CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluation of a new recursion induction principle for automated induction

Master's thesis in Computer Science – algorithms, languages and logic

ANDREAS WAHLSTRÖM
LINNÉA ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

MASTER'S THESIS 2017

Evaluation of a new recursion induction principle for automated induction

ANDREAS WAHLSTRÖM
LINNÉA ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Evaluation of a new recursion induction principle for automated induction
ANDREAS WAHLSTRÖM
LINNÉA ANDERSSON

Supervisor: Koen Claessen, Department of Computer Science and Engineering
Examiner: Carlo A. Furia, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Evaluation of a new recursion induction principle for automated induction
ANDREAS WAHLSTRÖM
LINNÉA ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Structural induction is a powerful tool for proving properties of functions with recursive structure, but it is useless when the functions do not preserve the structure of the input. Many of today's cutting-edge automated theorem provers use structural induction, which makes it impossible for them to prove properties about non-structurally recursive functions. We introduce a new induction principle, where the induction is done over a function application. This principle makes it possible to prove properties about non-structurally recursive functions automatically. We compare structural and application induction and the result indicates that application induction is the more powerful induction method, proving more properties, even though structural induction tends to be faster.

Keywords: Computer science, Automatic theorem proving, Application induction, Structural induction, Induction, Master's thesis.

Acknowledgements

First, we would like to thank our supervisor Koen Claessen, who also proposed the main idea behind this thesis, for his passionate participation. His advice and encouragement has been of great value to us and he made this experience both pleasant and rewarding.

We would also like to thank our examiner, Carlo A. Furia, for taking his time reading and giving comments on our thesis. Additionally we would like to express our gratitude to Johan Grönvall and Johan Wermensjö, who took the role as opponents for this work.

Finally, we would like to thank Chalmers University of Technology for giving us this opportunity.

Andreas Wahlström & Linnéa Andersson, Gothenburg, June 2017

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Induction is a principle used in mathematics and computer science to formally prove theorems or specifications. There are multiple methods to apply induction, such as structural induction, recursion induction and induction by size.

There are different reasons for why we would like to be able to formally prove specifications (properties) of a program. It could for example be used for testing and verifying the expected behaviour of the code or for proving whether a property holds [1].

Today, structural induction is a commonly used method when proving properties of structurally recursive functions, functions which, in their recursive call, use a substructure of their input. For an arbitrary variable `n`, structural induction tries to prove the correctness of a property over `n`, given that the property holds for all substructures of `n`. Let us look at the following example, written in Haskell [2], of proving the correctness of the sorting algorithm insertion sort.

```
prop_insertionSort ys = ordered (insertionSort ys)

insertionSort :: [Nat] -> [Nat]
insertionSort []     = []
insertionSort (x:xs) = insert x (insertionSort xs)

insert ::  Nat -> [Nat] -> [Nat]
insert x []                  = [x]
insert x (y:xs) | x <= y     = x : y : xs
                | otherwise = y : insert x xs
```

Listing 1.1: The property representing that insertion sort correctly sorts the input, with relevant functions

In listing 1.1 above, we have defined a property, `prop_insertionSort`. By

doing structural induction over the list **ys** we need to prove two cases, see listing 1.2 below. The first case is when **ys==[]** and it is called the base case. The second case is called the induction case and occurs when **ys==(x:xs)**.

Base case:	**ordered (insertionSort [])**
Induction case:	**ordered (insertionSort (x:xs))**

Listing 1.2: The base and induction case for proving **prop_inseritionSort**

The base case can be trivially proven by looking at the function definition. In the induction case we can assume that the property holds for **xs**, since **xs** is a substructure of **ys**. This gives us:

Hypothesis:	**ordered (insertionSort xs)**
Goal:	**ordered (insertionSort (x:xs))**

The goal can be proven given the lemma: **ordered as == ordered (insert a as)**.

Now, consider another sorting method: quick sort, with the following definition:

```
quickSort :: [Nat] -> [Nat]
quickSort []     = []
quickSort (x:xs) = quickSort (filter (x >=) xs)
                   ++ [x]
                   ++ quickSort (filter (x <) xs)
```

Listing 1.3: Quick sort for natural numbers

In the definition of **quickSort**, the function **filter** takes a predicate function and a list and then filters away all elements which does not fulfill the predicate.

If we would like to prove **ordered (quickSort ys)** using structural induction, we have the same base and induction case as **insertionSort**. Again, the base case is trivial to prove, while the induction case leads us to the following hypothesis and goal:

Hypothesis:	**ordered (quickSort xs)**
Goal:	**ordered (quickSort (x:xs))**

As can be seen in the definition of **quickSort**, we do not keep the structure of the input variable, **(x:xs)**. Instead we create two lists: **(filter (x >=) xs)** containing all elements in **xs** less than or equal to **x**, and **(filter (x <) xs)** containing all elements in **xs** greater than **x**. Therefore, the hypothesis cannot be used to prove the property and hence it cannot be proven by structural induction.

It would be more convenient to prove **ordered (quickSort (x:xs))** given the following:

Hypothesis:	**ordered (quickSort filter (x>=) xs)** **&& ordered (quickSort filter (x>=) xs)**
Goal:	**ordered (quickSort (x:xs))**

This might remind you of recursion induction, see section 3.1.2, a method not used often today. In this project we will examine and implement a modified version of recursion induction, hereafter called *application induction*. Recursion induction creates induction rules from the definition of a function, whereas application induction does induction over every application in the property. This is done by proving so called sub-properties, which are equivalent to the original property.

1.1 Aim

The aim of this project is to implement an automatic theorem prover, based on the new induction method *application induction*. We will not implement the logical reasoning, but use existing prover back-ends. This means we will prepare and modify the property, according to the induction method, but not derive the actual proof. We will examine how well application induction performs with respect to the research questions defined in the following section.

1.1.1 Research Questions

During the course of this project we would like to answer the following questions. They tie in closely to the performance of the new induction method but also touch upon the effect different prover back-ends provide.

- Structural versus application induction:
  - How much time does it take to prove properties?
  - Can structural induction prove any property that application induction cannot?

  - Can application induction prove some property that structural induction cannot?
  - Which kind of properties are proven?

- Performance of application induction:
  - **Variants of application induction** It is possible to prove multiple properties concurrently using application induction (see section 2.1.3) or to do deep application induction (see section 2.1.2), but what effect does this have on the general performance?
  - **Natural and conditional sub-properties.** One of the first steps in application induction is splitting a property into many sub-properties. These sub-properties can be divided into two categories: *natural* and *conditional*, see section 2.1.1. The conditional ones are more complicated, and the question arises whether they are actually necessary, or if the natural ones are enough.

- Does the performance of application induction change depending on which prover back-end that is used?

1.2 Scope

We only consider monomorphic functions in first-order-logic. We also assume the functions we apply induction on always terminate. Since our focus is on the induction itself we do not consider the completeness of the theory exploration tool, i.e., its ability to find the most relevant lemmas for the theory. This means that it in some cases might be relevant to introduce properties and lemmas by hand, when they are not discovered automatically.

To evaluate the induction methods with regards to the different provers, we create a benchmark suite. Due to the limitation of time and the fact that the files needs to be *completely* monomorphised (See section 5.1.1), the number of files is small. It is also difficult to find examples where the structure of the variables and functions differed from other examples.

1.3 Outline

In this thesis we will first, in chapter 2, give a more detailed problem formulation. Then, in chapter 3, we present the technical background necessary to understand the subsequent chapters. The next part, chapter 4, describes the method and the design choices made during the implementation. We then, in chapter 5, describe our benchmark suite. After that, we present our result in chapter 6, followed by chapter 7 where we conclude our work and discuss possible further research and work in the field.

Chapter 2

Problem formulation

In this chapter we will explain: the details of application induction via an example, what happens with properties over mutually recursive function, and the correctness of the new induction method.

2.1 Application Induction

In this section we will explain how application induction works by an example. We will show how it works when proving that quick sort returns a sorted list, that is:

```
φ xs = ordered (quickSort xs)
```

Listing 2.1: Property describing that **quickSort** returns a sorted list

Let us look at the function definition for **ordered** and recall the function definition for **quickSort**:

```
quickSort :: [Nat] -> [Nat]
quickSort []     = []
quickSort (x:xs) = quickSort (filter (x >=) xs)
                   ++ [x]
                   ++ quickSort (filter (x <) xs)

ordered :: [Nat] -> Bool
ordered []       = True
ordered (x:[])   = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

Listing 2.2: Quick sort for natural numbers

By looking at the function definitions of **quickSort** and **ordered**, we derive the following axioms:

Axioms for quickSort:
Axiom 1.1: ∀ **xs . xs == [] => quickSort xs == []**
Axiom 1.2: ∀ **xs,y,ys . xs == (y:ys) =>**

```
quickSort xs ==
    quickSort (filter (y >=) ys)
    ++ [y]
    ++ quickSort (filter (y <) ys)
```

Axioms for ordered:
Axiom 2.1: ∀ **xs . xs == [] => ordered xs == True**
Axiom 2.2: ∀ **xs y . xs == [y] => ordered xs == True**
Axiom 2.3: ∀ **xs,x,y,zs . xs == (x:y:zs) =>**

