Masther's thesis to obtain the degree
Master in innovation and research in informatics

Automatic inductive equational reasoning

Jan Mas Rovira

Supervised by Albert Rubio Gimeno

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament de Ciències de la Computació

Facultat d'informàtica de Barcelona
Barcelona, June 2018

Abstract

Functional languages are often praised for their expressive type systems and pure mathematical approach to programming, traits which make it easier to reason about them. Proving properties about our programs allows us to have strong guarantees on their correctness. However, informal hand written proofs are error prone, and computer-checked formal proofs are difficult and tedious. None of these methods are viable if thousands of theorems must be checked in a limited amount of time or if the proving process is part of an automatic process.

In this thesis we try to automate the reasoning process by designing a proof search algorithm which includes structural induction. We present Phileas, an automatic theorem prover of equations between functional expressions. Phileas parses properties specified in Haskell code and compiles them into a simpler internal language, then it tries to prove them. If it succeeds, it outputs the proof in a human readable format.

We evaluate Phileas on a large set of theorems. We also compare it to IsaPlanner and Zeno, similar automatic theorem provers. We found that our implementation is able to prove a considerable amount of *real world* theorems and is more robust than existing tools.

Acknowledgements

I would like to thank Albert, my supervisor, for his guidance during the process of writing this thesis.

I would also like to show my appreciation to Konstantinos, Maria, Arnau and Irene for proof-reading this report and making useful suggestions.

Contents

Part I

Introduction

A functional program defines a set of inductive types as well as a number of function definitions. A function definition is just an equality between two expressions. Functional programs compute their results from evaluating expressions as opposed to executing statements that alter a global state. These expressions are evaluated using rewrite rules, similar to evaluation of functions in traditional mathematics, without any side effect. A consequence of this is that evaluating an expression twice will have the same outcome. This basic mathematical principle allows us to reason about functional expressions as if they were mathematical expressions. Equational reasoning is a technique based on this principle that gives a framework to prove theorems about our programs. Since the only way to express repetition in a functional setting is recursion, we extend the logic framework with structural induction, a principle that allows us to reason about functions on finite recursive data structures.

Our aim is to implement an automatic inductive theorem prover—named Phileas—that is able to use this framework to prove properties about functional programs in a subset of first order logic. Data type definitions, functions and properties are all defined in a Haskell module. Phileas internally uses the GHC compiler, a state of the art compiler for the purely functional language Haskell, to parse and compile Haskell code to a syntactically simpler internal language which we named Phi.

Inductive theorem proving has a long history [1]–[4]. More recent work shows a promising future for automatic inductive theorem provers (AITPs). In particular, we put our attention on Zeno [5] and IsaPlanner [6]. Zeno is the AITP that inspired Phileas and it has a very similar pipeline. Zeno uses a heuristic based on the concept of critical paths to find relevant lemmas to the current theorem. IsaPlanner is an AITP in the form of a plugin for the Isabelle [7] proof assistant that uses the rippling technique [8] to guide rewrite rules application while ensuring termination. Section 11 provides a detailed comparison of Zeno, IsaPlanner and Phileas.

The research of this project—in automatic equational theorem proving—is motivated by its application in the context of model checking algorithms for systems composed of a set of processes that will potentially execute concurrently and access shared resources. Model checking algorithms try to prove properties about a given model, such as the absence of deadlocks, by exhaustively exploring its possible execution paths. The size of the search space makes simple brute force impractical. There are a number of techniques that try to face this problem with different approaches, one of them being *dynamic partial order reduction* [9]. DPOR reduces the search space by avoiding exploration of redundant paths. It does so by dynamically detecting independence between processes. If two processes A, B are independent, then there is no need to check both $A \mid B$ and $B \mid A$.

Recent research [10] presents *constrained dynamic partial reduction*, an extension to DPOR that is able to handle conditional independence between processes given by a set of equality constraints. That is, it provides some sufficient conditions under which two processes are independent. These conditions are given in the form of equalities between expressions in a functional language. This functional language is part of ABS [11], a popular and open source framework used to model concurrent systems and prove properties about them. The goal is to use Phileas to determine and prove which of these constraints will always be true. If a constraint is proven true for all possible values then it can be safely removed. Apart from avoiding the run-time check, removing a constraint may allow us to completely remove a node in the search tree, and hence showing exponential efficiency gains in the experimental evaluation.

This ABS framework together with the model checker based on CDPOR has successfully been applied to test Software Defined Networks (SDN) [12].

The most noteworthy achievements of this research are:

- Implementing Phileas, an AITP that is capable of automatically proving a large set of relevant equational theorems. Some of them useful in the use case of CDPOR.

- Fully support type polymorphism. A feature that is not easily implemented as shown by the fact that Zeno's attempt at doing so failed, as described in section 14

All code is publicly available at https://gitlab.com/janmasrovira/phileas. More details can be found in section 7.

