelle, or within other proof planners, such as λ*Clam* [23]. Such constructs include `MAP` and

`FOLD` which apply functions over the lazily evaluated search space. By using reference variables, these can also support the sharing of information between both *or* and *and* branches within the search space. For example, they can be used to express, in a concise manner, a generic mechanism for caching portions of search space. This approach, being based on ML functions, allows the language to be extended. This is done in a disciplined, domain-specific way by defining the basic language to be independent of any object-logic. Further techniques can then be organised into Isabelle's theory hierarchy which allows them to be inherited.

Proof planning also aims to provide descriptions of proof at varying levels of detail. IsaPlanner's representation of proof plans as declarative descriptions of Isar proofs supports this by allowing a user to 'unpack' a proof planning command into a script which may contain further proof planning commands. This allows proofs to be examined at varying levels of detail. Moreover, techniques can also be constructed this way, being designed to leave 'gaps' in the proof which will be returned to at a later point, or changed into assumptions of a modified conjecture. This allows IsaPlanner to be used to automatically construct complete or partial Isar proof scripts.

4 A Platform of Formalised Mathematical Theories

Perhaps one of the most obvious requirements for practical formalised mathematics is the existence of a sufficient mathematical vernacular. This is necessary even to state a conjecture. In proof assistants with support for powerful customisable tactics, such as the simplifier and classical reasoner in Isabelle, these tactics also require some configuration.

Isabelle's higher order logic sports a large theory library of formalised mathematics developed as conservative extensions of the object logic, which avoids introducing new axioms that otherwise weaken the guarantees of consistency. It includes developments within nonstandard analysis [24], a formalisation of Hilbert's axioms for geometry [25], and mechanisations of topology and vector spaces [26], among many other [27]. Recently, Isabelle has also successfully imported all the theories from the HOL system. While the theories of Isabelle/HOL are large with respect to most systems, and are still growing quickly, the other logics within Isabelle have much smaller theory developments and are growing at a slower pace. We are aware of only one system with significantly more mathematical theories than Isabelle, namely the Mizar system's huge journal of formalised mathematics.

However, even the mathematical libraries of these long established systems are not sufficiently developed for research level mathematics. We note that these systems still lack many important basic theories, for example Isabelle currently lacks a formalisation of linear algebra. Moreover, the existing theories do not contain many basic theorems, which requires users to derive them as needed. This is because users tend to only prove the theorems they need.

At present, formalisation is still a slow and difficult task. One approach to overcoming the need for such fully developed theories is to introduce new axioms. This provides a shortcut at the expense of the soundness guarantees. However, even if the user boldly asserts new axioms for a theory, systems with powerful user configurable proof tools, such as Isabelle, still involve the demanding task of configuring these tools appropriately.

We believe that a useful area of future research would be to investigate automatic configuration of proof tools, or alternatively to examine and implement other techniques which require less attention to their configuration. An example of such a tool, currently implemented in IsaPlanner, is a higher order version of rippling [21]. This allows any rule to be supplied without damaging the behaviour of the proof method.

The existence of large formalisations of complex mathematics provides strong support for the notion that the current formal systems are capable of expressing research level mathematics, and indeed are capable of being used to verify the correctness of research mathematics. However, to make this a practical affair requires significant support. In the following sections we examine the needed support.

5 Modularity

Support for developing and using theories in a modular fashion is essential for mathematics as the subject thrives on combining and relating existing theories in novel ways. Many areas of mathematics also reason about the theories themselves. For example, in abstract algebra, it is common to talk about groups and to reason about them. For mathematical reasoning about such theories, they must be treated as first class citizens.

Appropriate mechanisms for modularity can also be used to side-step the difficulties of developing large mathematical theories. The user can simply specify their theory as being dependent on a set of theorems that are assumed to have been proved earlier. Thus the provision of a suitable mechanisms for modularity is an important characteristic of a mathematical assistant.

In Isabelle, developments are organised into *theories*. These are the course-grained basic objects for organising mathematical development as well as storing constants, types, sorts, syntax information, theorems, and contextual information used by proof tools. Theories are outside the logic, in the sense that Isabelle cannot reason about a theory as an object in its own right.

Isabelle's theories provide local name spaces and support inheritance. Inheriting from several theories merges their contents, and merges the information used by the corresponding proof tools. Isar and IsaPlanner fit any extensions into the theory hierarchy and provide appropriate merging operations. This supports extending the Isar language and IsaPlanner techniques in a domain specific way.

The simplest mechanism for theorem reuse in Isabelle comes from the polymorphism in the logic. This allows types to be defined in terms of type variables which can then be instantiated. For example, lists are defined in this manner. Higher order unification then allows proof tools to match polymorphic theorems to any instance of the general type.

Modularity is also provided by Isabelle's axiomatic type classes [28]. These allow classes of types to be defined in terms of basic properties that hold for the class. From these, theorems can be proved about the objects within the type-class. Isabelle's unification supports type-classes and thus the proof tools fit naturally. Unfortunately, type classes are limited in their expressivity. For instance, they can only be dependent on a single type variable.

Isabelle's Locales provide another infrastructure for modular proof development [29] that is more expressive than type classes. This supports modularity using Isabelle's meta-logic in terms of parameters, that correspond to abstract constants, which are fixed over a collection of assumptions. For example, a semi-group can be specified as follows:

```
locale semigroup =
  fixes prod :: "'a ⇒ 'a ⇒ 'a" (infixl "·" 70)
  assumes assoc: (x · y) · z = x · (y · z)
```

Any theorem proved within this locale is implicitly assuming the statement `assoc` and each occurrence of the constant "·" is in fact a variable quantified using Isabelle's meta-level universal quantifier. For example, the theorem $(w \cdot (x \cdot y)) \cdot z = (w \cdot x) \cdot (y \cdot z)$ proved within the `semigroup` locale, corresponds to the meta-logical theorem:

$$\forall p.\ (p\ (p\ x\ y)\ z = p\ x\ (p\ yz) \Longrightarrow p\ (p\ w\ (p\ x\ y))\ z = p\ (p\ w\ x)\ (p\ yz))$$

An interesting feature of this kind of modularity is that Locales are expressions within Isabelle's meta-logic, and can thus be part of formulae. This allows a certain amount of reasoning about the modules themselves, as was needed for instance in Kammuller's formalisation of abstract algebra [30].