```
ordered xs == x <= y && ordered (y:zs)
```

We now create one property for each function application in φ, such that if one of these properties hold, we have proven the original property. In this case we have two different function applications, and thus two 'sub'-properties:

$p1_{\text{quickSort}}$ **xs = ordered (quickSort xs)**
$p2_{\text{ordered}}$ **xs = ∀ ys . xs == quickSort ys => ordered xs**

If we have a proof for either $p1_{\text{quickSort}}$ or $p2_{\text{ordered}}$, we have a proof for the original property, φ. Let us look at $p1_{\text{quickSort}}$, where we are doing induction over the application **quickSort**. By instantiating the definition of **quickSort** with **as**, we get the following axioms:

Axiom 1.3: **as == [] => quickSort as == []**
Axiom 1.4: ∀ **x,xs . as == (x:xs) =>**

```
quickSort as ==
    quickSort (filter (x >=) xs)
    ++ [x]
    ++ quickSort (filter (x <) xs)
```

We want to prove **ordered (quickSort as)** by induction over **quickSort**. From the definition of **quickSort**, this gives us two different cases, **as=[]** and **as=(x:xs)**. The first case is trivial, but in the second case we have to introduce an induction hypothesis. Figure 2.1 illustrates how properties are split, first into sub-properties and then into cases.

Application induction makes induction over an application, on the number of function calls until termination. Given that a program terminates, this means we can assume that a property holds for all arguments of the recursive function calls. Therefore, application induction allows us to assume that $p1_{quickSort}$ holds for `(filter (x >=) xs)` and `(filter (x <) xs)`, since they are the arguments to the two recursive function calls of `quickSort as`, see the function definition in listings 2.2. In the case of `as=(x:xs)` this gives us the following hypotheses and goal:

Hypothesis:
```
ordered (quickSort (filter (x >=) xs))
    && ordered (quickSort (filter (x <) xs))
```
Goal:
```
ordered (quickSort (x:xs))
```

Just as in many other cases of proving properties, this property cannot be proven without help-lemmas. Although, contrary to structural induction, we have some additional axioms we can make use of.

Later, in section 2.1.4, we present an informal argument to why application induction works.

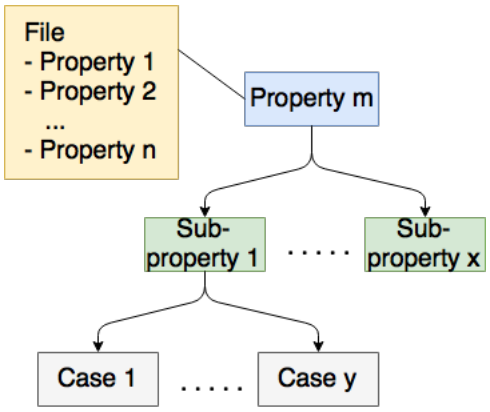

Figure 2.1: Illustration of how the property is split

2.1.1 Natural and conditional properties

As briefly mentioned in the introduction (chapter 1), splitting the original property may result in two kind of sub-properties. We call these *natural* and *conditional* properties. As an example we will use **`ordered (quickSort xs)`**, the same property as in the previous section.

In simple terms, a natural property occurs when we induct on an application with pure independent arguments. That is, when the arguments to the application are variables, which are independent of each other, and not another function application. An example of a natural property can be seen in listing 2.3, where the induction is done over **`quickSort`**. We call these properties *natural* since they retain the shape of the original property.

```
ordered (quickSort xs)
```

Listing 2.3: Example of a *natural* property

When the application chosen for induction has one or more applications as arguments, we end up with a conditional property, as shown in listing 2.4, where the induction is done over **`ordered`**. We call such a property *conditional* since it is of the shape: **`condition => property`**.

```
∀ xs . ys == quickSort xs => ordered ys
```

Listing 2.4: Example of a *conditional* property

2.1.2 Deep application induction

Suppose we are trying to prove **`ordered (quickSort as)`** over the function application **`ordered`**. The cases where **`quickSort as`** returns an empty list or a list with exactly one element are trivial. In the case where **`quickSort as=(x:y:zs)`**, we would get the following hypothesis and goal (with the previous definition of **`ordered`**):

```
Hypothesis: ∀ xs . (y:zs) == quickSort xs => ordered (y:zs)
Goal:       ∀ xs . as == quickSort xs => ordered as
```

Assuming **`quickSort as == [a,b,c]`** would, by unfolding it, give us the following:

```
ordered [a,b,c] =
(a ≤ b && ordered [b,c]) =
(a ≤ b && b ≤ c && ordered [c]) =
(a ≤ b && b ≤ c && True)
```

As can be seen, even though we only have one induction hypothesis, we have more than one call to the function **ordered**. A deep application induction creates hypotheses for all arguments to all calls to **ordered**, even though they are not on the right hand side of the function definition for the input argument. In the case of proving **ordered (quickSort [a,b,c])**, this gives us:

```
Hypothesis: ∀ xs. [b,c] == quickSort xs => ordered [b,c]
Hypothesis: ∀ xs. [c] == quickSort xs => ordered [c]
Goal:       ∀ xs. [a,b,c] == quickSort xs
                           => ordered [a,b,c]
```

2.1.3 Dependent application induction

In listing 2.5 two functions, **evens** and **odds**, are defined. The function **evens** returns all even indexed element in a list, whereas the function **odds** returns all odd indexed elements in a list.

```
evens :: [Nat] -> [Nat]
evens (x:xs) = x : odds xs
evens []     = []

odds :: [Nat] -> [Nat]
odds (x:xs) = evens xs
odds []     = []
```

Listing 2.5: The definitions of the two functions, **evens** and **odds**

Suppose we are trying to prove $\mathbf{p1}_{\mathbf{evens}}$ and $\mathbf{p2}_{\mathbf{odds}}$, shown in listing 2.6. As seen in the definitions of **evens** and **odds**, we do not have any recursive calls, instead, **evens** uses the function **odds** and vice versa. Hence, when proving $\mathbf{p1}_{\mathbf{evens}}$ or $\mathbf{p2}_{\mathbf{odds}}$ no induction hypotheses are created with application induction. Even though deep application induction would give hypotheses, we will here explore another method which we call dependent application induction.

```
p1 = ordered as => ordered (evens as)
p1_ord1 = ordered as => ordered (evens as)
p1_ord2 = ∀ bs . as = evens bs => (ordered bs => ordered as)
p1_evens = ordered as => ordered (evens as)


p2 = ordered as => ordered (odds as)
p2_ord1 = ordered as => ordered (odds as)
p2_ord2 = ∀ bs . as = odds bs => (ordered bs => ordered as)
p2_odds = ordered as => ordered (odds as)
```

Listing 2.6: The properties **p1** and **p2** with all their sub-properties

If we would like to prove $\texttt{p1}_{\texttt{evens}}$, it would probably be beneficial if we could assume that the related property $\texttt{p2}_{\texttt{odds}}$ holds and the other way around when proving $\texttt{p2}_{\texttt{odds}}$. In application induction we can assume the sub-property we are currently proving holds for the input to all recursive calls. In dependent application induction we remove the limitation of only being able to make assumptions over the current sub-property. We also look at all function calls in the definition of the function we induct on, not only the recursive ones. If **as=a':as'** this means that when trying to prove $\texttt{p1}_{\texttt{evens}}$, we are allowed to use all sub-properties over **odds** for the argument **as'**, hence we could assume **ordered as' => ordered (odds as')**.

If both $\texttt{p1}_{\texttt{evens}}$ and $\texttt{p2}_{\texttt{odds}}$ are proven we know that both original properties are valid. This since we, when trying to prove the properties, only assume that sub-properties hold for arguments to function calls with less steps until termination. Since we have assumed that the program terminates, we know that we will reach a base case and thus both properties must be valid.

More generally, if we find a set of properties such that they do not require any property outside the set to be valid, then we know that all properties in that set are valid.

2.1.4 Correctness of Application Induction

We will not give a proof of the correctness of application induction, but following we will provide an informal argument to why it is correct.

We know that if a function **f(x,y)** calls another function **g(z)**, then **g(z)** requires fewer calls than **f(x,y)** to terminate. Since everything terminates, we know we will always reach the case when there are no more function calls. If the property is valid for this case and for the case when the function application requires **n+1** calls, given that the property is valid for all function application with fewer than **n+1** calls, then the property is valid for all **n**.

Chapter 3

Background

This chapter aims to provide the theoretical background that may be necessary to fully understand further chapters. The background will provide information about: related work in induction, auxiliary libraries and tools, theory exploration, automated theorem proving, as well as existing theorem provers.

3.1 Related work

Induction is often used in mathematics, where it is used as a method for proving statements in an *arranged order*. It could for example be to prove equality of equations with natural numbers [3].

When it comes to proofs for properties about recursively defined structures, induction is a powerful tool, since there are many similarities between recursive definitions and the way induction approaches a proof attempt. In this section three induction methods, useful for proving properties about programs, will be explained.

3.1.1 Structural Induction

As mentioned in the introduction, structural induction is a method that can be used for proving properties about recursively defined structures. This method does induction over a variable, using the structure of the variable's data type [4] (for example a list). Given the structure of the induction variable, structural induction allows us to assume the property holds for any subset of the structure. Let us look at the following implementation of a list type and a list reverse function:

```
data [a] = []
         | a : [a]
```

Listing 3.1: The list data type

```
rev :: [a] -> [a]
rev as =
    case as of
        []       -> []
        (c : cs) -> rev cs ++  (c : [])
```

Listing 3.2: The list reverse function

Proving the property ∀ `as.  as == rev (rev as)` with structural induction yields a base case and an induction case. The base case is `as == []`, which is trivial to solve. The induction step is `as == (a:as')`. Since `as'` is a subset of `as`, the induction hypothesis becomes `as' == rev (rev as')`. Here parallels to mathematical induction over natural numbers can be found, where a theorem for `n+1` should be proven given the hypothesis that it holds for `n`.

If the base case and the induction step is proven with help of the induction hypothesis, then `as == rev (rev as)` has been successfully proven with structural induction.

3.1.2 Recursion Induction

Application induction is, as mentioned in chapter 1, a modified version of recursion induction. Recursion induction makes use of the recursive structure of the functions instead of the recursive structure of the variables, which structural induction does. Therefore, everything proven by recursion induction, as well as application induction, must terminate.

The difference between application induction and recursion induction is that recursive induction does not create sub-properties, instead it looks at the definition of the functions and create induction rules [5]. Therefore, the *conditional properties* (see section 2.1.1) are never proven by recursion induction.

In the example described in section 2.1, `ordered (quickSort xs)`, recursion induction only tries to prove the sub-property `ordered (quickSort xs)`, due to its unconstrained variables.

3.1.3 Induction by size

Induction by size, uses the size of the data structure to create its hypothesis. Assume property `p(x)` is to be proven and let `n` be the size of `x`. If the size of the data structure of `x` is defined in the range `[a,b)`, the base case would be to prove `p(x)`, where `n=a`, and the induction step would be to prove `p(x)` given

the hypothesis `p(x')` with size of `x'` less than `n`.

Induction by size can be used for different data structures, for example the size could be defined as the length of a list or a tree. When used with natural number it could be equal to the natural number it uses. Induction by size needs the size definition of each data structure, which might not always be as obvious as in the case of natural numbers or lists. How to define the size might be different depending on what property should be proven. Considering a tree, the size can be described by the number of nodes, the number of leaves, or the depth of the tree.

Application induction can also be considered as a variation of induction by size, with size being the number of steps until termination.

3.2 TIP - benchmarks & tools

To create a common benchmark-suite, Claessen et al. introduced *Tons of Inductive Problems*, or TIP [6]. TIP is a collection of benchmarks, which are more or less challenging, that aims to serve as a test-suite for inductive theorem provers. TIP also introduces a common format, extending and combining currently available formats, for specifying these benchmarks.

Beside the benchmark suit there is also a programming library and multiple executables, available at GitHub [7]. These executable mainly deal with translating theory between various existing formats. It supports Haskell, SMT and more. There is also support for modifying the existing theory (containing information about for example definitions, conjectures etc.), which sometimes is a prerequisite before translating the theory. The library specifies an API, in Haskell, for using the functionality directly from inside a program.

3.3 Theory Exploration

Theory Exploration is the practice of extending a mathematical theory from a set of basic concepts by progressively computing more and more complex theorems from the initial building blocks. Buchberger, a proponent of *computer-supported theory exploration*, argued that this was how a mathematician would normally begin their work [8].

3.3.1 TipSpec & QuickSpec

QuickSpec [9] is a program finding properties, or specifications, about a Haskell program. QuickSpec does not prove the properties it finds, but it does test them using a large amount of data. [10]

Among the tools provided by TIP there is one for theory exploration, called

TipSpec. TipSpec is based on QuickSpec and, given a theory, attempts to find as many relevant lemmas from the theory as possible. Of course, to avoid bloating the theory with too many speculated lemmas, it removes lemmas which it deem to be too similar to each other.

3.4 Automated theorem proving

Automated theorem proving (ATP) is a sub-field in automated mathematical reasoning. The main goal of ATP is to develop computer programs for proving mathematical theorems. Various techniques are used in this field, for example, mathematical induction, satisfiability modulo theory (SMT), and lean theorem proving.

There are multiple automatic theorem provers available today, among them we find Zeno, IsaPlanner, HipSpec and ACL2. Our version of application induction and HipSpec, which uses structural induction, have a very similar structure, which we will discuss more in the coming section.

3.4.1 HipSpec

HipSpec [11] is a program for proving properties of Haskell programs using first-order logic and structural induction. It automatically derives new properties and proves their correctness. HipSpec is built upon the automated induction prover, *Hip*, and conjecture generator, *QuickSpec*. A conjecture in this case refers to the equivalence of two terms.

HipSpec works by reading a Haskell program and, using QuickSpec, generates thousands of conjectures that Hip then attempts to prove or disprove. Proved conjectures can be used as lemmas which can then be used to prove more conjectures. If a conjectures has not been proved or disproved within a set timeout, it is considered as a *failed* conjecture. HipSpec also support user-written properties, which are tested after the conjectures from QuickSpec have been processed.

3.4.2 Zeno

Beside HipSpec, Zeno [12] might be the automatic theorem prover most similar to our application, but there are some differences. Zeno uses a top-down approach and hence finds auxiliary lemmas, but no additional help-lemmas can be given [13]. This can be a weakness when larger, more readable proofs should be created.

3.4.3 Approaches to Lemma Discovery

Many automatic theorem provers support some kind of lemmas discovery technique, for example Zeno, uses something called *lemma calculation*. This technique is used to replace goals or common sub terms in goals with a variable, which can be tested separately. This approach can be called a top-down approach. In contrast, HipSpec utilizes a bottom-up approach called theory exploration, as described in section 3.3.

3.5 Prover back-ends

As mentioned in the introduction (section 1.1) our application will by itself not derive any proofs. This task is instead given to dedicated theorem provers. Such theorem provers work by proving given conjectures in a theory. Given a theory, that includes axioms and definitions, it uses some strategy to find out whether one or more conjectures are true (to disprove a conjecture is a completely different problem). In the two following sections the provers used in this work will be described.

3.5.1 E - Theorem Prover

The E theorem prover (E-prover) tries to prove equality of a full first-order logic formula. E-prover first parses the problem and creates clauses and formulas. After the problem has been parsed, E-prover optionally prunes the clauses and formulas that will most likely not lead to a proof. E-prover then translate each formula into clausal form, which it afterwards pre-processes, to, for example, remove redundant clauses. After pre-processing, E-prover tries to solve the problem, possibly within a given timeout, stopping after it either deem the problem unsolvable or computes a proof. [14]

3.5.2 Z3 - Theorem prover

The Z3 [15] solver is an SMT theorem prover developed by Microsoft. Z3 is implemented in C and it is built upon, and combines, a number of other theorem provers. It also contains an E-matching engine to handle quantifiers. [16]

Chapter 4

Designing the theorem prover

We have implemented a program [17] for proving properties of SMT-LIB (version 1 & 2) and Haskell programs using either structural or application induction. In the input files the user can supply properties to be proven, which we call *user-specified properties*. Many non-trivial properties cannot be proven straight away using induction; instead their proofs require one or more auxiliary lemmas. Therefore, our program supports a type of lemma discovery, called theory exploration, described in section 3.3.

Some theorem provers, like IsaPlanner [18] and Zeno [13], use a top-down approach, as described in 3.4.3, while our prover, just as in HipSpec, is built with a bottom-up approach, i.e., we explore the theory for auxiliary lemmas before attempting any proofs. The theory exploration is done using TipSpec, which finds lemmas based on the function definitions in the source file (see section 3.3.1 for more details).

In the following sections we describe the design of our prover, which comprises: the induction methods and additional features.

4.1 General structure

We will now describe how our prover processes a given set of properties. From the theory with all properties we select one conjecture, which we try to prove with the specified induction method. If we succeed in proving a conjecture, we create a lemma and include it in the set of available help-lemmas. We can now use this lemma to prove other properties.

During one iteration we try to prove all properties, if none of them are proven we stop. Otherwise, some properties were proven and thus we have new lemmas. This means that we can attempt another iteration on the unproven properties.

Figure 4.1 illustrates the flow of proving a conjecture. First, the program

attempts to prove the property without induction. If this does not work then induction is applied to prove the property. No matter which induction method, we split the property into multiple sub-properties (for structural induction it depends on the variables in the property), each with multiple cases, and also create the hypotheses. The sub-properties are then tested until either, one has been proved, or all have unsuccessfully been attempted. A sub-property is said to have been proved when all its cases have been verified to be true.

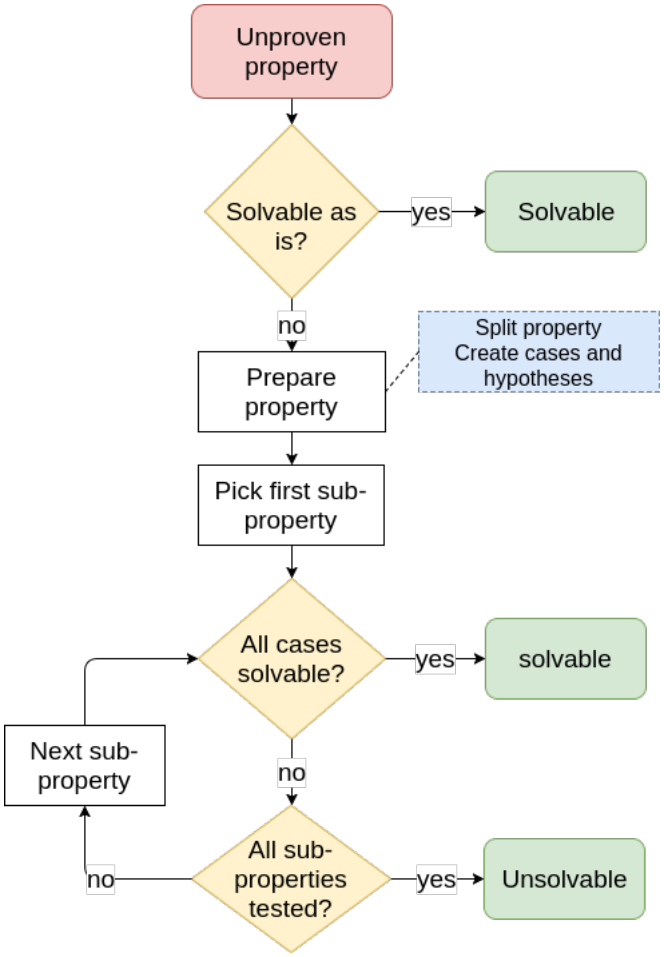

Figure 4.1: The program flow when trying to prove a property.

4.2 Structural Induction

While structural induction is not the focus of our theorem prover, it is still a very important benchmark for our own induction method. Here is a short summary of how structural induction was implemented.

When trying to prove a property with structural induction, we do induction over all variables, one at a time, in the goal we are trying to prove. It is possible to specify induction over multiple variables. In that case, all possible combinations of the specified number of variables will be tried.

4.3 Application Induction

Now, we will detail how application induction has been implemented. Let us assume that we have one property we would like to prove with application induction. We create one sub-property for each function application in the property. We will describe this step more carefully in the next section. After creating all sub-properties, we use a specified prover to try to prove these sub-properties until either, one is proven (we then have a proof of the property), or none of them are provable. In our prover we do not consider **==** to be a function to do induction over, because we used the predefined version and hence we did not have access to the definition.

4.3.1 Sub-properties

As mentioned earlier in chapter 2, one sub-property is created for each function application in the original property. Recall the example in chapter 1, where the goal is to prove that insertion sort returns an ordered list.

