**Abstract**

This Master thesis gives a new tool to automatically verify equational properties written in the functional programming language Haskell. The aim is to be able to reason about infinite and partial values available in Haskell from general recursion and lazy data structures. The novelty of this approach is to use automated theorem provers for first order logic by means of a translation from Haskell to first order theories. The properties are instantiated with different induction techniques applicable to non strict functional languages: structural induction, fixed point induction and the approximation lemma. As the target logic is untyped, Haskell features such as pattern matching and higher order functions needs to be dealt with special care. The results from using the tool on a test suite are convincing as the automated provers quickly deduce theorems for a variety of properties. To be able to compete on fair grounds with contemporary tools a system for adding lemmas would be required, which turned out to be difficult as theorems for finite and infinite values do not coexist peacefully out of the box. Our conclusion is that first order logic is able to express the various constructs in Haskell and that theorem provers perform well on the generated theories, and this without having to prove termination.

# Contents

# Chapter 1

# Introduction

Consider this data type of trees and its implementation of the `Monad` (Jones, 1995) functions:

```
data Tree a = Fork (Tree a) (Tree a) | Leaf a

return :: a -> Tree a
return = Leaf

(>>=) :: Tree a -> (a -> Tree b) -> Tree b
Fork u v >>= f = Fork (u >>= f) (v >>= f)
Leaf x   >>= f = f x
```

Could the elegant simplicity of bind possibly obey the three monad laws as stated by Wadler (1992)? Referential transparency in Haskell allows equational reasoning, but proving the monad laws by hand can be tiresome. Further, as bind is recursive, induction will be necessary. This is a situation where our tool `hip`, the Haskell Inductive Prover, can help. Write down the equations in a similar style as properties in QuickCheck (Claessen and Hughes, 2000). Run the tool on the source file which reports this after a few seconds:

```
Theorem: prop_return_right, t >>= return = t
  by structural induction on t and by approximation lemma

Theorem: prop_return_left, return x >>= f = f x
  by approximation lemma and by definitional equality

Theorem: prop_assoc, (t >>= f) >>= g = t >>= (\ x -> f x >>= g)
  by structural induction on t and by fixed point induction on >>=
```

The output `Theorem` means that the properties hold for infinite trees, in contrast to the possible result `Finite Theorem`. What is going on behind the scenes to prove these properties? The key components of this tool are:

1. a translation from Haskell to *first order logic*,
2. instantiating the properties with different *induction techniques*, and
3. running *automated theorem provers* on the generated theories.

The technology described and discussed in this thesis allows an automated way to prove *equational properties*. Moreover, it can reason about *infinite values* as the `Trees` above.

## 1.1 Aim

The aim of this thesis is to develop a tool able to do automated proving of Haskell properties. This should be realised by means of different induction techniques, a translation to first order logic and invocations to automated theorem provers. The tool should be able to reason and prove properties about lazy data structures, higher order functions and infinite and partial values. The project is restricted to equational properties to keep it tractable, and equality should coincide with equality in first order logic. This means that two values are equal if they are created by the same constructor, and the arguments to the constructors are all equal. As models of first order theories have extensional equality, this will also be asserted for Haskell functions.

## 1.2 Background

The target reader for this thesis typically has a good understanding of Haskell or some related functional language. Some familiarity of model theory and proof deduction in first order logic is needed, as well as a some knowledge of proofs by induction.

Automated theorem provers will be mainly considered as black boxes. Although different theorem provers accept different input formats describing first order theories, the theories will be written as ordinary logic formulae.

The rest of this thesis should be self contained.

## 1.3 Related Work

Zeno (Sonnex et al., 2011) is probably the most similar work to ours. Zeno is a tool that proves equational properties for Haskell programs, but unlike ours also supports type classes, and implications with equational antecedents, and can generalise properties. A notable contrast is that it does not handle infinite and partial values, which is a motivation goal of our work. In the Zeno setting, all values are total and functions are assumed to terminate. Another notable difference is that Zeno has its own theory for equational reasoning whereas our approach puts this work on automated theorem provers.

Zeno translates its proofs to Isabelle, a framework for different logics. Its most prominent use is as a theorem prover for Higher Order Logic, commonly abbreviated Isabelle/HOL (Nipkow et al., 2002). Different automated proving techniques has been implemented, most notably the heuristic Rippling (Dixon, 2005), found in IsaPlanner (Dixon and Johansson, 2007), which also relies on critics (Ireland and Bundy, 1995). Isabelle/HOL can also call theorem provers with Sledgehammer (Blanchette, 2011), and a comparison of different theorem provers in Sledgehammer has been done by Böhme and Nipkow (2010).

The proof assistant and programming language Coq resembles the function package in Isabelle in the sense that each data type declaration generates an induction principle and the user can write own induction procedures should it be too weak. Proofs can be assisted by tactics to automatically resolve some goals, and has an own language Ltac to write such tactics. A tactic for Rippling has been implemented for Coq (Wilson et al., 2010).

Coq has some support for corecursion, and so has the dependently language Agda (Norell, 2007). Its strong type system based on Martin-Löf type theory (Martin-Löf, 1980) with inductive families makes it a proof assistant thanks to the Curry-Howard correspondence. The support for corecursion in Agda has been investigated by Danielsson (2010).

In the work by Bove et al. (2011) Agda is used as a frontend to automated theorem provers enabling classical reasoning about functional programs. Reconstruction of proofs from the theorem prover Waldmeister to dependently typed code is investigated by Armstrong et al. (2011). The source language is Mella, syntactically similar to Agda but based on the Calculus of Constructions. A similar approach was taken by Lindblad and Benke (2004) does a search akin to Prolog, and can also handle induction. A contemporary implementation of this work is the Agsy proof search assistant for Agda.

These automated approaches to Agda do not handle infinite values. Techniques used in this thesis about corecursive functional programs are given for Haskell in Gibbons and Hutton (2005), and for Isabelle Paulson (1997), and in general by Gordon (1994). New approaches to stream processing can be found in the work by Hinze (2010).

A radically different way to reason about Haskell programs than this thesis does is investigated by Danielsson et al. (2006). The idea is to do reasoning in a language assumed to only be total, and under many circumstances the proofs also holds for partial values. Another approach based on operational semantics is given by Sands (1996). The proofs from this technique resembles those from fixed point induction, and the same techniques can also be used to derive locally improved functions.

## 1.4 Outline

The translation from Haskell to First Order Logic is described in Chapter 2. The problem is first tackled with a naive but straightforward attempt that fails and is then resolved. The different sections described different aspects of Haskell, such as pattern matching and higher order functions. Section 2.9 covers the mere basics of domain theory that is relevant for this translation and for the induction techniques. The Chapter is concluded with future work of parts not currently covered.

The proof techniques this tool uses are described in Chapter 3. Four different techniques are described and the outline in that chapter will guide you further. Each technique is explained, exemplified and proved or motivated to be correct.

The results of using this translation and the proof techniques on a test suite are presented is in Chapter 4. The different techniques are compared and the results are discussed. Chapter 5 describes possible future work and the last Chapter 6 concludes the thesis by looking back at what has been accomplished.

# Chapter 2

# Haskell to First Order Logic

To enable automated theorem provers to do equational reasoning of Haskell programs a translation to first order logic is needed. It is here referred to as a translation, but it could also be regarded as a compilation. The idea is to use constants and functions in first order logic to correspond to constructors and functions, and arguments to functions need to be universally quantified. We shall try to do a naïve attempt of a translation with these ideas and see how far it takes us.

## 2.1 Naïve Translation

We will use a data type of binary trees with an element at every branch, and consider some examples of functions defined on it. The definition of the data type is:

```
data Tree a = Fork (Tree a) a (Tree a) | Leaf
```

With the idea above, occurrences of the `Fork` constructor in the source code should be translated to a logic function $\mathrm{fork}$, and similarly a constant for `Leaf`. How should we then translate the `singleton` function, defined below?

```
singleton :: a -> Tree a
singleton x = Fork Leaf x Leaf
```

Following our intuition we make an universal quantification for `x`, and a new logic function for `singleton`. The result is this axiom[1]:

$$\forall\, x.\, \mathrm{singleton}(x) = \mathrm{fork}(\mathrm{leaf}, x, \mathrm{leaf})$$

To capture the intuition that values produced by different constructors are indeed different, appropriate axioms needs to be added. Without these, there will be models with only one element where everything is identified. The axioms added expressing that values created from different constructors are unequal will be called *disjoint constructor axioms*.

[1] Haskell functions and constructors are written in `monospace` and their counterpart in first order logic as $\mathrm{this}$. As customary when writing first order logic formulae, functions will be written in lowercase, and predicates with an initial capital. Hence constructors will be written as $\mathrm{fork}$ rather than $\mathrm{Fork}$.

For the data type `Tree`, one disjoint constructor axiom is generated for its two constructors:

$$\forall\, l, x, r.\, \text{leaf} \neq \text{fork}(l, x, r)$$

Constructors should also be injective to get regular models, and expressing such axioms is straightforward. For `Tree`, only `Fork` has arguments and this injectivity axiom is needed:

$$\forall\, l_0, l_1, x_0, x_1, r_0, r_1.\, \text{fork}(l_0, x_0, r_0) = \text{fork}(l_1, x_1, r_1) \rightarrow l_0 = l_1 \wedge x_0 = x_1 \wedge r_0 = r_1$$

In the next section, injectivity of constructors will a consequence of another axiom.

To describe pattern matching, consider a `mirror` function, which recursively swaps the left sub tree with the right and vice-versa. We follow our intuition to translate the pattern matching to these two axioms[2]:

```
mirror :: Tree a -> Tree a
mirror (Fork l x r) = Fork (mirror r) x (mirror l)
mirror Leaf         = Leaf
```

$$\text{I.} \quad \forall\, l, x, r.\, \text{mirror}(\text{fork}(l, x, r)) = \text{fork}(\text{mirror}(r), x, \text{mirror}(l))$$

$$\text{II.} \quad \text{mirror}(\text{leaf}) = \text{leaf}$$

A problem with this translation is that there are no axioms for other arguments of $\text{mirror}$ than leafs and forks, and there are models including other values than leafs and forks. Another problem is encountered for `singleton`'s left inverse `top` defined below, which returns the top element of a `Tree`. This function is a partial since the `Leaf` pattern is omitted:

```
top :: Tree a -> a
top (Fork l x r) = x
```

The translation must capture the pattern match failure that results from trying to evaluate `top` applied to a `Leaf`. We conclude that this naïve translation does not take us further, but we shall see in the next section how to fix these problems.

## 2.2 Bottom and Pattern Matching

We borrow the concept bottom from domain theory. It is denoted $\bot$ and is the least defined value, and corresponds to pattern match failures, use of `error` and `undefined` in the source code, and also non-terminating functions. For `top`, the idea is to add an axiom so that $\text{top}$ applied to anything that is not a `Fork` is bottom. This function is an example of such an axiomatisation[3]:

$$\text{I.} \quad \forall\, l, x, r.\, \text{top}(\text{fork}(l, x, r)) = x$$

$$\text{II.} \quad \forall\, t.\, (\nexists\, l, x, r.\, \text{fork}(l, x, r) = t) \rightarrow \text{top}(t) = \bot$$

Most theorem provers would as a preprocessing step skolemise the existential quantification in the second axiom. A new unary function would be introduced for $l$, $x$ and $r$,

[2] Axioms are enumerated by Roman numerals to tell them apart.

[3] This thesis uses the same convention for quantifiers as for lambda functions: they bind as far as possible.

depending on $t$, an arbitrary choice of names are $\mathrm{top}$ prepended to the original variable. The axiom then looks like this:

$$\text{II.}' \qquad \forall\, t.\, \mathrm{fork}(\mathrm{topl}(t), \mathrm{topx}(t), \mathrm{topr}(t))) \neq t \rightarrow \mathrm{top}(t) = \bot$$

For another function, like `mirror` above, one of the skolemised functions could be called $\mathrm{mirrorl}$. Since axioms of injective constructors are also added, a theorem prover could, in some steps, conclude that $\mathrm{mirrorl}(\mathrm{fork}(l, x, r)) = \mathrm{topl}(\mathrm{fork}(l, x, r)) = l$. Instead such skolemised *selector functions* are introduced manually. For the `Fork` constructor let us call them $\mathrm{fork}_0$, $\mathrm{fork}_1$ and $\mathrm{fork}_2$, and their axioms are:

$$\begin{aligned}
&\text{I.} && \forall\, l, x, r.\, \mathrm{fork}_0(\mathrm{fork}(l, x, r)) = l \\
&\text{II.} && \forall\, l, x, r.\, \mathrm{fork}_1(\mathrm{fork}(l, x, r)) = x \\
&\text{III.} && \forall\, l, x, r.\, \mathrm{fork}_2(\mathrm{fork}(l, x, r)) = r
\end{aligned}$$

The translation of `top` with these selector functions is:

$$\begin{aligned}
&\text{I.} && \forall\, l, x, r.\, \mathrm{top}(\mathrm{fork}(l, x, r)) = x \\
&\text{II.} && \forall\, t.\, (\mathrm{fork}(\mathrm{fork}_0(t), \mathrm{fork}_1(t), \mathrm{fork}_2(t)) \neq t) \rightarrow \mathrm{top}(t) = \bot
\end{aligned}$$

As a nice side effect, injectivity of constructors is implied the axioms of the skolemised selector functions. Assume we have $\mathrm{fork}(l_0, x_0, r_0) = \mathrm{fork}(l_1, x_1, r_1)$ then the first selector, $\mathrm{fork}_0$, gives $l_0 = l_1$. Analogously the second and the third give $x_0 = x_1$ and $r_0 = r_1$, respectively. Thus selector axioms are added in place of the injectivity axioms.

With the bottom constant in the theory, the axioms disjointedness are effected by this. It can be seen as an implicit constructor for every data type. For the `Tree` data type the axioms are:

$$\begin{aligned}
&\text{I.} && \forall\, l, x, r.\, \mathrm{fork}(l, x, r) \neq \mathrm{leaf} \\
&\text{II.} && \forall\, l, x, r.\, \mathrm{fork}(l, x, r) \neq \bot \\
&\text{III.} && \bot \neq \mathrm{leaf}
\end{aligned}$$

Now we have a good idea how to translate pattern matching, but in Haskell we can pattern match almost everywhere! How would we proceed to translate a function like this, taken from the implementation of `scanr` from the `Prelude`?

```
scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  f x q : qs
                     where qs = scanr f q0 xs
                           q = case qs of
                                 q : _ -> q
```

There is both pattern matching directly on the arguments, but also pattern matching in a case statements in the where function `q`. There can also be pattern matching in lambdas. To help with these difficulties, we define an intermediate language in the next section.

## 2.3 The Intermediate Language

To address the difficulties of pattern matching elsewhere than in the arguments of a function, a small intermediate language was designed that can only do pattern matching at a

very controlled location: in a case statement that is the entire body of a function, and all arms are simple expressions consisting of function and constructor applications and variables. As a first step, Haskell is translated to this language. This process includes several simplifications; pattern matching at other locations are moved to new top level definitions. Functions defined in let and where are raised to the top level, with the necessary variables in scope as additional arguments. The same is done for sections and lambda functions.