Although Locales provide a powerful tool for modularity and have been extensively used in many formalisations, they are still limited by Isabelle's inherent lack of support for quantifying over types. Locales provide a mechanism for modularity without having to assert new axioms. However, in Hindley/Milner style higher order logics as used in Isabelle [31–33], proof from an axiom is not equivalent to proof from an assumption. This is due to types implicitly being universally quantified over the whole statement. This means that modularity using locales is not exactly equivalent to the use of axioms. One solution to regain this equivalence is to extend higher order logic with quantification over types, as described by Melham [34].

Recently, Johnsen and Luth have used Isabelle's proof terms to provide a more expressive form of modularity than Isabelle's Locales and which effectively allows modularity involving type variables [35]. This is a promising approach to modularity, although it requires the generation of the full proof terms and is not currently integrated into Isabelle.

While these various mechanisms provide tools for modular development and combination of mathematical theories at the logical level, mathematical vernacular require also great flexibility in terms of the syntax used. In particular, providing a language for theory inheritance that allows modification of the syntax mechanisms would greatly improve the practical modularity of Isabelle. As noted by Laumann [26], this is currently one the main barriers to theory reuse.

Another difficult issue for modular proof development the integration and configuration of proof tools. When different theories which each configure a proof tool are combined, a merging operation must be provided to produce a new configuration for the tool in the merged theory. Isabelle provides basic support for merging configurations of its main proof tools. We believe that improved support for modular proof development is an important area of future work that could significantly improve the usability of Isabelle/Isar/IsaPlanner.

6 Knowledge Management and Theorem Navigation

In order to make formalisation a practical task it must be possible to refer to existing theorems, proof tools, and other mathematical objects easily. While the number of proof tools is usually not large enough to incur naming problems, theory libraries which contain thousands of theorems can make the task of finding a previously proved result surprisingly difficult. There are many concurrent approaches to tackling this problem:

Naming schemes provide a simple way to help users remember the name of previously proved theorems. While sensible naming schemes are very useful, they provide a somewhat adhoc approach and require significant discipline by and agreement between theory developers.

Name spaces give a disciplined approach to the naming of theorems, by supporting the overloading of theorem names. Isabelle provides local name spaces for its theories.

Search using syntactic properties provides a powerful tool for finding theorems, although it can take some time to learn the query language. Isabelle provides support, using higher order pattern matching, for finding previously proved results.

Theory browsing tools can provide a mechanisms to explore previously proved results. This is useful for both gaining an understanding of a theory and for looking up previously proved results. Isabelle provides a mixture of generated PDF documents as well as HTML files to support examining previously formalised theories.

Semantic markup adds additional information to mathematical objects which can then be used for search and presentation [36]. A trivial example of such markup is the distinction between lemmas, theorems and corollaries. In general, semantic markup provides the most sophisticated approach to mathematical knowledge management, although it is still unclear what the actual semantic information should be. This is an area of research which we believe could provide significant improvements to theory management.

Although Isabelle provides several mechanisms to help navigate through theories, we found that it is still often easier to re-prove relatively straightforward theorems than find them in the large libraries. This indicates that further work in supporting theory navigation and lookup is still needed.

Another issue of of importance to the management of mathematical knowledge, especially when modifying existing theories, is the tracking of dependencies. Autexier and Hutter [37] describe an approach to the management of change in software verification. Such tools can also be useful to the identification of dependencies that are otherwise hidden by proof tools. We believe that such analysis will be useful to mathematical investigation as it provides a meta-logical analysis of proofs and theories.

Isabelle currently tracks theory and theorem dependencies and can visualise them as a graph. However, it does not provide any further tools to support the management of change. At present, changes to a proof development must be made by hand and rerun to observe where failure occurs. Further tool support could significantly benefit mathematical formalisation which frequently involves correcting and modifying definitions.

7 Well-Behaved Exploratory Tools

One of the central requirements for formalisation is the convenience of using the proof tools. Powerful automation can prove many problems that a user might consider trivial. Such tools are often so powerful that they leave the user unsure of steps within the proof. This is partially because the automatic tools are capable of performing a huge number of proof steps very quickly. A further reason why a mathematician might have difficulty following automatic proof steps is because the proofs found by automatic methods, such as resolution, are generally considered unnatural. Although there have been attempts at dealing with such issues, such as Fiedler's work [38], generally, it is not clear that such proofs can be described in a concise and clear fashion.

Unfortunately, the same tools often still fail to solve problems that the user considers trivial. This shows that there is a gap between proof systems and users, in terms of what they can prove easily. Given that even mathematicians do not always agree with each other over which theorems are trivial, it seems unlikely that this issue can be solved by providing more automation.

We believe that while powerful, fully automatic tools can be extremely useful, the central requirement for supporting formalised mathematics is not for further automation. We propose that proof tools for mathematical assistants should focus on making 'intelligent' progress, giving helpful feedback, and providing robust behaviour. Although the notion of progress is an abstract one, in particular contexts it can be well defined. For example, rippling captures the abstract notion of getting closer to applying the induction hypothesis using a concrete measure. Helpful feedback can be given by tools that automatically analyse the proof state. For example, ripple analysis examines applying induction schemes based on the available lemmas and uses this to suggest an induction scheme. For its part, the robust behaviour of a technique is captured by the predictability of its application. For example, a technique, such as rippling, that always terminates, decreases some measure, and results in the same number of subgoals is considered more robust than one which may not terminate and results in an unpredictable number of subgoals.

The importance of syntactic representations when dealing with formalised mathematics highlights the significance of well-behaved tools. Making definitions and conjectures is an error prone task. This leads to users conjecturing statements that they believe to be trivially provable, but which turn out to involve subtle and complex proofs or even turn out to be false. Thus, even the most powerful of sound automatic tools will not always be able to meet the users' expectations. However a weaker tool that makes some progress can help the user understand why the conjecture is more difficult to prove than expected; for example because a lemma is needed. When the conjecture is trivially false, our proof assistant should be able to identify this. For example, Isabelle's quickcheck tool [39] can be employed by IsaPlanner to prune false conjectures.

7.1 A Hierarchy for Proof Tools' Behaviour

Generally, we argue for focusing on making proof tools exhibit better behaviour, especially when it comes to the examination of failed proof attempts. To this end, we categorise the result of a proof tool into the following levels:

(1) **Non-termination:**
The worst result for a user, short of being a bug in the proof system, is the non-termination of a tool. This is an unfortunate but common occurrence for new users of Isabelle who carelessly add rules to the simplifier.

