Contributions to AI4FM 2013

the 4th International Workshop on Artificial Intelligence for Formal Methods

by Gudmund Grov, Ewen Maclean & Leo Freitas (Eds.)
Preface

This report contains the contributions to the 4th international workshop on Artificial Intelligence for Formal Methods (AI4FM 2013). This was held as a satellite event of the 4th conference on Interactive Theorem Proving (ITP 2013), in Rennes (France) on July 22nd 2013. Previous AI4FM workshop has been held in Newcastle (2010), Edinburgh (2011) and Schloss Dagstuhl (2012).

The workshop consisted of one invited talk: **Sharing the Burden of (Dis)proof with Nitpick and Sledgehammer** by Jasmin Blanchette, and 9 regular talks. This report contains the abstract for each regular talk:

- **The social machine of mathematics** by Ursula Martin
- **Verifying the heap: an AI4FM case study** by Leo Freitas, Cliff Jones, Andrius Velykis and Iain Whiteside
- **Applying machine learning to setting time limits on decision procedure calls** by Zongyan Huang and Lawrence Paulson
- **Arís: Analogical Reasoning for reuse of Implementation & Specification** by Mihai Pitu, Daniela Grijincu, Peihan Li, Asif Saleem, Rosemary Monahan, Diarmuid O’Donoghue. This abstract was separated into 2 talks:
  - **Source Code Matching for reuse of Formal Specifications** by Daniela Grijincu
  - **Source Code Retrieval using Case Based Reasoning** by Mihai Pitu
- **Learning Domain-Specific Guidance for Theory Formation** by Jeremy Gow
- **Drawing Proof Strategies** by Gudmund Grov, Aleks Kissinger, Yuhui Lin, Ewen Maclean and Colin Farquhar
- **Analogical Lemma Speculation** by Ewen Maclean
- **Theory Exploration for Interactive Theorem Proving** by Moa Johansson

Gudmund Grov, Ewen Maclean and Leo Freitas
(workshop organisers)
The social machine of mathematics
Ursula Martin, Queen Mary, University of London
Ursula.Martin@ecs.qmul.ac.uk

“There is no ... mathematician so expert ... as to place entire confidence in his proof immediately on his discovery of it... Every time he runs over his proofs his confidence increases; but still more by the approbation of his friends.” David Hume, 1739

“Computers can ... create abstract social machines on the Web: processes in which the people do the creative work and the machine does the administration... The stage is set for an evolutionary growth of new social engines.” Tim Berners-Lee, 1999

“... and sometimes I realized that nothing that had ever been done before was any use at all. Then I just had to find something completely new; it’s a mystery where that comes from.” Andrew Wiles, 2000, on proving Fermat’s theorem

“Who would have guessed that the working record of a mathematical project would read like a thriller?” Tim Gowers/Michael Nielson, on collaborative online mathematics, Nature, 2009

“This is really the End.” Georges Gonthier, 2012 completes his 6 year formal verification of a major 255 page result in algebra, the odd-order-theorem

For centuries, the highest level of mathematics has been seen as an isolated creative activity, to produce a proof for review and acceptance by research peers. Mathematics is now at a remarkable inflexion point, with new technology radically extending the power and limits of individuals. “Crowdsourcing” pulls together diverse experts to solve problems; symbolic computation tackles huge routine calculations; and computers, using programs designed to verify hardware, check proofs that are just too long and complicated for any human to comprehend.

Yet these techniques are currently used in stand-alone fashion, lacking integration with each other or with human creativity or fallibility. Social machines are a new paradigm, identified by Berners-Lee [21], for viewing a combination of people and computers as a single problem-solving entity.

What if we developed a new vision, changing the way people do mathematics, and transforming the reach, pace, and impact of mathematics research, through creating a mathematics social machine — a combination of people, computers, and archives to create and apply mathematics?

Thus, for example, an industry researcher wanting to design a network with specific properties could quickly access diverse research skills and research; explore hypotheses; discuss possible solutions; obtain surety of correctness to a desired level; and create new mathematics that individual effort might never imagine or verify. Seamlessly integrated “under the hood” might be a mixture of diverse people and machines, formal and informal approaches, old and new mathematics, experiment and proof.

Much is known about the relevant ICT technologies:

- Collaborating: crowdsourced and open innovation
- Creating: AI for creativity, analogy and discovery
- Calculating: numeric and symbolic computation
- Verifying: formalization, reasoning and proof
- Sharing: knowledge management and interfaces

The obstacles to realising the vision seem to be not advances in any one of these domains, but rather:

- We do not have a high level understanding of the production of mathematics by people and machines, integrating the current diverse research approaches
- There is no shared view among the diverse research and user communities of what is and might be possible or desirable

In this note we sketch what we might do to address these challenges.

Background

Mathematics reach, pace and impact Mathematics and theoretical computer science, research underpins modern programming languages, secure systems, and the Web. Advances depend on hard foundational mathematics, and draw on newer areas such as statistics and dynamical systems, alongside traditional combinatorics and logic, supplemented by simulation and experiment. This potential reach of mathematics is increasing, thus increasing the challenge to researchers of deploying the right combinations of techniques within and beyond their own specialism to solve increasingly hard and broad problems, which are not stated in isolation but require an integrated approach. The rapid pace of technology creates key opportunities, if mathematicians are able to collaborate with each other, and with other computing disciplines, both academic and industry, to produce timely results. Increasing pace and reach has the potential to increase impact, if potential users of research can find the researchers and the research they need. Sometimes it can be easier to write a new paper than to find old results: the
past 10 years saw nearly 200K papers of relevance to theoretical computer science [2].

**Social machines** Social machines combine people and computers for emergent and collective problem-solving. Current examples include Google, Wikipedia and Galaxy Zoo, providing platforms for innovation, discovery, and commercial opportunity [34]. Future more ambitious social machines will combine deep social involvement and sophisticated automation [21], and are now the subject of major research. This approach builds on e-Science work such as Goble’s myExperiment [15], a collaborative research space for scientific workflow management and experiment: however such systems do not address mathematics.

**Mathematics and social machines** The production of mathematics provides an important, timely and exciting challenge for social machines research — with a variety of approaches to combining people and machines. In the past few years, systems for unstructured collaboration developed by researchers themselves have had a powerful impact: we call such systems *social mathematics.*

- in the summer of 2010 a paper was released plausibly claiming to prove one of the major challenges of theoretical computer science, that \( P \neq NP \). It was withdrawn after rapid analysis coordinated by senior scientist-bloggers
- *polymath* collaborative proofs, a new idea led by Gowers, use a wiki for collaboration among scientists from different backgrounds and have led to major advances [18]
- discussion fora, including new ideas such as user ratings for finding the right expert, allow rapid informal interaction and problem solving; in three years mathoverflow.net has hosted 27,000 conversations
- the widely used “Online Encyclopaedia of Integer Sequences” (OEIS) invokes subtle pattern matching against over 200K user-provided sequences on a few digits of input: so for example (3 1 4 1) returns \( \pi \) [7]
- the arXiv holds around 750K preprints in computer science, mathematics etc.. By providing open access ahead of journal submission, it has markedly increased the speed of refereeing, widely identified as a bottleneck to the pace of research [31]
- Innocentive [22], a site hosting open innovation and crowdsourcing challenges, has hosted around 1,500 challenges with a 57% success rate, of which around 10% were tagged as mathematics or ICT.

All can be viewed as social machines — for example OEIS involves users with queries or proposed new entries; volunteers curating the system; governance and funding mechanisms; as well as a database, matching engine and web interface.