The BNF for the language is this:

| | | | |
|---|---|---|---|
| Variables | $x$ | | |
| Functions | $f$ | | |
| Constructors | $C$ | | |
| Type variables | $\tau$ | | |
| Type constructors | $T$ | | |
| Declarations | $decl$ | $::= f\ \overline{x}$ = $body$ | function declaration |
| | | $\mid f\ ::\ t$ | type signature |
| | | $\mid$ `data` $T\ \overline{\tau}$ = $\overline{C\ \overline{t}}$ | data type declaration |
| Function body | $body$ | $::=$ `case` $e$ `of` $\overline{alt}$ | case body |
| | | $\mid e$ | expression body |
| Expressions | $e$ | $::= x$ | variable |
| | | $\mid f\ \overline{e}$ | function application |
| | | $\mid C\ \overline{e}$ | constructor application |
| Alternative | $alt$ | $::= pat \to e$ | branch without guard |
| | | $\mid pat$ `\|` $e \to e$ | branch with guard |
| Pattern | $p$ | $::= x$ | pattern variable |
| | | $\mid C\ \overline{p}$ | constructor pattern |
| Types | $t$ | $::= \tau$ | type variable |
| | | $\mid t\ \to\ t$ | function type |
| | | $\mid T\ \overline{\tau}$ | type constructor application |
| Programs | $prog$ | $::= \overline{decl}$ | |

This language is a strict subset of Haskell, and inherits its semantics. Repeated entities in the BNF are notated with an $\overline{\text{overline}}$. Data declarations are needed to generate axioms of disjointedness and selectors. Type signatures are ignored in the translation, but the proof techniques introduced later use this information.

A function is just a function name with a number of variables, and then a function body, which is either an expression of variables, functions and constructors, or a case statement with an expression scrutinee. Branches consists of a pattern, possibly with nested uses of constructors, and an optional guard, and in the arm is an expression. A notable exception from ordinary core languages is made here: nested cases are not allowed. This restriction will aid the translation. Nested cases will be lifted to top level definitions.

Now we need to distinguish between two translations: the intermediate translation from Haskell to the intermediate language, and the logic translation from this language to first order logic. The next section explains the first part.

## 2.4 The Intermediate Translation

This section describes the transformation from Haskell to the intermediate language. The main transformations are top level lifting of lambdas, local definitions and restricting pattern matching only in case statements.

**Argument pattern matching** A function that does pattern matching will be translated to one that takes in unmatched arguments and with a case in the body. The `mirror` function above is thus translated to this:

```
mirror :: Tree a -> Tree a
mirror t = case t of
   Fork l x r -> Fork (mirror r) x (mirror l)
   Leaf       -> Leaf
```

If you do pattern matching on several arguments, the scrutinee in the case will be a tuple of all the arguments.

**Local definitions** Where-clauses and let-expressions are raised to the top level, with all necessary variables as arguments. This example of an accumulator definition of multiplication of Peano natural numbers needs such a rewrite:

```
(*) :: Nat -> Nat -> Nat
x * y = go Zero x where go acc Zero    = acc
                       go acc (Suc n) = go (acc + y) n
```

The go function has the y in scope but not as argument so it is appended to the arguments to the top level lifted version of go:

```
go acc Zero    y = acc
go acc (Suc n) y = go (acc + y) n y

x * y = go Zero x y
```

Finally the pattern matching in go is translated to use a case expression:

```
go acc x y = case x of
     Zero  -> acc
     Suc n -> go (acc + y) n y
```

A similar translation is done for let expressions.

**Lambda functions** These are translated to top level definitions. Take this example of defining `fmap` in terms of the functions from the `Monad` type class as `liftM`:

```
liftM f m = m >>= \x -> return (f x)
```

In the lambda, `f` is a free variable so it becomes an argument to the new top level function called `lambda` below:

```
lambda f x = return (f x)

liftM f m = m >>= lambda f
```

An analogous translation as is done for lambdas is done for operator sections.

This concludes the translation to the intermediate language, and the rest of this chapter concentrates on the translation from it to first order logic. Note that sometimes code will for clarity be written with pattern matching on arguments directly, but it is implicitly assumed to be translated to a pattern matching in a case statement.

## 2.5 Pattern Matching Revisited

**Overlapping patterns** These needs to be removed to prevent generation of inconsistent theories. Example:

```
overlap :: Bool -> Bool
overlap b = case b of
              True -> True
              True -> False
```

Certainly, this cannot be translated to:

$$\begin{aligned} &\text{I.} && \text{overlap}(\text{true}) = \text{true} \\ &\text{II.} && \text{overlap}(\text{true}) = \text{false} \\ &\text{III.} && \forall\, b.\, b \neq \text{true} \rightarrow \text{overlap}(b) = \bot \end{aligned}$$

Reflexivity gives $\text{overlap}(\text{true}) = \text{overlap}(\text{true})$, transitivity of the equalities in the axioms I and II gives that $\text{true} = \text{false}$. Together with the axiom from disjoint constructors, $\text{true} \neq \text{false}$, we have a contradiction.

In Haskell, pattern matching is done from top to bottom of the definition, making the second match of `True` to never occur. Thus, the translation removes all patterns that are instances of a pattern above.

**Nested patterns and bottoms** The translation also handles patterns in more than one depth. At every location in a pattern where a constructor is matched against, a pattern with bottom at that spot is also added, defined to bottom. This Haskell function `even` determines if a list is of even length:

```
even :: List a -> Bool
even (Cons x (Cons y ys)) = even ys
even (Cons x xs)          = False
even Nil                  = True
```

For the sake of readability we use the constructors `Cons` and `Nil` for lists are used since the selectors $:_0$ and $:_1$ for the normal cons are hard to read.

Here, `even` should return $\bot$ when it is evaluated with an argument constructed with neither `Cons` nor `Nil` (recall that the logic is untyped.) This undefined value should also be returned if applied to `Cons x` $\bot$ for some `x`, since the `Cons` constructor is matched again on depth two. So there are two different situations at each depth. One is if there is a match any pattern (for `even`, it is the variable `xs` in the second pattern), new patterns are added that matches $\bot$. The other is if there is no wild pattern, a new one is added that goes to $\bot$.

No type information is needed to do this insertion, only inspection of the patterns is required. Could the bottoms be seen in the definition it would look like this:

```
even :: List a -> Bool
even (Cons x (Cons y ys)) = even ys
even (Cons x ⊥)           = ⊥
even (Cons x xs)          = False
even Nil                  = True
even _                    = ⊥
```

Haskell's behaviour of matching patterns from top to bottom is justified with implications ensuring the *upward agreement*. The axioms for this definitions are:

I. $\forall x, y, ys.\, \mathrm{even}(\mathrm{cons}(x, \mathrm{cons}(y, ys))) = \mathrm{even}(ys)$

II. $\forall x.\, \mathrm{even}(\mathrm{cons}(x, \bot)) = \bot$

III. $\forall x, xs.\, xs \neq \mathrm{cons}(\mathrm{cons}_0(xs), \mathrm{cons}_1(xs)) \wedge xs \neq \bot \rightarrow \mathrm{even}(\mathrm{cons}(x, xs)) = \mathrm{false}$

IV. $\mathrm{even}(\mathrm{nil}) = \mathrm{true}$

V. $\forall xs.\, xs \neq \mathrm{nil} \wedge xs \neq \mathrm{cons}(\mathrm{cons}_0(xs), \mathrm{cons}_1(xs)) \rightarrow \mathrm{even}(xs) = \bot$

The implications due to upward agreement are present in axioms III and V. This is needed for all wild patterns.

## 2.6 Functions as Arguments

In Haskell, functions readily take other functions as arguments, and functions can also be partially applied. To get the same behaviour in logic, each function is assigned a *function pointer*, and a new binary function is added to the language, written infix as $@$. For instance if there is a binary function `plus` then a constant called $\mathrm{plus.ptr}$ is added to the theory and this axiom:

$$\forall x, y.\, (\mathrm{plus.ptr}\, @\, x)\, @\, y = \mathrm{plus}(x, y)$$

When a function is only partially applied, or a function argument is applied, $@$ is used. Consider this Prelude function `iterate`:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

It is translated with $@$ in the following way, with the cons constructor `:` written infix:

$$\forall f\, x\, .\, \mathrm{iterate}(f, x) = x : \mathrm{iterate}(f, f\, @\, x)$$

Translaning all function applications could be done using $@$. However, this approach slows down the theorem provers significantly, so an optimisation is crucial. Functions will be regarded as having arity equal to the number of arguments on the left hand side in their definition. Should a function not get all of its arguments, appropriate use of $@$ is added, as in this function which increments all elements of the list by one using `map`:

```
incr = map (plus one)
```

As `incr` is written $\eta$-reduced, both `map` and `plus` are only partially applied. This is the translated axiom:

$$\text{incr} = \text{map.ptr} \mathbin{@} (\text{plus.ptr} \mathbin{@} \text{one})$$

If `incr` is applied to an argument $xs$, then `incr` is applied to more arguments than it takes, so we add $@$ so the corresponding formula becomes $\text{incr} \mathbin{@} xs$. By substituting the definition of $\text{incr}$ we get $(\text{map.ptr} \mathbin{@} (\text{plus.ptr} \mathbin{@} \text{one})) \mathbin{@} xs$ and the axiom of $\text{map.ptr}$ then equals this to $\text{map}(\text{plus.ptr} \mathbin{@} \text{one}, xs)$.

**Doing the impossible** Although it is not possible to quantify over functions in first order logic, this translation allows universal quantification of functions, allowing a way to reason syntactically about partially applied functions. On the model side, $@$ gives a way to interpret functions and universally quantify over them. If the function has a pointer defined, it just constrains $@$ on that pointer to do the same as the function.

## 2.7 Guards

Guards are treated similar to pattern matching. If a guard expression evaluates to `True`, that branch is picked. The expression could also evaluate to $\bot$, and then the result should be $\bot$. Let us consider the `filter` function:

```
filter :: (a -> Bool) -> List a -> List a
filter p (Cons x xs) | p x = Cons x (filter p xs)
filter p (Cons x xs)       = filter p xs
filter p Nil               = Nil
```

To translate this to logic it is needed to ensure that if `p x` evaluates to $\bot$, then so should the function. The axioms look like this:

I. $\forall p, x, xs.\, (p \mathbin{@} x) = \text{true} \rightarrow \text{filter}(p, \text{cons}(x, xs)) = \text{cons}(x, \text{filter}(p, xs))$

II. $\forall p, x, xs.\, (p \mathbin{@} x) = \bot \rightarrow \text{filter}(p, \text{cons}(x, xs)) = \bot$

III. $\forall p, x, xs.\, (p \mathbin{@} x) \neq \text{true} \wedge (p \mathbin{@} x) \neq \bot \rightarrow \text{filter}(p, \text{cons}(x, xs)) = \text{filter}(p, xs)$

IV. $\text{filter}(p, \text{nil}) = \text{nil}$

V. $\forall xs.\, xs \neq \text{nil} \wedge xs \neq \text{cons}(\text{cons}_0(xs), \text{cons}_1(xs)) \rightarrow \text{filter}(p, xs) = \bot$

## 2.8 Summary

The translation of different Haskell concepts is summarised in the table below:

| Haskell | First Order Logic |
|---|---|
| function | function or constant |
| constructor | function or constant |
| data type | disjoint constructors and selector axioms |
| pattern matching | overlap removal, bottoms insertion, upward agreement |
| guards | equality to true and bottom and upward agreement |
| partial application | $@$ on pointer constant |
| partially applied function | pointer constant and $@$ rule |
| sections, lambdas, let | new functions with variables in scope as arguments |

## 2.9 Domain Theory

This section is stand alone, and could be skipped especially if you already know the basics of domain theory: complete partial orders, monotonicity and continuity. It explains these concepts and discusses how it can be used to verify the translation, furthermore it is used as a reference in for later sections that rely on concepts from domain theory.

The values of every data type are ordered on how much "information" they contain. The least element bottom, denoted $\bot$, contains least information. It corresponds to all kinds of crashes in Haskell; use of `undefined`, non-termination and non exhaustive pattern matches. Different constructors hold different information, so they are not related by the ordering; this ordering is partial. Such orders are reflexive, transitive and anti-symmetric. The ordering is written $\sqsubseteq$, sometimes with a subscript indicating the type.

For the `Bool` data type the partial order can be drawn as a diagram in Figure 2.1. From the picture it is understood that $\bot$ is the least element, and the line from it to `False` means that $\bot \sqsubseteq$ `False`, since $\bot$ is below `False`. Correspondingly for `True`, the diagram tells us that $\bot \sqsubseteq$ `True`. It can also been seen that `True` $\not\sqsubseteq$ `False`; they are unrelated since there is no line between them. This kind of diagram is called a Hasse Diagram.

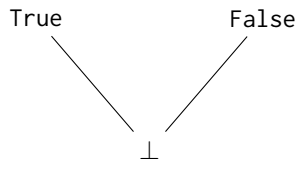


Figure 2.1: The order of `Bool` values.

For tuples and other constructors that take other data types as parameters, the ordering is:

$$(x_0, y_0) \sqsubseteq_{(a,b)} (x_1, y_1) \quad \text{iff} \quad x_0 \sqsubseteq_a x_1 \text{ and } y_0 \sqsubseteq_b y_1$$

The Hasse Diagram for the `(Bool,Bool)` values can be seen in Figure 2.2. Here `True` is abbreviated for `T` and similarly for `False`. It is not flat as the one for `Bool`; it can be seen as three dimensional. On the lowest layer the only value is $\bot$, on the next layer $(\bot, \bot)$. Above that the tuples with one $\bot$, and finally the total values at the top.

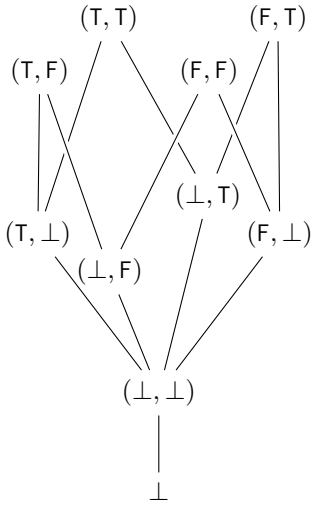


Figure 2.2: `(Bool,Bool)` order.

### 2.9.1 Monotonicity

An important property all safe Haskell functions have is that they are monotone with respect to this ordering.

**Definition** A function $f$ is *monotone* iff

$$\forall x, y. \quad x \sqsubseteq y \quad \Rightarrow \quad f(x) \sqsubseteq f(y).$$

Monotonicity can be understood in many ways. One way to see it is if you have two inputs to a function, one containing *less* information that the other, it is impossible to return *more* information from the input with less information.

One simple example of a consequence of monotonicity is the impossibility to make a function `isBottom :: a -> Bool`, returning `True` if the argument is bottom, and `False` otherwise:

$$\begin{aligned}
&\texttt{isBottom} :: \texttt{a} \to \texttt{Bool} \\
&\texttt{isBottom}\ \bot = \texttt{True} \\
&\texttt{isBottom}\ x\ = \texttt{False}, \qquad x \neq \bot
\end{aligned}$$

Since $\bot \sqsubseteq x$ for any $x$, then by monotonicity we must necessarily have

$$\texttt{isBottom}\ \bot \sqsubseteq \texttt{isBottom}\ x.$$

Take any non-bottom $x$, then this equation gives $\texttt{True} \sqsubseteq \texttt{False}$, which is false. Hence `isBottom` is not monotone.

### 2.9.2 Continuity

Another domain theoretic property that Haskell functions have is that they are continuous. This is a property that gives us insight in how functions behave on infinite input. To describe this, we need to consider the partial order of a data type with infinite values. The prime candidate `data Nat = Zero | Succ Nat` is used and Hasse Diagram can be seen in Figure 2.3.

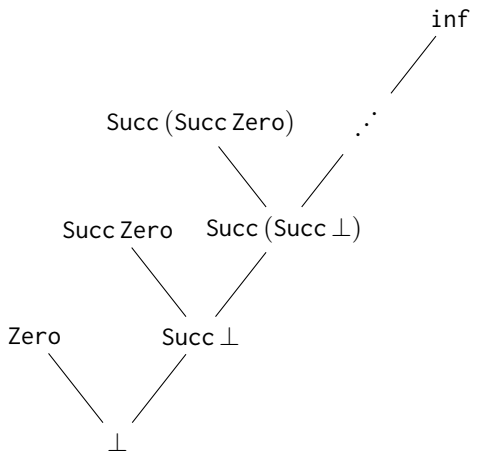


Figure 2.3: The (complete) partial order for `Nat`, with `inf = Succ inf`.