(2) **Uninformative result:**
Failure without helping the user is marginally better than non-termination — at least the user knows to try something else. Examples of tools that frequently act in this way include Isabelle's blast tactic and calls to first order theorem provers, although the latter also often fail to terminate.

Some applications of techniques result in a proof state that is not humanly readable, for instance it simply contains too many subgoal to understand what has happened. Such a result is often uninformative, and gives the user no additional benefit over outright failure.

(3) **Intelligent progress:**
While the above two results are considered 'bad' failure, a tactic that results in a new proof state, which has made some progress although it has failed to prove the goal, is considered 'good' failure.

Examples of this include some applications of Isabelle's simplifier as well as the rippling technique implemented in IsaPlanner. The latter typically preserves the structure of the goal during rewriting the step case of an inductive proof, while making progress towards applying the inductive hypothesis. Particularly successful applications of such well behaved tools allow automated examination of the failure which can be used to apply proof critics, as described by Ireland et al [12,40]

(4) **Proof or refutation:**
Successfully proving a goal is, of course, the ideal result from a sound proof tool, and symmetrically, although sometimes less welcome, is the refutation of a conjecture.

An orthogonal issue for proof tools is the ability to survey their effect: it is important that a mathematical assistant can explain itself to the mathematician. For a proof this should be a readable presentation. In terms of failure, this might be an explanation of the failure, or even of the progress that was made despite the failure. As well as explaining a the behaviour of a technique, we believe that the system should be able to give hints and observations to the user, or attempt the proof in the background as suggested by Meng and Paulson [41].

Another desirable side effect that can occur from proof planning is the automatic development of the user's theory, for instance by deriving lemmas [21]. Such lemmas are often useful in that they are automatically reused in other proof attempts within the theory and are standard results in interactive developments. Automatic theory formation happens when using proof critics that identify and prove useful lemmas, or refute false conjectures. This use of proof critics is dependent on having a clear notion of failure that can be analysed to develop a patch to the failed proof attempt.

Configuration also plays an important factor in the behaviour of many proof tools. For example, Isabelle's simplifier, which usually yields a simpler goal, will fail to terminate if carelessly configured. Thus, an important characteristic of proof tools is the simplicity of their configuration and the robustness of their behaviour upon misconfiguration. Most interactive tools are set up by supplying proved theorems or assumptions. Ideally, providing an additional theorem should not adversely effect the proof tool. If possible, the tool should notify the user of potentially dangerous additions. In general this is undecidable although in practice it is often still possible.

7.2 A Well Behaved Technique: Rippling

IsaPlanner focuses on providing tools that facilitate exploring the search space of proof and that try to provide the user with more information by making some kind of clear progress. In support of our approach and as an argument that such tools are not imaginary, we describe how the rippling technique, implemented in IsaPlanner, provides such an exploratory tool.

Rippling is a rewriting technique driven by a difference reduction heuristic. It has typically been used to automate inductive theorem proving by reducing the difference between the induction hypothesis and the step case conclusion. We will use it in this way.

The measure of difference used to guide rippling ensures its termination and provides a clear notion of progress and failure. Rippling also requires that part of the term's structure, referred to as the skeleton, is maintained throughout rewriting. This helps ensure that the final proof state is readable. When rippling fails to allow the inductive hypothesis to be used to rewrite the goal, the final proof state is the one with minimal difference to the inductive hypothesis. This often highlights the need for a specific lemma and has been used by Ireland et al to develop proof critics for rippling [12]. These try to patch the failed proof attempt, often by conjecturing new rules that would allow rippling to succeed.

We note that configuring rippling simply involves providing it with previously proved theorems. Furthermore, adding rules cannot significantly impair the techniques behaviour, unlike simplification which given an inappropriate rule leads to non-terminating applications. In this regard, rippling meets the ideals of a technique for our approach.

7.3 An Example Interaction with Rippling

We now present an example which shows user interaction with our proof centric mathematical assistant. We examine using induction and rippling, in IsaPlanner, to prove $(\sum_{i<n} 2 * i + 1) = n^2$. Initially users state the theorem they intend to prove with a `gap` element in place of the proof.

theorem `sum_of_odds:` $(\sum_{i<n} \ 2 \ * \ i \ + \ 1) \ = \ n^2$ `gap`

The user can then select a proof planning technique with which to explore the proof, automatically filling in the gap. This includes techniques not necessarily designed to solve the goal completely. For example, in IsaPlanner, the induction technique examines the inductively defined variables in the conjecture, and selects an appropriate induction scheme. This is then applied using Isabelle's induction tactic which results in the following automatically modified proof script:

theorem `sum_of_odds:` $(\sum_{i<n} \ 2 \ * \ i \ + \ 1) \ = \ n^2$
proof `(induct n)`
  show $(\sum_{i<0} \ 2 \ * \ i \ + \ 1) \ = \ 0^2$ **gap**
next
  fix n
  assume `IH:` $(\sum_{i<n} \ 2 \ * \ i \ + \ 1) \ = \ n^2$
  show $(\sum_{i<(Suc \ n)} \ 2 \ * \ i \ + \ 1) \ = \ (Suc \ n)^2$ **gap**
qed

This proof script contains further gaps indicating the open subgoals in the proof. The intention is that these will be solved later. Because the induction technique is designed only for exploration, it does not suggests techniques with which the gaps might be filled. More generally, the idea is to provide user-customisable levels of interaction. Supporting proof exploration is essentially the generation of proof scripts that reflect the capabilities of the theorem prover. The above example adds to the proof script based on the effect of the induction tactic. Such exploratory tools are analogous to computer algebra systems provision of computational machinery.

The above proof script transformation uses the declarative style of Isar which expresses the subgoals that need to be solved and the context in which they are to be proved, within the proof script. For example, the step case takes an arbitrary but fixed n, and assumes the inductive hypothesis. From this, the step case must be shown. As mentioned earlier, there is no need for an explicit presentation of the proof state resulting from Isabelle as it is shown by the context of the automatically generated Isar proof script.