**The social element** The social is crucial in the production of mathematics. Mackenzie’s sociological study of proof [25] confirmed Lakatos’s analysis of the role of error [23], and Hume’s assessment nearly 300 years earlier of the social nature of proof [10]. Williams’s notion of technological “artefacts” matches the way in which mathematical objects mutate as ideas are developed [41].

The work of cognitive scientists, sociologists, philosophers and the narrative accounts of mathematicians themselves, highlight the paradoxical nature of mathematical practice — while the goal of mathematics is to discover mathematical truths justified by rigorous argument, mathematical discovery involves “soft” aspects such as creativity, informal argument, error and analogy.

Collaborative systems such as *polymath* contribute to mathematics research, and also provide a rich evidence base for further understanding of mathematical practice. Our analysis of a *polymath* proof [35] found only 47% of the conversational “turns” were proof steps, with the rest being made up of conjectures, concept formation and the like.

At a recent learned society event organised by the proposer [1], leading mathematicians flagged the importance of collaborative systems that “think like a mathematician”, handle unstructured approaches such as the use of “sloppy” natural language, and the exchange of informal knowledge and intuition not recorded in papers, and engage diverse researchers in creative problem solving.

Yet if the mathematics social machine is to realise its potential, and scale up to large proofs, it will also need formal approaches. Verification through formalisation and proof, supported by decades of academic and industry research into theorem provers, is achieving remarkable breakthroughs, and providing rich archives of material for possible re-use:

- on 20th September 2012 Georges Gonthier announced that after six years effort he had completed a formalisation in the Coq theorem prover of one of the most important and longest proofs of 20th century algebra, the 255 page odd-order theorem [3]
- mathematician Tom Hales has almost completed a ten-year formalisation of his proof of the Kepler conjecture, using several theorem provers to confirm his major 1998 paper [19]
- hardware and software verification to ensure error-free systems is a major endeavour in companies like Intel and Microsoft [20], as well as supporting a number of specialist SMEs.
Other likely elements of a mathematics social machine would include the following, all currently major research activities in their own right:

- a variety of AI and cognitive science inspired approaches to "soft" aspects such as creativity, analogy and concept formation [13]. For example, mathematicians often mention the importance of error for creativity [1]: this has also stimulated Bundy’s recent AI work on ontology repair [26]
- symbolic and numeric computation, and associated data, provided by commercial systems such as Matlab and Maple, or research packages such as GAP: all already engage strongly with e-science
- digitised mathematical archives, using MKM, for example to support search, re-use and executable papers [24]. The National Academy of Sciences have just announced a major initiative [33]
- interfaces: people to machine, natural language to mathematics, and software to software

Capitalising on the substantial research underlying these achievements will inform thinking about the design space for mathematics social machines, for example:

- precise versus loose queries and knowledge
- human versus machine creativity
- specialist/niche versus general users
- logical validity versus cognitive appeal for output
- formal versus natural language for interaction
- generating versus checking conjectures or proofs
- formal versus informal proof
- "evolution" versus "revolution" for new systems
- governance, funding and longevity

Exploring these in the framework of social machines will include matters such as:

- Designing social computations Social machine models [21] view users as "entities" (cf agents or peers) and allow consideration of social interaction, enactment across the network, engagement and incentivisation, and methods of software composition that take into account evolving social aggregation. For mathematics this has far reaching implications — handling known patterns of practice, and enabling others as yet unimagined, as well as handling issues such as error and uncertainty, and variations in user beliefs.
- Accessing data and information Semantic web technology enables databases supporting provenance, annotation, citation and sophisticated search. Mathematics data includes papers, records of mathematical objects from systems such as Maple, and scripts from theorem provers. There has been considerable research in MKM [24], but current social mathematics systems have little such support. Yet effective search, mining and data re-use would transform both theoretical computer science research and commercial verification. Research questions are both technical, for example how to ensure annotation remains timely and correct, and social, for example many mathoverflow responses cite published work, raising the issue of why users prefer asking to searching.
- Accountability, provenance and trust Participants in social machines need to be able to trust the processes and data they engage with and share. Key concepts are provenance, knowing how data and results have been obtained, which contributes to accountability, ensuring that the source of any breakdown in trust can be identified and mitigated [40]. Current social mathematics systems are remarkably open — for example posting drafts on the arXiv ahead of journal submission is reported as speeding up refereeing and reducing priority disputes [1]. Trusting mathematical results requires considering provenance of the proof, a major issue in assessing the balance between formal and informal proofs, and the basis for research into proof certificates [30]. Privacy and trust are significant for commercial or government work, where revealing even broad interests may already be a security concern.

- Interactions among people, machines and data Interactions among people, machines and data are core to social machines, which have the potential to support novel forms of interaction and workflow which go beyond current practice, a focus of current social machine research [21]. Social mathematics shows a variety of communities, interactions and purposes — looking for information, solving problems, clarifying information and so on [35] — displaying much broader interactions than those supported by typical mathematical software. Lakatos identifies mathematical "moves" (eg responses to counterexamples) that are examples of mathematical workflow, and examining both polymath and the production of formal proofs has potential to reveal more [35]. In particular such workflows need to take account of informality and mistakes [12].

References
Verifying the heap: an Al4FM case study

Iain Whiteside, Leo Freitas, Cliff Jones, and Andrius Velykis
School of Computing Science
University of Newcastle
\{iain.whiteside,leo.freitas,cliff.jones,andrius.velykis\}@newcastle.ac.uk

Background
The Al4FM project aims to use “learning” techniques from artificial intelligence to record and abstract how experts do proofs in order to increase the proportion of cases where proofs are constructed without (or with minimal) human intervention. Most formal methods give rise to proof obligations (POs). The majority of such POs are discharged by automatic theorem provers, however some of them (typically 5-20%) still require guidance from a proof expert. Manually finding a proof can be a difficult process, and is often a bottleneck in industrial applications of formal methods. However, once a proof is found, many other POs can be discharged by following the same proof pattern. The goal of Al4FM is to automatically discharge POs of “similar form” by exploring the extra information deduced from the expert provided proof attempt.

At this stage in the project, we are in a position to put our tools and techniques to the test:

Proof Process The result of Velykis’ PhD project, analyses and captures the data from expert proof attempts in the WhyM format [5, 2].

PSGraph Grov’s graph-based strategy language for the Isabelle Proof Assistant enables a high-level representation extracted from proof attempts [1].

Lemma Analogy Maclean’s recent work (described in this workshop) enables, lemmas—acknowledged as a key part of an expert’s proof insights—to be speculated analogously to those given by an expert.

Sticking to the ethos of the project—to tackle real problems—we have decided to perform a case study in Al4FM. This paper reports on the progress so far.

VDM model of a heap
For our case study, we chose to verify a heap memory storage. The model we use is represented in VDM and based on [3, §7]. We chose this particular example because we had a hunch it would be nontrivial (this hunch was fully justified) and was simple to explain. The model itself consists of four levels of refinement: starting from a basic set-theoretic description and ending with code. There are two operations on the heap:

NEW given the current state of the heap and a desired size of memory to allocate, this operation updates the state of the heap and returns the starting location of an appropriate chunk. This operation is represented in VDM as:

DISPOSE is the dual operation and will, given a location and size as parameters, update the heap to return a chunk of memory.

In this paper, we will discuss two levels of refinement:

Level 0 represents the heap as a set of locations:

\[ \text{Loc} = \mathbb{N} \]
\[ \text{Free0} = \text{Loc-set} \]

At level 0, the NEW operation, for example, is represented as follows:

\[ \text{NEW} \ (s : \mathbb{N}_1) \ r : \text{Loc} \]
\[ \text{ext wr f : Free0} \]
\[ \text{pre} \ \exists l \in \text{Loc} \cdot \text{is-block}(l, s, f) \]
\[ \text{post} \ \text{is-block}(r, s, f) \land f = \text{locs-of}(r, s) \]
The operation takes a desired size \( s \) as input and modifies a state \( f \). The resulting output \( r \) is the start location of a suitable chunk. The operation is then specified indirectly with a precondition—that there exists a suitable block in the set—and a postcondition—that the result forms a block and that the updated state is equal to the original state without the set of locations just allocated.