```
insertionSort :: [Nat] -> [Nat]
insertionSort []       = []
insertionSort (x:xs)   = insert x (insertionSort xs)
```

Listing 4.1: Insertion sort for natural numbers

```
ordered :: [Nat] -> Bool
ordered []       = True
ordered (x:[])   = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

Listing 4.2: The function that checks whether a list is ordered

As can be seen, the function **insertionSort** calls another function **insert**. As for now, we will not give a definition the **insert** function, since it is not part of the property. We can use proven lemmas for **insert**, but we do not need to know the definition of the function.

The original property, which we want to prove, is defined as follows:

```
prop xs = ordered (insertionSort xs)
```

Listing 4.3: The property that **insertionSort** properly sorts the input

We split the property into two sub goals as seen in listing 4.4:

```
prop_ordered xs =
   ∀ ys . xs == insertionSort ys => ordered xs
prop_sort xs = ordered (insertionSort xs)
```

Listing 4.4: The sub properties to **prop**

As described in chapter 2, we create the sub-properties by looking at the functions' definitions. We can assume that our property hold for all recursive function calls on the right hand side of the pattern matching cases.

To verify the property for all different cases, we create one conjecture for each case and declare that all of them should hold. In listing 4.5 and 4.6, we show the hypotheses implied by the different possible cases, for **insertionsSort** and **ordered** respectively. In both listings we have instantiated the argument to the function application we do induction over with **as**.

```
as == []     : No hypothesis
as == (x:xs) : ordered (insertionSort xs)
```

Listing 4.5: The different cases, with hypotheses, for **insertionSort**

```
as == []          : No hypothesis
as == [x]         : No hypothesis
as == (x1:x2:xs) :
   ∀ ys . (x2:xs) == insertionSort ys => ordered (x2:xs)
```

Listing 4.6: The different cases, with hypotheses, for **ordered**

If we succeed in proving all the cases in either one of these two sub-properties, we have proved the original property.

4.3.2 Proving the cases

There are two different approaches in attempting to prove all cases. Either we prove the cases one by one (we call this approach *split cases*), or we join the cases together.

Listing 4.7 shows the hypotheses for the split-case approach, for induction on **ordered**.

```
Case 1: as == []
Case 2: as == [x]
Case 3: as == (x1:x2:xs) &&
  ∀ ys . (x2:xs) == insertionSort ys => ordered (x2:xs)
```

Listing 4.7: Split cases for **ordered**

Listing 4.8 shows how the sub property's hypothesis would be if we joined the different cases. As can be seen we are using the disjunction between the cases when we join them. The reason for this is because we are creating a global constant **ys** that the property should hold for, but since we do not know what **ys** is, we need to prove it for all different cases.

```
as == []
| as == [x]
| as == (x1:x2:xs) &&
  ∀ ys . (x2:xs) == insertionSort ys => ordered (x2:xs)
```

Listing 4.8: Joined cases for **ordered**

4.4 Back-ends and other functionality

We have decided to let the user decide whether the cases should be split into multiple hypotheses or be joined into one. This because the result of the two approaches varies depending on the prover back-end used.

The time taken while proving a conjecture can be very long. In order to test the performance of the different induction methods with the provers, it is possible to specify one or multiple timeouts. The timeout specifies the total time the prover back-end can use in a proof attempt. As long as at least one conjectures is proven during one iteration, over all conjectures, we keep the same timeout; otherwise the next timeout is used. We try all timeouts given to the program and return when all of them are tried.

The tool used for lemma discovery, TipSpec (see section 3.3.1), can at times be rather slow, much depending on the number of functions and definitions in the input file. Because of this we implemented a storing procedure for the generated files.

Now, if there is no existing TipSpec file for the current job, we create one, unless the user explicitly commands not to. This file is persistently stored and is used for any subsequent runs for the same job. This is to ensure that TipSpec does not run unnecessarily and also makes it possible to run the program (with different options) on the exact same file. Of course, if the original file has been modified, it is also possible, and necessary, to force an update of the TipSpec file.

Chapter 5

Benchmark suite

To evaluate application induction, using the two different provers, a benchmark suite has been created. All benchmark files are based on files from the TIP benchmark suite, see section 3.2. To better understand the results of this work, this chapter will first describe how the testing was performed and then the files used in the benchmark.

5.1 Evaluation

All files in the benchmark suite was run with four different configurations. All configuration were run with the split cases option since structural induction employed that approach:

- Structural induction using E-prover
- Structural induction using Z3
- Application induction using E-prover
- Application induction using Z3

The provers were first given one second to prove each case of each subproperty, then the properties which could not be proven within one second were tried with a five second timeout.

In the introduction it was mentioned that this project would not consider polymorphic function. TIP contains functionality to make a file monomorphic, which in the beginning was considered to be the method to handle polymorphic benchmark files. Unfortunately, this functionality turned out to be incomplete, thus we monomorphised the benchmarks by hand. The following section will describe this in more detailed.

5.1.1 Monomorphisation

The greatest problem with the monomorphisation has without a doubt been that we lost both lemmas and axioms. Without even an error message, some logic simply disappeared, probably due to TIP not being able to infer the types properly.

A requirement for monomorphisation is that is it possible to infer the types of the variables. In 5.1 we can deduce that both **x** and **y** are lists of natural numbers, since ordered takes a list of natural numbers as input. Whereas in listing 5.2 (++) has an input variable with polymorphic type and hence it cannot be deduced to a single type (or even multiple types).

```
prop_ord_concat x y =
    ordered (x ++ y) => (ordered x && ordered y)
```

Listing 5.1: Due to **ordered** taking a list of natural numbers as input, we know that both **x** and **y** has that type

```
prop_concat y = ([] ++ y == y ++ [])
```

Listing 5.2: **y** can be any type of list

5.1.2 Prover back-ends

The Z3 prover can handle polymorphism, hence, to make the benchmarks work for Z3, we could have solved the problem by handling polymorphism in our induction. E prover, on the other hand, does not support any polymorphism and thus, to be able to use E-prover, we required the benchmarks to be monomorphic.

Another problem related to polymorphism was that since Z3 contained built-in lemmas about certain function, it performed better than E-prover for properties about these functions. All such functions, **+**, **−**, $\leq$ etc., were what TIP also considered as built-in functions. The different in treatment for these functions made it so that TipSpec did not derive lemmas about them, thus E-prover had fewer lemmas than Z3 for certain proofs. Implementing all such built-in functions from scratch solved this problem since it allowed TipSpec to speculate lemmas for the functions, ensuring that the provers had access to the same lemmas.

5.2 Test cases

The benchmark suite consists of four files, which TipSpec generates more than 200 conjectures from. Each file contains both data structures as well as functions. As mentioned, due to problem with TIP and monomorphisation, all functions and data types were implemented by us (even the functionality existing in prelude, the base functionality in Haskell). The files used in the benchmark suite is included in appendix A.

All files contain different functions and data structures. The only exception are *quickSort* and *insertionSort*, this due to the reason that both structural induction and application induction was believed to prove that insertion sort returned a sorted list, whereas only application induction was believed to be able to prove that quick sort returned a sorted list.

The first file, *interleave*, contains two mutually recursive functions: **`evens`** and **`odds`**. The function **`evens`** takes a list with natural numbers and returns a list containing all even-indexed elements. The function does not have any recursive calls, it instead calls the function **`odds`**. Similarly, **`odds`** returns the list containing all odd-indexed elements of a given list, with help of the **`evens`** function. *Interleave*'s last function, **`interleave`**, takes two lists, **`x`** and **`y`**, as input and output a list where it alternates the elements from **`x`** and **`y`**. This function is recursively defined, but swaps the arguments.