As well as simply expanding an exploratory technique, proof planning can be used to start the unfolding of a proof planning technique without fully expanding proof attempts of the subgoals. This leaves an un-executed proof planning technique annotating the `gap` statements for each open subgoal. For example, using induction and rippling to the point where a proof critic will be applied, results in the following proof script:

```
theorem sum_of_odds: $(\sum_{i<n} 2 * i + 1) = n^2$
proof (induct n)
  show $(\sum_{i<0} 2 * i + 1) = 0^2$ by simp
next
  fix $n$
  assume IH: $(\sum_{i<n} 2 * i + 1) = n^2$
  have $(\sum_{i<n} 2 * i + 1) + Suc\ (2 * n) = (Suc\ n)^2$
    gap (conjecture_lemma $n^2 + Suc\ (2 * n) = (Suc\ n)^2$)
  thus $(\sum_{i<(Suc\ n)} 2 * i + 1) = (Suc\ n)^2$ by rippling
qed
```

This has rippled the goal to the point where the induction hypothesis can be applied from right to left. It has then used the proof critic machinery to conjecture a lemma. This technique has been defined to stop at this point, presenting the state to the user who can then ask the proof assistant to continue, or pursue a different approach, or manually prove the lemma. We have found that it is useful define techniques that stop and request user interaction at the point when lemmas are conjectured. This is because the automatic conjecturing of lemmas is a difficult and error prone task. It is common for such conjectures to be overly specialised or over generalised. Another reason to stop, which is the case in the above example, is that the correct conjecture is made but an automatic proof fails. If the technique was completely automatic, then it would backtrack over the failure without showing the user the correct suggested conjecture. More generally, the problem of managing the user interaction with planning is examined by approaches to mixed initiative planning [42].

Returning to the above example, if the needed lemma is manually proved and supplied to rippling then the proof can be done fully automatically. By exploring the techniques unfolding we have explored the problem and prove the goal but finally present the proof minimally without showing the final proof they arrived at. For the above example, this can be done as follows:

theorem `sum_of_odds:` $(\sum_{i<n} \ 2 \ * \ i \ + \ 1) \ = \ n^2$
by `(induction_and_rippling)`

Because the automatic tools search and find the proofs in terms of Isar proof script, they can also be expanded out to the level of detail the user requests, up to the point where Isabelle tactics are used. In Section 11, we describe how the user may maintain a full Isar description of the proof, while only presenting selected steps.

Recording the Proof Process

Although the final proof can be presented in a readable form, the process that was undergone to find the proof, including the proof planning expansions, is still hidden from the reader. This places the responsibility of expressing the proof process and the intuitions needed to find the proof, onto the mathematician.

The focus of our tool is on giving the mathematician flexibility in the expression of their proof, while maintaining a fully formal, introspectable underlying description. An avenue of future work might be to consider tools for recording the user's interaction and presenting the search process itself. This could lead to machine learning which attempts to mimic the users interactions.

Power vs Robustness

We remark that improving the automation of the induction and rippling technique is not always beneficial to the user. In particular, if the added automation, while being able to prove more theorems, also leads to non-termination on many problems then this can be sufficiently frustrating for the user that they would rather do the proofs by hand.

If the proof technique can allow itself to be unfolded lazily, generating incremental proof scripts, then this provides the best of both worlds: it allows the user to expand the proof manually, stopping attempts that do not seem fruitful, while maintaining a high degree of automation. IsaPlanner's technique language supports writing techniques in a hierarchical manner and subsequently stepping through them at a user-chosen level of detail. Techniques can also be written to identify steps which they consider to require user interaction. This provides a user-customisable level of interaction.

8 Support for Writing Reasoning Techniques

One of our aims is to provide the user with a 'scripting' language for writing common combinations of proof steps. Currently, IsaPlanner provides an extensible set of tools for automatically constructing and manipulating Isar proofs. This is based on an extensible and declarative representation the Isar language where proofs are tree structured. We provide basic techniques for adding different elements of the language to a proof. For instance, one of these primitive functions adds a "`show ...`" statement to an Isar proof.

In addition to this basic machinery for manipulating declarative proof scripts, we also provide tools that support higher-level notions for encoding techniques. Of particular importance to the expression of rippling was the provision of contextual information for holding the difference annotations and measures. To analyse proof planning failures, it is important that such extra-logical information is in a table of contextual information associated with a proof planning state, rather than as parameters to techniques. This is essential in order to support the later development of proof critics which require analysis of this non-logical information. The key feature of IsaPlanner's management of contextual information is that techniques to not need to know in advance what kinds of information other techniques use.

8.1 Tracing Proof Attempts

Beyond the constructs in the technique language, we provide a tool to trace through IsaPlanner's reasoning states to aid the development and introspection of techniques [43]. This tool allows the user to manually explore the or-branches in the search space and interact with the technique. Because the techniques are structured in a hierarchical manner, the user can also change the level of detail as the proof attempt is performed traced.

We remark that this was particularly useful in the development of rippling as it allowed us to examine in a step-by-step manner the unfolding of the rippling technique. We believe that this will also help in the future development of proof critics, by allowing the user to observe failure in a branch of the search space and explore applying different techniques.

We note that in tactic languages which execute the tactics in an eager fashion, this style of introspection into a technique's application is not possible. Thus, the ability to trace a technique's application represents a particular strength of IsaPlanner's approach to expressing patterns of reasoning. We are also unaware of a proof planner that provides this level of support for the development and introspection into proof planning attempts.

8.2 Contextual Information

A criticism of our use of contextual information within the proof planning state is that it introduces a discord between the proof script and the proof planning. It is no longer the case that proof planning can be stopped at an arbitrary point and continued again by only examining the proof script.

When the contextual information is small, such as the measure used by rippling, it is feasible to express it within the proof script. However, when the contextual information is more complex, for example a cache of previously seen proof states, then it becomes unreasonable to store the information in this way. In general, it seems that although proof planning can describe proof script translations, not every point during proof planning corresponds to a partial proof script. Moreover, given that this information may often be considered mathematically irrelevant, it might be considered unnatural to store it within proof script that are viewed by the mathematician. One approach is to allow proof planning information to be stored within proof scripts but hide them from the user. This is an area for future work.

8.3 Stylistic Choices in Expressing Proofs

Isar scripts provide the user with choices regarding the presentation of their proofs. Using the Isar language as the mechanism for exploration then introduces stylistic choices into the proof scripts generated by proof planning. For example, when should intermediate results be included within the main proof and when should they be considered as separate lemmas? The reason for such choices in encoding of techniques is that equality between Isar proof scripts is strict: proofs are considered different even if they differ only in the naming of variables.