**Level 1** refines the representation to show the heap as a finite map: taking start locations to sizes. Here we need an invariant on the properties of the map:

\[
\begin{align*}
\text{Free1} &= \text{Loc} \xrightarrow{\text{m}} \mathbb{N}, \\
\text{inv} (f) &\triangleq \forall l, f' \in \text{dom} f, \\
&l \neq f' \Rightarrow \text{is-disj}(\text{locs-of}(l, f(l)), \text{locs-of}(f', f(f'))) \land \\
&\forall i \in \text{dom} f \cdot (l + f(i)) \notin \text{dom} f
\end{align*}
\]

The first conjunct states that for every two start locations in the heap, the set of locations that they represent—their locs_of—must be disjoint. We call this property *Disjoint*. The second conjunct ensures that all heap chunks are *separate*: there is at least a one-location gap between them\(^1\).

### Isabelle model of a heap

Our first step was the formalisation of the model in the Isabelle Proof Assistant [4]. A small portion of the model, partially defining the level 1 invariant and *NEW* operation is shown:

```isabelle
definition F1-inv :: F1 \Rightarrow bool where F1-inv f \equiv Disjoint f \land sep f \land nat1-map f \land finite(dom f)
...
locale level1-basic = fixes f1 :: F1 and s1 :: nat assumes II-input-notempty: nat1 s1 and II-invariant : F1-inv f1
locale level1-new = level1-basic + assumes new1-precondition: new1-pre f1 s1
```

The invariant includes the *Disjoint* and *separate* properties described above, but also includes conditions on finiteness of the map and non-zeroness of the required length. These additional constrains are required because Isabelle maps are not finite in general and Isabelle does not have a specific datatype for non-zero naturals. The post-condition is specified separately as an Isabelle definition with parameters for the updated heap and the resulting location:

```isabelle
definition (in level1-new) new1-postcondition :: F1 \Rightarrow nat \Rightarrow bool where new1-postcondition f' r \equiv new1-post f1 s1 f' r \land F1-inv f'
```

The exact details of the formalisation are not required for this paper, but we hope this section gave a feel for the representation.

### Some examples of families of proof

Proving that the invariant holds after each operation is performed forms the bulk of the difficulty with proof obligations in this development, but we can clearly see that a lot of the proofs fall into families. As a very simple example, consider the proof obligations

```isabelle
lemma l-ar-sap shows sep({ x } \cdot q f)
```

\(^1\)If they were not separate, we would rather merge both chunks into a super chunk.
and

\textbf{lemma} \textit{l-union-sep}
\textbf{shows} \texttt{sep} \((\{ x \} \cdot \ominus f) \cup \{ x \mapsto y \})

Both of these goals, and the equivalents for the other parts of the invariant (and variants that we also come across) follow the same pattern of proof.

\textbf{Getting rid of difficult operators: an example proof pattern}
Previously identified by Freitas, this proof pattern has come up frequently in the heap verification and we believe that it could be learned and implemented using Maclean’s Lemma Analogy tool. Consider, for example, the lemma:

\textbf{lemma} \textit{l-dom-dom-ar:}
\textbf{shows} \texttt{dom}(\{ x \} \cdot \ominus f) = \texttt{dom} f - \{ x \}

which describes how domain anti-restriction affects the domain of the heap map. This lemma is often useful (for example, when arguing over elements of a map) as it removes the troublesome anti-restriction operator and replaces it with set minus (which Isabelle happens to know a lot about).

We postulate that this lemma could be used to speculate analogous lemmas on slightly different goals. For example:

\textbf{lemma} \textit{l-locs-of-dom-dom-ar:}
\textbf{shows} \texttt{locs}(\{ x \} \cdot \ominus f) = \texttt{locs} f - \texttt{locs-of} x (f \ x)

where the \texttt{locs} function is the union of \texttt{locs-of} over all elements in the map. We hope that information from an expert proof and an expert lemma (\textit{l-dom-dom-ar}) could be used to speculate an analogous lemma (\textit{l-locs-of-dom-dom-ar}) in a similar proof.

\textbf{Conclusion}
The results from this case study are promising. In preliminary analysis, we have identified several proof patterns and many families of proof that we believe can be learned and implemented as part of the project. Furthermore, it is clear that this was a non-trivial case study: our Isabelle development has already reached thousands of lines of proof, taken several man months and unearthed a number of subtle bugs in the specification. Finally, we have some significant amounts of data: we are excited to see what we can learn.

\textbf{References}
Applying machine learning to setting time limits on decision procedure calls

Zongyan Huang  Lawrence C. Paulson
zh242@cam.ac.uk  lp15@cam.ac.uk
The University of Cambridge Computer Laboratory, William Gates Building
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom

Abstract: MetiTarski is an automatic theorem prover which can prove inequalities involving special functions. It combines a resolution theorem prover (Metis) with a set of axioms and a collection of decision procedures for the theory of real closed fields (RCF). During MetiTarski’s proof search, it generates a series of RCF subproblems which are reduced to true or false using a RCF decision procedure. Many of the RCF subproblems do not contribute towards MetiTarski’s final proof, and time spent deciding their satisfiability is wasted. Setting a time limit on RCF calls reduces the wasted time but too tight a limit could affect MetiTarski’s proof. Machine learning has been applied to the problem-dependent selection of the best time limit setting for RCF decision procedure calls in the theorem prover MetiTarski.

1 Introduction

MetiTarski [1] is designed to automatically prove universally quantified logical statements involving special functions such as logarithms, sines, cosines, etc. MetiTarski works by eliminating special functions, substituting rational function upper or lower bounds, transforming parts of the problem into polynomial inequalities, and finally applying a decision procedure for the theory of RCF. The theory of RCF concerns (possibly quantified) boolean combinations of polynomial equations and inequalities over the real numbers.

In MetiTarski, RCF decision procedures are used to simplify clauses by deleting literals that are inconsistent with other algebraic facts. A clause is an atomic formula or its negation, which can be represented as a disjunction of literals. RCF decision procedures are also used to discard redundant clauses that follow algebraically from other clauses [2]. In MetiTarski’s proof search, the RCF tests typically dominate the overall processor time, while the time spent in resolution is much smaller. Only unsatisfiable RCF subproblems contribute to a MetiTarski proof. If the RCF time limit is too long, some problems may not be solved because the RCF decision procedure wastes too much time trying to prove the satisfiability of irrelevant RCF subproblems. However, if the RCF time limit is too short, important clause simplification steps may be missed when literal deletion RCF calls time out.

Machine Learning [3] uses statistical methods to infer information from training examples, which it can then apply to new problems. The Support Vector Machine (SVM) is a method used for classification or regression in the field of machine learning. SVMs are widely used because they can deal efficiently with high-dimensional data, and are flexible in modelling diverse sources of data [9].

2 Methodology

Two RCF decision procedures, Z3 [5] and Mathematica [10], were used in the experiment. Machine learning was applied to select which decision procedure to use and set the best time limit on RCF calls for a given MetiTarski problem. In each individual run, MetiTarski called Z3 or Mathematica with various time limits on RCF calls: 0.1s, 1s, 10s, 100s. A time limit of 100 CPU seconds was set for each proof attempt.