The second and third file, *quickSort* and *insertionSort*, contain implementations of the sorting methods quick sort respectively insertion sort. Both files contains functions such as concatenation of lists, less than or equal for natural numbers, and the sorting method itself. They also contain help-functions to check the order of the sorted list, filter all elements greater than (or, less or equal to) a given number, and counting the times a given number appears in a list.

The fourth file, *flatten*, contains tree and list data structures. It also contains functions to flatten a tree in four different ways, as well as a function to concatenate a list. Some of the functions to flatten the tree are structurally recursive, whereas some are not.

Chapter 6

Results

This chapter presents the results of this work. The chapter starts with an overview of the results from the benchmark suite, together with a comparison of the two induction methods over the whole suite. Then, the results are presented again but divided by benchmark file, along with explanations for those particular results.

6.1 Overall results

From table 6.1, it can be noticed that the results, from the benchmark files, differ quite a bit from file to file. There were also noticeable differences in what type of properties were provable by the different configurations, something discussed more deeply in section 6.2.

Table 6.1: Table showing the result of the benchmarks

	Application-E	**Application-Z3**	**Structural-E**	**Structural-Z3**
Flatten				
—time taken (s)	56.0	33.4	69.1	62.1
—solved	28	28	19	19
QSort				
—time taken (s)	5435.2	3223.3	3554.9	2327.1
—solved	116	78	123	59
ISort				
—time taken (s)	3636.7	1192.9	419.7	704.5
—solved	63	55	64	47
Interleave				
—time taken (s)	92.3	85.7	16.3	14.4
—solved	8	8	7	7
Total solved	215	169	213	132

The number of conjectures proved by structural and application induction, varies quite a bit between the two provers. In plot 6.2 and 6.1 each *dot* is one

conjecture that was proven.

Structural induction often proves a conjecture using its first sub-property, due to the fact that most functions pattern matches on their first variable. Therefore it was decided to only count the time for proving a sub-property, instead of also including the time for all failed sub-properties. Using a heuristic, application induction would likely also be able to pick the most probable sub-property.

The plots does not display conjectures which remain unproven by both induction methods. Furthermore, all conjectures were proven using induction, even though some might be provable without it. This decision was made due to the fact that some of the conjectures might need induction by one prover but, due to differences in previously proven conjectures, not by the other, which would not be a *fair* result.

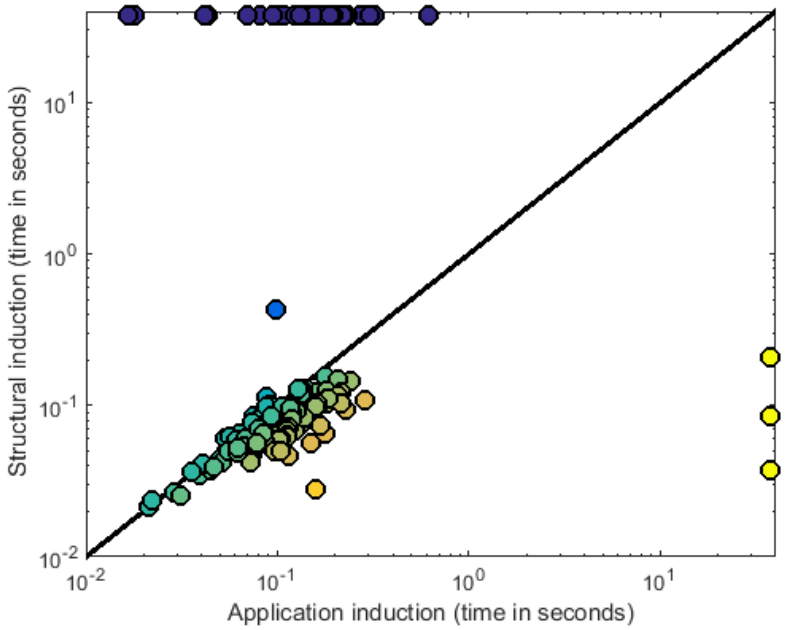

Figure 6.1: Scatter plot for all lemmas proven when using Z3 as back-end

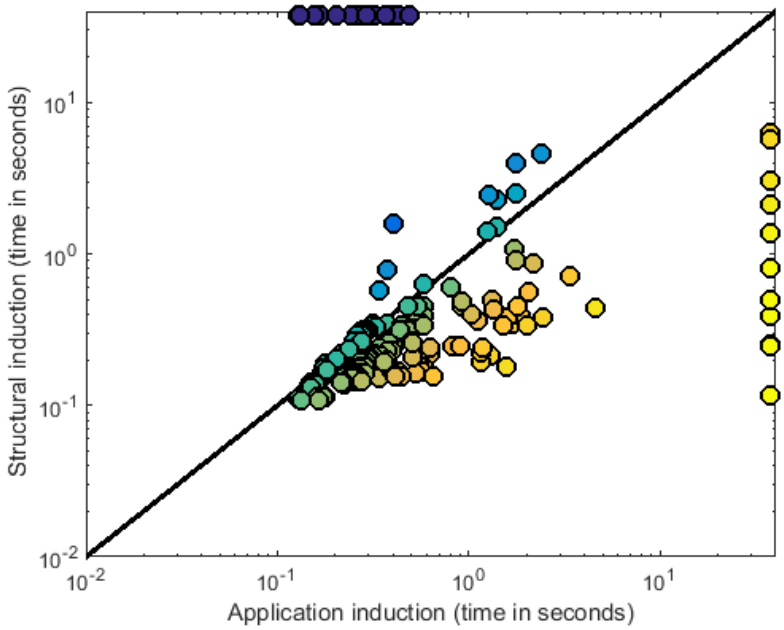

Figure 6.2: Scatter plot for all lemmas proven when using E-prover as back-end

The following list shows three interesting results derived from the information in the two plots:

1. Application induction solved more lemmas then structural induction. It might be hard to see in the plots, but in table 6.1 it can be seen that application induction solved more lemmas both using E-prover and Z3.
2. Structural induction was overall faster than application induction.
3. E-prover proved more lemmas than Z3, although Z3 proved the conjectures much faster than E-prover. The reason for Z3 being faster than E-prover might be due to greater preprocessing time.

Another thing noticed was that, sometimes, properties that are proven quickly when supplied with the required help-lemmas cannot be proven (or proven slowly) if further lemmas are provided. This behaviour could depend on how the prover chooses the strategy for the proof attempt, i.e., if the prover receives more information, in this case an extra lemma, it may decide upon another proving strategy. Still, in many cases the prover can still find a proof, just much more slowly. This was the case when proving that quick sort returned an ordered list. In appendix B, all lemmas required to prove quick sort are listed (and can be proved with application induction). If the order in which

we try to prove the lemmas is changed, it might not be possible to prove all properties.

To get a clearer image of what kind of properties the different methods can prove, the following section presents and examines results from the different benchmark files, one at the time.

6.2 Evaluation by file

The result for the number of conjectures solved by the different provers vary a lot depending on the structure of the functions in the property.

6.2.1 Insertion sort

Figure 6.3 shows the number of conjectures solved over time from the conjectures in the *insertionSort* benchmark. As can be seen, structural and application induction using E-prover solved almost the same number of conjectures, but structural induction was quite a bit faster.

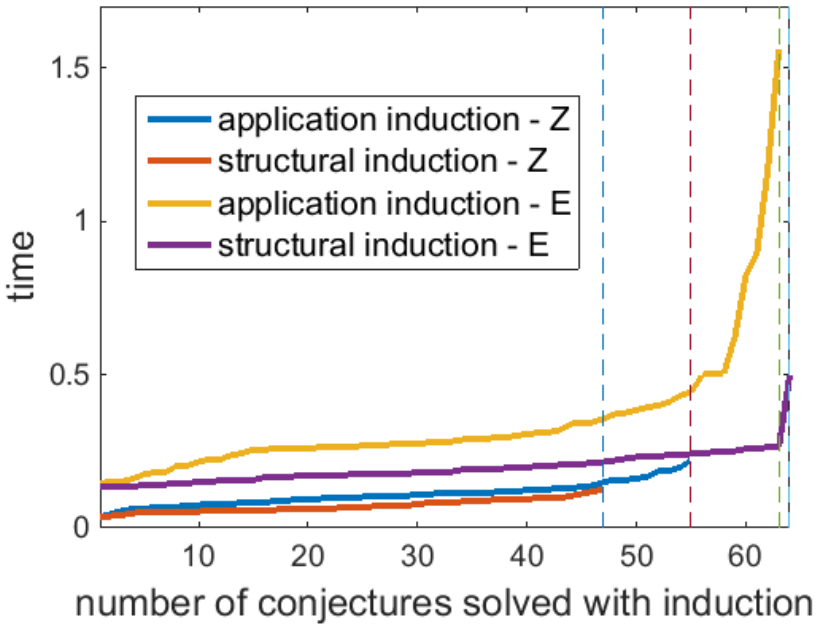

Figure 6.3: The insertion sort benchmark

Before testing application induction, our hypothesis was that the performance of the two different methods would be equivalent when proving properties about the functions in *insertionSort*. This due to the fact that all functions

in *insertionSort* are structurally recursive. The results varied greatly between the two provers while between the induction methods, there was only one conjecture provable only by structural induction.