A disadvantage of this strict representation is that techniques may not produce proof scripts in the style that the user wants. In the worst cases, the script may be complex and ugly. However, this approach also supports proof planning machinery that can transform one style of proof into another. This represents a kind of proof by analogy.

Similarly, proof planning can be used to modify proof scripts by applying proof critics. For example, an implementation of the induction critics described by Gow [44] might change the variable on which induction is being applied. We have not yet implemented techniques that construct analogous proofs as needed to modify the style of a proof script. The main difficulty has been in adapting Isar's parser to produce a suitable other parse tree.

9 Proof Plans as a Declarative Description of the Isar Language

Our approach to proof planning uses a declarative description of readable proof scripts as the notion of proof plan. This is an essential component of our tools for supporting exploration. Providing an efficient and extensible representation for such scripts is then an essential tool to support proof planning.

The extensibility of the Isar language inhibits IsaPlanner from using a datatype to directly express proof scripts with constructors for the elements of the Isar language. Thus the language of Isar scripts must be an interpreted object and manipulations of the proof script must account for unexpected commands in the script. Isar's notion of mode, as shown in Fig. 6, simplifies this problem. It allows elements of the language to be treated in terms of their manipulation of Isar's state.

To represent proof plans we use a tree structure where the leaf nodes are polymorphic elements of the Isar language. This allows new notations for proof to be added in a domain specific manner, and support for their use in proof planning to be added without having to modify any existing proof tools. We make use of Huet's zippers for tree representation and manipulation to provide an efficient tool for working with proof plans [45]. This is particularly useful as the most common operation on proof plans is a modification to gaps, expressed in leaf nodes of the proof plan.

In order to allow proof planning techniques to annotate gaps within the proof script, such techniques also need to be expressible within the Isar language. Similarly, arguments to the techniques also need to be represented. This highlights an important technical requirement of the proof centric approach. Proof scripts printed by the proof planner must be parse-able by proof checking machinery. This requires that pretty printed terms must be parseable, as must elements of the Isar language, tactics, and proof planning techniques.

One of the characteristics of Isar, is that it allows modification to syntax and to the proof tools within a proof attempt. This uses a rich notion of proof state for Isar that holds the data for the syntax machinery as well as for proof tools. The local data used by proof tools is essential in order to write some proofs in a suitably brief manner. In order for IsaPlanner to work with this rich notion of proof script, it must execute the elements in the proof plan as they are added. Without doing this it would no longer be a real representation of the proof. In particular, IsaPlanner might produce proof scripts that were not executable by Isar.

In order to delay part of a proof, but avoid having to re-execute large amounts of unchanged proof plan, we take advantage of Isar's notion of proof block in which the modification to proof tools and local syntax is locally scoped. In this way, we allow IsaPlanner to have a notion of delayed execution and partial proof, despite maintaining an executed proof script at all times.

Proof Representation for Replay

The ability to present the proofs at different levels of detail for the viewer brings into question the representation of proof stored in a file for replay: should the user try to minimise the size of the proofs by expressing them with a few powerful proof planning techniques; or should they expand the techniques to fully fleshed-out Isar scripts?

We observe that the more verbose the user makes the proof, by explicitly stating intermediate results, the more likely the proofs are to break if the definitions are modified. Being able to capture a proof using a proof planning technique allows the technique to find a new proof when definitions are changed. However, many proofs cannot be proved using a single proof planning technique. Most proofs require user interaction in selecting proof techniques and in the selection of lemmas to be proved. Moreover, the purpose of proving a theorem is often in order to present the proof. Thus we argue that tools for managing change should be used to update proofs when definitions change and the proof techniques should focus on helping the user find presentable proofs.

10 Interfaces

One of the essential challenges for a mathematical assistant and which may have the most significant effect on its adoption by mathematicians is the user interface. This should be integrated with the theorem database and other tools managing mathematical knowledge, as well as the proof planning machinery in order to support exploration of proof.

The Proof General project aims to provide a common interface to different systems [46]. It is currently the main interface to Isabelle, Isar, and IsaPlanner, as well as a many other systems, including Coq, Phox, and Lego. The typical interaction between a user and a proof system involves the user writing a proof script and executing it within the proof system, observing the result. Proof General provides mathematical symbols and maintains the synchronisation between the interface and the underlying proof tools.

Our approach to proof exploration provides a novel opportunity for user interaction with a proof checker. Rather than execute commands and observe the proof system's response in terms of subgoals, the subgoals can now be captured as part of the proof script. This removes the need for the traditional second window showing the proof system's open goals, thus simplifying the interface.

The IsaWin system is a graphical interface for Isabelle that provides an abstract visual presentation for theory development [47]. While it has helpful features such as proof by pointing, it lacks the sophisticated management of proof scripts. Aspinall and Lüth have recently proposed combining aspects of IsaWin with Proof general [48]. Our approach fits in with such a development by providing a suitable scripting language for the automatic formation and derivation of proof scripts.

Recently, the TeXmacs tool has been used to provide interfaces to computer algebra systems [49,50] as well as proof assistants such as Coq [51]. This is a particularly interesting opportunity for mathematical assistants since TeXmacs provides a WYSIWYG typesetting environment that can interact with proof systems. Future work includes examining how the proof centric approach could be supported by such an interface.

11 Presentation

A proof centric mathematical assistant must provide tools for presenting the proofs. Isabelle/Isar provides machinery for the generation of printable documents. This allows sections of the proof to be hidden thus giving the user precise control over the presentation while maintaining the fully formal nature of the proof.

To give the user the flexibility to present more than just the proofs, Isar theories support including Latex typesetting commands. This results in a notion of a formal proof document, which is an Isar proof script that can be parsed to produce a Latex document in which the proofs have been formally verified.

Isabelle's syntax mechanisms allow the use of Latex style mathematical symbols. Furthermore, Isabelle's underlying mixfix annotations provide a powerful mechanism for expressing formulae in a rich syntax while maintaining the ability to parse them. However, complex mathematical layouts are still not representable within the proof state. For example, the standard notations for matrices cannot be used within a proof. An approach to providing a richer mechanism for expressing mathematical symbols is outlined by Bertot [52].