The experiment was done on 826 MetiTarski problems in a variant of the TPTP format. The data was randomly split into three subsets, with approximately half of the problems placed in a learning set, a quarter put in a validation set used for kernel selection and parameter optimization, and the final quarter retained in a test set used for judging the effectiveness of the learning. For each MetiTarski problem, the best setting to use was determined by running Z3 or Mathematica with various RCF calls.
time limits and choosing the one which gave the fastest overall runtime. There were three main tasks involved in the experiment.

The first task was to identify features of the MetiTarski problems. Each MetiTarski problem was characterised by a vector of real numbers or features. Each vector of features was associated with label +1 (positive examples) or -1 (negative examples), indicating in which of two classes it was placed. Taking the setting of Z3 with 1s RCF time limit as an example, a corresponding learning set was derived with each problem labelled +1 if Z3 with 1s RCF time limit found a proof and was the fastest to do so, or -1 if it failed to find a proof or was not the fastest. The features used for the current experiment were of two types. The first type relate to various special functions: log, sin, cos, etc, each feature represents how complex the expressions containing special functions are. For example, the feature value relating to the log function was equal to 0 if log did not appear in the given problem, or equal to 0.5 if log appeared and only applied to a linear expression. All the other occurrences of log were regarded as complicated and the log feature value was set equal to 1. The other type of features are related to the number of variables in the given MetiTarski problem. These features were further differentiated as if they were equal to 0, 1, 2 or more. Having more variables in a problem will generally increase the difficulty of the proof search.

The second task was to select the best kernel function and parameter values. SVM-Light [6] is an implementation of SVMs in C, and was used to do the classification in this work. The accuracy of its model is largely dependent on the selection of the kernel functions and parameter values. In machine learning, the F$_1$-score [7] is often used to evaluate the performance of the classifier, which reaches its best value at 1 and its worst score at 0. A number of different kernel functions as well as variations of parameter values were tested using the learning and validation sets. Following the completion of this task, the best kernel function and model parameters for F$_1$-score maximization were selected for each classifier. The generated SVM models were applied to the test set to output the margin values [4], where the margin is a measure of the distance of the closest sample to the decision line.

The final task was to combine the models for different RCF time settings and compare the margin values produced by their classifiers. The margin values for each classifier were compared to predict which setting is the best for each problem. In an ideal case, only one classifier would return a positive result for any problem, where selecting a best setting is just a case of observing which classifier returns a positive result. However, in practice, more than one classifier will return a positive result for some problems, while no classifiers may return a positive result for others. Thus, the relative magnitudes of the classifiers were considered in the experiment. The classifier with most positive (or least negative) margin was selected.

3 Results

The experiment was done on 826 MetiTarski problems. The data was randomly split into a learning set (414 problems), a validation set (204 problems) and a test set (208 problems). The total number of problems proved out of 208 testing problems was used to measure the efficacy of the machine learned selection process. The learned selection was compared with fixed RCF time setting. The machine learned algorithm for selection performed better on our benchmark set than any of the individual fixed settings used in isolation. The selection results are given in the following bar chart.
4 Future work

Future work remains to be undertaken concerning on extension of the heuristic selection, which allows the choosing of the option settings within the decision procedures. For example, Mathematica has many configurable options, which gives us many possible heuristics to investigate.

Grant Passmore et al. [8], have shown that through detailed analysis of the RCF subproblems generated during MetiTarski’s proof search, specialised variants of RCF decision procedures can greatly outperform general-purpose methods on restricted classes of formulas. It would also be interesting to use a machine learning to learn to select the best stored models for assisting the proof of new RCF subproblems.

Exploring more feature types relevant to the classification processes is also essential for improving the efficacy of the machine learned selection process. After having defined a set of features, it is instructive to perform feature selection by removing features that do not contribute to the accuracy of the classifier. The results of the experiment will be analysed in order to find which measured features make a significant contribution to the learning and classification processes. This analysis will help to find why some heuristics work better with some types of problems and would also help with heuristic development.

References


Arís\textsuperscript{1}: Analogical Reasoning for reuse of Implementation & Specification.

Pitu M., Grijincu D., Li P., Saleem A., Monahan, R. O’Donoghue D.P.

Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland.

Introduction: Formal methods and formal verification of source code has been used extensively in the past few years to create dependable software systems. However, although formal languages like Spec# or JML are quite popular, the set of verified implementations remains small. Our work aims to automate some of the steps involved in writing specifications and their implementations, by reusing existing verified programs i.e. for a given implementation, we aim to retrieve similar verified code and then reapply the missing specification that accompanies that code. Similarly, for a given specification, we aim to retrieve code with a similar specification and use its implementation to generate the missing implementation.

Figure 1: Example: Similar implementations reusing specifications

Reasoning by Analogical Comparison: One of the more successful disciplines in artificial intelligence in recent years has been Case-Based Reasoning (CBR). While CBR traditionally uses relatively straightforward “cases”, retrieving similar implementations requires structure rich “cases” and necessitates CBR’s parent discipline of Analogical Reasoning (AR). Both AR and CBR solve problems not from first principles, but by using old solutions to solve new problems. We use these problem solving disciplines to assist the reuse of verified programs.

The canonical analogical algorithm is composed of the phases: Representation, Retrieval, Mapping, Validation and Induction. This paper focuses on Representation, Retrieval and Mapping. Currently in Arís, our model for verified program reuse, we focus on code retrieval. Representation depicts problems (and solutions) as static parse trees. We then retrieve similar parse trees for a given problem. Next, we identify the mapping between the two parse trees in order to generate the analogical inferences – transferring and adapting the retrieved specification to the given problem. Our parallel work on specifications explores specification matching and specification reuse. We briefly discuss this work later.

Figure 2: The Case-Base contains Source code implementations and corresponding formal specifications

Code Retrieval: When presented with a problem implementation, Arís beings by retrieving a similar implementation so as to re-use its specification. Each “case” is a software artefact which varies in granularity from methods, to classes or full applications. Our current model is implemented to perform retrieval using C# source code and corresponding Spec# (Leino & Müller, 2009) specifications. Retrieval ranks code artefacts by their similarity score with a given input artefact.

\textsuperscript{1} Meaning “again” in the Irish Language
(Mishne & De Rijke, 2004) successfully used conceptual graphs as a basis for source code retrieval. A conceptual graph (Sowa, 1994) is a bipartite, directed and finite graph, in which a node has an associated type (can be either a concept node or a relation node) and a referent value (the content inside the node). We use the concept nodes to model different structures and attributes from the source code (e.g. Loop concept) and relation nodes to show how they relate to one another (e.g. a concept node of type Method may have an ongoing edge to a relation node of type Parameter). The advantage of using this kind of representation is that it offers the possibility of exploring the semantic content of the source code while also analyzing its structural properties using graph-based techniques. In order to create the conceptual graph we first parsed the source code into an Abstract Syntax Tree (using the Microsoft Roslyn API) and then transformed the result into the higher level representation of a conceptual graph.

Source code retrieval is performed using distinct semantic and structural characteristics of the source code. In the semantic retrieval process, we express the meaning of the code through the use of API calls. API calls have been used as semantic anchors in (McMillan, Grechanik, & Poshyvanyk, 2012) as a successful method of retrieving similar software applications, because they have precisely defined semantics - unlike names of program variables, types and words that programmers use in comments (techniques used in the majority of existing source code retrieval systems). We further rely on the Vector Space Model (Salton, Wong, & Yang, 1975) to represent our documents (code artefacts) and use API calls as “words” in these documents to support semantic retrieval.

