CHALMERS

Automated Discovery of Conditional Lemmas in Hipster

Master of Science Thesis in Computer Science (Algorithms, Languages and Logic)

IRENE LOBO VALBUENA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

Automated Discovery of Conditional Lemmas in Hipster

IRENE LOBO VALBUENA

Examiner: PATRIK JANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Automated Discovery of Conditional Lemmas in Hipster

Irene Lobo Valbuena

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

Abstract

Hipster is a tool for theory exploration and automation of inductive proofs for the proof assistant Isabelle/HOL. The purpose of theory exploration is to automate the discovery (and proof) of new lemmas of interest within a theory development, enriching the background theory and providing necessary missing lemmas that might be required for other automated tactics to succeed.

Hipster has so far succeeded in incorporating automated discovery of equational conjectures and lemmas. This work presents an extension of Hipster adding support for *conditional lemmas*, required, for example, when reasoning about sorting algorithms and treating different branches of a proof along with their specific conditions.

The main focus is the implementation of a tactic for automated inductive proving via *recursion induction*, accompanied by an evaluation of the tool's capabilities. Recursion induction succeeds at automatically proving many of the discovered conditional conjectures, whereas a previously existing tactic was unable to. Additionally, recursion induction manages to set up inductive steps in a proof to render more immediately realisable subgoals by following computation order. Results show proving capabilities are increased not only with respect to conditional lemmas but also for more general equational lemmas.

Keywords: conditional lemmas, theory exploration, inductive theorem proving.

Topics: logic, functional programming.

Acknowledgements

First and foremost, I would like to thank my supervisor Moa Johansson for the initial project suggestion and her continued interest as I worked on it. Her reviewing of the present thesis and advice has been of great value to me, as well as her encouragement to present early stages of the work at the Second Workshop on Inductive Theorem Proving (March 2015, Chalmers University of Technology, Gothenburg, Sweden).

Thanks to the rest of the HipSpec team, Koen Claessen, Dan Rosén and Nicholas Smallbone; Víctor López Juan, for his interest in being my thesis opponent; my examiner Patrik Jansson; and the Computer Science and Engineering Department, Chalmers University of Technology, particularly the Software Technology division.

I would like to also thank Chalmers University of Technology and the Swedish state, for making it possible to take Master's studies at Chalmers.

Last, but not least, special thanks to Avlant Nilsson; Daniel Schoepe, David Muelas Recuenco, Ricardo Huertas Francisco and Raúl Cajías Hostos, amongst other friends, for their support and camaraderie at different times, and my family, whose efforts have nurtured my own: *os tengo presentes*.

Irene Lobo Valbuena, Göteborg, Sweden June 2015

Contents

Chapter 1

Introduction

Often, when proving a theorem, one finds oneself proving a separate, smaller lemma that will help complete the original theorem. But it might not be until deep into the proof that there is a need for an auxiliary lemma; other times the lemma required is not immediately obvious from the development of the proof.

Take for instance the definition of insertion sort `isort`:

```
fun insert :: "Nat ⇒ Nat List ⇒ Nat List" where
  "insert x [] = x # []"
| "insert x (z # xs) =
    (if x ≤ z
      then x # (z # xs)
      else z # (insert x xs))"

fun isort :: "Nat List ⇒ Nat List" where
  "isort [] = []"
| "isort (y # xs) = insert y (isort xs)"
```

And say we try to prove the correctness of part of its specification, that applying it to an input argument does indeed result in a list with elements in increasing order:

```
fun sorted :: "Nat List ⇒ Bool" where
  "sorted [] = True"
| "sorted (_ # []) = True"
| "sorted (r # (t # ts)) = (r ≤ t ∧ sorted (t # ts))"

theorem isortSorts: "sorted (isort ts)"
```

However, to be able to prove this, we need to be able to conjecture and prove an auxiliary lemma:

```
lemma isortInvariant : "sorted ys ⟹ sorted (insert x ys)"
```

Conditional lemmas like this one appear often in proofs for algorithm correctness and in propositions characterising the behaviour of operators and programs like the ones involved in insertion sort.

When using interactive computer systems to mechanise proof checking or aid in developing proofs, automatically generating side lemmas can help the working user and expand the background knowledge for some theory, easing automation of theorem proving. Automated conjecture and proof of conditional lemmas poses itself as a problem to be solved.

1.1 The objective

Hipster [27] is a tool designed for *theory exploration* for inductive theories in the proof assistant Isabelle/HOL. Its aim is to discover lemmas of interest and prove them within a formal system to achieve *inductive theorem proving automation*.

The original implementation of Hipster succeeded at discovering and proving equations about inductive datatypes. This work extends Hipster to add support for *conditional lemmas*.

With the discovery of conditional lemmas, propositions like insertion sort's correctness and many others could be postulated and proven automatically, becoming part of a theory knowledge and enabling the automation of other proofs.

1.2 Method

To improve Hipster's proof automation, not only for conditional lemmas but also for equational ones, *recursion induction* has been incorporated in the new automated proof procedure. Previously, its inductive proving employed structural induction. By applying recursion induction, that which follows the order of computation of function definitions, we achieve a structural setup of proofs, case distinctions and inductive steps more closely tailored to the problem to solve, whilst saving the automated generation of (generic) pre-defined induction schemata that could be followed.

Discovering conditional lemmas is possible via underlying components to Hipster's architecture, namely HipSpec [13] and QuickSpec [15]. HipSpec is a tool for automated derivation and proof of properties about functional programs (for the Haskell language) whilst QuickSpec generates possible specifications of Haskell programs. To be noted is that Hipster produces all of its proofs autonomously within Isabelle, whereas it uses QuickSpec's discovery features via HipSpec.

1.3 Scope and boundaries of the work

The domain of application of Hipster is that of inductive, algebraic datatypes and functions defined over them, most commonly recursive in structure.

Hipster's tactics can be used standalone in Isabelle/HOL proofs, without explicit exploration of a theory. Constructing proofs within a verified, formal system such as Isabelle guarantees their soundness.

The intended use of Hipster is during an interactive session, where the user specifies which functions, constants and predicates should be provided to the theory exploration process. Implicitly, this means the tactics count on theory exploration to have been performed previously or there existing part of a theory already developed in order to be able to obtain proofs to theorems in increasing degree of complexity.

Given its use of QuickSpec's capabilities, the generation of conjectures is bound to QuickSpec's coverage and efficiency.

1.4 Experiments

To evaluate Hipster's performance, use-cases in an interactive setting have been developed, of which we present a case study on insertion sort's correctness.

But to closely examine the new tactic's proving capability, tests from the recently released benchmark set Tons of Inductive Problems [14] (TIP) have been performed. Since we assume the tactic to be used along with exploration, these tests are run accordingly, counting on the possibility of exploring a problem's theory before proving it.

1.5 Outline

Chapter 2 gives an overview of theorem proving, with a focus on interactive settings and the benefit of them integrating automated reasoning tools. We introduce Hipster's architecture, its prior state of development and its working environment in Chapter 3, only giving an overview of relevant aspects of Isabelle/HOL to the rest of this text.

Next, we describe and discuss the use of recursion induction in the new tactic, and its relation to supporting conditional lemmas in Chapter 4. Chapter 5 presents Hipster's tactic as a whole and new options it supports, followed by an overview of the theory exploration process from Hipster's perspective in Chapter 6.

Experiments and their evaluation results are detailed in Chapter 7, both an account of the interactive development of a proof for insertion sort's correctness with Hipster and performance on a subset of TIP. A detailed discussion of the results and gains of recursion induction as well as other aspects of automating discovery and proof of lemmas in our setting is given in Chapter 8 along with future work to focus on in the development of Hipster.

Chapter 9 gives an overview of related work and we conclude in Chapter 10 with an overall view of Hipster's improvement and insight gained during this thesis' work.

Chapter 2

Theorem Proving Systems: A Survey

2.1 Formal science

The formalisation of mathematics in computerised systems is a long-standing problem in science, as is the development of approaches and tools towards automated reasoning. Amongst the first successful incursions into automating reasoning in a computer was the development of GPS (General Problem Solver) [35], dating as far back as 1957.

The significance of this line of work for software verification and proof checking has been great, both within science and already in the industry as well. Formalising mathematics allows for implementing and automating mathematical proofs achieving, for example, guarantees of correctness of software and protocols. Tools for software verification, such as KeY for Java [1], serve as a bridge between research development in automated reasoning and real world applications. Whereas developments like the proof for loop freedom of the AODV routing protocol, produced by means of the interactive theorem prover Isabelle/HOL [5], not only portray an important application but also serve as a testimonial of the benefits of mechanising proofs in theorem provers.

The importance of formalising mathematics transcends applications within software development: the collection of all mathematical works is extremely large and formalising it in a computer system with an appropriate representation would increase its availability for use in computer software and in the development of mathematics. More importantly, mathematical formalisation and mechanisation contributes to ensuring no mistakes are made in proofs, and has already been known to detect mistakes in known results, such as Fleuriot and Paulson's work did with Newton's development of the Kepler Problem [20].

Choosing how to formalise mathematics becomes complicated having to consider various aspects such as achieving a compromise between ease of automated handling of objects (for which generalisation of problem representation has been a question since the early days of GPS [19]) and readability of the presentation

given to a human user.

2.2 Interactive Theorem Proving

In particular, interactive theorem proving (ITP), performed within a proof assistant, can become a great tool for not only computer scientists but also for working mathematicians in their theory developments. By providing an environment where one gradually defines a theory context and explores properties, they give a basic setting similar to the one in which mathematics is developed whilst their formal setting ensures correctness. Mechanised proofs such as Gonthier's for the Four Colour Theorem [21] produced in the proof assistant Coq serve as evidence of interactive theorem proving's importance in proof checking of large and complicated mathematical results.

Different interactive theorem provers achieve formalisation and correctness in different ways. Some are strongly based on programming languages. Such is the case of Agda [38], a dependently typed functional language in which proofs are built by program construction, and ACL2 [28], a formal system based on first-order logic. Other systems are built atop a small, certified kernel. Such is the case for the aforementioned Coq, which is based on the Calculus of Constructions [16] (a type theory), and for a large family of provers based on Milner's LCF [33] (Logic for Computable Functions) theorem prover and the associated programming language ML. Amongst LCF provers one can find the succeeding family of HOL (Higher Order Logic) provers [22], to which the Isabelle proof assistant belongs.

LCF-style systems benefit from having a small trusted kernel upon which to build more complex and sophisticated reasoning procedures (often called tactics) which will be, so to speak, correct by construction (note that this correctness is only meant in terms of soundness but not necessarily completeness).

Despite ITP rising in popularity, as of today it is not that often that one finds mathematicians employing these tools, partly because of the time-cost involved in mathematical formalisation.