Part II

Phi language

Chapter 1

Syntax and informal overview

In this section we describe the Phi language, Phileas internal language. Phi has all the expressive power of modern functional languages such as Haskell, but its simpler structure allows for more convenient reasoning. It is based on GHC-Core [13], which in turn, is closely related to System F. GHC-Core is an intermediate language of the GHC[1] compiler. System F is an extension to typed lambda calculus with type polymorphism [14]. Figure 1.1 shows Phi's full grammar.

Below we briefly give an informal overview of some parts of the language. If the reader is familiar with functional languages, they may skip to the next section.

A Phi program consists of a set of **data types** and **function definitions**.

A data type definition declares a new type constructor and its data constructors. The constructors are used to build instances of such type. Phi supports only algebraic data types (ADTs). Consequently, there is no support for primitive types such as integers or arrays. ADTs are revisited in more detail in section 4. For example, the data type definition below shows the definition of booleans and lists.

```
data Bool = True | False
data List a = Nil | Cons a (List a)
```

Then, a list containing $True$ followed by $False$ is expressed by composing constructors thus: $Cons\ True\ (Cons\ False\ Nil)$[2].

A function definition declares a variable to be equal to some expression: $f = e$. This assignment should be read as a declaration of equivalence rather than a statement that modifies the contents of a variable.

An expression can be:

- A **defined variable** (or function) is a variable that has previously been defined. It could have been defined in a let expression or in a top level definition.
- An **application** $a\ b$. Note that a does not have to be a lambda or a defined function, it can be any (well typed) expression.
- A **lambda function** is used to abstract an expression over an argument. An applied lambda function $(\lambda x.\ e)\ a$ is equivalent to $e[a/x]$[3].

[1] GHC is the Glasgow Haskell Compiler, considered the standard Haskell compiler.

[2] In functional languages application is often implicit, so $a\ b$ means "apply a to b". In traditional notation we would write $a(b)$. Application is left associative, so $a\ b\ c$ is equivalent to $(a\ b)\ c$

[3] e[a/x] means "replace x by a in e".

- A **constructor** is a variable defined by a data type definition. For example $True$, $False$, $Cons$ or Nil. Constructors always start with a capital letter.
- A **case expression** is used to make a choice based on the structure of some value. A case is composed of a number of alternatives. Each alternative has a pattern and an associated expression. A pattern is a constructor followed by n pattern variables, where n is the arity of the constructor. There is a special alternative—the default alternative—that doesn't have a pattern. If the cased expression is a constructor application, it is pattern matched against all alternatives and the matching alternative expression is used. Then pattern variables are replaced by the cased constructor arguments. This is best understood by example, consider the following expression:

  ```
  case e of
    Cons x xs → xs
    _ → h
  ```

  then, if $e = Cons\ a\ b$, the expression is equivalent to b. If $e = Nil$, the expression is equivalent to h.

  See axioms Case match and Case default in figure 3.3.
- A **let** expression is useful to define auxiliary functions that should be in scope locally to some expression.

Every expression has an associated type. A type can be one of the following:

- A **function type** $a \mapsto b$ is the type of some expression that expects something of type a and returns something of type b.
- A **for all type** is similar to a lambda function at the type level. It is used to abstract a type over a type variable. For example, the type of the polymorphic id function is $\forall a.\ a \mapsto a$. An applied for all type $(\forall x.\ t)\ a$ is equivalent to $t[a/x]$.
- A **type variable** declared by a for all type.
- A **type application** $t\ r$. The left term can be either a for all type or a type variable.
- A **type constructor** is a type such as $Bool$ or $List$. New type constructors can be defined via type definitions.
- A **type constructor application** is used to specialize type constructors. For instance, $List\ Bool$ is a type constructor application that denotes the type of polymorphic lists instantiated to lists of booleans. It is essentially the same as type application where the left term is a type constructor.

The empty type—a type with no constructors—is forbidden. Allowing the empty type would make the reasoning process a lot more cumbersome [15].

As an example, the code fragment below shows the *map* function definition in Haskell and its Phi equivalent. This function applies a function to each element of a list.

```
-- map definition in Haskell
map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (a:as) = Cons (f a) (map a as)


-- map definition in Phi
map :: ∀ a. ∀ b. (a ↦ b) ↦ [a] ↦ [b]
map = λ@a. λ@b. λ(f :: a ↦ b). λ(ds :: [a]).
  case ds of
    Nil → Nil @ b
    Cons a1 as → (Cons @ b) (f a1) (((map @ a) @ b) f as)
```

Phileas has two built-in types: `[]` —the list type—and `Bool`. Equivalent types could easily be defined by the user, but only predefined types can be used in syntactic sugar constructions such as *if then else* expressions, guards and comprehension lists.