Another problem with Isar's formal document generation is that checking large documents is significantly slower than the ordinary Latex document preparation. This can make it a slow and painful task to correct errors in the typesetting. This problem is exacerbated by the lack of an interface for the errors generated during processing a proof document. This makes it impractical at present to use complex Latex presentation within the formal proof document. This is a technical but important issue if the system is to be used by mathematicians.

12 Related Work

Systems Based on Declarative Proof-Scripts

The Mizar project [4], which started in 1973, is an attempt to formally reconstruct mathematical vernacular through the use of a declarative, structured and readable representation. However unlike Isabelle/Isar, which is designed to be generic and extensible, the Mizar system is based on Tarski-Grothendieck set theory [53] and provides a fixed language. The Declare system also expresses proofs in a fixed declarative language but unlike Mizar it aims to be generic enough to be implemented for other proof systems [7].

A significant difference from most interactive theorem provers is that Mizar and Declare do not provide the user with a selection of tactics to choose from. Instead, both of these systems automatically prove the missing steps in a proof using an internal strategy. The user explicitly expresses the intermediate goals, assumptions, and which results are needed to show a goal. The proof assistant then processes these declarative proof scripts in a batch mode and reports to the user points where it considers the gaps in the proof to be too large or when it encounters errors in the syntax. Our approach can be seen as an extension of this paradigm which tries to ease and automate the process of writing these declarative proof scripts, as suggested by Syme [54].

Proof Planners

Another closely related system is the Omega system which employs proof planning and aims to assist main stream mathematicians [55,56]. As mentioned earlier, Omega focuses on connecting and integrating external proof tools. It also provides a customised interface for interacting with the system and tools to support using it as an educational teaching assistant.

The central difference in the approach taken by Omega is that it does not use a proof language that is designed to be human-readable and machine checkable. Instead, Omega uses an internal representation for the construction of proof plans, a formalised natural deduction calculus for their verification, and for presentation it can generate readable presentations at various levels of detail [38]. This represents an alternative approach to the one we have presented: rather than having a single language that unifies the users representation of proof with that used by the system, user of Omega views a presentation of proof that lacks information with respect to system's underlying representation.

Tactic-Script Based Proof Assistants

Tactic based interactive proof assistants provide support for the development of proof-checked mathematics. Systems such as Coq [57], HOL [58] and PVS [59], which employ procedural proof scripts as the primary means for interaction, support the exploration of proof through the application of tactics. However, the resulting proof scripts bear very little resemblance to mathematical vernacular.

Tactics in these systems are designed to be applied to a subgoal and result in new subgoals. These systems lack representation for proofs with gaps as an object which can be manipulated. This means that proof critics cannot be expressed in these systems. The partial evaluation of techniques in a manner that stores and supports their later continuation is also inexpressible. These limitations also apply to systems, such as Nuprl [60], which provide an integrated environment for working with proof, but still lack a formal representation of the proofs as objects which can be manipulated rather than just added to.

The main difference between existing tactic based provers, which focus on refining subgoals, and the approach that we have presented lies in our focus on tool support for producing a representation of proof which is human-readable, while maintaining its ability to be machine checked.

Computer Algebra based Systems

Computer algebra systems also aim to support mathematicians. Most of these systems provide tools for computation rather than proof. However the Theorema system [61], based on Mathematica, does aim to provide tools for writing proofs. An advantage of Theorema over other proof systems is its provision of tools for syntax that support two dimensional layouts, as used for example in the presentation of matrices.

The Theorema system also focuses on providing tools for mathematical exploration. Proofs in this system are written as Mathematica documents which combine informal text, definitions, computations, and proof. This is similar in style to to Isar proof scripts although it provides a more powerful environment for presentation.

Theorema supports schemas that ease the users proof development by generating parts of a document automatically. This bears similarity to our proof techniques in its ability to automatically generate proof scripts. The main difference is that our representation supports manipulation of proof scripts after they have been constructed. We also provide a foundational approach to their verification based on an LCF kernel.

Proof-Term Based Interactive Proof Assistants

The Agda system presents the user directly with the proof term which they incrementally fill in to meet the specification [62]. This is similar in style to the approach we have presented, in that the user is working directly with a representation of the proof. However, large proof attempts in Agda can easily become unreadable as there is no tactic-level abbreviations for proof terms. Another difference is that of the proof language itself: Isar is designed to be human-readable and in a natural deduction style whereas Agda presents the proof term which reads more like a functional program than a traditional mathematical text.

TPS

Another closely related approach is that taken by the TPS system. This uses Church's typed λcalculus as its underlying language for checking proofs but provides the user with a natural deduction based presentation and interface to working with proofs [63,64]. This makes the individual proof steps readable but does not allow a mathematical textbook-like presentation. In particular, it hides the structure of the proof and gives little flexibility in terms of its presentation.

The tactics of the TPS system are similar to our notion of techniques in that they construct part of the natural deduction style proof, which is the central object being developed. Furthermore, they provide tactics to perform the easy steps in the proof. However, their proof language does not contain unevaluated references to tactics which means that it cannot partially evaluate proof techniques.

13 Conclusions

In this article we have presented a proof-centric approach to mathematical assistants and described how the combination of the Isabelle proof assistant, the Isar language, and the IsaPlanner poof planner provides a concrete implementation. The underlying goal of this approach is to provide a mathematical assistant that unifies the tasks of proof exploration, certification, and presentation.

We have shown how proof planning can be used to automate the generation and manipulation of readable proof scripts. We have highlighted the need for exploratory tools that, while weaker in terms of the number of theorems they can prove, provide a clear notion of their intended behaviour and thus achieve a robust behaviour. We noted that improving the power of these techniques does not always lead to better behaviour. We described and illustrated this view using the rippling technique.

We also relate this work to issues of modularity, interface, and presentation. We note that a central requirement of formalised mathematics that needs further work is modular management of syntax. The central requirement for continuing this work is the development of a parser for the Isar language that produces parse trees that can be treated as IsaPlanner proof plans. Other areas of further work include: further development of mathematical theories, managing the dependencies between proofs to support the natural development and modification of theories, and improved support for finding previously proved theorems.

Acknowledgements

This research was funded by the EPSRC grant *A Generic Approach to Proof Planning* - GR/N37414/01. The authors would also like to thank Alan Bundy

for his helpful comments and the referees for their constructive feedback.