2.3 Automated Theorem Proving

The first complete and correct proof for the Four Colour Theorem was in fact possible thanks to the development of automated methods for proof search [2], a pioneering result from the use of computer aided reasoning.

Tools for specific kinds of reasoning, such as first-order (FO) logic theorem provers and satisfiability (SAT) solvers, along with the closely related satisfiability modulo theory (SMT) solvers, have been highly developed in the efficiency of their search algorithms. Renowned solvers include FO provers Vampire [43] and SPASS [48], and SMT solver CVC4 [17].

2.4 Theory exploration

ITP can highly benefit from efficient and well-developed specialised ATP techniques and tools to aid the user in his/her work.

Buchberger introduced the concept of theory exploration in [6]: he argues that the normal working of a mathematician starts by introducing some definitions and axioms which are the base for the theory studied, moving on to proving simple lemmas about these and arriving at more complex theorems by employing these or layering concepts already introduced. Theory exploration looks to automate this by attempting to conjecture new lemmas within a theory and prove them, in increasing order of complexity; a feature Buchberger has focused on in Theorema [8], a computer system for formal mathematics. When implemented in a proof assistant, the mathematical development becomes more in line with the natural flow of a mathematician's work and can also reveal properties of interest that could otherwise not be immediately thought of. So theory exploration hopes to enhance the mathematical development process by imitating, to some extent, the manner in which mathematics is typically done.

As outlined by Buchberger [7], desirable features (to his judgement) of systems for theory exploration are quite challenging to achieve, ranging from retention of both mathematical computational power and reasoning ability to the appeal of syntax and appropriate knowledge structuring. As stated there, no reasoning systems at the time met the features described and most likely still do not. It is not the goal of this project to fit in such suggested requirements (amongst other things, the scope of those is quite broad), but to continue in the line towards the incorporation of theory exploration systems in the workflow within interactive theorem provers.

There has been further work towards the incorporation of theory exploration in mathematical development, such as the system MATHsAiD, which discovers and proves lemmas from simple definitions and axioms [31]. Other tools have made incursions into incorporating lemma discovery in Isabelle, like IsaCosy, developed to synthesise inductive conjectures and lemmas from irreducible terms [26], and IsaScheme, which implements schema-based theory formation [34]. The Hipster relative HipSpec performs theory exploration via testing for functional programs in Haskell [13].

2.5 Inductive Theorem Proving

Theory exploration could be performed in either a bottom-up or top-down manner. Exploring a theory bottom-up looks to discover new lemmas starting from the generation of terms based on the smaller components of a theory (defined constant symbols), combining them and increasing the complexity of such terms gradually in the search for properties that could hold in a theory. Following a top-down approach would analyse new sub-goals encountered during a proof, trying to prove them independently as separate lemmas of the theory, often making use of failed proofs to guide the lemma discovery.

Recursion and induction's self-reference component and explicit incremental construction make them a pattern for which bottom-up theory exploration could

work well in automating discovery of new lemmas. Both HipSpec's and Hipster's work showed promising results for inductive theorem proving when adopting this theory exploration approach.

Alternative approaches to inductive theorem proving are taken by CVC4, which recently introduced inductive reasoning for SMT solvers [42], and the recently developed Pirate, which achieves automation of induction by extending the superposition calculus within a first-order prover [47].

Worth highlighting is work that preceded that of the previously mentioned IsaCoSy: an approach to guiding induction via rippling with case-analysis developed for Isabelle/HOL [25]. Said work was concerned with conditional statements as well, evaluating over theories based on natural numbers, lists and binary trees. In addition to it being a good source for conditional lemmas, the conclusions given in it also pointed towards improvements in handling the conjecturing of conditional lemmas to be of importance for automated theorem proving: from their corpus of theorems, many of those not solved by the techniques they presented required improved reasoning about side-conditions or conditional lemma generation.

Chapter 3

Hipster and its environment, Isabelle/HOL

3.1 Hipster

Hipster [27] is a theory exploration system for Isabelle which aims to discover missing lemmas in a given theory. It can automatically provide basic lemmas about a theory which can then be used both in automated and interactive proofs, contributing to building a corpus of lemmas within a theory gradually.

Its design is parameterised by two proving tactics, one meant for *hard* reasoning and another for easier *routine* reasoning, which it uses to determine which discovered and provable lemmas could be of interest and which could be trivial, respectively. Specifically, Hipster is currently instantiated to enable inductive theorem proving, but the system in itself may be used for any other kind of problem one chooses to explore.

For its induction instance, hard reasoning involves induction whilst routine reasoning corresponds to simplification optionally accompanied by first-order reasoning. Induction is necessary when reasoning about repetition, that is, when reasoning about patterns that use self-reference and iteration. Properties on recursive datatypes and functions require induction in their proofs, making it the proof procedure of choice when reasoning about functional programs and their correctness, being recursion an elementary component in their construction.

However, automating inductive proofs is a challenging problem: despite induction being semi-decidable, there is neither a definite way of defining in advance how many separate induction steps are required, nor which induction schemata are appropriate, nor which of the existing variables in a proposition one should induct on. In other words, an inductive proof may rely on other auxiliary lemmas, in turn requiring themselves inductive proofs. This poses an added challenge: discovering and proving those helper lemmas.

Hipster aims to counter this problem with theory exploration in a bottom-up fashion. It can be invoked in two modes: to automatically generate basic lemmas in a new theory development or to assist in a stuck proof attempt by discovering

new lemmas that would help in proving the current open goal. Its original scope was that of automating equational reasoning for inductive theories.

3.1.1 Architecture

Hipster works with Isabelle/HOL theories. A theory is simply a collection of datatypes and associated function definitions along with lemmas and theorems about these constructs. All of Hipster's reasoning and proving procedures take place within Isabelle, ensuring only sound proofs are constructed.

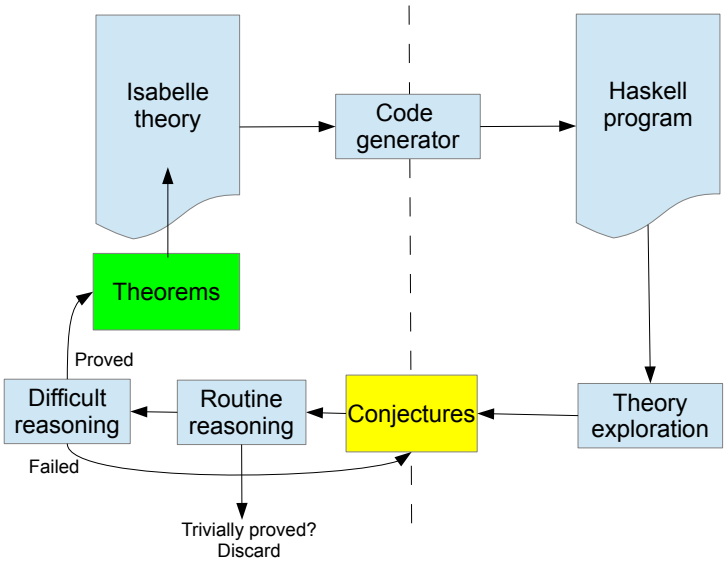

Figure 3.1: Hipster's architecture at a glance (reprinted from [27]).

To discover possible lemmas, Hipster first uses Isabelle/HOL's code generator to output corresponding Haskell code to the definitions and, after processing further the resulting Haskell module, employs QuickSpec's [15] automatic generation of Haskell program specifications (via HipSpec). QuickSpec is based on QuickCheck [12], a tool for random automated testing of Haskell program properties. QuickSpec generates expressions combining defined functions and uses QuickCheck to generate random test-cases; the evaluation on these tests is used to separate terms into equivalence classes. Lemmas discovered during theory exploration in Hipster hence depend on the kinds of schematic patterns and function combinations QuickSpec constructs.

The properties generated are equations; some support for generating laws subject to a premise (conditional propositions) exists, restricted to predicates of up to two arguments as pre-conditions to consider.

3.1.2 Conditional lemmas

Up till now, Hipster could discover and prove equational conjectures for Isabelle/HOL inductive theories. Whilst equational lemmas can give useful properties and insight into a mathematical structure, it is quite often the case that reasoning takes place within a specific frame or context where the structure of objects involved is restricted. For instance, the property `xs` $\neq$ `Nil` $\Longrightarrow$ `head (append xs ys) = head xs` for the definitions:

```
datatype 'a List = Nil | Cons 'a "'a List"

fun append :: "'a List ⇒ 'a List ⇒ 'a List" where
  "append Nil y = y"
| "append (Cons z xs) y = Cons z (append xs y)"

fun head :: "'a List ⇒ 'a" where
  "head (Cons t _) = t"
```

only makes sense and will be correct when the first list is constrained to not being empty.

There are diverse scenarios and constructs which require or give rise to constrained propositions, such as:

- branching on specific conditions which determine cases to consider during a proof attempt
- reasoning about algorithm correctness, where consecutive steps are to be taken into account for the changes they perform
- programming invariants
  - in datatypes, where sometimes invariant properties cannot be encoded in the constructors and instead depend on the arrangement of data values building an instance, making it necessary to verify that functions manipulating the datatype preserve the invariant
  - in functions and algorithms, where steps might be broken down into smaller functions which assume certain conditions on their input to be able to guarantee conditions over the output result.

The use of conditional lemmas can be said to be a necessity in breaking down reasoning into smaller units of focus. As depicted in the opening section, reasoning about sorting algorithms requires support for automated discovery and proof of conditional lemmas.

Discovery in itself of arbitrary conditional lemmas is a hard, combinatorial problem beyond the scope of this thesis. Without counting on pre-existing predicates, the options in synthesis of conditions soon start growing exponentially. And even with potential predicates existing in the theory, the choice of what arguments to apply them to (simple variables or arbitrary expressions) and the combinations of different predicates still complicate the task highly.

For this work, we focus on the particular case of having user specified conditions instead.

3.2 Isabelle/HOL

The proof assistant Isabelle [40] provides the environment in which Hipster constructs its proofs. Isabelle has a central meta-logic Isabelle/Pure [39] (a simply-typed intuitionistic higher-order logic with equality), which can be instantiated to an object-logic via its corresponding deduction calculus. In particular, Hipster works with Isabelle/HOL, the Higher Order Logic instance in which most Isabelle developments are carried out.

3.2.1 Practical layers

In terms of use, one can identify two main layers in Isabelle.

The top level, where notation is the closest to the usual mathematical one and with which the regular user interacts, is that corresponding to so-called Isabelle *theories* or modules. These contain proofs written in the Isar (Intelligible semi-automated reasoning) language, whose purpose is that of providing a structured proof document format for definitions, theorems and proofs [50].

The underlying system and lower level is implemented in Isabelle/ML, including its LCF-style kernel. It is a tactic-based proof system; tactics automate proof search procedures. Extensions and additional tactics are also implemented in Isabelle/ML.