6.2.1.1 Theory exploration gone wrong

The conjecture shown in listing 6.1 is the property which was only proved by structural induction, but not by application induction. The reason that the property could not be proved by application induction relates to the theory exploration by TipSpec, which in this case produced an unsatisfactory result.

```
Given:
       x + Zero == x
       (Succ x) + y == x + (Succ y)
       Succ (x + y) == x + (Succ y)
       x + y == y + x
Show:
    ((y + x) <= (z + x)) == (y <= z)
```

Listing 6.1: Property provable by structural but not application induction

Structural induction solved the conjecture in listing 6.1 by induction over `x`. Application induction could solve this property, but in the case of using E-prover it took longer time. Although, E-prover could solve the very similar one in listing 6.2 quickly. Using the lemma in listing 6.2, application induction could also directly solve the original property.

```
((x + y) <= (x + z)) == (y <= z)
```

Listing 6.2: Alternative version of lemma in listing 6.1

This indicate that the theory exploration unfortunately discovered a conjecture more suited for proof by structural induction. If the theory exploration instead discovered the conjecture in listing 6.2, then the situation might instead have been reversed. Possibly, the second property might actually have been discovered, but was then immediately discarded since it was too similar to the first one.

6.2.2 Quick sort

Figure 6.4 shows the number of conjectures solved over time for properties generated from the *quickSort* file. Although the result seems weak for application induction, it should be mentioned that none of the properties proven by structural induction and not by application induction were related to the sorting algorithm itself.

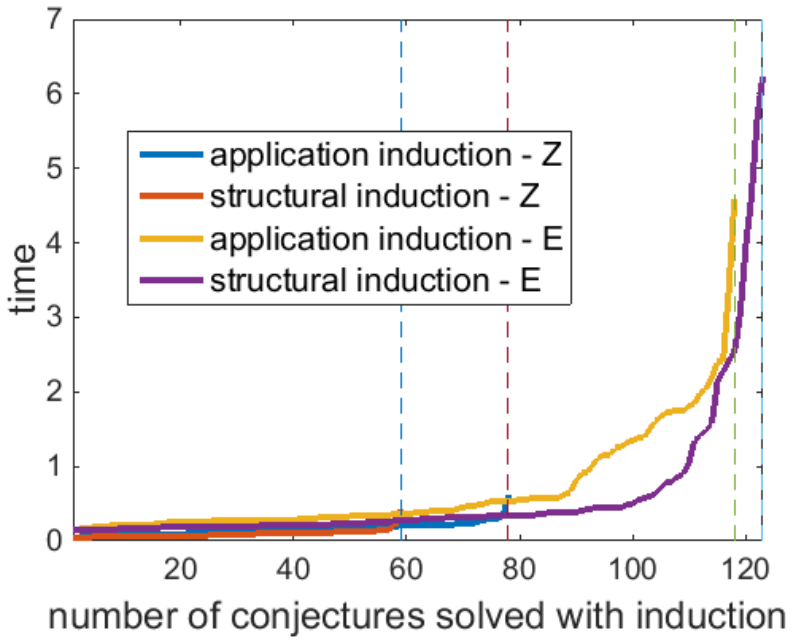

Figure 6.4: The quick sort benchmark

Our hypothesis was that only application induction would be able to prove that quick sort returned an ordered list, but while using TipSpec, none of the methods were able to do so. In the following section we will provide the results for proving the order of the sorted list using only user-specified lemmas.

6.2.2.1 Without lemma discovery

As mentioned in chapter 1.2, we decided that, if relevant, we were allowed to add help-lemmas if TipSpec did not generate them. In the case of proving that quick sort returns a sorted list, we decided to not use TipSpec at all, but only use user-specified lemmas. Given the lemmas in listing 6.3, application induction was able to prove that quick sort returns an ordered list. Some of these help-lemmas required other lemmas to be proven. For the full list of lemmas and definitions, see appendix B

```
lemma1 xs x    = smallerEq xs x == smallerEq (qsort xs) x
lemma2 xs ys z = smallerEq xs z && bigger ys z =>
   ordered (xs ++ [z] ++ ys) == ((ordered xs) && (ordered ys))
lemma3 xs x    = bigger (filterGT x xs) x
lemma4 xs x    = smallerEq (filterLEq x xs) x
lemma5 xs x    = bigger xs x == bigger (qsort xs) x
```

Listing 6.3: The help-lemmas necessary for proving **`ordered (quickSort xs)`**

The function **`smallerEq`** takes two argument, a list of natural number and a natural number. It returns true if all elements in the list is smaller or equal to the second argument. The function **`bigger`** takes two arguments, a list of natural number and a natural number. It returns true if all elements in the list is greater than the second argument. The filter functions, **`filterGT`** and **`filterLEq`**, both takes two arguments, a list and a natural number. The function **`filterGT`** returns a list with all number greater than the second argument, whereas **`filterLEq`** returns a list containing all elements less than or equal to the second argument.

Due to the fact that the provers took much longer time if many help-lemmas were given and Z3 already has some of the required functionality, we decided to do this proof using Z3 (with polymorphic functions). Application induction proved all 21 properties (20 help-lemmas and one conjecture), whereas structural induction was only able to prove 9 of 21.

6.2.3 Interleave

In figure 6.5 we see the results, number of conjectures solved over time, from the *interleave* benchmark. We can see that application induction has a minor lead compared to structural induction but there was also one property provable only by structural induction.

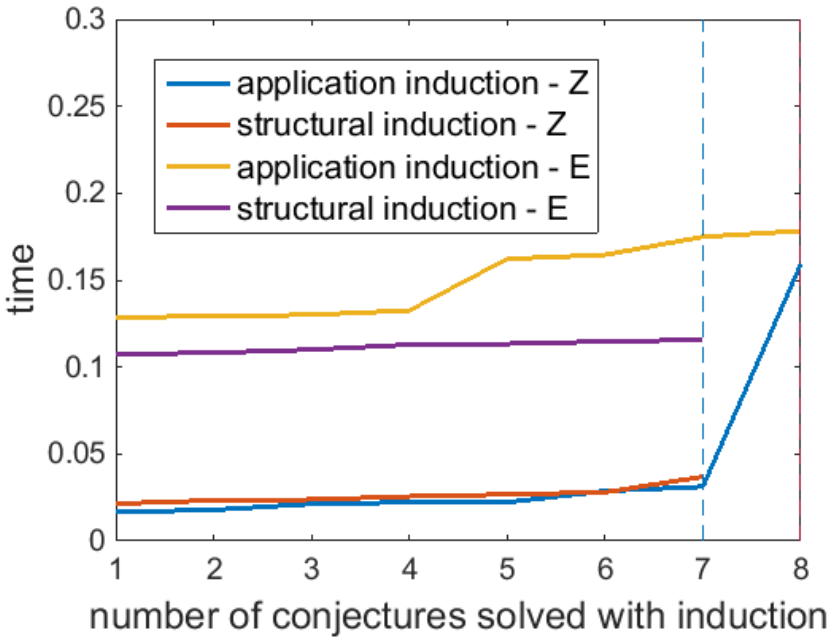

Figure 6.5: The interleave benchmark

The *interleave* benchmark was mainly used to test application induction on mutually recursive functions. Even though we did not implement any of the variants of application induction, which would work for mutually recursive functions, sections 2.1.2 and 2.1.3, an interesting result was discovered. This will be explored further in the next section.

6.2.3.1 Mutually recursive functions

Application induction depends on the recursive structure of the function and if there are no recursive calls then there are no hypotheses. Structural induction, on the other hand, makes induction over the variables and hence, as long as the variable contains recursive substructures, there will be hypotheses when using structural induction.

The following example shows the function **`evens`** and a related property. It is not possible to prove the property over **`interleave`**, but structural induction can prove it over the variable **`x`**. Since evens is not directly recursively defined, standard application induction is not able to prove this property. The same would apply when proving properties over the **`odds`** function.

```
property x = evens (interleave x x) == x

evens :: [Nat] -> [Nat]
evens (x:xs) = x : odds xs
evens []     = []

odds :: [Nat] -> [Nat]
odds (x:xs) = evens xs
odds []     = []
```

Although this property could not be proven using application induction, it was possible to prove it using deep application induction, see section 2.1.2. This example was solved by creating the hypotheses by hand and then use the back-ends to prove it. This shows us that deep application induction can improve the performance, but at the cost of more time. The longer time is due to the fact that the deep application induction add extra axioms when proving and, as mentioned in section 6.1, adding more axioms can make a big difference in the time taken to prove a property.

6.2.3.2 Recursion and application induction

One of our research question was if application induction can prove something recursion induction cannot.

As described in the example in section 6.2.3.1, **evens** and **odds** does not give any hypotheses when using application induction. The example below can therefore not be proven using recursive or application induction on the **evens** and **odds** functions. Although, it can be proven by induction with the conditional sub-property over the function application **interleave**. Hence, it is only possible to prove the property with application induction, and using a conditional sub-property.

```
interleave (evens xs) (odds xs) == xs
```

6.2.4 Flatten

In the benchmark file *flatten*, application induction was able to solve more conjectures than structural induction, see figure 6.6.