At the top we have the infinite value `inf`, defined in Haskell as `inf = Succ inf`. Here `inf` is the *limit* of a sequence of $\sqsubseteq$ inclusions. We will consider $\omega$-chains, chains with the same number of elements as $\omega$, the natural numbers.

An example of a chain which has the infinite natural number `inf` as its limit is:

$$\bot \sqsubseteq \mathsf{Succ}\,\bot \sqsubseteq \mathsf{Succ}\,(\mathsf{Succ}\,\bot) \sqsubseteq \mathsf{Succ}\,(\mathsf{Succ}\,(\mathsf{Succ}\,\bot)) \sqsubseteq \cdots$$

This chain could succinctly be written $\langle \mathsf{Succ}^n\,\bot \rangle_{n\in\omega}$. Here $\mathsf{Succ}^n$ means $n$ applications of the `Succ` constructor. The limit is written $\sqcup_{n\in\omega}(\mathsf{Succ}^n\,\bot)$ and is equal to `inf`, where $\sqcup$ is the least upper bound. All elements in the chain satisfy the property of being less than or equal to the limit: $\mathsf{Succ}^n\,\bot \sqsubseteq \mathtt{inf}$.

A partial order is a complete partial order iff there is a limit for every $\omega$ chain. All data types in Haskell are complete partial orders. Notice that the number of $\omega$ chains are infinite for all non-empty data types. For booleans, there are chains that start with *bottom* and shifts to `True` or `False` at some point. For the natural number data type, it is not necessary to add exactly one `Succ` constructor at every inclusion. For lists, the chains can be even more involving:

$$\mathtt{True}:\bot:\bot \sqsubseteq \mathtt{True}:\bot:\mathtt{True}:\bot \sqsubseteq \mathtt{True}:\mathtt{False}:\mathtt{True}:\bot \sqsubseteq \cdots$$

This could be the starting chain of an infinite list of booleans with `True` and `False` interleaved. Now, let us define continuity.

**Definition** A function $f$ is *continuous* iff it is monotone and preserves the $\sqcup$ of all $\omega$-chains: i.e. assume any chain $\langle x_n \rangle_{n\in\omega}$, then:

$$\sqcup_{n\in\omega}\,(f\,x_n) = f\,(\sqcup_{n\in\omega}\,x_n)$$

Just as with monotonicity, there are several ways to interpret this. One way is to say that what a function does on a chain, it must also do on the chain's limit, as with `map` on increasingly longer lists. Another is to say that a function cannot produce finite output by inspecting infinite input: there is no function `isFinite :: [a] -> Bool` returning `True` on finite lists and `False` on infinite lists. On the increasing chain

$$\bot \sqsubseteq x_0:\bot \sqsubseteq x_0:x_1:\bot \sqsubseteq \cdots$$

the function `isFinite` returns `True` (or $\bot$), but the limit should return `False`, so this is not a continuous function.

An interesting formulation of Church's Thesis in terms of continuity is given by Plotkin Plotkin (1983):

*A function is continuous iff it is physically feasible.*

This means that all computable functions are continuous, and the other way around. The conclusion for us is that all Haskell functions are continuous.

### 2.9.3 Unsafe Haskell

In GHC, you can use `unsafePerformIO` and `catch` from `Control.Exception` and other tricks to "unsafely" catch errors (bottoms). With this machinery it is possible to write a function `isBottom :: a -> Bool` to catch calls to `undefined`, pattern match failures, etcetera. In addition, some non-termination can also be determined in Haskell because of the *blackhole* run time object that replaces a *thunk* that is being currently evaluated. But this machinery does

not and indeed cannot cover all non terminating functions because of the undecidability of the halting problem.

The domain theoretic results remain; in this setting $\perp$ can be seen as another, albeit inconveniently inspected, constructor to every data type. All patterns are exhaustive: every function has an implicit match any pattern to $\perp$. Then we add a *true* least element to the domain denoting the uncatchable bottoms; undeterminable non termination. With this setting all Haskell functions are continuous with respect to the *true* bottoms. However, proving properties about such programs is difficult as the order of evaluation now matters. For the rest of this thesis, only pure and "safe" Haskell functions are considered.

### 2.9.4 Monotonicity as Verification

A way to verify the translation is to add axioms to the generated theory describing the $\sqsubseteq$ relation, and axioms that asserts that each function is monotone. An automated theorem prover could not easily show that it is a satisfiable theory since it will normally only have infinite models. However, a long run without any counter model could be seen as a witness for a successful translation in this respect. On the other hand, continuity is a concept that is hard to express in first order logic. We can come close with an axiomatisation of set theory, but it is beyond the scope of this thesis.

## 2.10 Future Work

Haskell is a big language, and translating it all in one go is a daunting task. Therefore, some restrictions were settled to be able to focus on proving rather than translating. The goal was to add enough of the Haskell language to enable to prove interesting properties, but much of the widely available sugar in Haskell was omitted since it does not add extra expressibility. This means that list comprehensions and are not supported but can be added by their respective rewriting rules. `Type` definitions should be unrolled, so they could be used in the signature for properties. Type classes is probably the most interesting thing to add, and is discussed in Section 5.3 in future work.

Another interesting but omitted feature are the built-in types `Int`, `Integer`, `Double`, `Char`, etc. For `Integer` appropriate axioms could be added that hold for $\mathbb{Z}$, the canonical infinite discretely ordered commutative ring. The other data types do not enjoy such well behaved properties because of different bit sizes and overflow and precision errors.

Syntactic features for controlling lazy and strict evaluation namely irrefutable patterns, `seq` and bang patterns, and richer pattern matching in form of pattern bindings are discussed below.

### 2.10.1 Irrefutable Patterns and Pattern Bindings

Irrefutable patterns can be defined in terms of projections, examples are `fst`, `snd`, `head`, `fromJust` defined in the standard library. Each irrefutable pattern is translated to a constant, and when you use the variables in the pattern, you translate it to appropriate use of projections. One example is the translation of the `uncurry` function:

```
uncurry f ~(x,y) = f x y        ⇒    uncurry f t = f (fst t) (snd t)
```

The irrefutable pattern `~(x,y)` is replaced with the new constant `t`, and in the body of the function, `x` is replaced with the strict projection `fst t`, and similarly for `y`.

Top level patterns, also called pattern bindings, can also be written in terms of such projections. The whole pattern is replaced with a constant, and when the variables from the pattern are used, you again replace it with projections. This is how it could look for a simple `fromJust . lookup` implementation:

```
unsafeLookup x xs = v           ⇒    unsafeLookup x xs = fromJust t
  where Just v = lookup x xs           where t = lookup x xs
```

The strict projections would not rely on the user having `fst` or `fromJust` in scope, they can automatically be generated by inspection of the data type definition.

### 2.10.2 Bang Patterns and `seq`

The translations for bang patterns and `seq` are also straightforward. `seq` defined by bang patterns is:

```
seq :: a -> b -> b
seq !x y = y
```

The axioms for a translation of `seq` needs to ensure that if `x` evaluates to $\bot$, then `seq x` also evaluates to $\bot$. The two axioms for this functions are:

$$\text{I.} \quad \forall y.\, seq(\bot, y) = \bot$$
$$\text{II.} \quad \forall x, y.\, x \neq \bot \rightarrow seq(x, y) = y$$

Either you implement bang patterns in this fashion, or you do the same translation as GHC for bang patterns: for each strict variable, you add a `seq` for that variable for the expression of that pattern, and you simply add the axioms for `seq` to the theory if the program uses it or bang patterns. For data types with strictness fields one proceeds by adding `seq` when constructing elements.

### 2.10.3 Pattern Guards

Patterns guards is a GHC specific extension to Haskell which allows arbitrary pattern matching on the result from an expression in a guard. An example is this elaboration of the `lookup` function from the `Prelude`, which applies a function to the element, if found:

```
transformLookup :: Eq k => k -> [(k,v)] -> (k -> v -> b) -> Maybe b
transformLookup k xs f | Just v <- lookup k xs = Just (f k v)
                       | otherwise             = Nothing
```

If the look up returns `Just`, `v` is already bound and can be used in the expression of the right hand side. This is very similar to normal guards, as they are a special case of pattern guards: the guard `f x | p x` is expressed as `f x | True <- p x`. The current translation of guards checks if `p x` is `True`, and then "picks" this branch, or is $\bot$. This could be done for constructors, bottoms would need to be added in the guard branches as is currently done for ordinary patterns.

# Chapter 3

# Proof Techniques

The proof developed in this thesis is called `hip`, the Haskell Inductive Prover. To use it, properties are inserted to the source code where the definitions of the relevant functions are. As an example, this is how the associativity of list concatenation can be entered:

```
import Prelude ()
import AutoPrelude

(++) :: [a] -> [a] -> [a]
[]   ++ ys = ys
x:xs ++ ys = x:(xs ++ ys)

prop_app_assoc :: [a] -> [a] -> [a] -> Prop [a]
prop_app_assoc xs ys zs = xs ++ (ys ++ zs) =:= (xs ++ ys) ++ zs
```

There is no module system implemented, so all definitions must be present in one source file. The usual imports from `Prelude` are hidden. The arguments are universally quantified, so this property means:

$$\forall\, xs, ys, zs.\, xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

The equality is interpreted as equality on the constructor level: two values are identified if they are constructed with the same constructor and their arguments are also equal. The function `=:=` and the type constructor `Prop` come from the imported module `AutoPrelude`. The type argument to `Prop` is the type of the equality, so the type of `=:=` is:

```
(=:=) :: a -> a -> Prop a
```

The type signature of properties cannot be omitted as this is used for some proof techniques. Using `hip` is then a matter of saving the file, for instance to `ListProps.hs`, and executing this statement in your favourite terminal:

```
hip ListProps.hs
```

The program will report to you which proof methods succeeded on proving this property, if any. In this case, it is provable with all three inductive techniques.

By importing `AutoPrelude` the properties in the file are also testable with QuickCheck, given that there are appropriate `Eq` and `Arbitrary` instances provided.

The rest of this chapter explains the different proof methods supported in this tool. Some properties are a direct consequence from the definitions in your file. How to prove such properties is described in Section 3.1 about definitional equality. The three other techniques uses induction in different ways. Structural Induction is explained in Section 3.2, which uses the structure of the data types a property quantifies over. Another method which does induction on the recursive structure of the program is Fixed Point Induction, introduced in Section 3.3. A more subtle way of induction is used in the Approximation Lemma, Section 3.4, where the structure of the data type of the equality is approximated.

## 3.1 Definitional Equality

Some properties cannot or need not use induction or some more sophisticated technique, since they are true by definition. Examples are properties for fully polymorphic functions such as this definition of `id` in the SK-calculus:

```
s f g x = f x (g x)
k x y = x
id x = x

prop_skk_id :: Prop (a -> a)
prop_skk_id = s k k =:= id
```

The generated conjecture needs to be stated with function pointers as this:

$$(\text{s-ptr} \mathbin{@} \text{k-ptr}) \mathbin{@} \text{k-ptr} = \text{id-ptr}$$

However, it is not provable in this form, but needs an axiom introduced in the next section.

### 3.1.1 Extensional Equality and `seq`

To prove the previous property we also need to have extensional equality, postulated with this following axiom:

$$\forall f, g.\, (\forall x.\, f \mathbin{@} x = g \mathbin{@} x) \rightarrow f = g$$

This axiom identifies function pointers and functions composed with @. One problem with extensional equality in Haskell, is that the presence of `seq` weakens it. `seq` is a built in function with the following behaviour:

```
seq :: a -> b -> b
seq ⊥ y = ⊥
seq x y = y,    x ≠ ⊥
```

It forces the first argument to weak head normal form and returns the second. For our purposes, it is only important if the first argument is $\bot$, then the function also returns $\bot$ as it is strict in its first argument.

With `seq` it is possibly break extensional equality and distinguish between functions that are otherwise observationally equal. An example of a pair of such functions are:

```
f = ⊥
g = \x -> ⊥
```

Because `seq f ()` evaluates to $\bot$, and `seq g ()` evaluates to `()`, but on any argument `f` and `g` gets, they both return $\bot$. Here we also need an extra axiom, which says that anything applied to $\bot$ is $\bot$:

$$\forall x. \bot \mathbin{@} x = \bot$$

However, `seq` is the only function that can tell such functions apart, so we will ignore its presence in Haskell. In the future, there could be added as a switch `--enable-seq`, which weakens extensional equality appropriately.

If extensional equality is assumed we also have the property that `Prop (a -> b)` is equivalent to `a -> Prop b`, by letting the property have an extra argument that is applied to the left and right hand side of the equality. This has two benefits. Firstly, it can trigger other proof methods should `a` or `b` be concrete types (the former for induction and the latter for approximation lemma). Secondly, for many properties the extensionality axiom which introduces extra steps in the proof search is not needed.

The property about `s k k` is then instead translated to:

$$\forall x. \mathrm{s}(\text{k-ptr}, \text{k-ptr}, x) = \mathrm{id}(x)$$

This gives a little less unnecessary overhead to the theorem provers.

### 3.1.2 Future Work: Concrete Concerns

This only works on non-concrete types because of the way bottoms are added. One example of such a problem is this standard definition of `&&`, and a property stating its right identity:

```
True  && a = a
False && _ = False

prop_or_right_identity :: Bool -> Prop Bool
prop_or_right_identity x = x && True =:= x
```

The translation of `&&` makes any element in the domain that is not the introduced constants $\mathrm{false}$ or $\mathrm{true}$ for `Bool`'s constructors, equal $\bot$. Now consider the translation of the property:

$$\forall x. x \mathbin{\&\&} \mathrm{true} = x$$

Now this equation is false, take a model with another element $\diamond$ in the domain:

$$\diamond \mathbin{\&\&} \mathrm{true} = \bot$$

The consequence of examples like the one above is that the proof principle of definitional equality is only used on abstract types, as they cannot be strict without `seq`. Should any argument be concrete, this technique is not applied.

Do not fear: the property above is trivially proved by induction, as induction for `Bool` and other non recursive data types degenerate into mere case enumeration. However, it can only be seen as a meta theorem as it cannot simply be added as a lemma in the theory. This phenomena is described in more detail in Section 5.2.1. Structural induction is described in detail in the next section.

## 3.2 Structural induction

Properties that are true merely by rewriting the definitions are neither abundant nor especially interesting. The fundamental concept of constructors and pattern matching in Haskell is closely related to induction. The values of an argument with a concrete type can only range over the data type's different constructors. Such a case analysis combined with induction is especially strong.

Induction is a very fundamental concept of mathematics and is directly or indirectly an axiom in axiomatisation of mathematics. We shall take the Peano Arithmetic, $\mathcal{PA}$, axioms for true. This theory has the natural numbers as standard model with a small vocabulary consisting only of the constant $0$, the unary successor function $s$, and binary plus and multiplication. Induction in $\mathcal{PA}$ allows proving properties that hold for all natural numbers. The first proof obligation in a proof by induction is to prove that the property holds for zero. The second is if it holds for an arbitrary but fixed number then it must hold for its successor. The induction schema is formally written as this:

$$\frac{\overbrace{P(0)}^{\text{base}} \qquad \overbrace{\forall x.\ \underbrace{P(x)}_{\text{hypothesis}} \rightarrow \underbrace{P(s(x))}_{\text{conclusion}}}^{\text{step}}}{\forall x.\, P(x)}$$

Different parts of this rule has been highlighted that are common names for induction proofs: the induction base, the induction step with the assumed hypothesis and the obligated conclusion. This is a axiom *schema* since it is not possible to quantify over a predicate in first order logic. Rather, it is a infinite set of axioms, one for each (well-formed) formula instantiated in place for $P$. Generally, theorem provers do not instantiate schemata themselves so it has to be done manually, with an appropriate formula for $P$. In this thesis, the predicates will be equalities that depend on the argument to $P$. For example, to show that two functions $f$ and $g$ are equal, $P(x)$ is defined to be $f(x) = g(x)$.