References

[1] M. Wenzel, Isar - a generic interpretative approach to readable formal proof documents, in: Theorem Proving in Higher Order Logics, Vol. 1690 of LNCS, 1999, pp. 167–184.
URL `citeseer.nj.nec.com/wenzel99isar.html`

[2] L. Paulson, Isabelle: A generic theorem prover, Springer-Verlag, 1994.

[3] L. Dixon, J. D. Fleuriot, IsaPlanner: A prototype proof planner in Isabelle, in: Conference on Automated Deduction, Vol. 2741 of LNCS, 2003, pp. 279–283.

[4] P. Rudnicki., An overview of the mizar project, in: 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad, 1992, pp. 311–332.
URL `http://www.mizar.org/`

[5] V. Zammit, On the Readability of Machine Checkable Formal Proofs, Ph.D. thesis, University of Kent (March 1999).
URL `http://www.cs.ukc.ac.uk/pubs/1999/909`

[6] S. Autexier, C. Benzmüller, A. Fiedler, H. Horacek, B. Q. Vo, Assertion-level proof representation with under-specification, Electronic in Theoretical Computer Science 93 (2003) 5–23.

[7] D. Syme, Declarative theorem proving for operational semantics, Ph.D. thesis, University of Cambridge (1999).

[8] A. Abel, B.-Y. E. Chang, F. Pfenning, Human-readable machine-verifiable proofs for teaching constructive logic, Tech. rep., Università degli Studi Siena, Dipartimento di Ingegneria dell'Informazione, in Proceedings of the Workshop on Proof Transformation and Presentation and Proof Complexities (PTP'01) (2001).
URL `http://www.tcs.informatik.uni-muenchen.de/~abel/ptp01.pdf`

[9] J. Harrison, A mizar mode for HOL, in: J. von Wright, J. Grundy, J. Harrison (Eds.), Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96, Vol. 1125 of Lecture Notes in Computer Science, Springer-Verlag, Turku, Finland, 1996, pp. 203–220.

[10] F. Weidijk, Mizar light for HOL light, in: R. J. Boulton, P. B. Jackson (Eds.), 14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001, Vol. 2152 of Lecture Notes in Computer Science, Springer, Edinburgh, Scotland, 2001, pp. 378–393.
URL
`http://link.springer.de/link/service/series/0558/tocs/t2152.htm`

[11] F. Wiedijk, Formal proof sketches, in: TYPES, 2003, pp. 378–393.

[12] A. Ireland, A. Bundy, Productive use of failure in inductive proof, Journal of Automated Reasoning 16 (1–2) (1996) 79–111.

[13] A. Ireland, M. Jackson, G. Reid, Interactive proof critics., Formal Asp. Comput. 11 (3) (1999) 302–325.

[14] L. C. Paulson, Isabelle's Logics, Computer Laboratory, University of Cambridge, isabelle2002 Edition (2002).

[15] L. C. Paulson, Generic automatic proof tools, in: R. Veroff (Ed.), Automated Reasoning and Its Applications, MIT Press, 1997, pp. 23–47.
URL `citeseer.nj.nec.com/paulson97generic.html`

[16] L. Paulson, A Generic Tableau Prover and its Integration with Isabelle, Journal of Universal Computer Science 5 (3) (1999) 73–87.

[17] E. Melis, A. Meier, Proof Planning with Multiple Strategies, in: J. Loyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. S. P. Stuckey (Eds.), First International Conference on Computational Logic (CL-2000), Vol. 1861 of LNAI, London, UK, 2000, pp. 644–659.

[18] A. Bundy, The use of explicit plans to guide inductive proofs, in: Conference on Automated Deduction, 1988, pp. 111–120.
URL `citeseer.nj.nec.com/bundy88use.html`

[19] A. Bundy, A science of reasoning, Computational Logic - Essays in Honor of Alan Robinson 6 (1991) 178–198.
URL `citeseer.nj.nec.com/bundy91science.html`

[20] L. Dixon, A proof planning framework for Isabelle, Ph.D. thesis, School of Informatics, University of Edinburgh, (Forthcoming) (Jan 2005).

[21] L. Dixon, J. D. Fleuriot, Higher order rippling in IsaPlanner, in: Theorem Proving in Higher Order Logics, Vol. 3223 of LNCS, 2004, pp. 83–98.

[22] A. Bundy, D. Basin, D. Hutter, A. Ireland, Rippling: Meta-level Guidance for Mathematical Reasoning, Springer-Verlag, 2005.

[23] J. D. C. Richardson, A. Smaill, I. Green, System description: proof planning in higher-order logic with Lambda-Clam, in: Conference on Automated Deduction, Vol. 1421 of LNCS, 1998, pp. 129–133.

[24] J. D. Fleuriot, A Combination of Geometry Theorem Proving and Nonstandard Analysis, with Application to Newton's Principia, Springer-Verlag, 2001.

[25] L. I. Meikle, J. D. Fleuriot, Formalizing Hilbert's Grundlagen in Isabelle/Isar., in: Theorem Proving in Higher Order Logics, 2003, pp. 319–334.

[26] C. Laumann, An idealistic formalization of Stokes' theorem: Pedagogical math in Isabelle/Isar, Master's thesis, School of Informatics, University of Edinburgh (2004).

[27] Isabelle archive of formal proof, http://afp.sourceforge.net/ (2004).

[28] M. Wenzel, Type classes and overloading in higher-order logic., in: Theorem Proving in Higher Order Logics, 1997, pp. 307–322.

[29] C. Ballarin, Locales and locale expressions in Isabelle/Isar., in: TYPES, 2003, pp. 34–50.

[30] F. Kammüller, Modular reasoning in Isabelle, Ph.D. thesis, University of Cambridge (August 1999).

[31] J. R. Hindley, The principal type-scheme of an object in combinatory logic, Trans. American Math. Soc 146 (1969) 29–60.

[32] R. Milner, A theory of type polymorphism in programming., J. Comput. Syst. Sci. 17 (3) (1978) 348–375.

[33] T. Nipkow, Order-sorted polymorphism in Isabelle, in: G. Huet, G. Plotkin (Eds.), Logical Environments, Cambridge University Press, 1993, pp. 164–188.