Whilst Hipster is implemented in Isabelle/ML, its immediate intended use is at the top Isabelle theory level, but its functionality could potentially be used in the development of other proof procedures in combination with other Isabelle/ML tools.

The meta-logic Already at the theory level, Isabelle's meta-logic coexists with the object-logic being used, in our case HOL. The former is seen in the formulation of theorems whilst the latter may be employed for the construction of HOL terms (which in turn may appear in a theorem or lemma).

Isabelle's meta-logic has the following connectives:

- $/\backslash$: universal quantification (or abstraction over some parameter)
- $\Longrightarrow$: implication (dependency between proofs)
- $\equiv$: definitional equality

Some Isabelle/HOL syntax remarks The so-called *outer syntax* (see §2.1 in [37]) is the syntax employed to define a theory. Embedded in the theory definition syntax, one encounters an *inner syntax*, whose terms are to be surrounded by double quotes `"..."`. HOL types, terms and formulae are written with this inner syntax.

With respect to HOL syntax, shorter arrows $\Rightarrow$ are those denoting function types and are not to be confused with the meta-logic implication connective introduced above. Additionally, free or unbound variables in HOL are implicitly universally quantified at the meta-logic.

3.2.2 Tactics and automation

Isabelle/HOL has a series of automated tactics and tools which can assist greatly the user in their work. On the one hand, counterexample generators such as Nitpick [3] and Quickcheck [9] (which works in a similar manner to the Haskell QuickCheck [12] tool) help in quick diagnosis towards identifying cases which falsify a statement at hand. On the other hand, amongst the automated tools which work towards the construction of proofs, Sledgehammer [41] uses machine learning in choosing existing lemmas which are likely to be useful to prove a given goal. These lemmas are then run through a series of very efficient external first-order provers (such as Vampire, SPASS or E [44]) and uses their output to reconstruct a minimised proof using an Isabelle/HOL first-order resolution tactic based on Metis [23].

A common trend in all these tools is that of interfacing with other efficient, well-established external tools such as the FOL SAT-based constraint solver Kodkod in Nitpick [46] or the FOL automated theorem prover SPASS in Sledgehammer [4]. Hipster follows in this trend for its conjecture generation, but relies solely on Isabelle/HOL for its proof construction.

Chapter 4

Inductive theories and recursion induction

4.1 Induction

To equationally reason about algebraic inductive datatypes, the main proof method of choice is induction. Such datatype definitions give a recursion pattern from which an induction principle can be derived, the structural induction principle. The inductive definition sets the base-case initial objects (those which do not refer back to the structure being defined) and the inductive combinations of objects of the datatype (those which are recursive on the datatype) used in constructing the datatype.

Structural induction, as a proof principle, states that for any one predicate P over an inductive datatype, if P holds true for all the initial objects (base cases), and also holds for all recursive constructions given it holds for the subparts involved in them (induction hypothesis), then P will hold for every instance of the datatype. Expressed as an inference rule, in the case of a datatype expressed in terms of itself alone, we would have:

StructuralInd

$$\frac{\begin{array}{c} P(C_1) \quad ... \quad P(C_m) \\ \forall b_1,\ b_2,\ ...\ (P(b_1) \implies P(b_2) \implies ... \implies P(C_{m+1}\ b_1\ b_2\ ...))\ ... \end{array}}{\forall x\ P(x)}$$

Where C_i are constructors (for initial or recursive objects) and b_i are variables of the type the induction is described over. For instance, for the datatypes

```
datatype Nat = Z | S Nat
datatype 'a List = Nil | Cons 'a "'a List"
datatype 'a Tree = Leaf | Node "'a Tree" 'a "'a Tree"
```

one obtains the induction principles

NatInd

$$\frac{P(Z) \qquad \forall n\ (P(n) \implies P(S\ n))}{\forall x\ P(x)}$$

ListInd
$$\frac{P(Nil) \qquad \forall t,\ ts\ (P(ts) \implies P(Cons\ t\ ts))}{\forall x\ P(x)}$$

TreeInd
$$\frac{P(Leaf) \qquad \forall ts,\ t,\ rs\ (P(ts) \implies P(rs) \implies P(Node\ ts\ t\ rs))}{\forall x\ P(x)}$$

These inference rules are in fact theorems, and as such they are implemented in our working framework Isabelle/HOL. Hence, they are appropriate for deductive reasoning about potentially infinite structures and recursion.

4.2 Recursion induction

Functions recursing over inductive datatypes need not strictly follow the structural order of the datatype's recursion. The recursion pattern they follow might be more specific, or even less so, than the structure's definition's pattern.

For instance, take the functions on lists `sorted`, `init` and `last` for which we assume the relation less-or-equal on naturals `le` to be defined (see Section 5.1):

```
fun sorted :: "Nat List ⇒ Bool" where
  "sorted Nil = True"
| "sorted (Cons _ Nil) = True"
| "sorted (Cons r (Cons t ts)) =
      (if (le r t) then sorted (Cons t ts)
                   else False)"

fun init :: "'a List ⇒ 'a List" where
  "init Nil = Nil"
| "init (Cons _ Nil) = Nil"
| "init (Cons t ts) = Cons t (init ts)"

fun last :: "'a List ⇒ 'a" where
  "last (Cons t Nil) = t"
| "last (Cons _ ts) = last ts"
```

The recursion in the first two functions breaks down the cases into more detailed ones by considering certain amounts of elements in a list to define the action to take. From the function definition's recursive structure we can derive two alternative induction principles. Note how even though the recursion on the argument list is the same for either scheme, the `if` constructor in `sorted`'s definition leads to its guard condition to play a part in the scheme:

SortedInd
$$\frac{P(Nil) \qquad \forall u\ P(Cons\ u\ Nil) \qquad \forall t,\ r,\ ts\ ((le\ \ t\ r \implies P(Cons\ r\ ts)) \implies P(Cons\ t\ (Cons\ r\ ts)))}{\forall x\ P(x)}$$

InitInd
$$\frac{P(Nil) \qquad \forall u\ P(Cons\ u\ Nil) \qquad \forall r,\ t,\ ts\ (P(Cons\ r\ ts) \implies P(Cons\ t\ (Cons\ r\ ts)))}{\forall x\ P(x)}$$

If, alternatively, the predicate defining a list as sorted were to be expressed as the equivalent:

```
fun sorted_bis :: "Nat List ⇒ Bool" where
  "sorted_bis Nil = True"
| "sorted_bis (Cons _ Nil) = True"
| "sorted_bis (Cons r (Cons t ts)) = (le r t ∧ sorted_bis (Cons t ts))"
```

the corresponding recursion induction scheme would be the same as `init`'s, in accordance with their equal recursive structure on lists:

SortedBisInd

$$\frac{P(Nil) \qquad \forall u\ P(Cons\ u\ Nil) \qquad \forall r,\ t,\ ts\ (P(Cons\ r\ ts) \implies P(Cons\ t\ (Cons\ r\ ts)))}{\forall x\ P(x)}$$

Meanwhile, for `last` the case for an empty list is not taken into account. The partiality of this third example does not prevent it from defining an induction principle; however, for it to be an actual theorem which can be used as a proof principle, completion of the inference rule with missing cases is necessary for the universal statement over all lists to hold.

So, instead of having:

PartialLast

$$\frac{\forall u\ P(Cons\ u\ Nil) \qquad \forall r,\ t,\ ts\ (P(Cons\ r\ ts) \implies P(Cons\ t\ (Cons\ r\ ts)))}{\forall x\ P(x)}$$

one would take:

CompleteLast

$$\frac{\forall u\ P(Cons\ u\ Nil) \qquad \forall r,\ t,\ ts\ (P(Cons\ r\ ts) \implies P(Cons\ t\ (Cons\ r\ ts))) \qquad P(Nil)}{\forall x\ P(x)}$$

Hence, `last`'s induction scheme turns out to be equivalent to `init`'s too.

Whilst structural induction is appropriate to reason about cases in terms of constructors, it fails to be specific enough for other recursion patterns. Schemata arising from recursion patterns can isolate sub-units not represented in a datatype's structure as being atomic, such as lists with at least two elements in the previous examples.

Induction following these recursion schemata, called *recursion induction* or *computation induction*, allows for reasoning in the proof of a lemma to be driven and shaped by an arbitrary induction scheme derived from a recursive definition. In particular, Hipster explores the proofs one could construct by using recursion inductions defined by operations appearing in the statement to be proven.

This new approach also responds to the aim of exploiting Isabelle/HOL's existing features. In this case, that of deriving induction schemata from recursive functions, which Isabelle/HOL automatically does from the termination order of recursive functions extracting the schemata as theorems [29].

Recursion induction in a proof

To depict the potential difference between recursion and structural induction, let us take the already introduced lemma `sorted xs` $\Longrightarrow$ `sorted (insert x xs)`. Applying structural induction on the list `xs` would produce the subgoals:

1. `sorted Nil` $\Longrightarrow$ `sorted (insert x Nil)`
2. $\bigwedge$ `a y`. `(sorted y` $\Longrightarrow$ `sorted (insert x y))` $\Longrightarrow$ `sorted (Cons a y)` $\Longrightarrow$
   `sorted (insert x (Cons a y))`

Whilst `sorted`'s recursion induction scheme would yield:

1. `sorted Nil` $\Longrightarrow$ `sorted (insert x [])`
2. $\bigwedge$ `u`. `sorted (Cons u Nil)` $\Longrightarrow$ `sorted (insert x (Cons u Nil))`
3. $\bigwedge$ `r t ts`. `(sorted (Cons t ts)` $\Longrightarrow$ `sorted (insert x (Cons t ts)))` $\Longrightarrow$
   `sorted (Cons r (Cons t ts))` $\Longrightarrow$
   `sorted (insert x (Cons r (Cons t ts)))`

The latter set of subgoals leads to an immediate proof of the main lemma thanks to its steps mirroring the actual predicate definition, hence having a correspondence with its simplification rules. In contrast, the former, even though it intuitively looks immediate to prove, is not sufficiently generalised nor does it specify any intermediate result on inserting an element on a concrete non-empty list (in our case, the singleton list) which would enable to prove the second subgoal for any arbitrary list. Structural induction is in some way a weaker scheme and additional case-splits or lemmas would be required to close the proof.

4.3 Recursion induction for conditionals

4.3.1 Considering the structure of a program through data

Side-conditions one might encounter or require to handle in order to prove interesting lemmas can be of very different nature. Within the scope of inductive theories, these will very often either:

- state an explicit condition on the constructors of (typed) parameters, e.g. for a list, `xs` $\neq$ `Nil`
- employ a predicate defined in the current theory (potentially recursive, following some pattern on the datatype's structure), e.g. the predicated for sortedness, `sorted xs`

Naturally, these conditions might be applied to variables or to arbitrary expressions which could contain function applications, resulting in variables we can induct on being nested deeper in the condition or combined with other expressions.