To prove a property with induction in Haskell, induction can be carried out on the number of constructors in a value. It is possible to get many base cases, one for each non-recursive constructor and then a step case for each recursive constructor. As an example, we can consider one of the simplest recursive data structure in Haskell, the Peano Natural numbers in Haskell, defined as `data Nat = Zero | Succ Nat`. This axiom schema is used and follows the same structure as in $\mathcal{PA}$:

$$\frac{P(\texttt{Zero}) \qquad \forall x.\, P(x) \rightarrow P(\texttt{Succ}\ x)}{\forall x.\, x \text{ finite and total} \rightarrow P(x)}$$

In the schema above, a total value means that it does not contain any bottoms, whereas being only finite is weaker; values such as $\texttt{1:2:}\bot$ are regarded as finite, but not total.

Any data type gives rise to a structural induction schemata. These are the axiom schemata for lists and the `Tree` type defined in Section 2.1:

$$\frac{P(\texttt{[]}) \qquad \forall x, xs.\, P(xs) \rightarrow P(x\texttt{:}xs)}{\forall xs.\, xs \text{ finite and total} \rightarrow P(xs)} \qquad \frac{P(\texttt{Empty}) \qquad \forall l, x, r.\, P(l) \wedge P(r) \rightarrow P(\texttt{Fork}\ l\ x\ r)}{\forall t.\, t \text{ finite and total} \rightarrow P(t)}$$

For indexed sum of products data types the translation is straightforward: given a data type `T`, then for every constructor `K`, all recursive arguments of type `T` to `K` are hypotheses.

### 3.2.1 Partial and Infinite Values

It is also possible to use induction to prove properties about all infinite and partial values of a data types. The predicate must then be *admissible*, and is formally defined below. Informally, it means that the predicate preserves $\omega$-chains of $\sqsubseteq$. Such chains are described in more detail in Section 2.9, where this example of such a chain was given:

$$\bot \sqsubseteq \texttt{Succ}\ \bot \sqsubseteq \texttt{Succ}\ (\texttt{Succ}\ \bot) \sqsubseteq \texttt{Succ}\ (\texttt{Succ}\ (\texttt{Succ}\ \bot)) \sqsubseteq \cdots$$

This chain has the limit `inf`, defined as `inf = Succ inf`. If $P$ is an admissible predicate, then if $P$ holds for every element in that chain then $P(\texttt{inf})$. We can now define admissible.

**Definition** A predicate $P$ is *admissible* for every $\omega$-chain $\langle x_n \rangle_{n \in \omega}$, this holds:

$$(\forall\, n.\, P(x_n)) \Rightarrow P(\sqcup_{n \in \omega}(x_n))$$

The properties concerned about in this thesis are about equality, and equality of continuous functions is can be proved to be admissible. Assume two such functions $f$ and $g$, and define the predicate as $P(x) :\Leftrightarrow f(x) = g(x)$. Take any chain $\langle x_n \rangle_{n \in \omega}$, and assume $P$ holds for every element in the chain, then:

$$\begin{array}{lcl}
 & & \{\text{assumption}\} \\
\forall\, n.\, P(x_n) & & \\
 & \Leftrightarrow & \{\text{definition}\} \\
\forall\, n.\, f(x_n) = g(x_n) & & \\
 & \Rightarrow & \{\text{limits}\} \\
\sqcup_n\, (f(x_n)) = \sqcup_n (g(x_n)) & & \\
 & \Leftrightarrow & \{\text{continuity}\} \\
f(\sqcup_n (x_n)) = g(\sqcup_n (x_n)) & & \\
 & \Leftrightarrow & \{\text{definition}\} \\
P(\sqcup_n (x_n)) & &
\end{array}$$

This result generalises to equalities of compositions of continuous functions. One way to show this is to use that the continuous functions over complete partial orders form a category with products and exponentials. These are the necessary components of composition and application.

To conclude: if you want to show that an admissible property holds for partial and infinite values you also need to consider $\bot$ as a constructor for the data type. An example of this is the induction principle for possibly partial and infinite lists:

$$\frac{P(\bot) \qquad P(\texttt{[]}) \qquad \forall\, x, xs.\, P(xs) \to P(x : xs) \qquad P \text{ admissible}}{\forall\, xs.\, P(xs)}$$

### 3.2.2 Generalisations

The structural induction schemata seen so far only uses each constructor once and that does not work for proving properties about functions defined with recursion with a bigger depth. For instance, the induction on Peano numbers in Haskell can be adjusted to a constructor depth two in this way[1]:

$$\frac{P(\bot) \qquad P(\texttt{Zero}) \qquad P(\texttt{Succ Zero}) \qquad \forall\, x.\, P(x) \wedge P(\texttt{Succ}\ x) \to P(\texttt{Succ}\ (\texttt{Succ}\ x))}{\forall\, x.\, P(x)}$$

Given how many constructors you maximally want to unroll for your data type, you have to prove the property for all combinations of constructors up to including that limit, but for an induction step with a conclusion with $i$ constructors, the induction hypothesis is the conjunction of all combinations with strictly less than $i$ constructors.

This unrolling is used to use induction on more than one arguments, by using the tuple constructor. The base and the step cases of induction on two natural numbers is this:

I. $$P(\texttt{Zero}, \texttt{Zero})$$

II. $$\forall\, x.\, P(x, \texttt{Zero}) \to P(\texttt{Succ}\ x, \texttt{Zero})$$

III. $$\forall\, y.\, P(\texttt{Zero}, y) \to P(\texttt{Zero}, \texttt{Succ}\ y)$$

IV. $$\forall\, x, y.\, P(x, y) \wedge P(\texttt{Succ}\ x, y) \wedge P(x, \texttt{Succ}\ y) \to P(\texttt{Succ}\ x, \texttt{Succ}\ y)$$

This induction schema can be used to show the commutativity of natural number addition defined recursively on the first argument. The predicate is defined as $P(x, y) \Leftrightarrow x + y = y + x$. This property can be proved with induction on the first argument, but with two lemmas. These two lemmas follow from the stronger induction schema above:

1. $\forall\, x.\, x + \texttt{Zero} = x$. This corresponds the hypothesis $P(x, \texttt{Zero})$.
2. $\forall\, x, y.\, \texttt{Succ}\ x + y = x + \texttt{Succ}\ y$. Derivable from two of the hypotheses in axiom IV:

$$\begin{aligned}
lhs &= \texttt{Succ}\ x + y && \{\text{definition of } +\}\\
&= \texttt{Succ}\ (x + y) && \{\text{hypothesis } P(x, y)\}\\
&= \texttt{Succ}\ (y + x) && \{\text{definition of } +\}\\
&= \texttt{Succ}\ y + x && \{\text{hypothesis } P(x, \texttt{Succ}\ y)\}\\
&= x + \texttt{Succ}\ y = rhs
\end{aligned}$$

The commutativity of plus does only hold for total values since we have:

$$\bot = \bot + \texttt{Succ Zero} \neq \texttt{Succ Zero} + \bot = \texttt{Succ}\ \bot$$

Unrolling data types in this way to construct hypotheses also work well with data types defined in terms of other data types. An example is the definition of $\mathbb{Z}$ defined with a disjoint union as $\mathbb{N} + \mathbb{N}$, which can be defined as this in Haskell:

[1] In this section, predicates will silently be assumed admissible to avoid cluttering the induction schemata.

```
data Z = Pos Nat | Neg Nat
```

A value of `Pos` $x$ denotes the integer $+x$ and `Neg` $y$ the integer $-(1 + y)$. This interpretation avoids having two values denoting $0$. With the technique above, the induction principle for total `Z` with two constructors is:

$$\frac{\begin{array}{ll} P(\texttt{Pos Zero}) & \forall\, x.\, P(\texttt{Pos}\ x) \wedge P(\texttt{Neg}\ x) \rightarrow P(\texttt{Pos}\ (\texttt{Succ}\ x)) \\ P(\texttt{Neg Zero}) & \forall\, x.\, P(\texttt{Pos}\ x) \wedge P(\texttt{Neg}\ x) \rightarrow P(\texttt{Neg}\ (\texttt{Succ}\ x)) \end{array}}{\forall\, x.\, P(x)}$$

### 3.2.3 Skolemised Hypotheses

Each induction step can be expressed with skolemised variables instead of universally quantified variables. This means that a new constant is introduced for each variable. They can then be referred to in different axioms in the theory. For example, when adding the `Branch` step for an induction proof over the `Tree` data type, the theorem prover could be run like this:

$$T \vdash \forall\, l, x, r.\, P(l) \wedge P(r) \rightarrow P(\mathrm{branch}(l, x, r))$$

Skolemisation of each universally quantified variable allows to instead prove this equivalent judgement:

$$T, P(l), P(r) \vdash P(\mathrm{branch}(l, x, r))$$

This prevents long formulas should there be many hypotheses. It also makes the implementation of the simple induction technique with one argument and one constructor straightforward. For each constructor $C$ with $n$ arguments, we make a new call to a theorem prover with $n$ skolem variables $x_1 \cdots x_n$ and try to prove $P(x_1, \cdots, x_n)$. Additionally, for each argument with assigned variable $x_i$, if the type of this variable is the same as the constructor $C$'s, $P(x_i)$ is added as an axiom to the theory: this an induction hypothesis.

The predicate is inlined: there is no introduction of $P$ and its definition, it is replaced with the property. Suppose we want to prove the property

$$\forall\, t.\, \mathrm{mirror}(\mathrm{mirror}(t)) = t$$

where $t$ : `Tree a`. The `Branch` case generates this call to the theorem prover:

$$\begin{aligned} \mathcal{T}\ &, \mathrm{mirror}(\mathrm{mirror}(x_1)) = x_1 \\ &, \mathrm{mirror}(\mathrm{mirror}(x_3)) = x_3 \\ &\vdash \mathrm{mirror}(\mathrm{mirror}(\mathrm{branch}(x_1, x_2, x_3))) = \mathrm{branch}(x_1, x_2, x_3) \end{aligned}$$

Here $\mathcal{T}$ is the theory for this Haskell program: it includes the definition of `mirror`, along with axioms that `Branch`, `Empty` and $\bot$ are disjoint. Hypotheses for $x_1$ and $x_3$ are added since the first and the third argument for the `Branch` constructor is `Tree a`. For $x_2$, nothing is added since the type of this value is just `a`. For this property, in total three invocations to a theorem prover is carried out: one for each of the two constructors, and one for bottom.

### 3.2.4 Future work

**Exponential Types** It is also possible to use structural induction for exponential data types. Examples are higher-order abstract syntax, another is the Brouwer ordinals defined by Dixon (2005) as:

```
data Ord = Zero | Succ Ord | Lim (Nat -> Ord)
```

The induction schema for this data type is:

$$\frac{P(\bot) \qquad P(\texttt{Zero}) \qquad \forall\, x.\, P(x) \to P(\texttt{Succ}\ x) \qquad \forall\, f.\, (\forall\, x.\, P(f\ x)) \to P(\texttt{Lim}\ f)}{\forall\, x.\, P(x)}$$

Generation of such schemata was never implemented because of too few examples with exponential data types.

**Lexicographic Induction** In the rest of this section only induction in $\mathcal{PA}$ is discussed. Again, $\mathcal{PA}$ has $0$ as zero and the successor function $s$. If you need induction on two variables, to show $\forall\, x, y.\, P(x, y)$, an other way than generalised above is to use lexicographic induction. First you do induction on $x$ to get the proof obligations $\forall\, y.\, P(0, y)$ and $(\forall\, y.\, P(x, y)) \to (\forall\, y.\, P(s(x), y))$. Notice that in the step case, $y$ is universally quantified in both the hypothesis and in the conclusion. In both the base and in the step you can now do induction on $y$ to get this inference rule:

$$\frac{P(0,0) \qquad \dfrac{P(0,y)}{P(0,s(y))} \qquad \dfrac{\forall\, y'.\, P(x,y')}{P(s(x),0)} \qquad \dfrac{\forall\, y'.\, P(x,y') \qquad P(s(x),y)}{P(s(x),s(y))}}{\forall\, x, y.\, P(x, y)}$$

Generation of lexicographic induction schemata is not yet implemented since it was not clear which examples would benefit from it, and the many combinations of how to apply this principle: in the example above induction is first on $x$, and then on $y$, but the other way around is also legit. If a property has $n$ variables that could be used for induction, and you use all variables, you have $n!$ ways of doing lexicographic induction, and this is disregarding subsets of variables. A solution to this problem, as well as the problem with how many constructors you want to use in the generalised induction, is to allow the user to annotate in the source code the desired way to do induction: furthermore, it does not blend well with the rest being a fully automated verification of properties.

**Automated Depth** Another way to do this is to encode different induction ways by means of axioms in the theory itself. A simple example of this is to prove a property $P$ on one Peano natural number, and choose how many base cases in form of $P(0)$, $P(1)$, up to $P(n)$ to get a stronger induction hypothesis $P(x) \wedge P(x+1) \wedge \cdots \wedge P(x+n-1) \to P(x+n)$. It is possible to axiomatise such a "machine" in first order logic as this:

$$\text{Q-0}\ \frac{}{Q(0,x)} \qquad \text{Q-S}\ \frac{P(x) \qquad Q(n,s(x))}{Q(s(n),x)} \qquad \text{+-0}\ \frac{}{0+x=x} \qquad \text{+-S}\ \frac{}{s(n)+x=s(n+x)}$$

$$\frac{\exists\, n\,.\, Q(n,0) \wedge \forall\, x.\, Q(n,x) \to P(n+x)}{\forall\, x.\, P(x)}$$

The $n$ in the existential quantifier is how deep the induction needs to go. $Q$ is a predicate which both gives a suitable induction base and hypothesis for $n$. For the base case, prove $P(0)$, $P(1)$, up to $P(n-1)$. In the induction step,$P(x)$, $P(x)$, up to $P(x+n-1)$ are hypotheses. The consequent is $P(n+x)$, so some axioms for $+$ are borrowed from $\mathcal{PA}$. The degenerate case come from $n=0$:

$$\frac{\text{Q-0}\ \dfrac{}{Q(0,0)} \qquad \forall\, x.\ \text{Q-0}\ \dfrac{}{Q(0,x)} \to \text{+-0}\ \dfrac{P(x)}{P(0+x)}}{\forall\, x.\, P(x)}$$

With $n=1$ the normal $\mathcal{PA}$-induction is obtained.
A little bit of expanding is necessary for $n=2$ to obtain:

$$\frac{\dfrac{\text{Q-S}\ \dfrac{P(0) \qquad \text{Q-S}\ \dfrac{P(1) \qquad \text{Q-0}\ \dfrac{}{Q(0,2)}}{Q(1,1)}}{Q(2,0)} \qquad \forall\, x.\ \text{Q-S}\ \dfrac{P(x) \qquad \text{Q-S}\ \dfrac{P(s(x)) \qquad \text{Q-0}\ \dfrac{}{Q(0,s(s(x)))}}{Q(1,s(x))}}{Q(2,x)} \to \text{+-S}\ \dfrac{\text{+-0}\ \dfrac{P(s(s(x)))}{P(s(s(x+0)))}}{P(1+s(x))}}{P(2+x)} \qquad P(0) \qquad P(1) \qquad \forall\, x.\, P(x) \wedge P(s(x)) \to P(s(s(x)))}{\forall\, x.\, P(x)}$$

The complexity is greatly increased with more irregular data types than natural numbers, and it is unclear how well theorem provers would be able to handle this.