In the structural retrieval process, we focus on source code topology as represented in these conceptual graphs and by analyzing metrics of these representations (for example, number of nodes, number of loops, loop size, connectivity (O'Donoghue & Crean, 2002)). Based on conceptual graphs, we extract content vectors (Gentner & Forbus, 1991) which express the structure of the code and the number of concepts (for example, loops, statements, variable declarations, etc.). Because our database of source code artefacts will potentially be very large, we need an efficient way of retrieving the most similar cases. We propose using the K-means clustering algorithm in order to create sub-groups of content vectors and perform K-nearest neighbours only on the “closest” sub-group to a given input content vector, thus speeding up the structural retrieval process. We then combine semantic retrieval with structural retrieval and impose different constraints: depending on the type (method, class, application) of the input artefact (query) - retrieving only artefacts of the same type. The outcome of retrieval then, is a ranked list with a small number of the most similar artefacts found in the repository.

**Code Mapping:** This next mapping task is a per-cursor to solution generation, finding detailed correspondences between the old solution and the new problem. We use the term source to refer to each retrieved (candidate) solution in turn and the term target to refer to our unspecified problem code. The objective here is to identify the best mapping between the two isomorphic graphs, where the two programs use different identifier names. But mapping should also cater for homomorphic graphs, allowing different structures to be mapped together. For code matching we propose to use an incremental matching algorithm based on the Incremental Analogy Machine (IAM) (Keane, Ledgeday, & Duff, 1994). Although methods for comparing conceptual graphs have been proposed before, many of them focus on matching identical graphs or subgraphs (like Sowa’s set of projections and morphisms) or they rely on various parameters that have to be empirically determined (Mishne & De Rijke used parameters like concept and relation weights, matching depth etc). IAM begins by matching the two largest sub-trees within the source and target graphs. This forms a seed mapping and then additional structures from the source and target are added iteratively forming a single mapping between the new code and the previously specified code.

**Mapping Constraints:** Specific constraints guide the formation of this mapping. Gentner’s (1982) 1-to-1 constraint ensures that the mapping remains consistent. We also impose further constraints on those concepts that are mapped between the source and target code. We might ensure for example that variables are matched with variables and loops with loops. Isomorphic subgroups that share the same structural properties are then formed based on one-to-one correspondences between the graph concepts. However, we plan to map graphs that don’t share the exact same structure, but that may be related (see for example the two graphs in Figure 3). We are currently exploring with mappings between two non-isomorphic (homomorphic) sub-graphs from the source and target parse trees.

![Figure 3: Two matched program graphs](image-url)
Finding the appropriate “root” concept (the most referenced node in the graph) from which to start the matching process is a challenging aspect of using the IAM algorithm. We will address this issue by assigning node ranks (using a graph metric akin to Page Rank like the one proposed in (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012)) and see whether this will lead us to conclusive mappings between the source and target domains. The generated mapping is then evaluated to decide whether or not it is near-optimal and the algorithm may choose to backtrack to select alternative seed mappings – or the mapping may be abandoned if sufficient similarity cannot be found.

**Inference:** Once IAM has found a suitable mapping, it generates the analogical inferences in order to generate the required specifications for the given code. Analogical inferences are generated using a surprisingly simple algorithm for pattern completion called CWSG - Copy With Substitution and Generation (Holyoak et al., 1994). CWSG transfers the additional specifications from the retrieved code and adds it to the target code – substituting source code items with the mapped equivalents. This should allow our target/problem code to be formally verified using the newly generated specification. Optionally, we can also retain the newly formally verified source code artefacts for further use.

**Parallel Research on Specification Matching and Reuse:** In addition to our work on code reuse we are exploring the reuse of specifications. For a given specification, we aim to retrieve code with a similar specification and use the retrieved implementation to generate the “missing” implementation. The work here has two approaches: the matching of specifications based on the description of the program’s behaviour and the reuse of the same specifications for implementations that differ only in their use of data structure.

The first approach focuses on the specification matching of software components (Zaremski & Wing 1997) using a hierarchy of definitions for precondition and postcondition matching. We apply these definitions to specifications of C# code that are written in Spec# (Barnett et al. 2005) using the underlying static verifier Boogie (Leino et al. 2005) and the SMT solver Z3 (De Moura and Bjørner, 2008) to determine matches and to verify the correctness of our specification implementation pairs. Results to date are promising with a mixture of method specification matches allowing the retrieval and verification of similar specifications and their associated implementations.

Our second approach focuses on reuse of specifications and program verifications. Software clients should only be concerned with specifications and do not need to know details about the implementation and the verification process. This separation of concerns is achieved via data abstraction where we provide an abstract view of a program which we can provide to the client without exposing implementation details. As a result, each specification may have many implementations, each differing in terms of the underlying data structure used in their implementations. The theory of data refinement guarantees that this difference of data structure does not adversely affect the correctness of the programs with respect to their specifications. Our research here explores the reuse of specifications, and the automatic generation of the associated proof obligations, when an implementation is replaced by another implementation that is written in terms of an alternative data structure. We focus on the Dafny language (Leino, 2010), which is closely related to the Spec# language and is verified using the Boogie static verifier. The advantage of using Dafny is that it offers updatable ghost variables which can be used to verify the correctness of a data refinement.

**Future work:** The basic assumption underlying our work is that similar implementations have similar specifications. Further work is ongoing to ascertain the veracity of this hypothesis – and if true, what degrees of “similarity” are required. At the moment our work is fragmented by slightly different tools and approaches. In the future, we hope to combine these approaches within Aris to achieve a fully integrated platform with analogical reasoning as its core, allowing for reuse of implementations, specifications and their verifications.

**References**


HR is a theory formation system, which (semi-)automatically develop existing formal theories, adding new concepts, conjectures and examples, and which has numerous applications in formal methods and beyond [2]. However, obtaining reasonable results in a new domain often requires the system to be properly configured, either by an expert user or through considerable trial-and-error. This can be overwhelming for novices: HR’s GUI has a “Search” panel with 159 parameters (checkboxes, text fields etc.), with even more features available to those willing to investigate the source code.

We report here on some initial experiments on getting HR to automatically learn a simple form of domain-specific guidance, in the context of Event-B lemma generation. Llano et al. [3] showed how HR could be used to generate invariants (conjectures) that allowed Event-B proof obligations to be automatically discharged during the model refinement process. Below, we use one of her examples to illustrate how information about the utility of existing conjectures can be exploited to reduce further search in the same domain.

The general mechanism is as follows: whenever a theoretical entity is found to be useful in a particular domain (e.g. a conjecture discharges a proof obligation), the concepts which lead to its creation are added to a valued list $V$. We define $D(V)$ as the list of theory steps used to derive the elements of $V$. For subsequent searches in that domain, HR’s agenda of theory formation steps can be ordered by i) the maximum complexity of the concepts in each step and ii) the number of times the step’s rule appears in $D(V)$. Hence production rules found to successful in a domain are promoted in the search. Following Llano et al., we use a breadth-first exploration of the theory space, though rule orderings could be used with other strategies, e.g. HR’s interestingness-based heuristic search. Our approach is implemented in a modified version of HR, and demonstrates a simple form of domain-specific learning in theory formation, which we hope to develop in future work.

**Mondex example**

Mondex is a card-based system for handling electronic cash transfers. Butler and Yadav present a formal development in Event-B which incrementally refines an abstract specification of the Mondex system into a detailed design [1]. Each refinement step results in a set of proof

---

This work was supported by the Dependable System Group at Heriot-Watt University, through EPSRC Platform Grant EP/J001058/1. Thanks in particular to Teresa Llano, Andrew Ireland and Simon Colton for their invaluable advice and encouragement.
configured could generate a number of rule orderings. For simplicity, we focus on the ordering application rules.

The hand-crafted macros used by Llano et al. (Llano, personal communication). Five production rules are enabled: exists, numrelation, compose, disjunct and negate.