Support for proving of conditional lemmas has been added by considering the structure of types and functions closer throughout a proof via the use of recursion induction. In such a way, the amount of known patterns and already given information affecting a proof is increased rather seamlessly.

In a given proof, following the induction schemata defined by functions occurring in the condition makes it possible to account for its effect on the structure of its parameters in a direct way. Yet this is not guaranteed to produce a complete proof. In consequence, recursion induction schemata derived from elements in the conclusion are also taken into consideration. This increases the scope of Hipster's original approach for equational lemmas too.

4.3.2 Instantiating recursion induction schemata

Applying recursion induction requires careful instantiation of the free variables present in a lemma on which to induct. Recursion schemata are typed in appropriate correspondence to the function they derive from and its type definition. So, for instance, having the already defined list concatenation `append` and a zipping function:

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a, 'b) list" where
  "zip Nil _ = Nil"
| "zip (Cons _ _) Nil = Nil"
| "zip (Cons z xs) (Cons t ys) = Cons (z, t) (zip xs ys)"
```

And the induction schemata for `append` and `zip` respectively:

AppendInd

$$\frac{\forall ts\ P(Nil,\ ts) \qquad \forall z,\ rs,\ ts\ (P(rs, ts) \implies P(Cons\ z\ rs,\ ts))}{\forall xs,\ ys\ P(xs,\ ys)}$$

ZipInd

$$\frac{\forall ts\ P(Nil,\ ts) \qquad \forall z,\ rs\ \ P(Cons\ z\ rs, Nil) \qquad \forall z,\ y,\ rs,\ ts\ \ (P(rs,\ ts) \implies P(Cons\ z\ rs,\ Cons\ y\ ts))}{\forall xs,\ ys\ \ P(xs,\ ys)}$$

Say we happen to have the term `zip (append x y) z` in some lemma, with free variables `x`, `y` and `z`, which without any further restriction have inferred types `x` :: `'a list`, `y` :: `'a list`, `z` :: `'b list`. We would hence only apply induction following `append`'s induction scheme on variables `x` and `y` (the derived induction principle will take predicates `P` :: `'a list ⇒ 'a list ⇒ bool`, of the same argument type as `append`). Whilst `zip`'s induction scheme could be applied to either the pair `(x, z)` or the pair `(y, z)`.

As seen in this very example, schemata also capture patterns combining several constructs and variables. When performing the corresponding recursion induction, the instantiation does not necessarily have to be complete and one may specify fewer variables than those present in the scheme to induct on. Whilst this might be appropriate at times, terms which are hard to prove and might require inductions of their own can be produced. Such is the case if, for instance, other free variables occur in terms involving the originator function of a scheme along with the variables instantiated for an induction. Hence, by default we consider as complete as possible instantiations first when exploring the proof for a given lemma; these may still be partial instantiations.

Not all free inductable variables present in a lemma are necessarily instantiated for the recursion induction. In such cases, the induction proof should hold for any arbitrary values of these and therefore we generalise over them

when setting up the recursion induction. The aim is to obtain new subgoals as general as possible where variables not inducted on are universally quantified. Otherwise, the inductive step of a proof may turn out to have a too specific and strong induction hypothesis condition.

Take for instance the following definition for the maximum between two natural numbers and its corresponding recursion induction principle:

```
fun max :: "Nat ⇒ Nat ⇒ Nat" where
  "max Z y = y"
| "max (S z) Z = S z"
| "max (S z) (S x) = S (max z x)"
```

MaxInd

$$\frac{\forall a\ P(Z,\ a) \qquad \forall z\ P(S\ z, Z) \qquad \forall a,\ b\ (P(a,\ b) \Longrightarrow\ P(S\ a,\ S\ b))}{\forall x,\ y\ P(x,\ y)}$$

Consider the proposition:

```
max (max a b) c = max a (max b c)
```

When said recursion induction is instantiated to the variables `a` and `b` without making `c` arbitrary, the inductive step of the proof results in the open subgoal:

```
/\ z x. max (max z x) c = max z (max x c) ⟹
        max (max (S z) (S x)) c = max (S z) (max (S x) c)
```

In contrast, if `c` is in fact set to be arbitrary, the inductive step translates to:

```
/\ z x c. (/\ c'. max (max z x) c' = max z (max x c')) ⟹
          max (max (S z) (S x)) c = max (S z) (max (S x) c)
```

A weaker, more general premise is favourable in finding a way to derive the final conclusion. Particularly, after induction Hipster employs a first-order solver (see Chapter 5); the more general the assumptions it can make the higher the chances of finding good instantiations and combinations of those premises to solve the open goal.

Recursion schemata due to function definitions include other non-inductive argument types, in addition to the pattern of recursion on the inductables. In the setting of Isabelle/HOL this means that higher-order variables could appear, as in:

```
fun map :: "('a ⇒ 'b) => 'a List ⇒ 'b List" where
  "map _ Nil = Nil"
| "map f (Cons t ts) = Cons (f t) (map f ts)"
```

MapInd

$$\frac{\forall f\ P(f,\ Nil) \qquad \forall f,\ t,\ ts\ (P(f, ts)\ \Longrightarrow\ P(f,\ Cons\ t\ ts))}{\forall g,\ ys\ P(g,\ ys)}$$

Despite not changing recursive call after recursive call with respect to its structure, the higher-order parameter, without which `map`'s induction principle

would simply correspond to structural induction, has to be treated specially. It should be fixed throughout the rest of the proof if `map`'s recursion induction is to be applied: we instantiate the inductive rule application with some variable or constant symbol of the appropriate higher-order type present in the proposition at hand. Otherwise, in later steps of a proof there exists the risk of entering a higher-order unification loop, since higher-order unification is not decidable.

Chapter 5

A tactic (not only) for conditionals

Hipster employs the Isabelle automated proof procedures *simp* (simplification), *metis* (resolution and superposition for first-order logic with equality [23]) and *induction*. From these, Hipster constructs its own generalised tactics. The specific new tactic (and component) in Hipster implements the choice and instantiation of recursion induction, followed by said simplification and first-order reasoning.

5.1 A single induction step

Given Hipster's intended use, it has been decided for the tactic to perform a single induction per individual proof attempt. Since the aim is for Hipster to enable gradual development in a theory, this is appropriate: lemmas requiring more than a single induction shall rely on smaller lemmas about other inductive results instead of relying on a larger proof with several induction steps.

Theory exploration is the key part to the system that makes this connection to inductive sub-lemmas possible, by discovering them instead of descending into a combinatorial problem of successive induction attempts.

Structural induction is also a potential induction scheme to follow. There are cases where following other schemata derived from functions appearing in a lemma might not set up the proof's branches in a solvable way, whilst structural induction will do. Say we define the relation less-or-equal for natural numbers as:

```
fun le :: "Nat ⇒ Nat ⇒ bool" where
  "le Z _ = True"
| "le _ Z = False"
| "le (S x) (S y) = le x y"
```

which has the associated recursion induction scheme

$$\textsc{LeInd}\ \frac{\forall n\ P(Z,\ n) \qquad \forall n\ P(S\ n,\ Z) \qquad \forall n,\ m\ (P(n,\ m) \implies P(S\ n,\ S\ m))}{\forall x,\ y\ P(x,\ y)}$$

When proving reflexivity for less-or-equal `le x x`, applying the predicate's recursion induction scheme leads to opening the subgoals:

1. `/\ y. le y y`
2. `/\ v. le Z Z`
3. `/\ x y. le x x ⟹ le (S x) (S x)`

Whilst a subset of these would prove the intended lemma, the new subgoal 1 restates it. Such a goal will not be discharged by first-order reasoning reasoning and the automated proof would fail. On the other hand, structural induction would yield subgoals easily discharged by simplification:

1. `le Z Z`
2. `/\ x. le x x ⟹ le (S x) (S x)`

If, however, the property was expressed in an alternative way with a conditional:

`y = x ⟹ le x y`

Structural induction would not suffice to prove the lemma whilst recursion induction due to the definition of `le` would in this case set up proof subgoals easily discharged using simplification.

5.2 Rewriting, simplification and first-order reasoning

After applying recursion induction, Isabelle's simplification tactic is applied to the current proof state, allowing it to simplify each open goal. It is then followed by the first-order reasoner *metis* which is applied as well once to each open goal. These tactics may be supplied with additional lemmas upon application.

A notable issue is that of termination: simplification, rewriting procedures and *metis* depend on the rules made available to them to terminate within a reasonable time. To save situations where these procedures execute for unpredictable amounts of time, a timeout is set to their application. Even though this still provides an effective tactic, there is a question as to what should define the interruption of a proof attempt: apart from execution time, the number of steps taken in a proof or the depth of proof search could also be considered. The magnitude of each limit would still be a question, and the feasibility of imposing limits to the number of steps and the depth of proof search has not been explored yet.

Rules derived from definitions (whether datatype or function definitions) are automatically added to the simplification set of rules in Isabelle. Therefore, by default, no additional lemmas are supplied to the simplifier (see discussion in Section 8.5). It is not always immediate to know whether a given rule in combination with the rules already considered in simplification will cause a

loop. A simple example lemma with such an effect is `le (S n) m` $\Longrightarrow$ `le n m = True`, where having a larger premise expression (with respect to term size) than conclusion could provoke simplification to not terminate (see §2.5.3 in [36]). This specific case would probably be straight-forward to detect with a simple algorithm to check term sizes, but the same might not hold for other conditional expressions.

In contrast, the new tactic feeds *metis* lemmas found via theory exploration and any additional ones explicitly supplied (for example, by the user during an interactive session). Any simplifications which could have been made if these lemmas had been given to the simplifier as well are normally identified by the first-order solver too. The problem of non-termination or very long execution is also present in this case, and the question of which combination of lemmas to use beyond the scope of this work (see discussion in Section 8.5).

5.3 Tactic options

To enable a more flexible tactic for interactive use, a series of setup options have been made available

- adjustable timeout for *metis*
- possibility of setting filtering functions for selection of theory explored lemmas in:
  - simplification: by default, the filter leaves out all explored lemmas
  - *metis*: by default, the filter picks all lemmas discovered by theory exploration and all lemmas explicitly stated in the tactic application
- *metis* has different options, but for the purpose of Hipster only the type encoding of the problem to solve has thus far had an effect; particularly, although disabled by default, it is possible to set the type encoding to *metis*' option *full_types* which makes the first-order proof search fully typed; this slows down *metis*, so the timeout is to be adjusted accordingly

In a first theory exploration attempt for given constructs in a theory, however, employing the default settings is recommendable since, for the most part of what has so far been tested, making adjustments to them has only been necessary on a few occasions.

Having the ability to control the sets of lemmas passed on to the simplifier and *metis* opens up the possibility to experiment with relevance filtering. Relevance filtering looks to reduce the search space for a resolution problem by using heuristics to determine which lemmas from a given set are unlikely to contribute in finding a proof for a theorem.

Chapter 6

Exploring a theory

During a theory exploration, an exchange between Isabelle/HOL definitions and Haskell code takes place via Isabelle's code exporting functionalities and HipSpec's output. Attempts to construct formal proofs in Isabelle/HOL for the conjectures rendered by co-joint work of HipSpec and QuickSpec then follow.

The aforementioned Hipster tactic options may be adjusted to set up the exploration and its proofs.

Even though the new default, complete tactic for induction performs simplification followed by *metis*, it is possible for a user to employ different tactics and easily construct new ones based on the now supported recursion induction tactic. Similarly, a tactic employing structural induction but no recursion induction is also available and can as well be combined with other kinds of reasoning posterior to the induction phase.

As introduced in Section 3.1, the same way induction tactics can be tailored, Hipster's exploration procedure can be adjusted via the tactics parameterising it. The user has the possibility to easily set the pair of *routine* reasoning, that which defines triviality for some setting, and *hard* reasoning tactics to their convenience.

For the present study, trivial reasoning was set to be non-inductive, concretely, the kind of reasoning applied after setting up the induction: simplification and first-order proving.

6.1 Exploration

Exploration can be triggered with one of two commands: `hipster` or `hipster_cond`. Both of them are to be provided with the names of the definitions one wishes to theory explore and extract properties about. When given several of them, they are explored jointly leading to possibly discovering relations between them and not only properties of individual operations.

To discover conditional lemmas, the latter command `hipster_cond` is to be used. Given the complexity of deciding or discovering useful predicates (i.e. predicates which in fact affect the way other operations behave when applied to

common variables), Hipster currently takes a user-supplied predicate which is to be provided as the first argument to the conditional exploration command.

For instance, we might want to explore the functions `le` and equality for naturals `eqN` jointly for conditional lemmas with `le` as the predicate to explore with. With `eqN` defined as:

```
fun eqN :: "Nat => Nat => Nat" where
  "eqN Z Z = True"
| "eqN Z _ = False"
| "eqN _ Z = False"
| "eqN (S x) (S y) = eqN x y"
```

Hipster would be invoked providing in first place the predicate to employ as a condition, followed by the rest of functions to explore:

```
hipster_cond le eqN
```

This would yield a series of lemmas not present yet in the theory, from those involving either of the functions to those involving both, and both with and without conditionals. Some example lemmas for this exploration are:

```
lemma lemma_a [thy_expl]: "eqN x y = eqN y x"
by (hipster_induct_schemes le.simps eqN.simps Nat.exhaust)

lemma lemma_ac [thy_expl]: "eqN x Z = le x Z"
by (hipster_induct_schemes le.simps eqN.simps Nat.exhaust)

lemma lemma_ag [thy_expl]: "le x y ⟹ eqN x y = le y x"
by (hipster_induct_schemes le.simps eqN.simps Nat.exhaust)

lemma lemma_ai [thy_expl]: "le z y ∧ le x z ⟹ le x y = True"
by (hipster_induct_schemes le.simps eqN.simps Nat.exhaust)
```

The application of `hipster_induct_schemes` is a call to Hipster's new recursion induction tactic. Theory exploration provides the proving tactics with two kinds of rules: `x.simps`, the simplification rules for some definition `x`, and `T.exhaust`, associated to a datatype `T` and given as case distinction rules.

Limitations from the underlying QuickSpec and HipSpec shape what is feasible to explore. QuickSpec currently only supports unary and binary predicates as conditions, and having a single predicate per exploration. The number of atomic predicate applications allowed in the premise of discovered lemmas can be set freely in Hipster. Nonetheless, due to efficiency constraints and the openness of the question of constrained data generation, as assessed in [11], allowing either one or two atomic predicates in lemmas' side-conditions is what we consider in this work and recommend for the use of Hipster as of now.

6.2 Proof loop

Once conjectures are collected from HipSpec's output, the next step is to prove as many as possible and filter out trivial properties. Just as the original Hipster

did, the order of conjectures generated by HipSpec is kept the same during this process. One of the important properties of this order is that of placing the most general conjectures first, listing specific instances later. For instance, for natural numbers and addition between them, `(x + y) + z = x + (y + z)` would be listed before `(x + y) + y = x + (y + y)`, where the free variable `y` substitutes for `z`. By doing this, when a more general proposition is provable without considering specific instances of it, the following specific instantiations become trivially provable avoiding bringing redundant and trivial properties into a theory development. It also avoids performing redundant proofs for properties easily derivable from others.

For each examined proposition appropriate simplification rules for datatypes derived from the theory's definitions may be needed for the first-order reasoning we apply, or any other reasoning for that matter. But, more importantly for theory exploration's purpose, previously discovered theorems are made available to automated tactics applied after the induction. In Section 6.1, the label `thy_expl` in the declaration of each lemma defines each corresponding lemma as belonging to the set of theory explored lemmas, allowing Hipster's tactics to look them up.

Options provided for the tactic will affect the proof loop, given the existence of timeouts and theorem filtering. Consequently, some failed proof attempts may still succeed (with the given tactic) after some adjustment of options.

Chapter 7

Evaluation

Evaluating automated tools for interactive theorem proving necessarily has to consider some degree of interaction. Their performance with respect to their intended use is otherwise hard to analyse. Scope of proving capability has been the focus, an analysis which had not been performed for Hipster to the same extent prior to this work.

Two forms of evaluation were taken:

- Case studies on algebraic data types and operations on them. In particular, focusing on inductive theories for:
  - natural numbers: with basic arithmetic operations and comparison
  - lists: with basic (polymorphic) functions (`len`, `last`, `take`, `zip`, `count`, etc.)
  - sorting: proving insertion sort's correctness on lists.
- Evaluation on a subset of TIP (Tons of Inductive Problems [14]), a new set of benchmarks and challenge problems for inductive theorem provers, recently made available.

7.1 Interactive case

As introduced already, automating the proof of correctness for insertion sort was a motivating case. To showcase Hipster's handling of conditional lemmas and their necessity, we present its theory development here[1].

Assuming the introduced datatype definitions for lists and natural numbers and the less-or-equal operator `le` (and no prior, additional lemmas), based on the function definitions:

```
fun sorted :: "Nat List ⇒ Bool" where
  "sorted Nil = True"
```

[1]Source code for Hipster and examples presented are available online: `https://github.com/moajohansson/IsaHipster`

```
| "sorted (Cons _ Nil) = True"
| "sorted (Cons r (Cons t ts)) = (le r t ∧ sorted (Cons t ts))"

fun insert :: "Nat ⇒ Nat List ⇒ Nat List" where
  "insert r Nil = Cons r Nil"
| "insert r (Cons t ts) =
    (if (le r t) then Cons r (Cons t ts)
                 else Cons t (insert r ts))"

fun isort :: "Nat List ⇒ Nat List" where
  "isort Nil = Nil"
| "isort (Cons t ts) = insert t (isort ts)"
```

The theorem corresponding to the algorithm's correctness is:

```
theorem isortSorts: "sorted (isort ts)"
```

Running exploration directed to solve the theorem alone will not find appropriate sub-lemmas that would together immediately prove it. Since we have two different predicates used in the definition of terms appearing in the main goal theorem, `le` and `sorted`, apart from exploring the theory for equational lemmas we should consider running an exploration to discover conditional ones.

Starting from the components, exploration for conditional lemmas about `le` would be invoked with the command:

```
hipster_cond le
```

It yields 7 lemmas, all proven by Hipster's tactic, of which 4 are conditionals. These four lemmas already require recursion induction in their proofs and are:

```
lemma lemma_ac [thy_expl]: "le x y ⟹ le x (S y) = True"
by (hipster_induct_schemes le.simps Nat.exhaust)

lemma lemma_ad [thy_expl]: "le y x ⟹ le (S x) y = False"
by (hipster_induct_schemes le.simps Nat.exhaust)

lemma lemma_ae [thy_expl]: "le y x ∧ le x y ⟹ x = y"
by (hipster_induct_schemes le.simps Nat.exhaust)

lemma lemma_af [thy_expl]: "le z y ∧ le x z ⟹ le x y = True"
by (hipster_induct_schemes le.simps Nat.exhaust)
```

Concretely, the critical lemmas for the rest of the proof are the first two. The latter two correspond to reflexivity and transitivity for less-or-equal, lemmas whose proofs were not automatic with Hipster before.

QuickSpec does not produce conjectures with inequalities nor negations, thus it can be useful to explicitly explore the negation of predicates in a theory. In the case for `le` we get new lemmas:

```
lemma lemma_ah [thy_expl]: "le (S x) y = (¬ le y x)"
by (hipster_induct_schemes le.simps Nat.exhaust)
```

```
lemma lemma_ai [thy_expl]: "(¬ le x y) ⟹ le x Z = False"
by (hipster_induct_schemes le.simps Nat.exhaust)

lemma lemma_aj [thy_expl]: "(¬ le y z) ∧ (¬ le x y) ⟹ le x (S Z) = False"
by (hipster_induct_schemes le.simps Nat.exhaust)
```

Next, one would think of running exploration on the functions isort, insert and sorted, taking sorted as a possible side-condition:

```
hipster_cond sorted isort insert
```

However, none of the most general and interesting lemmas discovered by said exploration, such as isort (isort x) = isort x turn out to be provable yet. On occasions, non-fully typed searches for first-order proofs will not succeed, despite all necessary helping lemmas being available. Adjusting the system's options to have *metis*' search use full types gives the proof for a vital discovered lemma, the invariant guaranteeing correctness to insertion sort:

```
lemma isortInvariant [thy_expl]: "sorted ys ⟹ sorted (insert x ys) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
```

During this last exploration, our main theorem is in fact discovered along with other interesting lemmas, all of which can be now proven automatically by using the sub-lemma isortInvariant:

```
theorem isortSorts [thy_expl]: "sorted (isort x) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)

lemma isortFixes [thy_expl]: "sorted x ⟹ isort x = x"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)

lemma insertComm [thy_expl]: "insert x (insert y z) = insert y (insert x z)"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
```

A single invocation to the recursion induction tactic manages to prove these statements, simplifying the interaction with the proof assistant. The recursion induction approach therefore proves to be of good aid in automating proofs. The lemma isortInvariant is crucial in the previous proof, highlighting once again the need for support of conditional lemmas in automated inductive proving. In its case, it is the induction scheme arising from the recursion in sorted that allows to complete the proof.

Lastly, the benefit of an interactive environment is being able to parameterise the proving methods with adequate options as one works along. This supervised automation can be of great utility to a user during a formal theory development.

As a side note, if we were to use Isabelle/HOL's pre-defined types for natural numbers and lists, the exploration would have been different. In such a case, we would have been able to count on the notion of order defined for naturals, and exploring the lemmas about less-or-equal would not have been necessary. Conditional exploration on sorted along with isort and insert would have sufficed to obtain the missing auxiliary lemma isortInvariant and complete the theorem's proof. Enabling full typed search for *metis* would not have been necessary either.

7.2 Tons of Inductive Problems (TIP) benchmarks

From TIP, we evaluate Hipster's proving ability over two subsets of problems employed in previous works on inductive theorem proving: Johansson, Dixon and Bundy's work on case-analysis for rippling [25] (we denote it *case-analysis*), and prior work by Ireland and Bundy on employing proof failure to guide lemma discovery and patch inductive proofs [24] (we denote it *prod-failure*).

7.2.1 Method

To evaluate performance on TIP problems, we consider a set up where each problem is considered in isolation. Only the theory definitions required to express it are given and no auxiliary lemmas are provided to the prover, just those discovered for the isolated problem.

For each problem, theory explorations were first run on individual functions appearing in its definition, jointly with their auxiliary functions. After, if the goal theorem had not been proven yet, exploration was run on groups of functions co-occurring in the problem's formulation. Not all problems required an exhaustive exploration about all functions appearing in them, so more functions were added to explorations as required. For instance, some problems were immediately solvable by Hipster's new tactic with no prior exploration, whilst others only needed exploration for a predicate involved in a conditional.

As explained earlier, conditional lemma discovery is limited to explore a single predicate to define the premise at a time. For the present set of problems this has sufficed.

Additionally, to test Hipster's capacity when working on strictly newly defined theories and datatypes, no assumptions nor properties from theories in Isabelle/HOL were considered during proof search. As an example, natural numbers are not Isabelle/HOL's already available ones, but redefined in cases they are used. Hence, predefined notions of orderings and other properties do not play a part in proofs obscuring the results of Hipster's actual work. In this way, we only consider as the base starting point a set of definitional statements, aligning with the purpose of proving based on structure and construction of programs.

Examples of theorems found in the evaluation set are:

```
le (count n xs) (count n (append xs ys))

filter p (append xs ys) = append (filter p xs) (filter p ys)

zip (append xs ys) zs
  = append (zip xs (take (length xs) zs)) (zip ys (drop (length xs) zs))

rotate (length x) x = x
```

7.2.2 Results

From the benchmarks, the following statistics, which we later compare with other provers' (Chapter 9), were gathered:

	Case-analysis		Prod-failure	
	EQ	COND	EQ	COND
Total number of benchmarks	71	15	38	12
Fully automated	67	14	29	12
Partially automated	4	0	6	0
Prior exploration	33	4	34	4
Conditional exploration	6	2	2	3
Recursion induction	29	6	1	1
Recursion induction on sub-lemmas	16	1	5	0
Filter theorems for FO prover	3	0	0	0

Table 7.1: Total number of problems evaluated from the benchmarks (all problems from *case-analysis* and *prod-failure*) and Hipster's performance on them. Columns *EQ* and *COND* collect results for equational and conditional lemmas respectively.

The resulting theory files to the tests are available online for both the *case-analysis* and *prod-failure* sets[2].

Fully automated Full automation refers to solving a problem entirely only with lemmas discovered by Hipster's theory exploration and Hipster's automated recursion induction tactic. Overall, the rate of fully automated provability on the benchmark set is observed to be high: 90%. For conditional lemmas in the test set this rate is higher, 96%.

Partially automated A number of theorems (problems 52, 53, 72, 74 from *case-analysis*; and 2, 4, 5, 20, 22, 23 from *prod-failure*) required one of the following two similar lemmas, not automatically proven in a first instance (neither by structural nor recursion induction):

```
length (append x y) = length (append y x)
count z (append x y) = count z (append y x)
```

These two lemmas are discovered. In both cases, a single additional lemma which is not discovered would allow to complete the proof. Specifically, the respective lemmas were:

```
length (append xs (Cons y ys)) = S (length (append xs ys))
count z (append xs (Cons z ys)) = S (count z (append xs ys))
```

For efficiency reasons, lemmas are only constructed with terms up to a certain depth, 3, during theory exploration in Hipster using the underlying HipSpec/QuickSpec, whilst the required lemma has term depth 4. With depth limit

[2] `https://github.com/moajohansson/IsaHipster`

4 QuickSpec can generate such a conjecture. These auxiliary lemmas are proven by Hipster's tactic and once they are added to the affected problem theories, the remaining proofs are all automated by Hipster.

Examples of these partially automated theorems are:

```
rev (drop i xs) = take (minus (length xs) i) (rev xs)
half (length (append x y)) = half (length (append y x))
count n xs = count n (rev xs)
```

If we include the function for addition on natural numbers, however, enough lemmas are automatically discovered to prove `length (append x y) = length (append y x)` and `count z (append x y) = count z (append y x)` without needing any other intervention (only including addition in the theory exploration for each of these, few, theorems; see Section 8.2 for details).

Theory exploration Over half of the problems required prior lemma discovery. This indicates the benefit of theory exploration since otherwise their provability in this context would not have been feasible. Equational lemmas were much more likely than conditionals to need exploration. Equations are unrestricted and can be more general, potentially requiring more steps to assess their different case branches. However, the number of conditional problems in these benchmarks is low, and one cannot draw conclusions as to whether exploration becomes less necessary, or not, when a statement to be proven is constrained by a side-condition.

A smaller subset of problems were provable with the aid of conditional exploration, namely those involving functions defined in terms of some predicate.

Recursion induction Whereas recursion induction might not have been necessary as often as theory exploration (whether for the main theorem or any auxiliary lemmas), its impact is still notable.

Overall, there seems to be a trade-off between using weaker induction schemes (structural induction) and reducing the number and complexity of needed auxiliary lemmas. Structural induction was always attempted first by the tactic, meaning theorems solved via recursion induction (around a third of the benchmarks) would have not been solved otherwise, at least not with the degree of exploration carried out.

The results suggest recursion induction can save on exploration time, since it provides appropriate induction patterns that avoid the need for sub-lemmas about specific constructor combinations.

Tactic options In a few cases, the default tactic options were not adequate for a proof to be constructed.

Firstly, when theories are explored not all discovered lemmas will be meaningful or useful to solve the main theorem at hand. Nonetheless, in our automated setting these are also imported into the theory, forming part of the set of theorems employed by the tactic during first-order reasoning. With an increasing number of lemmas to consider, automated tactics such as *metis* slow

down and might not find a proof before the set timeout (our interest is to find proofs relatively fast). In a regular session this would not be a particular issue since one may interactively remove from the set of theory explored lemmas those which are not relevant for a specific theorem.

This portrays the relevance of supporting lemma filtering for the different building blocks of a general tactic, which we discuss later (Section 8.5). Such an option in Hipster is new and still primitive: one can filter lemmas in terms of what the proposition to prove at hand is, although a study on possible filtering approaches was outside the aim of this work. Nonetheless, a user could freely define such filter functions for Hipster, a useful feature we intend to develop further.

In the 3 cases encountered (problems 22, 31 and 73 from *case-analysis*), an existing fallback solution to *metis'* application in which no discovered lemmas were provided and with a higher timeout was enough. These 3 problems required recursion induction in their proofs too.

Secondly, a couple of cases involving lemmas on sorting required to set *metis* to use fully typed proof search (problems 77 and 78 from *case-analysis*), which correspond to theorems related to the proof of insertion sort's correctness (as seen as well in the case study in Section 7.1).

Technical limitations The current underlying implementation for theory exploration has some inefficiencies in the random term generation performed by QuickSpec. In Hipster, random datatype generators for the Haskell definitions exported from Isabelle are derived automatically and hence are bound to be more inefficient than a manually defined one. For values or functions with rapid growth, their performance scales badly.

This led to three specific problems in *prod-failure* to not be feasible to explore during evaluation and hence fail:

```
fac x = qfac x one
mult x y = mult2 x y Z
exp x y = qexp x y one
```

In these lemmas, `fac` and `exp` are typical definitions of the factorial and exponentiation operators, whilst `qfac` and `qexp` are their corresponding tail-recursive versions. Similarly with `mult` and `mult2` for multiplication, where the latter has a third accumulator parameter.

The commutativity and associativity of addition and multiplication are part of what caused a very high number of terms to be generated for exhaustive evaluation in QuickSpec, as well as combinatorial issues coming from a three argument function. Work towards a second version of QuickSpec, however, has increased efficiency to a great extent and future integration with the new version will bring those benefits to Hipster too.

Chapter 8

Discussion and further work

Results gathered during evaluation and observed during individual theory development show recursion induction allows for better automation in the search for inductive proofs. They also point to new directions in which to advance automated inductive proving.

8.1 Gains from recursion induction

Overall, recursion induction improves automation of induction by tailoring the scheme of induction used directly to the terms appearing in a proposition. In our benchmark testing, around a third of the studied problems required recursion induction in their proofs.

Even though one of the original thoughts was to allow conditions on lemmas to give structure to a proof's induction cases, during evaluation it was observed to be most often the case for functions in the conclusion statement to drive the proof with their definition structure. Such is the case of (having `lt` as the less-than operator on natural numbers):

`lt n (length xs)` $\Longrightarrow$ `last (drop n xs) = last xs`

for which the recursion scheme of `last` led to a complete proof. For conditional theorems like this one, where the outermost operator applied in the conclusion recurses over a different structure or argument than the outermost operator of the premise, recursion induction deriving from the conclusion's structure was observed to be the successful scheme to follow.

The current tactic does not employ a specific heuristic in choosing a recursion induction scheme. Studying possible heuristics to employ, especially for theorems with much larger terms, is future work and this last observation a starting point to it. Whether an outermost function should be considered before inner ones is also a factor that the present work has not studied but could influence speed in finding the appropriate induction scheme.

The experiments highlight the usefulness of recursion induction in general, not just for conditional theorems. A notable gain, affecting both conditional and

equational lemmas, is that of having the capability of performing simultaneous induction, whereas previously only structural inductions on a single variable were performed by Hipster. Simultaneous induction schemata are those inducting over more than one variable at a time, whether those variables are of the same type or not. Such is the case of the scheme derived from the definition of the list function `zip`, which corresponds to the special pattern of parallel induction on 2 lists:

ZipInd

$$\frac{\begin{array}{c}\forall ts\ P(Nil,\ ts) \qquad \forall z,\ rs\ \ P(Cons\ z\ rs, Nil) \\ \forall z,\ y,\ rs,\ ts\ \ (P(rs,\ ts) \implies P(Cons\ z\ rs,\ Cons\ y\ ts))\end{array}}{\forall xs,\ ys\ \ P(xs,\ ys)}$$

With that possibility, theorems like:

```
zip (append xs ys) zs =
  append (zip xs (take (length xs) zs)) (zip ys (drop (length xs) zs))
```

are provable, after some initial exploration of other functions involved. And for the following alternative related conditional theorem, the proof becomes direct and needs no prior theory exploration:

```
length a = length b ⟹
  append (zip a b) (zip c d) = zip (append a c) (append b d)
```

An additional gain is therefore that of more concise proofs where less logical steps are required.

Neither of these lemmas was provable before, even having done exploration for all the occurring functions in them. Hipster's prior structural induction approach could not capture in a scheme the relation between two variables. In these two cases, `zip` traverses its arguments taking steps on both at the same time, a pattern we can only capture with some form of simultaneous induction. Instead of synthesising a series of possible simultaneous induction schemata, recursion induction gives us an immediate choice which is also closer to the problem at hand.

8.1.1 Other patterns

The work so far has focused on regular induction patterns, but still indicates that more complicated kinds of inductive proving could gain from the use of recursion induction.

Mutual induction and co-induction proofs could gain in the same way from the use of recursion induction. Currently, Hipster does not handle datatypes or functions with these recursion patterns in any particular way and will hence fail at proving properties involving such definitions. Nonetheless, its current tactic structure would allow for a straight-forward first implementation to assess the possible benefit of using recursion induction for mutually recursive and co-inductive patterns.

8.1.2 Creating schemata

It would be a possibility to have Hipster generate arbitrary induction schemata, depending on the variables it encounters in a theorem to prove and the structural induction schemata over their types. This would make the choice for schemata broader and increase likelihood of encountering a successful scheme. Of course, one would face the challenge of implementing an automated proof for arbitrary, on-demand constructed induction rules, since induction schemata ought to be proven theorems in Isabelle in order to be applied. Alternatively, induction schemata proven by the user could be taken by Hipster as a parameter.

It is uncertain, though, how much the automated proof could benefit from this. Right now, by using induction patterns derived from symbols involved in a pending goal, we both reduce the search space for schemata and employ schemata adapted to the relations function application creates between variables in the goal theorem.

8.2 Relating functions

For the set of problems evaluated, to complete the proof of 10 theorems one of two specific lemmas, which were not proven automatically despite being discovered, was key, as indicated in Table 7.1 and further explained in Subsection 7.2.2:

```
length (append x y) = length (append y x)
count z (append x y) = count z (append y x)
```

When, however, explored jointly with addition for natural numbers `plus`, other discovered terms associating `plus` to `append` and `count` respectively, and addition's own properties, make these two lemmas provable. Concretely, the following are crucial:

```
length (append x y) = plus (length x) (length y)
count z (append x y) = plus (count z x) (count z y)
plus x y = plus y x
```

The target benchmark problems were defined in terms of `length` or `count`, `append` and other functions depending on `append` (such as `rev`), but none of them depended on the definition for addition on natural numbers. Hence, theory exploration does not include addition to the exploration automatically.

In an interactive setting, a user would of course be able to see a possible pattern with functions portraying similar relations to each other and interactively invoke an appropriate theory exploration.

One can observe that both `append` and `plus` cover their respective 2 arguments linearly with the same pattern. For both cases, the outermost function applied, `length` and `count` respectively, relates the two argument datatypes linearly too. Furthermore, these will in fact act as relating functions between `append` and `plus`. Adding `plus` interactively, Hipster discovers and proves automatically the stated crucial lemmas. Recursion induction schemata and their

comparison could be a way for automating the association of functions on different datatypes with each other and with a relating function, potentially helping in the decision of which independent definitions to theory explore jointly.

These examples seem to indicate that recursion induction may not suffice when a non-commutative operation nested within another has commuting arguments at each side of an equality. At least not in the absence of smaller related lemmas corresponding to subgoals. This seems reasonable: the structure of the terms at each side of the equality will differ upon induction.

8.3 Depth-limited discovery

An unavoidable problem for lemma discovery is that of needing to set a limitation to the terms generated.

For larger discovered expressions, other seemingly more complicated (with more symbols) but more specific expressions might be useful as sub-lemmas towards their proof. Currently, the depth of terms' expression trees considered during theory exploration via HipSpec and QuickSpec is bounded for efficiency reasons.

Taking the previous example, this leads to an expression like the following to not be discovered:

```
length (append ys (Cons x xs)) = S (length (append ys xs))
```

This conjecture can be proven automatically by Hipster's tactic, and can be used as a sub-lemma to prove one of the main propositions referred to in the previous section (Section 8.2). The depth of each of the expressions at either side of the equality, however, is 4 whereas expression depth limit is at 3.

We already spoke then about finding relator functions being a question of interest. For some cases, though, a different discovery process could enable the same end results whilst being a more direct approach.

For instance, assuming a depth limit of n, a heuristic for relevant term generation would be to allow tree expression depths of $n + 1$ for expressions with a type constructor as one of its nodes. Potentially, this would yield non-trivial variations on discovered lemmas, like the one above, that can break down reasoning further into steps a single induction could handle.

For the most part, these issues have been handled in a newer version of the tool behind conjecture generation: QuickSpec 2, soon to be released. Schemata are employed in term synthesis meaning much greater depths can be reached in less time, as they present in a recent technical report[1]. Hipster will undoubtedly benefit from this once HipSpec is also fully integrated with QuickSpec 2.

[1] http://www.cse.chalmers.se/~nicsma/papers/quickspec2.pdf

8.4 Chaining inductions

As presented, Hipster's approach is to apply a single induction at a time. Hipster could be extended further with the possibility of chaining inductions in an automated way. Limits to the number of consecutive inductions to perform can be naturally imposed by the number of existing, distinct free variables in a proposition allowing for a single instantiation of each variable in all the successive recursion inductions performed. However, this still eventually leads to a combinatorial problem, especially so when considering different recursion induction schemata as potential induction methods.

A different approach is HipSpec's: all inductions are structural, but it may apply structural induction simultaneously over more than one variable. The resulting induction schemata have a structuring effect similar to that of consecutive structural inductions although they differ in an important aspect: their inductive step. This approach yields very good results on the subset of TIP tested and presented here as well.

Being it that recursion induction can consider several variables simultaneously, though, Hipster's new tactic can make up for some of the induction chaining. For instance, for the theorem:

```
take n (drop m xs) = drop m (take (plus n m) xs)
```

Hipster can prove this theorem after simple theory exploration by `take`'s recursion induction scheme, instantiating it with `m` and `xs`, whilst, comparatively, HipSpec employs simultaneous structural induction on `n`, `m` and finally `xs` (forming a more complex induction scheme) after a corresponding theory exploration. It must be noted in this case, for the comparison to be meaningful, that Hipster requires fewer auxiliary lemmas discovered for said problem; in fact, for Hipster it suffices with individual exploration of the functions appearing whilst reported HipSpec results show auxiliary lemmas combining different functions being discovered and proven too towards the proof of the problem.

Worthy of noting is also the fact that recursion induction need not be necessarily replaceable by chained inductions, as for instance, the already presented simultaneous induction due to `zip` breaks down arguments in parallel not in sequence.

Aside from these aspects, as a tool meant for an interactive system, it is preferable for Hipster's tactic's work to not perform more than one *unit* of reasoning per application. Its current application involves several sub-units making it somewhat complex, but only up to a point where it can define whether a specific recursion induction would be successful. Any further inductions required should be solved in sub-lemmas so as to minimise complexity of automated proofs.

8.5 Exploration theorem set

As already touched upon, which theory explored lemmas should further be employed, when applying first-order reasoning or other methods, following the recursion induction instantiation is an important question to further study.

Adding too many or irrelevant ones slows down the automated proof search. This search not only depends on the goal theorem at hand in each proof, but also on which other theorems are in the exploration theorem set.

Hipster currently leaves this mostly up to the user to define and modify via simple options if wanted. Work on Sledgehammer's development led to the design of relevance filtering methods [32], attending in particular at those generated by means of automated, mechanical procedures. These methods use machine learning to automatically select seemingly relevant lemmas for a given goal. Being it that Hipster generates lemmas mechanically, and with a growing set of discovered lemmas during a theory development, incorporating those methods would not only speed up the tactic's performance, but also lead to a higher success rate. Additionally, the proof loop during exploration in itself would also benefit from it.

In contrast to the previous Hipster induction tactic, which also provided simplification procedures with the theory explored lemmas, the new tactic does not do so. By providing the first-order prover with the function simplification rules and theory explored lemmas, the same sub-goals can be proven. The difference is we do not risk non-termination of the simplifier because of unsafe discovered lemmas for rewriting.

8.6 Structured proofs

Proofs delivered by Hipster are applications of the implemented tactic.

When working with Isabelle and within its community, especially with the introduction of the Isar language, there is an emphasis on the benefits of producing structured proofs [51]. Unlike a tactic application style, Isar aims to enable writing human-readable mechanised proofs, minimising the explicit use of proof-search based methods in final proof documents. Apart from improving readability and logic understanding from the reader's perspective, this approach produces easier to maintain proofs.

In Hipster's current infrastructure, extracting the specific recursion induction and instantiation which was successful is feasible and straight-forward. A future aim is for Hipster's tactic not only to prove but also to translate and export its reasoning procedure as a structured proof for the user to use in the final proof document, following in the style for structured induction presented in [49] and aligning with the general aim to have structured proofs as a standard, serving as better documentation of what steps the tactic is actually taking too. This would have the added benefit of a more streamlined replaying of already found proofs, without requiring a new proof search.

In this line, exploiting other Isabelle tools' possibilities would help in better proof construction. Concretely, and again, Sledgehammer would help in minimising the amount of lemmas employed during first-order reasoning in these structured proofs.

Chapter 9

Related work

Induction schemata and procedures vary from automated reasoning system to system, which we briefly review and compare here.

Whilst Hipster interfaces with HipSpec for theory exploration, none of the proving capabilities of Hipster depend on HipSpec. HipSpec instantiates induction schemata based on structural induction, with the possibility of it being simultaneous, and then employs first-order provers to complete the proofs, whilst Hipster constructs a proof within Isabelle/HOL.

SMT solvers have started, as of recent, incorporating induction proving capabilities. These tools follow a top-down approach in conjecture generation, most commonly via strengthening of the original conjecture or subgoals arising in a proof. Such is the case for CVC4 and Pirate, already introduced in Section 2.5. Pirate's induction application is guided by a heuristic based on superposition, and it performs conjecture strengthening via generalisation of sub-terms in expressions. A particularity of Pirate is it employs strong structural induction.

Of relevance are also Isabelle/HOL's other two systems for induction and inductive theory lemma generation: IsaCoSy [26] and IsaScheme [34]. In contrast to Hipster, IsaCoSy, as referenced earlier, synthesises irreducible terms as conjectures, filtering out by counter-example checking those which are refutable, and then employs IsaPlanner's induction techniques to prove remaining ones [18]. Not only is lemma generation, and hence theory exploration, different to Hipster's, but also the approach in induction proving: IsaPlanner employs proof planning critics based on rippling. IsaScheme tackles lemma generation by means of using schemata, which are automatically instantiated but are to be provided by the user. A comparison in terms of conjecture generation between HipSpec/Hipster and these two systems shows the lemmas found are similar for all three [13, 27].

In comparison to other (automated) inductive provers, the new Hipster is the only one (to the best of my knowledge) to employ recursion induction. Additionally, other theory exploration systems remain as automated tools whilst Hipster aims to be an interactive tool to enable automation of certain proving activities that is still flexible enough for tailored usage, allowing to implement new proving tactics to apply during exploration and permitting adjustment

of the existing ones. Other exploration systems do not support discovery of conditional lemmas either, although Pirate, not a theory exploration system, does have support for conditional theorems as well.

The test suites Hipster has been evaluated on serve as a good point of comparison with other inductive provers. The following table shows the number of problems solved by some of the introduced provers, summarising the results published in the already mentioned [47, 42, 13], and including those of Zeno's performance, a tool for proving equational inductive properties of Haskell programs [45], the ACL2 Sedan prover [10], the Dafny system [30] and the proof-critics based CLAM prover [24] (from which *prod-failure* benchmarks are extracted).

	Case-analysis			Prod-failure	
Hipster	80	(84)	[85]	41	(47)
CLAM	-			41	
HipSpec	80			44	(47)
Zeno	82			21	
ACL2 Sedan	74			-	
IsaPlanner	47			-	
Dafny	45			-	
CVC4	80			40	
Pirate	85		[86]		(47)

Figures in parenthesis indicate number of successful proofs after some adaptation of settings. Additionally, a last theorem not present in all benchmarks was brought up in [47] (not yet published) as missed by some of the cited work: `lt x y` $\Longrightarrow$ `elem x (ins y xs)` $\longleftrightarrow$ `elem x xs`; in square brackets we state the results that are known when such a theorem is added. Note that the only work including this problem is the cited Pirate and our own.

It is of interest to note that from the test suite we denote *prod-failure*, of a total of 50 problems the cases where Hipster fails (problems 33-35, spoken of in Subsection 7.2.2) are different to HipSpec's and Pirate's, with the exception of 33 in Pirate's case. and due to issues in the theory exploration. It so is the case as well that HipSpec and Pirate adapt their settings for those problems in the cited results. Particularly, HipSpec employed adjusted settings in these three cases, due to memory usage in QuickSpec, to be able to discover appropriate helping lemmas. Hipster has not been evaluated with adjusted settings at the HipSpec/QuickSpec level and hence the exploration phase was not feasible to perform for them. With similar settings to HipSpec's, there is a probability problems 33-35 would be solvable in Hipster too.

With respect to HipSpec, with which exploration results are shared, the failed problems for the first test suite (*case-analysis*, with a total of 85 problems or 86 if the stated missing problem is considered) are different too, for the exception of problem 85 which so far only Pirate has shown to be able to prove. This indicates recursion induction can play an important role in inductive proving.

Hipster performs on par to other state of the art automated inductive theorem provers even if its intended use is as an interactive tool, a motivating factor towards developing further the use of recursion induction.

Chapter 10

Conclusion

We had set out to add support for conditional lemma discovery and proof in Hipster, an interactive theory exploration tool for Isabelle/HOL, with a focus on inductive theories. The main contribution has been two-fold:

- a (novel) approach to inductive proof automation through application of recursion induction (with automated selection of schemata), making the most of the environment for our interactive tool, Isabelle, and producing certified proofs
- an extension of Hipster's equational lemma handling with support for conditional lemma exploration and proof in Hipster, necessary in reasoning about (recursive) programs and algorithm correctness.

In addition, we provide an evaluation of Hipster's new capabilities on a set of benchmarks recently presented and used in the literature of automated inductive proving, as well as demonstrate its usefulness in an interactive setting. The benchmark evaluation not only serves as evidence of our system's potential and capabilities, not shown previously to the same extent, but also sets it to be on par with the latest and very recent work on automated inductive proving whilst employing a kind of induction not applied by others. These benchmarks also reflect Hipster's gained ability to discover and prove conditional, inductive lemmas.

Recursion induction gives Hipster a possibility it did not have before: following simultaneous induction schemata. This has strengthened Hipster's ability to prove equational lemmas as well. Furthermore, after this experience with recursion induction, support for other sorts of induction such as mutual induction and co-induction stand as immediate extensions to implement for Hipster.

An important current limitation is efficiency in relation to the depth of expressions explored. However, the new QuickSpec 2 (to be released) is showing a great improvement in this respect and with its future integration with HipSpec/Hipster some of the barriers thus encountered will be overcome.

We further provide new parameters for the system that flexibilise it further, for the user to tailor it to their needs and specific problem setting, and

leave an open door to future work. New avenues include the development and application of heuristics and relevance filtering (of lemmas) techniques to further advance proof automation. Hipster could provide a good infrastructure for experimentation in these topics.

Overall, results suggest that theory exploration and recursion induction can be employed successfully in combination towards automated inductive proving, not only for conditional lemmas, but in general. Most importantly, they reaffirm Hipster's advantage in an interactive setting.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified software: theories, tools and experiments*, volume 8471 of *LNCS*, pages 55–71, Berlin, Germany, 2014. Springer.

[2] Kenneth Appel, Wolfgang Haken, and Jonh A. Koch. Every planar map is four-colorable. *Illinois Journal of Mathematics*, 21:429–567, 1977.

[3] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 131–146. Springer Berlin Heidelberg, 2010.

[4] Jasmin Christian Blanchette, Andrei Popescu, Daniel Wand, and Christoph Weidenbach. More SPASS with Isabelle. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 345–360. Springer Berlin Heidelberg, 2012.

[5] Timothy Bourke, Rob J. van Glabbeek, and Peter Höfner. A mechanized proof of loop freedom of the (untimed) AODV routing protocol. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, pages 47–63, 2014.

[6] Bruno Buchberger. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, Seria Matematica-Informatica*, XXXVIII(2):9–32, 2000. Selected papers of the 2nd International Workshop on Symbolic and Numeric Algorithms in Scientific Computing, Oct. 4-6, 2000, Timisoara, Romania (T. Jebelean, V. Negru, A. Popovici eds.).

[7] Bruno Buchberger. Algorithm-supported mathematical theory exploration: A personal view and strategy. In *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004*, pages 236–250, 2004.

[8] Bruno Buchberger, Adrian Crăciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus

Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470 – 504, 2006. Towards Computer Aided Mathematics.

[9] Lukas Bulwahn. The new Quickcheck for Isabelle. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, volume 7679 of *LNCS*, pages 92–108. Springer Berlin Heidelberg, 2012.

[10] Harsh Raju Chamarthi, Peter Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 Sedan theorem proving system. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *LNCS*, pages 291–295. Springer Berlin Heidelberg, 2011.

[11] Koen Claessen, Jonas Duregård, and Michal H. Palka. Generating constrained random data with uniform distribution. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 18–34, 2014.

[12] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM Press.

[13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of the Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

[14] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of inductive problems. In *Proceedings of the Conference on Intelligent Computer Mathematics (CICM)*, volume 9150 of *LNCS*. Springer, 2015.

[15] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs (TAP)*, pages 6–21, 2010.

[16] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[17] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of CVC4: How it works, and how to use it. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 4:7–4:7, Austin, TX, 2014. FMCAD Inc.

[18] Lucas Dixon and Moa Johansson. IsaPlanner 2: A proof planner in Isabelle. Technical report, 2007. DReaM Technical Report (System description). `http://dream.inf.ed.ac.uk/projects/isaplanner/docs/isaplanner-v2-07.pdf`.

[19] George W. Ernst and Allen Newell. Some issues of representation in a General Problem Solver. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 583–600, New York, NY, USA, 1967. ACM.

[20] Jacques D. Fleuriot and Lawrence C. Paulson. Proving Newton's Propositio Kepleriana using geometry and nonstandard analysis in Isabelle. In *Automated Deduction in Geometry*, volume 1669 of *LNCS*, pages 47–66. Springer Berlin Heidelberg, 1999.

[21] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, page 333, 2007.

[22] Mike Gordon. From LCF to HOL: A short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[23] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[24] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[25] Moa Johansson, Lucas Dixon, and Alan Bundy. Case-analysis for rippling and inductive proof. In *Proceedings of the International Conference on Interactive Theorem Provers (ITP)*, pages 291–306, 2010.

[26] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

[27] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Proceedings of the Conference on Intelligent Computer Mathematics (CICM)*, volume 8543 of *LNCS*, pages 108–122. Springer International Publishing, 2014.

[28] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[29] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 152–167. Springer Berlin Heidelberg, 2011.

[30] K. Rustan M. Leino. Automating induction with an SMT solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 315–331, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Roy L. McCasland, Alan Bundy, and Patrick F. Smith. Ascertaining mathematical theorems. *Electronic Notes in Theoretical Computer Science*, 151(1):21 – 38, 2006. Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005).

[32] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41 – 57, 2009. Special Issue: Empirically Successful Computerized Reasoning.

[33] Robin Milner. Logic for Computable Functions: Description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[34] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637–1646, 2012.

[35] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

[36] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*, May 2015. Updated with every new version of Isabelle (and Isabelle/HOL). `http://isabelle.in.tum.de/doc/prog-prove.pdf`.

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. Latest online version 25 May 2015. `http://isabelle.in.tum.de/doc/tutorial.pdf`.

[38] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[39] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[40] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *Logic and Computer Science*, 31:361–386, 1990.

[41] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *Proceedings of the 8th International Workshop on the Implementation of Logics*, 2010.

[42] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 8931 of *LNCS*, pages 80–98. Springer Berlin Heidelberg, 2015.

[43] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2,3):91–110, 2002.

[44] Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2,3):111–126, 2002.

[45] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Cormac Flanagan and Barbara König, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *LNCS*, pages 407–421. Springer Berlin Heidelberg, 2012.

[46] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 632–647. Springer Berlin Heidelberg, 2007.

[47] Daniel Wand and Christoph Weidenbach. Automatic induction inside superposition. `https://people.mpi-inf.mpg.de/~dwand/datasup/draft.pdf`.

[48] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier Science and MIT Press, Amsterdam, The Netherlands, 2001.

[49] Makarius Wenzel. Structured induction proofs in Isabelle/Isar. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management*, volume 4108 of *LNCS*, pages 17–30. Springer Berlin Heidelberg, 2006.

[50] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 167–183. Springer Berlin Heidelberg, 1999.

[51] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD thesis, Technische Universität München, 2001.