## 3.3 Fixed Point Induction

Structural induction is applicable when at least one argument is of a concrete type, such as lists or trees. There are also properties where all arguments are of abstract types. A canonical example is the map-iterate property::

$$\forall\, f, x.\, \texttt{map}\ f\ (\texttt{iterate}\ f\ x) = \texttt{iterate}\ f\ (f\ x)$$

Here $f$ is any function of type $a \to a$, and $x$ is a value of type $a$. This example is further investigated in Section 3.3.3 below, but it is already clear that this property cannot be proved with structural induction since there is no argument of a concrete type.

Enter fixed point induction. It gives a way of performing induction on the recursive structure of the program. In short, if the property regards a function $f$, the hypothesis is that the property holds for all the recursive calls in the definition of $f$, and the goal is to prove that it holds for $f$. The origin of the name is the use of the fixed point combinator, which is traditionally defined in Haskell as `fix`:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Fixed point induction can prove properties about `fix f` for some `f`, and the next section gives a method to rewrite any recursive functions in terms of `fix`. Then properties about recursive functions in general can be proved by fixed point induction.

The fixed point in question is the solution of the fixed point equation $x = f\ x$. The function `fix` solves this equation: substitute $x$ for `fix` $x$, then the left side evaluates to $f$ (`fix` $f$) in one step, which is then equal to the right side. This is the origin of the name of the combinator `fix`: this is a fixed point of the equation, furthermore it is the least fixed point (Plotkin, 1983).

### 3.3.1 Rewriting Functions in Terms of `fix`

Any self-recursive function can be rewritten in terms of `fix`. This is a mechanical translation which simply prepends a new argument which is used for each recursive call of the function. This is exemplified for the `map` function in Figure 3.1 below. A new argument `m` is introduced and the recursive call uses `m` instead of `map`.

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map f [] = []
```

```
map' m f (x:xs) = f x : m f xs
map' m f [] = []

map :: (a -> b) -> [a] -> [b]
map = fix map'
```

Figure 3.1: The standard definition of `map` and a definition in terms of `fix`

The correctness of the translation of `map` in Figure 3.1 is immediate: `fix map'` evaluates to `map' (fix map')` by the definition of `fix`, which means that the recursive function `m` is `fix map'`. By construction it is then equal to the original definition of `map`.

Generally, for a function `f` with $n \geq 0$ arguments $\mathtt{x}_1$ to $\mathtt{x}_n$ and a body $e[\mathtt{f}, \mathtt{x}_1, \ldots, \mathtt{x}_n]$ with these variables free, the definition in terms of `fix` is:

$$\mathtt{f}\ \mathtt{x}_1 \ldots \mathtt{x}_n = e[\mathtt{f}, \mathtt{x}_1, \ldots, \mathtt{x}_n] \quad \Rightarrow \quad \mathtt{f}^\bullet\ \mathtt{f}^\circ\ \mathtt{x}_1 \ldots \mathtt{x}_n = e[\mathtt{f}^\circ, \mathtt{x}_1, \ldots, \mathtt{x}_n], \quad \mathtt{f} = \mathtt{fix}\ \mathtt{f}^\bullet$$

The funny notation with filled and outlined circles carry some meaning. The black dot in $\mathtt{f}^\bullet$ indicates that this function has an implementation, mnemonic filled with implementation. The outlined circle in $\mathtt{f}^\circ$ means that this function does not have any implementation, it is empty. However, when `fix` $\mathtt{f}^\bullet$ unrolls this will replace $\mathtt{f}^\circ$.

As promised, fix point induction proves properties about a function written in terms of `fix`, and its inference rule will be stated in the next section.

### 3.3.2 Inference Rule

Fixed point induction can show a property $P$ about `fix f` for some function `f`. The base case is to show that $P$ holds when the function is replaced with $\bot$, which corresponds to zero "unrolls" of the function. For the step case, assume $P$ holds for a function $x$, and the goal is to show $P(\texttt{f}\ x)$. Intuitively, we can see $x$ as a number of "enrolling" of the function and we need to show for the next. Moreover, it can also be seen as $x$ corresponds to all the recursive occurrences in the body of the function, and we need to show it for the real function.

Using the notation from Gibbons and Hutton (2005), we now state its inference rule:

$$\frac{P(\bot) \qquad P(x) \to P(f\ x) \qquad P \text{ admissible}}{P(\text{fix} f)}$$

Admissible predicates was introduced in Section 3.2.1, and just as structural induction with the bottom base case, this induction technique also has the admissibility requirement on the predicate. Again, predicates from equality properties are admissible.

An interesting property of fixed point induction is that it does not care about types. Indeed, it works in an untyped setting. In addition, it can exploit arbitrary recursive structures of the function. A caveat is that it can only prove properties that must hold for infinite and partial values.

Fixed point induction is a consequence of induction of natural numbers. The proof for this relies on the fact that $\sqcup_n(f^n\ \bot) = \texttt{fix}\ f$, where $f^n$ is $n$ self-applications of $f$. This is true since `fix` is defined as $f$ applied to it self. The proof also uses induction over natural numbers and that $f^0\ \bot = \bot$, and the admissibility of $P$. Proof:

$$\begin{aligned}
& P(\bot) \wedge \forall x.\, P(x) \to P(fx) \\
& \qquad\qquad \Leftrightarrow \quad \{f^0\ \bot = \bot\} \\
& P(f^0\ \bot) \wedge \forall x.\, P(x) \to P(fx) \\
& \qquad\qquad \Rightarrow \quad \{\text{instantiation of } x \text{ to } f^n\} \\
& P(f^0\ \bot) \wedge \forall n.\, P(f^n\ \bot) \to P(f^{n+1}\ \bot) \\
& \qquad\qquad \Rightarrow \quad \{\text{induction over } \mathbb{N}\} \\
& \quad \forall n.\, P(f^n\ \bot) \\
& \qquad\qquad \Rightarrow \quad \{P \text{ admissible}\} \\
& \quad\ P(\sqcup_n(f^n\ \bot)) \\
& \qquad\qquad \Rightarrow \quad \{\text{definition of } \texttt{fix}\} \\
& \quad\ P(\texttt{fix}\ f)
\end{aligned}$$

One reason to introduce fixed point induction is to avoid the natural numbers in $\forall n.\, P(f^n\bot)$ to prove $P(\texttt{fix}\ f)$.

The necessary theory ought to be explained by now, so the next section shows how to apply fixed point induction on the example in the introduction.

### 3.3.3 Example: map-iterate

For properties that do not have any arguments with a concrete type, structural induction is not applicable. The Haskell function `iterate` is a that makes an infinite list from a seed, by repeated application of a function, i.e `iterate f x` is the list `x:f x:f (f x):···`. It is related to Haskell function `map` in the map-iterate property, stated as follows:

$$\forall\, f, x.\, \texttt{map}\ f\ (\texttt{iterate}\ f\ x) = \texttt{iterate}\ f\ (f\ x)$$

The standard definition of `map` was given above and `iterate` is defined as this:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

The behaviour of `map` is to apply a function to every element of a list. We see that we cannot use structural induction here, since both $f$ and $x$ are abstract, but the `map-iterate` property can be proved by fixpoint induction on `iterate`. First, we rewrite this function in terms of `fix`:

```
iterate = fix iter
iter i f x = x : i f (f x)
```

The predicate $P$ is defined to be $P(i) \Leftrightarrow \forall\, f, x.\, \texttt{map}\ f\ (i\ f\ x) = i\ f\ (f\ x)$. If the base and the step case is shown $P(\texttt{fix iter})$ can be concluded, which by definition is $P(\texttt{iterate})$.

The base case is $P(\bot)$. Since `map` is strict in its second argument, it is both the left side and right side evaluate to $\bot$. The for the step case we have to show $P(\texttt{i}) \rightarrow P(\texttt{iter i})$. The proof without explicitly writing out function pointer applications looks like this:

$$\begin{aligned}
lhs &= \texttt{map}\ f\ (\texttt{iter i}\ f\ x) && \{\text{definition of } \texttt{iter}\}\\
&= \texttt{map}\ f\ (x\texttt{:i}\ f\ (f\ x)) && \{\text{definition of } \texttt{map}\}\\
&= f\ x\texttt{:map}\ f\ (\texttt{i}\ f\ (f\ x))) && \{\text{induction hypothesis}\}\\
&= f\ x\texttt{:i}\ f\ (f\ (f\ x))) && \{\text{definition of } \texttt{iter}\}\\
&= \texttt{iter i}\ f\ (f\ x) = rhs
\end{aligned}$$

Now fixed point induction gives the `map-iterate` property.

### 3.3.4 Simplification

The mechanical translation introduced above for self-recursive functions introduces a new function with an additional argument, the "non-recursive" version of itself. The generated first order function would use the new argument as a function pointer, which in turn means a lot of use of the @. This can be seen in the cons case for `map`:

$$\forall\, m, x, xs.\, \text{map}'(m, x{:}xs) = (f\ @\ x){:}((m\ @\ f)\ @\ xs)$$

This gives unnecessary overhead to the automated theorem provers. There is another approach. It avoids introducing these function pointers and the additional argument to every function. Consider the translation of a function `f` with the same setting as before: $n \geq 0$ arguments $\texttt{x}_1$ to $\texttt{x}_n$ and a body $e[\texttt{f}, \texttt{x}_1, \ldots, \texttt{x}_n]$ with these variables free, instead this translation is made:

$$\texttt{f x}_1 \ldots \texttt{x}_n \texttt{ = } e[\texttt{f}, \texttt{x}_1, \ldots, \texttt{x}_n] \qquad \Rightarrow \qquad \texttt{f}^\bullet \texttt{ x}_1 \ldots \texttt{x}_n \texttt{ = } e[\texttt{f}^\circ, \texttt{x}_1, \ldots, \texttt{x}_n]$$

Two new functions are introduced to the vocabulary: $\texttt{f}^\bullet$ and $\texttt{f}^\circ$. The empty circle $^\circ$ describes that this function is empty (lacks implementation,) and the filled circle $^\bullet$ means that this function has an implementation. This implementation is in terms of the unfilled version. The fixed point induction schema can now be stated using these functions:

$$\frac{P(\bot) \qquad P(f^\circ) \to P(f^\bullet) \qquad P \text{ admissible}}{P(f)}$$

The empty circle in $f^\circ$ indicates that it does not have any implementation, but the induction hypothesis is asserted for it. The induction conclusion is to prove the property for $f^\bullet$, where the recursive call to $f$ is replaced with $f^\circ$. This simplification is done since it is better suited for the theorem provers.

Further, it is now necessary to "inline" the predicate in the generated theory. The reason is that $f^\bullet$ and $f^\circ$ are first order functions rather than expressible as pointers, and predicates that quantify over functions are not first order expressible. Such "inlining" is also beneficial for theorem provers as unnecessary predicates are not introduced to the theory.

**Mutually recursive functions** It is also possible to rewrite a group of mutually recursive functions by packing the "non-recursive" copies of themselves in a tuple. The inference rule for two functions at the same time, possibly mutually recursive then looks like this:

$$\frac{P(\bot, \bot) \qquad P(f^\circ, g^\circ) \to P(f^\bullet, g^\bullet) \qquad P \text{ admissible}}{P(f, g)}$$

This also works when the functions are not mutually recursive. An example is the map-iterate property which can be proved by fixating both `map` and `iterate`.

### 3.3.5 Erroneous Use of Fixed Point Induction

It is crucial that $P$ is admissible to avoid deriving falsities. This section gives a simple example of what can happen otherwise. Recall the definition of the infinite natural number:

```
inf = Succ inf
```

A predicate that is not admissible will deliberately be used to "prove" that `inf` $\neq$ `Succ inf`. The predicate is $P(x) \Leftrightarrow x \neq \text{succ}(x)$. Furthermore, `inf` is translated to first order logic and rewritten in terms of $^\circ$ and $^\bullet$, as introduced in the previous section:

$$\text{inf}^\bullet = \text{succ}(\text{inf}^\circ)$$

Now we proceed by "fixed point induction". Base case: $\bot \neq \text{succ}(\bot)$ is true by the axioms of disjoint constructors. Step case: assume $\text{inf}^\circ \neq \text{succ}(\text{inf}^\circ)$. Injectivity of succ gives $\text{succ}(\text{inf}^\circ) \neq \text{succ}(\text{succ}(\text{inf}^\circ))$. By the definition of $\text{inf}^\bullet$, which is then equal to the goal $\text{inf}^\bullet \neq \text{succ}(\text{inf}^\bullet)$.

The conclusion `inf` $\neq$ `Succ inf` is clearly nonsense as it directly contradicts the definition of `inf`! A consequence of this example is that inequality is in general not admissible.

### 3.3.6 Candidate Selection

Faced with the following property saying that if you drop $n$ elements from a list the length of this is the same as the length of the original list minus $n$, which functions should one do fixed point induction on?

```
prop_length_drop :: [a] -> Nat -> Prop Nat
prop_length_drop xs n = length (drop n xs) =:= length xs - n
```

The answer is to do fixed point induction on `drop`, and on `-`. So far no better way to tackle this is used than to try fixed point induction on all subsets of recursive functions mentioned in the property.

### 3.3.7 Future Work

Just as with structural induction, it is also possible to use fixed point in more than one "depth", giving for instance this inference rule:

$$\frac{P(\bot) \qquad P(f\ \bot) \qquad P(x) \wedge P(f\ x) \rightarrow P(f\ (f\ x)) \qquad P \text{ admissible}}{P(\text{fix} f)}$$

It is also possible to use such an encoding as in "Automated depth" in Section 3.2.4 to let the theorem prover determine the depth. As an example, the map-iterate property impossible to show with `map` redefined to `doublemap`, defined below, with ordinary one depth fixed point induction.

```
doublemap :: (a -> b) -> [a] -> [b]
doublemap f []       = []
doublemap f [x]      = [f x]
doublemap f (x:y:zs) = f x : f y : doublemap f zs
```

Although `doublemap` is behaviourally equivalent to `map` on total lists, it makes the induction hypothesis in fixed point induction too weak.

An issue with the candidate selection is that is some selections are immediate dead ends. An example is fixating functions on only one side of the equality, then the base case will generally never succeed unless the other side is constant bottom. A heuristic to find good candidates would be beneficial.

## 3.4 Approximation Lemma

The approximation lemma is another standard technique for proving properties about corecursive programs. It has similar properties as fixed point induction: it is applicable on functions that produce infinite values and with no argument viable for structural induction. However, it is simpler to apply. (Gibbons and Hutton, 2005). The key idea for equality over lists is to show that all prefixes of the sides coincide; i.e their approximations are equal.

The approximation lemma supersedes another proof technique for infinite lists, the classical take lemma (Bird and Wadler, 1988) by being easier to apply and generalise: unlike the take lemma, it can be applied to equalities of any polynomial data type (Hutton and Gibbons, 2001). The definitions of `approx` for lists and `take` can be seen in Figure 3.2.