**Config B** Config A with all applicable production rules enabled. Experiments indicated that eight rules could potentially be applied in this theory: the five from config A, plus size, arithmetic and forall.

**Config C** Config B with the rule ordering compose > disjunct > other enabled.

Config A found 7 of the 9 target conjectures, and generated stronger variants (equivalences) for the remaining two (T2 and T3). Each conjecture required between 60 (T5) and 5524 (T7)
theory formation steps. Enabling all the production rules (Config B) lead to more than twice the amount of search for six conjectures, and more than the step limit for the remaining three. Our rule ordering approach (Config C) demonstrated lower costs than B, allowing all 9 targets to be found even though all rules were enabled. Figure 1 shows the search costs for B and C, relative to A. For conjectures requiring only 1-step concepts, B and C both find the target, with C outperforming A and B, and B requiring about 2-3 times as much search as A. For more complex conjectures, requiring 2-step concepts, C finds the target at much higher cost than A, and B fails to find it within 10,000 steps. In summary, our learned config C outperforms the naive config B, though is ultimately inferior to the hand-crafted config A, which has the advantage of turning some useless rules off completely.

In further work, we hope to test this approach more rigourously — we’re currently working with Llano to recreate the rest of the target conjectures that were used (but not documented) in [3]. We also hope to explore more sophisticated forms of domain-specific knowledge including: patterns of rule applications, e.g. the recurrence of the $x \land (y \lor z)$ concept structure in the Mondex examples; learning analogies between theory predicates, e.g. the similar roles played by $abortepa$ and $abortepv$; and learning measure weights in heuristic search.

References


Drawing Proof Strategies – the PSGraph Language & Tool
Gudmund Grov, Yuhui Lin & Colin Farquhar (Heriot-Watt University)
Aleks Kissinger (University of Oxford)
Ewen Maclean (University of Edinburgh)

Abstract
Complex automated proof strategies are often difficult to analyse, extract, visualise, modify, and debug. Traditional tactic languages, often based on stack-based goal propagation, make it easy to write proofs that obscure the flow of goals between tactics and are fragile to minor changes in input, proof structure or changes to tactics themselves. Here, we outline PSGraph [2], a graphical language for writing proof strategies. Strategies are constructed visually by “wiring together” collections of tactics and evaluated by propagating goal nodes through the diagram. It is supported by the PSGraph tool. We argue that the combination of both procedural and declarative views of proofs, making it more amendable to analysis and generalisations (across similar proofs) – and briefly outlines some techniques for generalisations. More details of the work described here can be found in [2, 3, 1]

1 Introduction
Most tactic languages for interactive theorem provers have no means to distinguish goals. Thus when composing tactics, we have no choice but to rely on the order in which goals arrive, thus making them brittle to minor changes. For example, consider a case where we expect three outputs of tactic $t_1$, where the first two are sent to $t_2$ and the last to $t_3$. A small improvement of $t_1$ may result in only two sub-goals. This “improvement” causes $t_2$ to be applied to the second goal when it should have been $t_3$. The tactic $t_2$ may then fail or create unexpected new sub-goals that cause some later tactic to fail.

As a result: (1) it is often difficult to compose tactics in such a way that all sub-goals are sent to the correct target tactic, especially when different goals should be handled differently; (2) when a large tactic fails, it is hard to analyse where the failure occurred; and (3) the reliance of goal order means that learning new tactics from existing proofs have not been as successful as it has been for discovering relevant hypothesis in automated theorem provers.

Here, we introduce the PSGraph language for representing proof strategies [2]. We argue that this language has three advantages over more traditional tactic languages:

(i) to improve robustness of proof strategies with static goal typing and type-safe tactic “wirings”; (ii) to improve the ability to dynamically inspect, analyse, and modify strategies, especially when things go wrong; and (iii) to enable learning of new tactics from proofs.

We then introduce the PSGraph tool which implements the language. The implementation is generic across theorem provers, but here we focus on the instantiation in Isabelle. Details of this work can be found in [2]. At the end we briefly discuss one particular use of the language – to generalise proofs into proof strategies [3].

2 The PSGraph Language
Traditional tactics are essentially untyped: they are functions from a goal to a conjunction of possible sub-goals. In many programming languages, types are used statically to rule out many “obvious” errors. For example, in (typed) functional languages, a type error will occur when one tries to compose two functions which do not have a unifiable type. In an untyped tactic language, this kind of “round-peg-square-hole” situation will not manifest until run-time.
For errors that cannot be found statically, it is very hard to inspect and analyse the failures during debugging. In the above example, if \( t_2 \) creates sub-goals that tactics later in the proof do not expect, the error may be reported in a completely different place. Without a clear handle on the flow of goals through the proof, finding the real source of the error could be very difficult indeed.

To make proofs more robust, structured proofs show each intermediate goal of a proof, and are thus more robust for changes in the sense one can see exactly at which point a failed proof breaks down. However, for two proofs in the same “family” the goals will vary, thus a proof strategy cannot include the exact expected goal.

To overcome this, PSGraph has a concept of the goal types. A goal type is essentially a predicate on a (sub-)goal, and this (sub-)goal is of the goal type if the associated predicate holds. The underlying theory is generic with respect to the goal type used, and we have currently developed two goal types:

- in [2] a simple goaltype is used, where the user has to provide the implementation of the predicate. A goaltype there is simple a conjunction of such user-provided predicates.
- in [3] a more complex goal type is used. This is used to highlight relevant hypotheses, and enable much more static analysis. This is more suitable for analysis, proof extraction and generalisations, but may be too complex when drawing strategies by hand.

PSGraph is defined using the mathematical formalism of string diagrams. A string diagram consists of boxes and wires, where the wires are used to connect the boxes. Both boxes and wires can contain data, and data on the edges provides a type-safe mechanism of composing two graphs. Crucially, string diagrams allow dangling edges. If such edge has no source, then this becomes an input for the graph, and dually, if an edge has no destination then it is the output of the graph.

The edges in a PSGraph contains a goal type, while a box could either be a tactic or a (sub-)goal. There are two types of tactics. The first type is known as a graph tactic, which is simply a node holding one or more graphs which be unfolded. This is used to introduce hierarchies to enhance readability. A second usage, in the case it holds more than one child graph, is to represent branching in the search space, as there are multiple ways of unfolding such boxes. The other type of tactic is an atomic tactic. This corresponds to a tactic of the underlying theorem prover.

Evaluation is achieved by sending goals down wires, and it evaluation is completed when all the goals are on the output edges of a wire. For this, the (sub-)goal node is used. One step of evaluation for a tactic (box) \( t \), with an input wire containing a goal node \( g \), works as follows: \( t \) is applied to \( g \) and \( g \) is removed from the graph. Each sub-goal produces is then matched to each output edge. If it succeeds then it is added to the graph. Note that if a sub-goal does not match any of the output edges then the step fails, and if it matches multiple outputs, then this will introduce branching in the search space (one for each match).

3 The PSGraph Tool

The PSGraph language has been implemented in the PSGraph tool\(^1\). It is implemented on top of Quantomatic\(^2\), and supports both Isabelle and, to some extent ProofPower, as well as both the goal types discussed above. Figure 1 illustrates one usage of the tool – as an Isabelle method. Here, the \texttt{ipsgraph} (interactive psgraph) method is called from the Isabelle/PIDE interface. This communicates with the graphical interface of PSGraph, which enables one to step through the proof. There is also an automated version of this method called \texttt{psgraph}. \([1]\) discusses generation of structured proof scripts in Isabelle/Isar from an application of a psgraph method.

\(^1\)See \url{https://github.com/ggrov/psgraph/}.