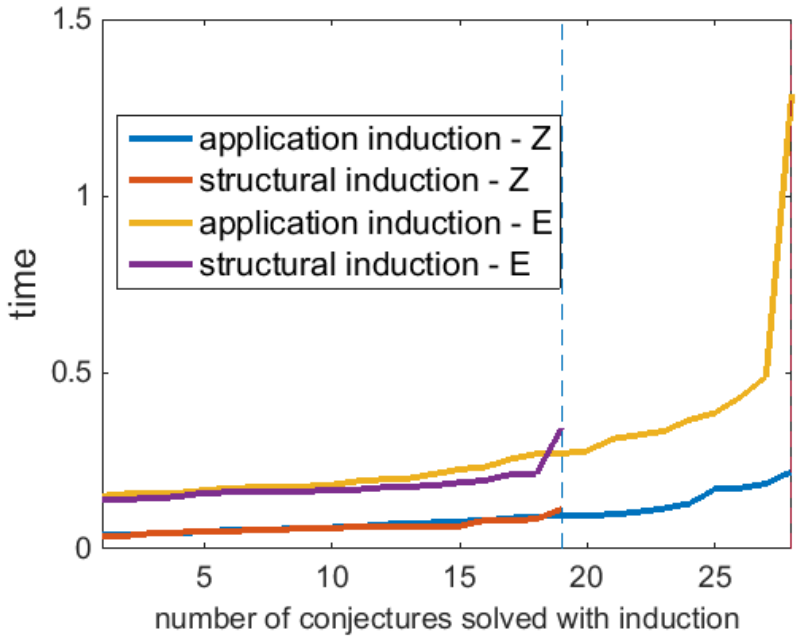

Figure 6.6: The flatten benchmark

Our hypothesis was that application induction would be able to solve more properties, due to the fact that some of the functions defined in the file are not structurally recursive. In listing 6.4, the functions **`flatten0`** and **`flatten2`** are two examples of structurally recursive functions, whereas **`flatten3`** is a non-structurally recursive function.

```
flatten0 :: Tree -> [Nat]
flatten0 Nil          = []
flatten0 (Node p x q) = flatten0 p ++ [x] ++ flatten0 q

flatten2 :: Tree -> [Nat] -> [Nat]
flatten2 Nil          ys = ys
flatten2 (Node p x q) ys = flatten2 p (x : flatten2 q ys)

flatten3 :: Tree -> [Nat]
flatten3 Nil                      = []
flatten3 (Node (Node p x q) y r) =
               flatten3 (Node p x (Node q y r))
flatten3 (Node Nil x q)           = x : flatten3 q
```

Listing 6.4: Three functions from *flatten*

The result turned out to be as expected, only application induction was able

to prove properties about the non-structurally recursive functions. One such property, **prop03**, is listed in listing 6.5. The property **prop02**, on the other hand, can be solved by both structural and application induction.

```
prop03 x = flatten0 x == flatten3 x
prop02 x = flatten0 x == flatten2 x []
```

Listing 6.5: Two properties about the functions in listing 6.4

Chapter 7

Conclusion

In general, application induction has given us some promising results and does solve more properties than structural induction, especially for properties of non-structurally recursive functions. Disregarding bloated theories with too many lemmas, we see an encouraging future ahead for application induction. Considering our results we believe that using a top-down approach could alleviate many of the problems we encountered during this work.

In this chapter we condense our experience and results into answer to the research questions introduced in section 1.1.1. Afterward we also speculate about the directions further research should take when it comes to application induction.

7.1 Research Questions

In this section we aim to present our answers to all research questions proposed in this thesis.

7.1.1 Structural vs. application induction

In the beginning our hypothesis was that application induction could prove at least all properties provable by structural induction. Unfortunately, this was not the case, as can be observed from the results. Another hypothesis was that application induction could prove properties about non-structurally recursive functions. This hypothesis was confirmed as seen in, for example, section 6.2.4. Let us now look at the questions about the comparison of the two induction methods.

- **How much time does it take to prove properties?**

  As seen in chapter 6, structural induction is generally faster than application induction. Most likely, this depends on the structure of the

hypothesis. This is also one of the reasons why structural induction is able to prove properties which application induction cannot prove in a reasonable time limit.

- **Can structural induction prove any conjecture that application cannot?**

  Yes, but all properties that were investigated could be solved by application induction if one of the following actions were taken:

  - Longer time was given, since application induction in general is slower.
  - Fewer help-lemmas were in the file, see section 6.1 for more info.
  - Manually introduce help-lemmas, since the theory exploration sometimes pruned necessary lemmas, see section 6.2.1.1.
  - Use deep application induction. This is necessary if there are no *direct* recursive calls, see section 6.2.3.1.

- **Can application induction prove some conjecture that structural induction cannot?**

  Yes, in cases where the functions are non-structurally recursive, this was often the case. For an example see section 6.2.4.

- **Which kind of properties are proven?**

  For the structurally recursive functions, structural induction is to be preferred, mostly due to the time it takes to prove the properties. When it comes to non-structurally recursive functions, application induction is necessary to be able to prove the properties using induction.

7.1.2 Performance of application induction

- **What effect does the variants of application induction have on the performance?**

  Due to the time limit of this project, the implementation of proving properties concurrently was postponed. The reason was also due to the fact that the back-end provers struggled when the number of help-lemmas increased, see chapter 6 for examples.

  We did not have the time to implement deep application induction but, as shown in section 6.2.3.1, deep application was tested on one of the benchmark files. There it was shown that deep application induction could prove a property which normal application induction could not. This, we believe, is a result worth further consideration.

- **Are conditional properties necessary?**

  The biggest difference between recursive induction and application induction, as mentioned in chapter 3.1.2, is that application induction can handle *conditional* properties. In section 6.2.3.2 it was shown that conditional properties are sometimes necessary to prove a property. This result tells us that application induction is, at least in some cases, better than recursion induction.

7.1.3 Prover back-ends

The difference in performance for the two prover back-ends is clear. E-prover seems to prove more properties than Z3, but when Z3 proved a property it did it faster. It was noted that the provers were very sensitive and given more lemmas, they often proved less. When given more lemmas they seemed to not use the hypotheses, since they were given as normal axioms. It would be interesting to somehow promote the hypothesis to the prover, hinting that it should be prioritized.

7.2 Future research

The conclusions in this report, lead us to believe there is a bright future for application induction. We have recorded our thoughts and opinions about the future of application induction, as well as for any theorem prover implementing it. We hope that this text may inspire and guide any who is interested in continuing our research.

As we have repeatedly mentioned, the provers are sensitive to the number of help-lemmas provided, and thus more help-lemmas are not always to be preferred. This makes us believe that a good heuristic, for choosing relevant help-lemmas, might grant huge improvements.

For our prover we decided on the bottom up approach using theory exploration. In hindsight, this might actually not have been the optimal choice, since it, to some extent, gives rise to the problems described above. Using the top down approach instead, would help in reducing the amount of extra lemmas. To keep using the bottom up approach we would need to somehow change the theory exploration to ensure that the lemmas discovered are the ones we want, as illustrated in section 6.2.2.

As seen from the results, structural induction tends to be faster, while application induction is able to prove properties that structural induction is not able to. Thus, at the current stage of implementation, we would recommend a mix of application and structural induction, to receive the benefits of both. Such a mix should probably focus on structural induction but use application induction when unable to prove a property. Alternatively, by developing a

heuristic, one could perhaps decide upon one of the methods before attempting the proof. By having access to the function definition, it would be possible to observe 1) whether the function preserves the structures, and 2) whether there are any recursive calls available. These points should be enough to create a simple heuristic for choosing which induction method to use.

Bibliography

[1] M. Johansson, "Reasoning about Functional Programs: Exploring, Testing and Inductive Proofs." Talk at Off the Beaten Track 2017, feb 2017.

[2] haskell.org, "Haskell - An advanced, purely functional programming language." https://www.haskell.org/. Accessed 2017-06-13.

[3] J. A. Bather, "Mathematical induction," 1994.

[4] R. M. Burstall, "Proving properties of programs by structural induction," *The Computer Journal*, vol. 12, no. 1, pp. 41–48, 1969.

[5] I. Lobo Valbuena, "Automated discovery of conditional lemmas in hipster," 2015. 54.

[6] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, *TIP: Tons of Inductive Problems*, pp. 333–337. Cham: Springer International Publishing, 2015.

[7] TIP, "TIP: Tons of Inductive Problems." https://tip-org.github.io/. Accessed 2017-05-27.

[8] B. Buchberger, "Theory exploration with Theorema," *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, vol. 38, no. 2, pp. 9–32, 2000.

[9] N. Smallbone, "Quickspec: equational laws for free!." https://github.com/nick8325/quickspec. Accessed 2017-06-02.

[10] K. Claessen, N. Smallbone, and J. Hughes, "Quickspec: Guessing formal specifications using testing," in *International Conference on Tests and Proofs*, pp. 6–21, Springer, 2010.

[11] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *International Conference on Automated Deduction*, pp. 392–406, Springer, 2013.

[12] W. Sonnex, "Zeno: An automated proof system for Haskell programs." http://hackage.haskell.org/package/zeno. Accessed 2017-06-02.

[13] W. Sonnex, S. Drossopoulou, and S. Eisenbach, *Zeno: An Automated Prover for Properties of Recursive Data Structures*, pp. 407–421. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[14] S. Schulz, "System description: E 1.8," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 735–743, Springer, 2013.

[15] M. Research, "The Z3 Theorem Prover." `https://github.com/Z3Prover/z3`. Accessed 2017-06-02.

[16] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*, pp. 337–340. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[17] A. Wahlström and L. Andersson, "Application Induction." `https://github.com/LinneaAndersson/FunInd.git`. Accessed 2017-06-02.