```
take :: Nat -> [a] -> [a]
take Zero     _      = []
take (Succ n) []     = []
take (Succ n) (x:xs) = x : take n xs
```

```
approx :: Nat -> [a] -> [a]
approx (Succ n) []     = []
approx (Succ n) (x:xs) = x : approx n xs
```

Figure 3.2: Definition of `take` and `approx`

Both functions in Figure 3.2 can be seen as approximating a list to a given length. The difference is the last value in the list. `take` ends the list with `[]`. Since there is no `Zero` case for `approx`, it ends it with $\bot$. The idea of these techniques is then to show that show that two lists are equal by showing that their prefix or approximation coincides for all natural numbers.

Now we can state the approximation lemma:

$$xs = ys \quad \Leftrightarrow \quad \forall\, n \in \mathbb{N}.\, \mathsf{approx}\, n\, xs = \mathsf{approx}\, n\, ys \tag{3.1}$$

Equation 3.1 quantifies over the true natural numbers[2] (i.e. total and finite), rather than the Haskell naturals. To prove a non-trivial equality to a proof by induction over natural numbers. The base case for $0$ is always trivially true by reflexivity, as the approximation of the two lists is $\bot = \bot$. For the step case, the right to left implication is (trivially) true by substitution, and the other direction hinges on the lemma that better and better approximations form a chain with limit `id`, as illustrated in Equation 3.2 below.

$$\mathsf{approx}\, 0 \sqsubseteq \mathsf{approx}\, 1 \sqsubseteq \cdots \sqsubseteq \mathsf{approx}\, n \sqsubseteq \mathsf{approx}\, (\mathsf{Succ}\, n) \sqsubseteq \cdots \sqsubseteq \mathtt{id} \tag{3.2}$$

The inclusions in Equation 3.2 are easily given by induction on natural numbers and the limit by structural induction on lists. For other polynomial data types, this lemma is established by the structural induction induced on that data type. The desired implication is then readily deduced:

$$\begin{aligned}
\forall\, n.\, \mathsf{approx}\, n\, xs &= \forall\, n.\, \mathsf{approx}\, n\, ys \\
&\Rightarrow \quad \{\text{limits}\} \\
\sqcup_n\, (\mathsf{approx}\, n\, xs) &= \sqcup_n\, (\mathsf{approx}\, n\, ys) \\
&\Leftrightarrow \quad \{\text{continuity of application}\} \\
\sqcup_n\, (\mathsf{approx}\, n)\, xs &= \sqcup_n\, (\mathsf{approx}\, n)\, ys \\
&\Leftrightarrow \quad \{\text{Equation 3.2}\} \\
\mathtt{id}\, xs &= \mathtt{id}\, ys \\
&\Leftrightarrow \quad \{\text{definition of } \mathtt{id}\} \\
xs &= ys
\end{aligned}$$

This concludes the proof of the approximation lemma. The take lemma is stated just the same but with `take` instead of `approx`. However, it does not enjoy the simple chain property in Equation 3.2, so the proof is a bit more involving. Furthermore, generalising `approx` to

[2]The true natural numbers refers here means the standard model of $\mathcal{PA}$.

other sum of product data types is simple, an example is given in the next section. For the `take` lemma it is not as simple: for some data types there is no single nullary constructor in place for the empty list.

### 3.4.1 Example: Mirroring an Expression

Consider these definitions of a modest but prototypical expression data type, and its mirroring function:

```
data Expr = Add Expr Expr | Value Nat

mirror :: Expr -> Expr
mirror (Add e1 e2) = Add (mirror e2) (mirror e1)
mirror (Value n)   = Value n

prop_mirror_involutive :: Expr -> Prop Expr
prop_mirror_involutive e = e =:= mirror (mirror e)
```

This property that `mirror` is involutive is provable by the approximation lemma. The approximation function for `Expr` can be automatically generated by approximating each self-recursive constructor. This relies on the fact that the type of the equality is known, and here is the reason why the type needs to be stated in the properties[3]. For our data type `Expr`, the generated function approximates the sub-expressions in `Add` and the `Nat` in `Value` is not further approximated:

```
approx :: Nat -> Expr -> Expr
approx (Succ n) (Add e1 e2) = Add (approx n e1) (approx n e2)
approx (Succ n) (Value n)   = Value n
```

Since `Expr` does not have a nullary constructor, deriving a `take` function for it is impossible. Furthermore, fixed point induction fails on this property: choosing either or both occurrences of `mirror` on the right side is constant bottom for the base case, and the left side is the identity.

As always in proofs by approximation lemma, we proceed by induction over natural numbers, and the base case is always trivial: true by reflexivity as both sides are $\bot$. For this reason, only the step case needs to be proved:

$$\frac{\forall\, e.\, \mathtt{approx}\, n\, e = \mathtt{approx}\, n\, (\mathtt{mirror}\, (\mathtt{mirror}\, e))}{\forall\, e.\, \mathtt{approx}\, (\mathtt{Succ}\, n)\, e = \mathtt{approx}\, (\mathtt{Succ}\, n)\, (\mathtt{mirror}\, (\mathtt{mirror}\, e))}$$

An important property of the induction hypothesis is the universal quantification of the expression $e$, unlike the fixed natural number $n$. The proof is by cases, since for any argument, `mirror` will return a value of $\bot$, `Add` or `Value`. The *bot* case is trivial as `mirror` is strict in its first argument, and the `Value` case is also straight forward as mirroring is the identity on this input.

[3] Indeed, the types could be inferred.

The Add case is slightly more involving. The reasoning is as follows:

$$\begin{aligned}
lhs &= \texttt{approx}\,(\texttt{Succ}\,n)\,(\texttt{Add}\,e_1\,e_2) && \{\text{definition of } \texttt{approx}\}\\
&= \texttt{Add}\,(\texttt{approx}\,n\,e_1)\,(\texttt{approx}\,n\,e_2) && \{\text{induction hypothesis}\}\\
&= \texttt{Add}\,(\texttt{approx}\,n\,(\texttt{mirror}\,e_1))\,(\texttt{approx}\,n\,(\texttt{mirror}\,e_2)) && \{\text{definition of } \texttt{approx}\}\\
&= \texttt{approx}\,(\texttt{Succ}\,n)\,(\texttt{Add}\,(\texttt{mirror}e_1)\,(\texttt{mirror}e_2)) = rhs
\end{aligned}$$

As the all the cases are proved, the approximation lemma proves the property to be a theorem. The next section will investigate the relation between approximation lemma and fixed point induction. It will be shown that this section's technique can be expressed in terms of fixed point induction. This makes this method lighter for theorem provers.

### 3.4.2 Approximation Lemma is Fixpoint Induction

This technique is already simple and widely applicable, however it can further be simplified. Using it with theorem provers in the form introduced about relies on the auxiliary structure of $\mathcal{PA}$ natural numbers which then needs to be added to the theory. This can be removed by the observation that the approximation lemma can be expressed as fixed point induction. The fixed function is a recursive form of the identity function, which will be called `ind` for its close resemblance of induction. For lists and the `Expr` data type, `ind` can be seen in Figure 3.3.

```
ind[a] :: [a] -> [a]
ind[a] [] = []
ind[a] (x:xs) = x:ind[a] xs
```

```
indExpr :: Expr -> Expr
indExpr (Value n) = Value n
indExpr (Add e1 e2) = Add (indExpr e1) (indExpr e2)
```

Figure 3.3: Definition of `ind` for lists and `Expr`

Each `ind` function constructed in this way is indeed an identity function, equivalent to the implementation `ind x = x` when disregarding time and space complexity. Now, to prove an equality for two expressions $e_1$ and $e_2$ with some variables $x_1$ to $x_n$ free:

$$\forall\, x_1, \ldots, x_n.\, e_1[x_1, \ldots, x_n] \;=\; e_2[x_1, \ldots, x_n],$$

one can simply use fixed point induction over $\mathrm{ind}$ to prove:

$$\forall\, x_1, \ldots, x_n.\, \mathrm{ind}(e_1[x_1, \ldots, x_n]) \;=\; \mathrm{ind}(e_2[x_1, \ldots, x_n])$$

where $\mathrm{ind}$ is a such a specialised recursive identity function over the data type of the equality. The step case in induction $P(ind^{\circ}) \to P(ind^{\bullet})$ is then exactly the same strength as the approximation lemma with natural numbers. With implicit universal quantification, we get this simplified inference rule for the approximation lemma:

$$\frac{\mathrm{ind}^{\circ}\; e_1 = \mathrm{ind}^{\circ}\; e_2 \to \mathrm{ind}^{\bullet}\; e_1 = \mathrm{ind}^{\bullet}\; e_2}{e_1 = e_2}$$

Just as fix point induction was introduced to reason about chains without explicitly mentioning natural numbers, this technique follows the same pattern, as it does does not rely on natural numbers.

### 3.4.3 Future Work: Total Approximation Lemma

It is also possible to adjust the approximation lemmas to prove properties about total values. A property that is true for total list is the idempotence of nub. This is a version of nub on booleans, and with this property stated:

```
nub :: [Bool] -> [Bool]
nub (True :True :xs) = nub (True:xs)
nub (False:False:xs) = nub (False:xs)
nub (x:xs)           = x:nub xs
nub _                = []

prop_nub_idem :: [Bool] -> Prop [Bool]
prop_nub_idem xs = nub (nub xs) =:= nub xs
```

This is not a theorem for infinite and partial lists: Consider the list `True:False:`$\bot$. One application of nub gives `True:`$\bot$, and two gives $\bot$ immediately. Similar results are obtained for the infinite and total list `repeat True`, however this property is true for total finite lists and a way to adjust the approximation lemma to prove such properties is to add a predicate indicating totality. A value is total if it does not contain any bottoms. This predicate can be approximated is first order equality, and these axioms sketches out the approach for the totality theory relevant for the nub function:

I. $\neg Total(\bot)$

II. $Total(\texttt{[]})$

III. $\forall x, xs. Total(x) \wedge Total(xs) \leftrightarrow Total(x\texttt{:}xs)$

IV. $\forall xs. Total(xs) \rightarrow Total(\mathsf{nub}\, xs)$

The last axiom should only be added after a proof of termination proof. However, without an axiomatisation of set theory, it is impossible to express totality for infinite length lists. The axioms above are then only an approximation and will have models where infinite total lists are $Total$ and also $\neg Total$. Hence, out setting, $Total$ must necessarily mean finite. This is further discussed in Section 5.2.2.

Furthermore, the admissibility of this approach needs to be proved. A simple and general case is assumed, where the predicate is defined to be

$$P(y) \Leftrightarrow \forall x. Q(x) \rightarrow R(x, y),$$

and the restriction is that $R$ is admissible in the second argument. This is the setting above with $Q$ in place of $Total$ and $R$ the admissible equality predicate with $x$ as a free variable. Assume a $\sqsubseteq$-chain $\langle y_n \rangle_{n \in \omega}$ that satisfies $P$. Take some $x_0$ such that

$$\forall n. Q(x_0) \rightarrow R(x_0, y_n)$$

If not $Q(x_0)$, then for all $y$ we have $R(x_0, y)$, and in particular when $y = \sqcup_n(y_n)$ we get $P(\sqcup_n(y_n))$. If $Q(x_0)$, then $R(x_0, \sqcup_n(y_n))$ because $R$ is admissible in its second argument. In both cases we have $P(\sqcup_n(y_n))$, hence $P$ is admissible. This technique with a predicate for totality also works for fixed point induction. Additionally, the predicate $Q$ can also be a predicate indicating type.

# Chapter 4

# Results and Discussion

This chapter describes the test suite used in this project. A large test with five theorem provers was carried out on this test suite, and this section describes the setup and results.

## 4.1 Test Suite

To carry out a throughout testing of the system a large test suite was developed and collected from other sources. It consists of over 25 Haskell source files, and includes over 500 equality properties. The files of can be divided into groups with different aims of complexity, suitable proving methods and different aspects of Haskell. All files of the test suite are briefly described in Table 4.1, and available in the project's repository[1], but some parts and design decisions are highlighted in the rest of this section.

The test suite includes many properties about natural numbers, because they constitute the minimal recursive data type with a base case. The properties originates from the Zeno test suite, but are extended with different versions with addition and multiplication defined in different ways, the definitions for addition viewed in Figure 4.1 below.

Nat

```
Z   + y = y
S x + y = S (x + y)
```

NatSwap

```
Z   + y = y
S x + y = S (y + x)
```

Nat2ndArg

```
x + Z   = x
x + S y = S (x + y)
```

NatAcc

```
Z   + y = y
S x + y = x + S y
```

NatDouble

```
Z       + y = y
S Z     + y = S y
S (S x) + y = S (S (x + y))
```

NatDoubleSlow

```
Z       + y = y
S Z     + y = S y
S (S x) + y = S (S x + y)
```

NatStrict

```
Z   + Z   = Z
Z   + S x = S x
S x + y   = S (x + y)
```

Figure 4.1: The test suite's different versions of addition over `data Nat = Z | S Nat`.

[1] See the `testsuite` directory in `http://github.com/danr/hip/`

| Group | File | Description |
|---|---|---|
| Fundamental | `Bool` | Simple properties about and, or and not |
| | `Expr` | Expressions with mirror, size and eval |
| | `Functions` | Function composition, currying |
| | `Integers` | Integers as a disjoint union of Nats |
| | `Reverse` | Reverse with and without accumulator |
| | `Queues` | Queues with $O(1)$ pop and enqueue |
| Infinite values | `Infinite` | Infinite list and tree properties |
| | `Sequences` | Infinite sequences |
| | `Streams` | Streams from (Hinze, 2010) |
| Miscellaneous | `PatternMatching` | `||` and `mirror` in different ways |
| | `Fix` | Even and odd defined using `fix` |
| | `Tricky` | Properties that hold for total infinite lists |
| Monads | `MonadEnv` | The environment monad |
| | `MonadMaybe` | The maybe monad |
| | `MonadState` | The state monad |
| Natural numbers | `Nat` | Natural numbers from the Zeno test suite |
| | `Nat2ndArg` | Induction on the second argument |
| | `NatAcc` | Accumulator definitions |
| | `NatDouble` | Depth two definitions |
| | `NatDoubleSlow` | Depth two, recursion in one depth |
| | `NatStrict` | Strict in both arguments |
| | `NatSwap` | Arguments swapped in the recursive call |
| Other Work | `IWC` | Inductive challenge problems from `http://www.csc.liv.ac.uk/~lad/research/challenges` |
| | `ProductiveUseOfFailure` | From (Ireland and Bundy, 1995) |
| | `ZenoLists` | The list part of Zeno's test suite |
| | `Ordinals` | Ordinals as defined by Dixon (2005) |
| Sorting | `InsertionSort` | Insertion sort |
| | `MergeSort` | Merge sort from `Data.List` |

Table 4.1: Description of the files in the test suite, by group.

To test reasoning about higher order functions the file `Functions` states standard properties about function combinators, and the file `MonadEnv` defines the reader monad. To make it a little more complicated there is also a definition of the state monad in `MonadState`.

Properties about functions that produce infinite lists and trees can be found in the file `Infinite`, and many of these are not provable with structural induction. There are some more complex properties about infinite lists in `Streams`.

Many properties are easier to prove with other properties available as lemmas in the generated theory. Some properties even requires this. An example are the properties about insertion sort in the test suite. It includes properties such that the resulting list after sorting is sorted. Insertion sort works with a helper function `insert` that inserts an element in a sorted list at its right position. Lemmas for `insert` are needed to prove properties about insertion sort.

Unfortunately there is no information available about which properties require lemmas, nor is there any data of which properties should be provable with a given proof technique. To produce such information it would be necessary to prove that a given property is *not* provable in a certain way, and this is out of scope of this thesis.

## 4.2 Setup

This section describes the setup for the exhaustive run of the test suite. Five theorem provers were used and they are summarised in Table 4.2.

| Name | Version | Reference |
|---|---|---|
| Vampire | 0.6 | `http://www.vprover.org/` |
| E | 1.0-004 Temi | Schulz (2002) |
| prover9 | 2009-02A | McCune (2010) |
| SPASS | 3.5 | `http://www.spass-prover.org/` |
| equinox | 6.0.1alpha | Claessen (2011) |

Table 4.2: The theorem provers used on the test suite

The provers were run on a 2.40GHz Intel Xeon CPU. All properties were tested for each theorem prover, and also one run which tried every invocation with all provers. To make this tractable, some restrictions were needed. The time limit for each invocation was 5 seconds. The maximum depth and induction variables for structural induction was 2, and to prevent combinatorial explosion the number of induction steps was limited to 20, and a maximum of 500 hypotheses were generated. For fixed point induction there is a similar problem since there are many possible candidate functions, so the maximum number of invocations with different fixed functions were 100 for each property.

This was the setup used when testing the tool on the test suite. The rest of this chapter explains the results, comparing the different proof techniques, the used theorem provers and the success times for the theorem prover invocations.

## 4.3 Results for Proof Techniques

Of the 540 properties of the test suite, 111 could only be proved when universally quantifying only finite values. Those theorems will be referred to as Finite Theorems. All of the

were proved with structural induction without a bottom base case as it is the only applicable technique implemented for them.

An additional 214 properties were proved with the four proof techniques for infinite values. These are succinctly referred to as Theorems. This section will discuss the performance of the proof methods on properties which were stated as Theorems. The overlap between the different techniques can be viewed in the Venn diagram in Figure 4.2.

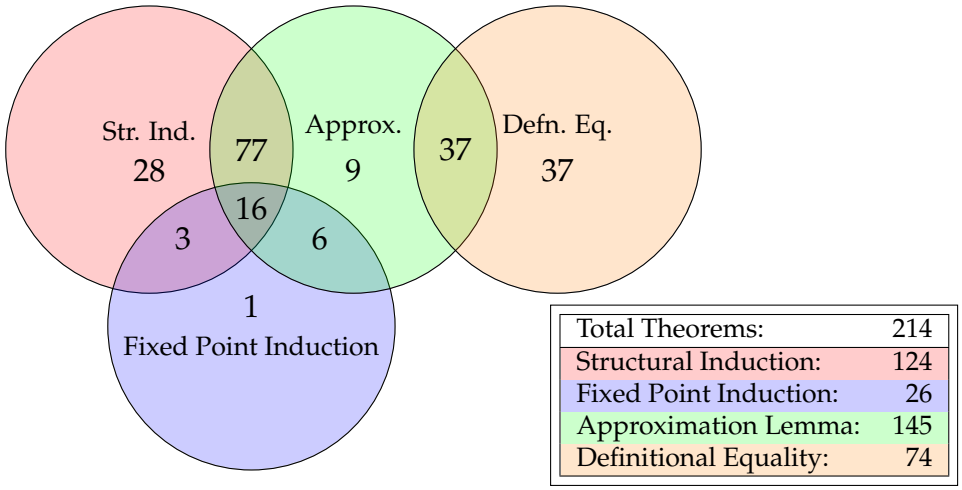


Figure 4.2: Venn diagram of intersection between proved theorems for different techniques. The data is from the 214 proved theorems when using all provers.

There are a number of observations to do from this figure. The different parts of the diagram is studied in the rest of this section.

### 4.3.1 Definitional Equality and the Approximation Lemma

Firstly, definitional equality only overlaps with the approximation lemma. The reason for this is that this technique is only applied when all arguments are abstract, as outlined in Section 3.1.2. Further, fixed point induction is only used for recursive functions and properties about such functions generally need induction. Approximation lemma is applicable when the type of the equality is concrete, so that is the source of the overlap.

The properties provable with definitional equality and the approximation lemma naturally comes from properties regarding only functions from files `Functions`, `MonadEnv` and `MonadState`, where definitional equality alone proves the properties where the type of the equality is not concrete. Two unit test properties could for these reasons also only be proved with definitional equality, like this one from `Queues`:

```
prop_34 :: a -> Prop a
prop_34 x = top (enqueue x empty) =:= x
```

Indeed, this property is not applicable for proving with any other technique. There were about 10 unit test properties provable with both approximation lemma and definitional equality.

**Only Approximation Lemma** The 9 properties provable with only approximation lemma can be divided in two parts. The first part consists of a property from `Streams`:

```
zero :: Stream Nat
zero = pure Zero

one :: Stream Nat
one = pure (Succ Zero)

prop_one :: Prop (Stream Nat)
prop_one = Succ <$> zero =:= one
```

Here, `pure` is the infinite stream constructed by repeating a value, and `<$>` is map over streams. This could be proved with fixed point induction, but it currently does not do any inlining. Should `zero` be inlined to `pure Z` and similarly for `one`, the property is provable with fixed point induction as well. Note that fixed point induction is instantiated when the theory is generated, so it does not help that the theorem provers can deduce that `zero` = `pure Zero`.

The other 8 properties only proved with approximation lemma come from `MonadState`. This monad is written without a newtype wrapper, and in many different ways. The implementations of bind with a strict `uncurry` definition and the property of right return identity are stated in this file as this:

```
-- Strict in its second argument. The normal definition uses a irrefutable pattern.
uncurry f (a,b) = f a b

(>>=) :: State s a -> (a -> State s b) -> State s b
(m >>= f) s = uncurry f (m s)

return :: a -> State s a
return x s = (x,s)

prop_return_right_strict :: (s -> (a,s)) -> Prop (s -> (a,s))
prop_return_right_strict f = (f >>= return) =:= f
```

The property will be expanded to `(f >>= return) s =:= f s`. Interestingly, proving this with the approximation lemma works as a typing predicate: it will provide the information that `f s` is either bottom or `(a,b)` for some `a` and `b`. This information is not available for the untyped setting in definitional equality and thus gets stuck.

**Approximation Lemma and Fixed Point Induction** The properties provable with only these two techniques all come from `Infinite`, and include properties about `map`, `repeat`, `iterate` and similar constructions for Trees rather than lists. Neither of these properties have concrete arguments to do structural induction on.

### 4.3.2 Structural Induction

The properties provable with only structural induction include properties from `Fix`, where ordinary functions are written in terms of `fix`, where fixed point induction ironically does not work. In addition, many properties come are about natural numbers and lists.

Some structural induction and approximation lemma theorems are essentially doing proof by plain equality but with cases, from `Bool`. However, most of them come from properties about lists and natural numbers, integers and the `Maybe` monad.

**Structural Induction and Fixed Point Induction** The three properties provable with only these two techniques point induction come from `IWC`, `Infinite` and `Nat` respectively. The first is a definition of proving the equivalence between a mutually recursive definition of an even predicate over natural numbers and a self-recursive definition in depth two. The first is readily not provable with the approximation lemma as it approximates the resulting `Bool`, and a stronger induction hypothesis is needed.

The second property is one of the functor laws for `fmap` for trees:

```
prop_fmap_comp :: (b -> c) -> (a -> b) -> Tree a -> Prop (Tree c)
prop_fmap_comp f g t = fmap (f . g) t =:= fmap f (fmap g t)
```

Interestingly, this property is provable with the approximation lemma, but it takes more than 5 seconds: the E theorem prover produces a proof in 18s.

The last property does not seem to be suitable for approximation:

```
prop_max_assoc :: Nat -> Nat -> Nat -> Prop Nat
prop_max_assoc a b c = max (max a b) c =:= max a (max b c)
```

**Using all inductive techniques** The properties provable with all three techniques are associativity proofs, symmetry of `max` from `Nat`, some properties about lists from `ZenoLists` and some lemmas from `ProductiveUseOfFailure`, including:

```
prop_L7 :: a -> a -> [a] -> Nat -> Nat -> Nat -> Prop [a]
prop_L7 x y zs m n o = drop (S o) (drop n (drop (S m) (x:y:zs)))
                   =:= drop (S o) (drop n (drop m (x:zs)))
```

### 4.3.3 Fixed Point Induction and Exponential Types

There was only one property only provable with fixed point induction, namely the property of associative addition for Brouwer ordinals from Dixon (2005) in the test file `Ordinals`. The relevant definitions are these:

```
data Ord = Zero | Suc Ord | Lim (Nat -> Ord)

(+) :: Ord -> Ord -> Ord
Zero  + b = b
Suc a + b = Suc (a + b)
Lim f + b = Lim (\n -> f n + b)

prop_assoc_plus :: Ord -> Ord -> Ord -> Prop Ord
prop_assoc_plus a b c = a + (b + c) =:= (a + b) + c
```

The `Ord` data type is exponential, and complete support for such types were never implemented for structural induction and the approximation lemma (see Section 3.2.4). With fixed point induction, this support comes for free. For the proof of associativity the fixed functions are the first addition in both sides, with this induction hypothesis:

$$\forall\, a, b, c.\, a +^{\circ} (b + c) = (a +^{\circ} b) + c$$

The interesting case is for the limit $\lim(f)$ in the step part of proof by fixed point induction. A proof that liberally uses lambdas instead of explicit pointers is given below:

$$
\begin{aligned}
lhs &= \lim(f) +^{\bullet} \big(b + c\big) && \{\text{definition of } +^{\bullet}\}\\
&= \lim(\lambda n.f(n) +^{\circ} \big(b + c\big)) && \{\text{induction hypothesis}\}\\
&= \lim(\lambda n.(f(n) +^{\circ} b) + c) && \{\beta\text{-expansion}\}\\
&= \lim(\lambda n.(\lambda m.f(m) +^{\circ} b)(n) + c) && \{\text{definition of } +\}\\
&= \big(\lim(\lambda m.f(m) +^{\circ} b\big) + c && \{\text{definition of } +^{\bullet}\}\\
&= \big(\lim(f) +^{\bullet} b\big) + c = \ rhs
\end{aligned}
$$

In hindsight, it is not so surprising that this property is provable with fixed point induction, but its applicability on exponential data types was actually not thought of during its implementation.

## 4.4 Results for Different Theorem Provers

In this section we present how well the different provers succeeded on the big run of the test suite. As in the previous section, successes for methods which handle all values are called Theorems, and for methods that only work on finite and total values are called Finite Theorems.

To compare the provers and the different proof methods on this test suite, the number of proved properties for each prover and proof method is illustrated in Table 4.3. The entries in the row captioned Theorem shows how many theorems were proved out of the total. The following four columns for definitional equality, structural induction, approximation lemma and fixpoint induction tells how many properties out of those could be proved with this method. The following two columns are similar, though for finite theorems where the only applicable method is structural induction.

| Prover | Theorem | defn.eq. | induction | approx | fixpoint | Finite Thm. | induction |
|---|---|---|---|---|---|---|---|
| E | 203/540 | 68/203 | 120/203 | 136/203 | 23/203 | 99/540 | 99/99 |
| prover9 | 194/540 | 69/194 | 114/194 | 92/194 | 8/194 | 93/540 | 93/93 |
| SPASS | 198/540 | 65/198 | 122/198 | 111/198 | 21/198 | 94/540 | 94/94 |
| Vampire | 208/540 | 71/208 | 122/208 | 133/208 | 25/208 | 104/540 | 104/104 |
| equinox | 78/540 | 15/78 | 57/78 | 56/78 | 3/78 | 39/540 | 39/39 |
| all provers | 214/540 | 74/214 | 124/214 | 145/214 | 26/214 | 111/540 | 111/111 |

Table 4.3: Number of proved properties per prover and proof method. Only the Theorem is counted for properties proved as both Theorems and Finite Theorems.

**Discussion** In general, all provers tested in this experiment but equinox performed almost equally well on this problem set. For those four in the top the differences are small, but the most successful is Vampire. It is closely followed by E, which performs slightly better on the approximation lemma. One of the notable differences is that prover9 performs worse proofs by the approximation lemma and fixed point induction. The theorem provers has been considered as black-boxes and examining these results in terms of how the theorem provers operate is out of scope of this thesis.

## 4.5 Proving Time

Most invocations to the theorem provers are solved in a few milliseconds, and few can take up to 500 ms. SPASS had the largest success time of all invocation: about 1500ms, but that was the only success over one second. The cumulative amount of success times can be viewed in Figure 4.3. Each method can give rise to several different ways of proving a property, an example is fixed point induction on different functions, and each of these can consist of one or many invocations to a theorem prover, an example is the base and the different step cases in a proof by structural induction. All invocations from a technique with a given setting that ended in success are counted in the figure.

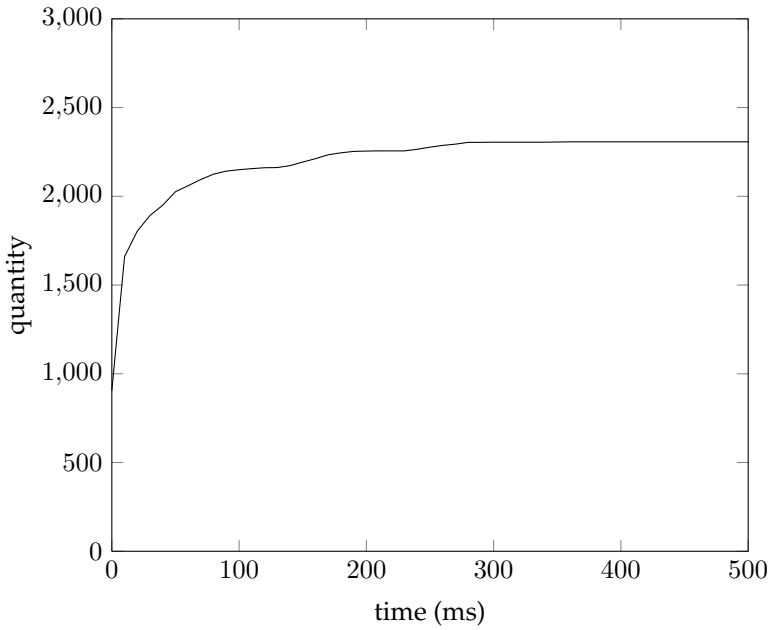


Figure 4.3: Cumulative amount of successes over time, using all provers and methods. Only successes from a method resulting in a theorem are counted.

**Discussion** The figure conveys that more than the majority of invocations for successes take less than 50ms, and after 300ms almost no new successes are encountered. However, as noticed in Section 4.3.2, there are also examples where much longer invocation times are needed. A way of guiding the theorem provers in the search space of proofs is discussed in Section 5.6.

# Chapter 5

# Future work

The chapter addresses some limitations of our approach and ideas how to get around them.

## 5.1 Pattern Matching Re-Revisited

The current translation of pattern matching is described in Section 2.5. One of the strengths of it is the translation of wild patterns. Consider this function for equality of a data type with three elements:

```
data Tri = A | B | C

equal :: Tri -> Tri -> Bool
equal A A = True
equal B B = True
equal C C = True
equal _ _ = False
```

The translation adds two more cases that go to bottom, one where the first argument is bottom, and one where the second one is bottom. All other values that are not the same `Tri` and not bottom go to `False`.

If there for models of this function which includes another value $\diamond$, other values than those of this data type and bottom, must have $\mathrm{equal}(\diamond, x) = \mathrm{false}$, for all non bottom $x$, including $\diamond$. This is not a problem per se, since the Haskell source is well typed, and `equal` will not be applied to anything but `Tri`s and bottoms.

One weakness of that approach is that two functions that are extensionally equivalent in Haskell are not in the generated theory. Due to its untyped nature, results can differ when applied to non sense arguments. An example of this behaviour occurs for these two implementations of the boolean and function in Figure 5.1.