\(^2\)See \url{https://sites.google.com/site/quantomatic/}.
The use of AI for Generalising PSGraph Strategies

One of our key motivations for this language is to support extraction of proof strategies from exemplar proofs, and this is where the AI comes into play. In [3] we have made a start for this by developing methods for in particular generalising goal types and tactic boxes as well as loop discovery. The latter is particularly interesting, as we achieve this by a combination of goal type generalisation, tactic generalisation and graph rewriting. An important property of this is that we can (statically) discover the termination condition for a loop. Developing further techniques for such generalisations is the PhD topic for one of the authors (Farquhar).

In the future we will also integrate the work discussed here, with two other papers submitted to the AI4FM 2013 workshop: (i) ‘Analogical Lemma Speculation’ (Maclean) which we believe can be applied to discover related lemmas in case of cut-rule we were not able to generalise properly; and (ii) ‘Verifying the heap: an AI4FM case study’ (Whiteside et al), where we plan to represent the captured strategies in PSGraph, and utilise the proof process discussed there to guide generalisations.

Acknowledgements. Thanks to member of the AI4FM project, Lucas Dixon and Alex Merry. This work has been supported by EPSRC grants: EP/H023852, EP/H024204 and EP/J001058, the John Templeton Foundation, and the Office of Navel Research.

References


1 Motivation

One of the main ideas that underlies the work discussed here is that the strategy for completing a proof can be explicit or implicit:

**Explicit** strategies are those where each step of the proof is prescribed exactly.

**Implicit** strategies refer to those where the power of the prover and its search strategy means that the choice of rules to apply determines the success or failure of a proof attempt.

Even in the implicit case, some form of strategy is employed but the precise order of application of rules is not of interest.

This paper concerns the implicit case, where we are concerned more with trying to find the right rules to allow the proof to succeed. We present an idea for lemma generation which combines the notion of analogy and theory formation.

The motivation for lemma generation via analogy comes from the main hypothesis of the AI4FM project, combined with an as yet unresolved problem with combinatorial explosion in the generation of lemmas.

In the AI4FM project, the hypothesis is stated as:

we believe that it is possible to (devise a high-level strategy language for proofs and) extract strategies from successful proofs that will facilitate automatic proofs of related proof obligations.

Currently there is an implementation of a “Strategy Language” in Isabelle, which allows proofs to be annotated with “features” and for strategies to be extracted from example proofs [1]. The idea is that the extracted strategy should be able to be “replayed” on other proofs.

The intuition is that during a mechanised proof of a large problem, which involves many smaller proofs, if an expert shows how one example can be proved, a strategy can be extracted from this which accounts for a good percentage of the remaining proofs from the problem.

In this note we translate this idea to the notion of implicit strategies by attempting to discover lemmas by analogy. We attempt to construct lemmas from one template example.

2 Terminology

We work within the context of an Automated Theorem Prover, where a current theorem is being studies and a proof attempted. In the case where it is blocked, the system searches for a similar “analogous” theorem and gleans information from its proof. We use the following terminology:

**Target Theorem** This is the theorem which is to be proved.

**Source Theorem** This is a similar theorem which has been proved and acts as an analogous example to the Target Theorem.

**Source Lemma** This is a user-defined lemma which was employed in the proof of the Source Theorem.

**Target Lemma** This is a lemma which is to be discovered and should be analogous to the Source Lemma.

For this presentation we do not describe in detail the way in which the Source Lemma and Source Theorem are chosen. The work follows the ideas of [3], which provides a way to determine similarity between proofs. The lemmas we prove in this exposition are all non-conditional, which is a restriction of the current system. Dealing with conditionals is further work.
Table 1: Equivalences between functions of the Source and Target Lemmas

3 Lemma Analogy

In order to elucidate our approach we use a running example taken from the work on formalising the Java Virtual Machine in ACL2 [2]. The examples given here are provided by a tool written specifically for ACL2 which incorporates Statistical Machine Learning. For example, we take our Source Theorem to be

\[ fact(n) = qfact(n, 1) \]

where \( fact \) is a primitively recursive factorial function, and \( qfact \) is a tail recursive factorial function where the accumulator argument is here set to 1. The proof of this theorem employs a user-defined generalisation lemma – the Source Lemma:

\[ fact(n) \cdot a = qfact(n, a) \]

Now we are faced with trying to prove the Target Theorem:

\[ qfib(n, 0, 1) = fib(n) \]

where \( fib(n) \) gives the \( n \)th number of the Fibonacci sequence, and \( qfib \) is a tail recursive version with two accumulator variables set to 0 and 1.

We omit the definitions of the defined functions for space reasons as they are standard in the literature. The Source Lemma is the key to proving the Source Theorem – it is a typical generalisation used in proving equivalences between tail recursive and primitively recursive functions. The goal is to efficiently generate the analogical lemma for the Target Theorem.

3.1 Function symbols

Firstly we identify the common symbols to the theory of naturals - these are

\[ + \quad \ast \quad - \quad 1 \quad 0 \]

then we attempt to create equivalences between the Source and Target Theorem symbols. Here we focus on the equivalence: A set of tables of equivalences such as these are generated in order to construct the analogical lemma. Common functions symbols are constrained to be equivalent to common function symbols, so \( \ast \) cannot be replaced with \( fib \) for example. Table 1 shows just one possible equivalence.

3.2 Tree recomposition

The term tree of the Source Lemma is decomposed and recomposed with the function symbols from the equivalence table shown in 1. Where the arity is different, a new variable is inserted at each possible location of the correct type (in this case just natural numbers), and all permutations are generated. Where the lemma is an equality with one function symbol on one side, no permutation is calculated as this produces duplicates in alpha-equivalences. The tree recomposition shown in the left side of Figure 3.3 shows a first attempt at tree recomposition. Each candidate lemma is passed through a simple counterexample checker which attribute values to each variable and checks if the lemma is invalid. In the case of the lemma on the left hand side of Figure 3.3:

\[ fib(n) \cdot a = qfib(n, m, a) \]

the candidate lemma can immediately be ruled out as a candidate due to the existence of a single variable which does not appear on the other side of the equality. If no potentially valid candidate lemma is found then a more complicated tree recomposition (called mutation) takes place.
3.3 Tree mutation

If no generated lemma passes the testing phase, then a secondary, more involved process takes place where the tree is “mutated” by complicating the tree to add more structure to the nodes. The process expands each leaf node of the tree where a variable is not found, and then in a final phase expands the top node to introduce extra structure to account for theorems such as common distributivity laws. Such a tree reconstruction is shown in the right side of Figure 3.3. This leads to the correct lemma to be used:

\[ qfib(n, m, a) = fib(n) \cdot a + fib(n - 1) \cdot m \]

![Figure 1: A correct and incorrect recomposition of the Source Lemma using symbols from the Target Theorem](image)

4 Results

We have tested the system on a set of theorems from the ACL2 implementation of the Java Virtual Machine:

- \( qfact(n, 1) = fact(n) \)
- \( qpower(n, 1) = power(n) \)
- \( qexpt(n, m, 1) = expt(n, m) \)
- \( qmul(n, m, 1) = mul(n, m) \)
- \( qsumsquare(n, 0) = sumsquare(n) \)
- \( qsum(n, 0) = sum(n) \)
- \( qfib(n, 0, 1) = fib(n) \)

By choosing all possible Source and Target combinations, the correct lemmas are suggested immediately except in the case where the Source Lemma has a deeper term tree than the Target Lemma. This is a failing of the system. However, the process has discovered some interesting results, which are unexpected, such as

\[
qfact(n, m) = (fact(n - 1) * n) + (fact(n) * (m - 1))
\]
\[
qpower(n, m) = (power(n - 1) * m) + (power(n - 1) * m)
\]

where \( power(n) \) denotes \( 2^n \) and \( qpower \) is a tail-recursive version.