[34] T.F. Melham, The HOL logic extended with quantification over type variables, in: L.J.M. Claesen, M.J.C. Gordon (Eds.), International Workshop on Higher Order Logic Theorem Proving and its Applications, North-Holland, Leuven, Belgium, 1992, pp. 3–18.
URL `citeseer.ist.psu.edu/melham93hol.html`

[35] E. B. Johnsen, C. Lüth, Theorem reuse by proof term transformation, in: K. Slind, A. Bunker, G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics, Vol. 3223 of Lecture Notes in Computer Science, Springer, 2004, pp. 152–167.

[36] F. Guidi, I. Schena, A query language for a metadata framework about mathematical resources., in: MKM, 2003, pp. 105–118.

[37] S. Autexier, D. Hutter, H. Mantel, A. Schairer, Towards an evolutionary formal software-development using CASL, in: Workshop on Algebraic Development Techniques, 1999, pp. 73–88.
URL `citeseer.ist.psu.edu/article/autexier99towards.html`

[38] A. Fiedler, User-adaptive proof explanation, Ph.D. thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany (2001).

[39] S. B. T. Nipkow, Random testing in Isabelle/HOL, in: J. Cuellar, Z. Liu (Eds.), Software Engineering and Formal Methods (SEFM 2004), IEEE Computer Society, 2004, pp. 230–239.

[40] A. Ireland, The use of planning critics in mechanizing inductive proofs, in: Logic Programming and Automated Reasoning, 1992, pp. 178–189.
URL `citeseer.nj.nec.com/ireland92use.html`

[41] J. Meng, L. C. Paulson, Experiments on supporting interactive proof using resolution, in: D. Basin, M. Rusinowitch (Eds.), Second International Joint Conference on Automated Reasoning, IJCAR 2004, Springer, 2004, pp. 372–384.

[42] M. Burstein, D. McDermott, Issues in the development of human-computer mixed-initiative planning, in: Gorayska, Mey (Eds.), Cognitive Technology: In Search of a Humane Interface, North Holland, 1996, pp. 283–303.

[43] L. Dixon, Interactive hierarchical tracing of techniques in IsaPlanner, in: User Interfaces for Theroem Provers (UITP'05), ENTCS, 2005.

[44] J. Gow, The dynamic creation of induction rules using proof planning, Ph.D. thesis, School of Informatics, University of Edinburgh (2004).

[45] G. Huet, The zipper, Journal of Functional Programming 7 (5) (1997) 549–554.

[46] D. Aspinall, T. Kleymann, Proof General Manual, University of Edinburgh (2004).

[47] C. Lüth, T. H, Kolyang, B. Krieg-Brückner, TAS and IsaWin: Tools for transformational program developkment and theorem proving, in: J.-P. Finance (Ed.), Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99, no. 1577 in Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 239– 243.

[48] D. Aspinall, C. Lüth, Proof general meets isawin, in: User Interfaces for Theroem Provers (UITP'03), ENTCS, 2003.

[49] J. van der Hoeven, Gnu texmacs: A free, structured, wysiwyg and technical text editor, in: D. Flipo (Ed.), Le document au XXI-ième siècle, Vol. 39–40, Metz, 2001, pp. 39–50, actes du congrès GUTenberg.

[50] J. van der Hoeven, Gnu TeXmacs, http://www.texmacs.org (1998–2002).

[51] P. Audebaud, L. Rideau, TexMacs as authoring tool for publication and diffusion of formal developments, in: User Interfaces for Theroem Provers (UITP'03), ENTCS, 2003.

[52] A. Amerkad, Y. Bertot, L. Pottier, L. Rideau, Mathematics and proof presentation in Pcoq, in: The Workshop on Proof Transformation and Presentation and Proof Complexities (PTP'01), 2001, pp. 1–15.
URL `http://www.inria.fr/rrrt/rr-4313.html`

[53] A. Trybulec, Tarski Grothendieck Set Theory, Journal of Formalized Mathematics Axiomatics.

[54] D. Syme, Declare: A prototype declarative proof system for higher order logic tech report, comp lab, univ of camb, 1997. (1997).
URL `citeseer.ist.psu.edu/syme97declare.html`

[55] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. H. Siekmann, V. Sorge, Omega: Towards a mathematical assistant., in: Conference on Automated Deduction, 1997, pp. 252–255.

[56] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, M. Pollet, Proof development in OMEGA: The irrationality of square root of 2, in: F. Kamareddine (Ed.), Thirty Five Years of Automating Mathematics, Kluwer Applied Logic series (28), Kluwer Academic Publishers, 2003, pp. 271–314, iSBN 1-4020-1656-5.

[57] The Coq Development Team, The Coq Proof Assistant Reference Manual – Version V8.0 (Apr. 2004).
URL `http://coq.inria.fr`

[58] M. J. C. Gordon, T. F. Melham (Eds.), Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993.
URL `http://www.dcs.glasgow.ac.uk/ tfm/HOLbook.html`

[59] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, PVS System Guide, Computer Science Laboratory, SRI International, Menlo Park, CA (Sep. 1999).

[60] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, S. F. Smith, Implementing Mathematics with the Nuprl Development System, Prentice-Hall, NJ, 1986.
URL `citeseer.ist.psu.edu/constable86implementing.html`

[61] B. Buchberger, T. Jebelean, W. Windsteiger, T. Kutsia, K. Nakagawa, J. Robu, F. Piroi, A. Craciun, N. Popov, G. Kusper, M. Rosenkranz, L. Kovacs, C. Kocsis, F 1302: THEOREMA: Proving, Solving and Computing in General Domains, in: P. Paule, U. Langer (Eds.), Special Research Program (SFB) F 013, Numerical and Symbolic Scientific Computing, Proposal for Continuation, Part I: Progress Report, April 2001-September 2003, Johannes Kepler University Linz, Austria, 2003, pp. 148–170.

[62] C. Coquand, The AGDA Proof System Homepage, `http://www.cs.chalmers.se/~catarina/agda/` (1998).

[63] P. B. Andrews, M. Bishop, C. E. Brown, System description: Tps: A theorem proving system for type theory, in: D. McAllester (Ed.), Proceedings of the 17th International Conference on Automated Deduction, Vol. 1831 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Pittsburgh, PA, USA, 2000, pp. 164–169.

[64] P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, H. Xi, TPS: A theorem proving system for classical type theory, Journal of Automated Reasoning 16 (1996) 321–353.