```
and :: Bool -> Bool -> Bool
and True  b = b
and False _ = False
```

```
and' :: Bool -> Bool -> Bool
and' True  b = b
and' _     _ = False
```

Figure 5.1: Two definitions of boolean and, `and` and `and'`

Let us analyse the models of this program in Figure 5.1 with an additional value $\diamond$ in the domain. The translation makes $\mathrm{and}(\diamond, \mathrm{true}) = \bot$, since there is an implicit wild pattern to $\bot$. For the other function we have $\mathrm{and}'(\diamond, \mathrm{true}) = \mathrm{false}$, as the wild pattern already goes to $\mathrm{false}$. Because of this differences, the approximation lemma cannot prove their extensional equality. Although this example is a quite simple, it is easy to imagine more complex cases where different pattern matching techniques makes no difference to the Haskell program, but to the translation.

This suggests that wild patterns should be expanded to pattern for all their constructors, and add a match any pattern that goes to bottom. This make the the meta theorem $\forall\, x, y.\, \mathrm{and}(x, y) = \mathrm{and}'(x, y)$ consistent with rest of the theory. Further, it would be provable by the approximation lemma, and in more complex cases with recursive functions, by fixed point induction. This translation is also assumed to be a little easier to implement. The down side is that functions such as `equal` above would generate $O(n^2)$ axioms for a data type with $n$ constructors. The presence of GADTs and other type extensions such as type families would requite type information and add a lot of complexity.

## 5.2 Lemmas

For many properties, especially more advanced ones, it is crucial to be able to use lemmas to obtain a proof. The proof techniques used in this thesis are just not strong enough. For structural induction, it is possible to do induction in more than one depth. This sometimes make the hypotheses equally strong as the required lemmas, as in the example of plus commutativity of natural numbers in Section 3.2.2. But this approach fails for properties about multiplication: it is essential to use lemmas about addition.

The concept of adding lemmas is of course quite simple. Assume your program has two properties `prop_a` and `prop_b`, and the second needs the first as a lemma. If `prop_a` succeeds, then the should program just add that property to the theory generated for `prop_b`, and try it again. Unfortunately, adding a property to a theory needs to be carried out with care. For properties that hold for infinite and partial values it is actually a bit simpler than for those that are only true for finite values. These settings are addressed one by one below.

### 5.2.1 Lemmas from Theorems

One property that holds for partial values is our example of the right identity of `&&`:

$$\forall\, x.\, x \;\&\&\; \mathrm{true} = x$$

Adding this meta theorem to the theory would make it inconsistent; in a model with an extra value $\diamond$, we have that $\diamond \;\&\&\; \mathrm{true} = \bot$. However, we can create a function that forces a value to be a `Bool` or $\bot$ as this:

| | | |
|---|---|---|
| I. | $\mathrm{force}(\mathrm{true})$ | $= \mathrm{true}$ |
| II. | $\mathrm{force}(\mathrm{false})$ | $= \mathrm{false}$ |
| III. | $\forall\, x.\, \mathrm{force}(x)$ | $= \bot \vee x = \mathrm{false} \vee x = \mathrm{true}$ |

The theory would be consistent if instead this reformulation of the theorem is added:

$$\forall\, x.\, \mathrm{force}(x) \;\&\&\; \mathrm{true} = \mathrm{force}(x)$$

It is easy to generalise $\mathrm{force}$ beyond booleans. Make it a binary function with first argument a description of the type. Each simply kinded type would be given a constant, and higher-kinded types functions, an example would be $\mathrm{list}(\alpha)$ for lists of $\alpha$:

$$\begin{array}{lrl}
\text{I.} & \forall\,\alpha.\,\mathrm{force}(\mathrm{list}(\alpha),\mathrm{nil}) & = \mathrm{nil} \\
\text{II.} & \forall\,\alpha, x, xs.\,\mathrm{force}(\mathrm{list}(\alpha),\mathrm{cons}(x,xs)) & = \mathrm{cons}(\mathrm{force}(\alpha,x),\mathrm{force}(\mathrm{list}(\alpha),xs) \\
\text{III.} & \forall\,\alpha, xs.\,\mathrm{force}(\mathrm{list}(\alpha),xs) & = \bot \\
& \multicolumn{2}{r}{\vee xs = \mathrm{nil} \vee xs = \mathrm{cons}(\mathrm{cons}_0(xs),\mathrm{cons}_1(xs))}
\end{array}$$

Using functions and predicates effectively to witness type information has been studied by Claessen et al. (2011) and by Blanchette et al. (2011).

Proofs by definitional equality would also benefit from type tags. The the current limitations are discussed section 3.1.2. Furthermore, the extensional equality of the different implementations of the or function in Figure 5.1 is not provable with approximation lemma with the current translation. However, stated with $\mathrm{force}$-tags it should be possible. This could give us the best of two worlds: minimal translation of pattern matching and wider applicability of untyped proof methods.

### 5.2.2 Lemmas from Finite Theorems

The $\mathrm{force}$ function from the previous section is not easily generalised to remove partiality from a value. A way to enforce totality is to introduce a predicate $\mathrm{Total}$ indicating totality.

The axioms below are an example of how to axiomatise $\mathrm{Total}$ for natural numbers:

$$\begin{array}{ll}
\text{I.} & \neg\,\mathrm{Total}(\bot) \\
\text{II.} & \mathrm{Total}(\mathrm{zero}) \\
\text{III.} & \forall\,x.\,\mathrm{Total}(x) \rightarrow \mathrm{Total}(\mathrm{succ}(x))
\end{array}$$

As with the $\mathrm{force}$ function, it could be wise to add an argument to $\mathrm{Total}$ indicating the type, but for the rest of this section we will only consider the (Haskell) natural numbers. A property such as the commutativity of plus could be expressed with this meta theorem:

$$\forall\,x, y.\,\mathrm{Total}(x) \wedge \mathrm{Total}(y) \rightarrow x + y = y + x$$

We would also like to express infiniteness. Our canonical example of a an infinite value is the natural number `inf = Succ inf`. This is the only infinite natural number so a predicate $\mathrm{Inf}$ for infinite natural numbers is easy to axiomatise:

$$\forall\,x.\,\mathrm{Inf}(x) \leftrightarrow (x = \mathrm{succ}(x))$$

Although defining $\mathrm{Inf}$ is straightforward for natural numbers, further complexities are added for other data types and introduction schemata are needed. But since Haskell functions are continuous and finite, every infinite value is produced from finite information (`repeat`, `iterate`, `enumFrom`), so a predicate $\mathrm{Inf}$ should be enough for our purposes, and $\mathrm{Inf}$ should be defined so that it excludes partial values, that is satisfying $\mathrm{Inf}(x) \rightarrow \mathrm{Total}(x)$.

We can now define finiteness of a value $x$ as $\mathrm{Fin}(x) \leftrightarrow \mathrm{Total}(x) \wedge \neg\mathrm{Inf}(x)$. The results from a function would be necessary to axiomatise to make such axioms usable. To exemplify why an axiomatisation is not trivial, consider the complexities of the plus function:

$$\begin{array}{lllll} \text{I.} & \forall x, y.\, \mathrm{Fin}(x) \wedge \mathrm{Fin}(y) & \leftrightarrow & \mathrm{Fin}(x+y) \\ \text{II.} & \forall x, y.\, \mathrm{Inf}(x) \vee (\mathrm{Fin}(x) \wedge \mathrm{Inf}(y)) & \leftrightarrow & \mathrm{Inf}(x+y) \end{array}$$

Axiom I asserts that $x + y$ is finite iff $x$ and $y$ are. Further, $x + y$ is infinite iff $x$ is or $x$ is finite and $y$ is infinite. This is expressed in Axiom II. How to prove such statuses for the return value of a function in terms of $\mathrm{Fin}$, $\mathrm{Total}$ and $\mathrm{Inf}$ for different statuses of arguments is an open problem. Possible sources for inspiration is the work by Escardó (2008), where topological compactness for data types play a significant rôle.

### 5.2.3 Speculating Lemmas

The previous sections assumed that the necessary lemmas are present as properties in the source file. In practice, this assumption puts a lot of burden on the programmer to both figure out required lemmas and to state them, it is not always clear which lemmas are required to prove a given property. This section discusses some ideas to generate candidate lemmas.

Inductive proofs by rippling enables techniques such as critics (Ireland and Bundy, 1995), which makes lemmas and generalisations when rippling fails. As our approach relies on theorem provers, it is very hard to extract some information from a timeout of a proof by induction.

Another approach is to do property speculation, implemented for Isabelle (Johansson et al., 2009) and Haskell (Claessen et al., 2010). The idea is to use the signatures from the functions in the source file and it try to find equality properties by creating small syntax trees of the functions by testing. Such equalities are well suited as lemmas for more complex properties, and it would be very interesting to extend our approach with a lemma synthesis.

## 5.3 Type Classes

There are some obstacles to support type classes. One is introduced by the technicalities of type inference to decide instances in the program. Another is how to express type classes in logic. An approach would be to use dictionary passing, inlined for concrete types.

A third obstacle is to decide which axioms to use for the functions from a type class, as there is no set agreement on how strong the rules are that we expect from some instances to obey. An example is the Eq class, whose instances typically only constitute an equivalence relation for finite values.

## 5.4 Material Implication and Existential Quantifiers

To be able to prove more complex properties we would like to be able to prove properties with implications and existential quantification. This example about soundness for a prefix predicate is given by Runciman et al. (2008):

```
prop_isPrefixSound :: Eq a => [a] -> [a] -> Prop (Bool :=> Exists [a] Bool)
prop_isPrefixSound xs ys = isPrefix xs ys ==> exists (\xs' -> xs ++ xs' == ys)
```

Finite structural induction could be extended to prove such a property. However, implication is not an admissible property which means that other coinductive proof techniques are not applicable. Existential quantification over functions would require a set of function combinators in the theory to construct a witness.

## 5.5 Other Proof Techniques

Each of the proof methods have their own future work chapter, but there are other techniques not implemented. One is recursion induction (Brady, 1977), where you prove that two functions are equal by asserting that one of them fulfills the same equations as the other. Another is bisimulation (Capretta, 2010), which could possibly be extended to prove properties about total but infinite values. To prove properties about infinite streams, an automated approach using anamorphisms and idiom laws as in Hinze (2010) could be used.

## 5.6 Faster Proof Searches via Predicates

It is possible to add annotations in the equations generated for functions to make the theorem prover not unroll unnecessary definitions and regard these equalities more like definitions. This would avoid the theorem provers to get "lost" in the search space, and in some cases also allow finite models. The predicate indicates that an expression is suitable of reducing to weak head normal form. This predicate would be need to be added at various places in the translated theory, an example is that functions doing pattern matching would need to indicate that the expression under scrutiny is suitable for reduction.

# Chapter 6

# Conclusion

In this thesis we developed a tool able to prove equational properties for Haskell programs. This was accomplished by means of a translation to first order logic and instantiation of several different induction techniques for functional programs. The proof search is carried out by automated theorem provers. The translation handles partial values and some of the available proof techniques can prove properties about corecursive programs generating infinite values, and properties that hold for infinite and partial arguments.

An exhaustive test of the system was carried out on a test suite developed as a part of this project. The test suite consists of over 500 properties. The approach of using automated theorem provers on this kind of problems turned out to be successful; the proved theorems often took much less time than 100 milliseconds for the theorem provers. For the properties in the test suite, over 200 theorems were proved to hold for infinite and partial values, and for total and finite values more than 100. There were properties that could be proved with many techniques, but every technique had some properties that it could exclusively prove.

The future work includes adding a system of adding proved properties as lemmas for proving other properties. This turned out to be difficult for two reasons. First, the translation of pattern matching needs to be carried out in a certain way. Second, adding lemmas that only holds for finite values needs to be tagged by means of predicates or functions in the generated theory that witnesses their finiteness. To appropriately use these tags, it would be necessary to prove termination of functions, and conversely explain which arguments are finite given that a function terminate with a finite value. To generate suitable lemmas it would be interesting to automatically synthesise conjectures, and verify the specifications by testing before proving. Such tools for specifications are QuickSpec (Claessen et al., 2010) and IsaCosy (Johansson et al., 2009). This would relief the user of the burden of figuring out exactly which lemmas are needed to prove desired properties.

Finally, two of Haskell's beauties is its purity and simplicity, and the simple and effective technique of equational reasoning is readily available thanks to referential transparency. We believe that this tool is on the right track of using these properties to accomplish automatic verification of Haskell programs. It should be noted that the tool is able to prove properties without type information and without proving termination for functions. As this thesis shows, first order logic is expressible enough to capture many different aspects of Haskell, including closures of higher order functions, even though no native support for quantifying over functions is available in the logic. By using automated theorem provers the burden of reimplementing a proof system is avoided.

# Bibliography


Armstrong, A., Foster, S., and Struth, G. (2011). Dependently typed programming based on automated theorem proving. *CoRR*, abs/1112.3833.

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*.

Blanchette, J. C. (2011). *Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL*.

Blanchette, J. C., Böhme, S., and Smallbone, N. (2011). Monotonicity or how to encode polymorphic types safely and efficiently.

Böhme, S. and Nipkow, T. (2010). Sledgehammer: Judgement day. In *IJCAR*, pages 107–121.

Bove, A., Dybjer, P., and Sicard-Ramírez, A. (2011). Combining interactive and automatic reasoning in first order theories of functional programs. Available from: `http://www1.eafit.edu.co/asicard/pubs/fotc.pdf`.

Brady, M. (1977). Hints on proofs by recursion induction. *Comput. J.*, 20(4):353–355.

Capretta, V. (2010). Bisimulations generated from corecursive equations. *Electron. Notes Theor. Comput. Sci.*, 265:245–258. Available from: `http://dx.doi.org/10.1016/j.entcs.2010.08.015`.

Claessen, K. (2011). The anatomy of equinox - an extensible automated reasoning tool for first-order logic and beyond - (talk abstract). In *CADE*, pages 1–3.

Claessen, K. and Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279.

Claessen, K., Lilliesträm, A., and Smallbone, N. (2011). Sort it out with monotonicity: translating between many-sorted and unsorted first-order logic. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 207–221, Berlin, Heidelberg. Springer-Verlag. Available from: `http://dl.acm.org/citation.cfm?id=2032266.2032283`.

Claessen, K., Smallbone, N., and Hughes, J. (2010). Quickspec: Guessing formal specifications using testing. In *TAP*, pages 6–21.

Danielsson, N. A. (2010). Beating the productivity checker using embedded languages. Available from: `http://www.cse.chalmers.se/~nad/publications/danielsson-productivity.pdf`.

Danielsson, N. A., Hughes, J., and Jansson, P. (2006). Fast and loose reasoning is morally correct. Available from: http://www.cse.chalmers.se/~nad/publications/danielsson-popl2006-tr.pdf.

Dixon, L. (2005). *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh.

Dixon, L. and Johansson, M. (2007). Isaplanner 2: A proof planner in isabelle. DReaM Technical Report (System description). Available from: http://dream.inf.ed.ac.uk/projects/isaplanner/docs/isaplanner-v2-07.pdf.

Escardó, M. (2008). Exhaustible sets in higher-type computation. *Logical Methods in Computer Science*, 4.

Gibbons, J. and Hutton, G. (2005). Proof methods for corecursive programs. *Fundamenta Informatica*, 66(4):353–366. Available from: http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/corecursive.pdf.

Gordon, A. D. (1994). A tutorial on co-induction and functional programming. In *In Glasgow Functional Programming Workshop*, pages 78–95. Springer. Available from: http://kyhcs.ustcsz.edu.cn/~smyang/report/coinduction/gordon94tutorial.pdf.

Hinze, R. (2010). Concrete stream calculus: An extended study. *Journal of Functional Programming*, pages 463–535. Available from: http://www.cs.ox.ac.uk/ralf.hinze/publications/CSC.pdf.

Hutton, G. and Gibbons, J. (2001). The generic approximation lemma. *Information Processing Letters*, 79:2001. Available from: http://eprints.nottingham.ac.uk/225/1/approx.pdf.

Ireland, A. and Bundy, A. (1995). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:16–1.

Johansson, M., Dixon, L., and Bundy, A. (2009). Isacosy: Synthesis of inductive theorems.

Jones, M. P. (1995). Functional programming with overloading and higher-order polymorphism.

Lindblad, F. and Benke, M. (2004). A tool for automated theorem proving in agda. In *TYPES*, pages 154–169.

Martin-Löf, P. (1980). *Intuitionistic Type Theory*.

McCune, W. (2010). Prover9 and mace4. Available from: http://www.cs.unm.edu/~mccune/prover9/.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Paulson, L. C. (1997). Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7. Available from: http://arxiv.org/pdf/cs/9711105.

Plotkin, G. (1983). Domains. Available from: http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps.

Runciman, C., Naylor, M., and Lindblad, F. (2008). Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell*, pages 37–48.

Sands, D. (1996). Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18:175–234. Available from: http://repository.readscheme.org/ftp/papers/topps/D-221.pdf.

Schulz, S. (2002). E - a brainiac theorem prover.

Sonnex, W., Drossopoulou, S., and Eisenbach, S. (2011). Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report. Available from: http://pubs.doc.ic.ac.uk/zeno/.

Wadler, P. (1992). The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/143165.143169.

Wilson, S., Fleuriot, J., and Smaill, A. (2010). Inductive proof automation for coq (draft).