References


Theory Exploration for Interactive Theorem Proving

Moa Johansson
Chalmers University of Technology

Abstract

Theory exploration is an automated reasoning technique for discovering and proving interesting properties about some set of given functions, constants and datatypes. In this note we describe ongoing work on integrating the HipSpec theory exploration system with the interactive prover Isabelle. We believe that such an integration would be beneficial for several reasons. In an interactive proof attempt a natural application would be to allow the user to ask for some suggestions of new lemmas that might help the current proof development. Theory exploration may also be used to automatically generate and prove some basic lemmas as a first step in a new theory development. Furthermore, when the theory exploration system is used as a stand-alone system, it should output a checkable proofs, for instance for Isabelle, so that sessions can be saved for future use.

1 Introduction

Theory exploration is an automated reasoning technique for discovering and proving interesting properties about some set of given functions, constants and datatypes. In the context of theorem proving, theory exploration systems are typically used to generate candidate conjectures satisfying some interestingness criteria, using automated testing or counter-example finding to produce candidates likely to be theorems.

Many theory formation systems, such as MATHsAiD [6], IsaCoSy [5] and IsaScheme [7] are automatic and attempt to produce a background theory with ‘interesting’ lemmas which are likely to be useful to in later proof attempts. However, while both IsaCoSy and IsaScheme are built on top of the interactive prover Isabelle [8], neither can be called from an interactive proof attempt. This problem is due to long runtimes, IsaCoSy and IsaScheme are generally too slow. We have recently developed a new theory exploration system called HipSpec, which discovers and proves properties about Haskell programs by induction [2]. HipSpec is considerably faster than previous systems, and thus a candidate for integration in an interactive system.

Large theories with many functions and datatypes is a major challenge for automated theory exploration systems. To deal with theories with many hundreds, or even thousands, of functions requires some form of modular exploration and division into smaller sub-theories. In the interactive scenario we can to some extent circumvent this problem. Where the user asks the system for suggestions of lemmas applicable to the current subgoal, the input to the theory exploration system can be directly specified by the user or limited to functions relevant to that subgoal.

In this note, we describe a new project aiming at integrating theory exploration and interactive theorem provers:

- Theory exploration has so far mainly been used in conjunction with automated theorem provers. Many proof developments are however too complex to be fully automated. By integrating theory exploration with an interactive theorem prover we make theory exploration available to a wider audience.
- We aim to develop a tool integrated with the interactive proof assistant Isabelle which allow a user who is stuck in a proof attempt to ask for some candidate lemmas which could help the current proof attempt. It is crucial that the system responds quickly and does not produce too many (or too few) candidates.
- We would like theory exploration systems such as HipSpec to produce a sensible output format, so that proofs can be independently checked and lemmas recorded in a library. Producing Isabelle theory files is one option.
2 Background: The HipSpec System

HipSpec is an inductive theorem prover and theory exploration system for discovering and proving properties about Haskell programs. HipSpec takes a Haskell program annotated with properties which the user wish to prove and first generate and prove a set of equational theorems about the functions and datatypes present. These are added to its background theory and user specified properties are then proved in this richer theory. Experimental results have been encouraging, HipSpec performs very well and proved more theorems automatically than other state-of-the-art inductive theorem provers [2].

Figure 1: Architecture of the HipSpec system.

HipSpec combines the inductive prover Hip [10] with the QuickSpec system [3]. An overview of HipSpec is shown in Figure 1. HipSpec first translates the function definitions and datatypes from the Haskell program into first-order logic to form an initial theory about the program. QuickSpec also reads in the available function symbols and datatypes and proceeds to generate new terms from these. Using the automated testing framework from the QuickCheck system [1], QuickSpec divides these terms into equivalence classes, from which equational conjectures are extracted. These conjectures are passed back to Hip, which applies a suitable induction scheme and feeds the resulting proof-obligations, along with the first-order background theory, to an automated first-order prover, for instance Z3 [4]. If the proof succeeds, the new theorem is added to the background theory, and may be used as a lemma in subsequent proof attempts. If a proof fails, HipSpec tries other conjectures and may return to open ones later, when more lemmas have been added to the background theory.

3 Integrating HipSpec and Isabelle

Isabelle’s higher-order logic is essentially a (terminating and finite) subset of the functional programming language Haskell, and Isabelle’s code generator can translate Isabelle theories into Haskell programs. HipSpec then needs to monomorphise any polymorphic types in order to translate it to first-order logic, after which it can generate lemmas about the functions corresponding to the Isabelle theory. A very early prototype allowing HipSpec to be called from Isabelle has been implemented, but a lot of implementational work remain, in particular to import lemmas discovered back into Isabelle.

HipSpec does not produce proofs, as it relies on external provers as black boxes. We have however experimented with using Z3’s capability to produce unsatisfiable cores in order to report back which lemmas were used in a proof. Using this information, we plan to experiment with a lightweight inductive tactic for Isabelle, which should apply induction and simplification using the relevant lemmas, thus verifying the proof in Isabelle as an added soundness check. Given the right lemmas, we expect such a simple induction tactic would succeed in proving many of the conjectures which HipSpec has discovered. This is similar to how the Sledgehammer system works [9], it sends a conjecture from Isabelle to external provers and then replays the proof using Isabelle’s internal prover Metis. Once we have an Isabelle tactic for HipSpec we can use it in several ways:

Automated induction: HipSpec takes the current goal and applies theory exploration and induction. If it succeeds, it reports which induction scheme and lemmas were used to its corresponding tactic, which replays the proof in Isabelle. If any new lemmas were found, information to replay their proofs should also be produced.
**Generate background lemmas:** The user specifies a set of Isabelle functions and datatypes which are of interest. HipSpec applies theory exploration to these and generate a set of interesting lemmas along with instructions to verify the proofs inside Isabelle. This could be done as a first step in a new theory development, to automatically generate many basic lemmas before the user tackle more complicated theorems. Alternatively, the user may call theory exploration to help in an ongoing proof-attempt where the user is stuck. In this case, theory exploration is even more restricted, it should only generate lemmas which really do apply to the goal, anything else is uninteresting. The user should be presented with a short list of options to choose from.

HipSpec use random testing to generate conjectures, hence all conjectures have passed a large number of tests before being submitted to the prover. In an interactive setting, even conjectures HipSpec has failed to prove automatically can be of interest, with the user interactively supplying the proof, should she/he find the conjecture is useful.

**Produce Isabelle theories about Haskell programs:** Ultimately we would like HipSpec to be useful for verifying real Haskell programs. As mentioned, HipSpec currently use a black-box external prover and does not output any proofs. For verification purposes we may want some additional reassurance. Therefore, we suggest that HipSpec should have the option to output its results Isabelle theories. The user can then inspect the Isabelle theory, and recheck the proofs independently if required. Lemmas about libraries can also more easily be imported if required. Translating from Haskell to Isabelle is however not as straight forward as the other way around, Haskell is a lazy language and also support infinite datatypes and non-terminating functions, which Isabelle/HOL does not. Hence, in the first instance we will be limited to the finite and terminating subset of Haskell.

4 Summary

We believe that theory exploration would be very useful also in an interactive setting. It could assist the user with suggestions of lemmas relevant to the current proof attempt, or simply to generate basic lemmas in a new theory. By letting the user specify the functions and datatypes passed to theory exploration, we can control the search space and explore larger and more complex theories that are currently beyond the scope of the fully automatic version. We are currently working on integrating the theory exploration system HipSpec with the interactive prover Isabelle.

References