[18] L. Dixon and J. Fleuriot, *IsaPlanner: A Prototype Proof Planner in Isabelle*, pp. 279–283. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

Appendix A

The benchmark files

A.1 Utils.Types

A file which defines different types.

```
module Utils.Types where

data Nat = Zero | Succ Nat

data TreeList = TNil | TCons Tree TreeList
data NatList = NNil | NCons Nat NatList

data Tree = Node (Tree) Nat (Tree) | Niil
```

A.2 quick sort

```
{-# LANGUAGE ScopedTypeVariables #-}
module Sort where

import Tip
import Utils.Types

(<=*) :: Nat -> Nat -> Bool
Zero       <=* b          = True
a          <=* Zero       = False
(Succ a)   <=* (Succ b)   = a <=* b

(++*) :: NatList -> NatList -> NatList
NNil       ++* bs = bs
(NCons a as)   ++* bs = NCons a $ as ++* bs
```

```
(+*) :: Nat -> Nat -> Nat
Zero      +* b = b
(Succ a)  +* b =Succ $ a +* b


qsort ::  NatList -> NatList
qsort NNil     = NNil
qsort (NCons x xs) = qsort (filterLEq x xs) ++*
                (NCons x NNil) ++* qsort (filterGT x xs)

filterLEq, filterGT :: Nat -> NatList -> NatList
filterLEq a NNil = NNil
filterLEq a (NCons b bs)
    | b <=* a = NCons b $ filterLEq a bs
    | otherwise = filterLEq a bs

filterGT a NNil = NNil
filterGT a (NCons b bs)
    | not (b <=* a) = NCons b $ filterGT a bs
    | otherwise = filterGT a bs

smallerEq :: NatList -> Nat -> Bool
smallerEq NNil _     = True
smallerEq (NCons x xs) y = x <=* y && smallerEq xs y

bigger :: NatList -> Nat -> Bool
bigger NNil _     = True
bigger (NCons x xs) y = (not (x <=* y)) && bigger xs y

ordered :: NatList -> Bool
ordered NNil       = True
ordered (NCons x NNil)      = True
ordered (NCons x (NCons y xs)) = x <=* y && ordered (NCons y xs)

count :: Nat -> NatList -> Nat
count x NNil = Zero
count x (NCons y ys)
  | x == y = (Succ Zero) +* count x ys
  | otherwise = count x ys

prop_count x xs = count x xs === count x (qsort xs)
prop_QSortSorts xs = bool $ ordered (qsort xs)
prop_FALSE xs = ordered (qsort xs) === ordered xs
```

A.3 Insertion sort

```
module ISort where
import Tip
import Utils.Types

(++*) :: NatList -> NatList -> NatList
NNil       ++* bs = bs
(NCons a as)  ++* bs = NCons a $ as ++* bs

(+*) :: Nat -> Nat -> Nat
Zero       +* b = b
(Succ a)  +* b =Succ $ a +* b

(<=*) :: Nat -> Nat -> Bool
Zero       <=* b          = True
a          <=* Zero       = False
(Succ a)  <=* (Succ b)  = a <=* b

sort :: NatList -> NatList
sort = isort

isort ::  NatList -> NatList
isort NNil     = NNil
isort (NCons x xs) = insert x (isort xs)

insert ::  Nat -> NatList -> NatList
insert x NNil                  = (NCons x NNil)
insert x (NCons y xs) | x <=* y    = NCons x $ NCons y xs
                | otherwise = NCons y $ insert x xs

prop_countCount x ys zs = count x (ys ++* zs) ===
                          count x ys +* count x zs
prop_ISortSorts xs = ordered (isort xs) === True
prop_ISortCount x xs = count x (isort xs) === count x xs
prop_FALSE xs = ordered (isort xs) === ordered xs

ordered :: NatList -> Bool
ordered NNil        = True
ordered (NCons x NNil)      = True
ordered (NCons x (NCons y xs)) = x <=* y && ordered (NCons y xs)

count :: Nat -> NatList -> Nat
count x NNil = Zero
count x (NCons y ys)
  | x == y = (Succ Zero) +* count x ys
  | otherwise = count x ys
```

A.4 Interleave

```
{-# LANGUAGE ScopedTypeVariables #-}
module Interleave where

import Tip
import Utils.Types


{-# NOINLINE evens #-}
evens :: NatList -> NatList
evens (NCons x xs) = NCons x $ odds xs
evens NNil     = NNil

{-# NOINLINE odds #-}
odds :: NatList -> NatList
odds (NCons x xs) = evens xs
odds NNil     = NNil

interleave :: NatList -> NatList -> NatList
interleave (NCons x xs) ys = NCons x $ interleave ys xs
interleave NNil     ys = ys


prop_Interleave xs =
  interleave (evens xs) (odds xs) === xs
```

A.5 Flatten

```
module Tree where

import Tip
import Utils.Types (Tree(..),NatList(..),TreeList(..),Nat(..))


(++*) :: NatList -> NatList -> NatList
NNil      ++* bs = bs
(NCons a as)  ++* bs = NCons a $ as ++* bs

concatMapF0 :: TreeList -> NatList
concatMapF0 TNil      = NNil
concatMapF0 (TCons a as) = (flatten0 a) ++* (concatMapF0 as)

flatten0 :: Tree -> NatList
flatten0 Niil          = NNil
flatten0 (Node p x q) = flatten0 p ++* (NCons x NNil) ++* flatten0 q

flatten1 :: TreeList -> NatList
flatten1 TNil                  = NNil
flatten1 (TCons Niil ps) = flatten1 ps
flatten1 (TCons (Node Niil x q) ps) = NCons x $ flatten1 (TCons q ps)
flatten1 (TCons (Node p x q) ps) =
               flatten1 (TCons p $ TCons (Node Niil x q) ps)

flatten2 :: Tree -> NatList -> NatList
flatten2 Niil          ys = ys
flatten2 (Node p x q) ys = flatten2 p (NCons x $ flatten2 q ys)

flatten3 :: Tree -> NatList
flatten3 Niil                    = NNil
flatten3 (Node (Node p x q) y r) = flatten3 (Node p x (Node q y r))
flatten3 (Node Niil x q)         = NCons x $ flatten3 q

prop_Flatten1 p =
  flatten1 (TCons p TNil) === flatten0 p
prop_Flatten1List ps =
  flatten1 ps === concatMapF0 ps
prop_Flatten2 p =
  flatten2 p NNil === flatten0 p
prop_PROVE_FALSE p a =
  flatten3 (Node p a p) === flatten0 p
prop_Flatten3 p =
  flatten3 p === flatten0 p
```

Appendix B

Quicksort with user specified properties

```
{-# LANGUAGE ScopedTypeVariables #-}
module Sort where
import Tip

qsort ::  [Int] -> [Int]
qsort []     = []
qsort (x:xs) = qsort (filterLEq x xs) ++ [x] ++ qsort (filterGT x xs)

filterLEq, filterGT :: Int -> [Int] -> [Int]
filterLEq a [] = []
filterLEq a (b:bs)
    | b<=a = b:filterLEq a bs
    | otherwise = filterLEq a bs

filterGT a [] = []
filterGT a (b:bs)
    | b>a = b:filterGT a bs
    | otherwise = filterGT a bs

smallerEq :: [Int] -> Int -> Bool
smallerEq [] _     = True
smallerEq (x:xs) y = x <= y && smallerEq xs y

bigger :: [Int] -> Int -> Bool
bigger [] _     = True
bigger (x:xs) y = x > y && bigger xs y

ordered :: [Int] -> Bool
ordered []       = True
ordered [x]      = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

```
count :: Int -> [Int] -> Int
count x [] = 0
count x (y:ys)
  | x == y = 1 + count x ys
  | otherwise = count x ys

property xs = bool $ ordered (qsort xs)

lemma1 a b c    = a++(b++c) === (a++b)++c
lemma2 x as bs  = count x as + count x bs === count x (as ++ bs)
lemma3 y        = count y [] === count y (qsort [])
lemma4 y x xs   = count y (x:xs) ===
                  count y (filterLEq x xs) +
                          count y [x] + count y (filterGT x xs)
lemma5 x xs     = count x (qsort xs) === count x xs
lemma6 a b x    = bigger (a++b) x === ((bigger a x) && (bigger b x) )
lemma7 a b x    = smallerEq (a++b) x ===
                  ((smallerEq a x) && (smallerEq b x))
lemma8 ys x     = (smallerEq ys x) === (filterGT x ys == [])
lemma9 ys x     = (bigger ys x) === (filterLEq x ys == [])
lemma10 y ys x  = smallerEq (y:ys) x ===
                   smallerEq (filterLEq y ys ++ [y] ++ filterGT y ys) x
lemma11 y ys x  = bigger (y:ys) x ===
                   bigger (filterLEq y ys ++ [y] ++ filterGT y ys) x
lemma12 xs x    = smallerEq xs x === smallerEq (qsort xs) x
lemma13 xs x    = bigger xs x === bigger (qsort xs) x
lemma14 x xs    = (ordered xs && smallerEq xs x) ==> ordered (xs++[x])
lemma15 x xs    = ordered (xs++[x]) ==> (ordered xs && smallerEq xs x)
lemma16 x xs    = (ordered xs && bigger xs x) ==> ordered (x:xs)
lemma17 a b     = ordered (a ++ b) ==> ((ordered a) && (ordered b))
lemma18 x xs    = bool $ smallerEq (filterLEq x xs) x
lemma19 x xs    = bool $ bigger (filterGT x xs) x
lemma20 xs ys z = smallerEq xs z && bigger ys z ==>
                     ordered (xs ++ [z] ++ ys) ===
                     ((ordered xs) && (ordered ys))
```