Throughout this report we assume that all structures are finite. Reasoning about infinite structures, such as infinite lists, usually requires the use of coinduction [16] and this falls out of the scope of this project.

$$
\begin{array}{rll}
\mathit{Program} = & & \\
& \mathit{DataTypeDefinition}^{*}\ \mathit{Binds} & \text{Phi program} \\
\mathit{DataTypeDef} = & & \\
& \texttt{data}\ \mathit{TypeConstructor}\ \mathit{TypeVar}^{*}\ = & \\
& \quad \mathit{ConstructorDef}\ (\texttt{|}\ \mathit{ConstructorDef}^{*}) & \text{Data type definition} \\
\mathit{ConstructorDef} = & & \\
& \mathit{Constructor}\ \mathit{Type}^{*} & \text{Constructor definition} \\
\mathit{Binds} = & & \\
& (\mathit{DefinedVar} = \mathit{Expression})^{*} & \text{Variable definition} \\
\mathit{Expression} = & & \\
& \mathit{Var} & \text{Variable} \\
& \mid \lambda(\mathit{UniversalVar} :: \mathit{Type}).\ \mathit{Expression} & \text{Lambda asbtraction} \\
& \mid \mathit{Expression}\ \mathit{Expression} & \text{Application} \\
& \mid \texttt{case}\ \mathit{Expression}\ \texttt{of}\ \mathit{CaseAlt}^{+} & \text{Case} \\
& \mid \texttt{let}\ \mathit{Binds}\ \texttt{in}\ \mathit{Expression} & \text{Let abstraction} \\
& \mid \lambda\texttt{@}\mathit{TypeVar}.\ \mathit{Expression} & \text{Type asbtraction} \\
& \mid \mathit{Expression}\ \texttt{@}\ \mathit{Type} & \text{Type application} \\
\mathit{Var} = & & \\
& \mathit{UniversalVar} & \text{Bound by a pattern or a lambda} \\
& \mid \mathit{DefinedVar} & \text{Defined variable} \\
& \mid \mathit{Constructor} & \text{Constructor variable} \\
\mathit{CaseAlt} = & & \\
& \mathit{Constructor}\ \mathit{UniversalVar}^{*} \rightarrow \mathit{Expression} & \text{Pattern match} \\
\mathit{Type} = & & \\
& \mathit{TypeVar} & \text{Type variable} \\
& \mid \mathit{Type} \mapsto \mathit{Type} & \text{Function type} \\
& \mid \forall \mathit{TypeVar}.\ \mathit{Type} & \text{Type abstraction} \\
& \mid \mathit{Type}\ \texttt{@}\ \mathit{Type} & \text{Type application} \\
& \mid \mathit{TypeConstructor}\ \mathit{Type}^{*} & \text{Type constructor application}
\end{array}
$$

Figure 1.1: Phi language definition.

Chapter 2

Typing rules

Inferring the most general type of an expression in presence of polymorphic types and mutually recursive let bindings is a laborious job and for a similar language such as System F it is an undecidable problem [17]. Fortunately GHC implements this task for GHC-Core and annotates each variable with its type. This information is carried to Phi in the compilation process which greatly facilitates the type inference of expressions. Because of this, we present a simplified version of the Let type inference rule.

The environment Γ contains variables and polymorphic type variables in scope. The $::$ operator denotes the typing relation. A construction of the form $\Gamma \vdash e :: \tau$ is called a typing judgment and it should be read as "e has type τ in type environment Γ". Greek letters σ, φ, τ denote types. Greek letter κ denotes a kind[1]. Kind rules are not presented since kinds are ignored in the reasoning process.

$$\textsc{Variable}\quad \frac{var :: \tau \in \Gamma}{\Gamma \vdash var :: \tau} \qquad \textsc{Abstraction}\quad \frac{\Gamma, x :: \tau \vdash e :: \sigma}{\Gamma \vdash \lambda(x :: \tau).\ e :: \tau \mapsto \sigma} \qquad \textsc{Application}\quad \frac{\Gamma \vdash f :: \tau \mapsto \sigma \qquad \Gamma \vdash x :: \tau}{\Gamma \vdash f\ x :: \sigma}$$

$$\textsc{Type abstraction}\quad \frac{\Gamma, \tau :: \kappa \vdash e :: \sigma}{\Gamma \vdash \lambda@\tau.\ e :: \forall\tau.\ \sigma} \qquad \textsc{Type application}\quad \frac{\Gamma \vdash e :: \forall\tau.\ \sigma \qquad \varphi :: \kappa \in \Gamma}{\Gamma \vdash f\ @\ \varphi :: \sigma[\varphi/\tau]}$$

$$\textsc{Case}\quad \frac{\Gamma \vdash a_1 :: \tau\ \ldots\ \Gamma \vdash a_n :: \tau \qquad \Gamma \vdash d :: \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\ p_1 \to a_1;\ \ldots; p_n \to a_n; \_ \to d\ \} :: \tau}$$

$$\textsc{Let}\quad \frac{\Gamma, d_1 :: \tau_1, \ldots, v_n :: \tau_n \vdash e :: \sigma}{\Gamma \vdash (\texttt{let}\ v_1 = d1;\ \ldots\ ; v_n = d_n \texttt{in}\ e) :: \sigma}$$

Figure 2.1: Type inference rules.

[1] A kind is the type of a type.

Part III

Automating proofs

Chapter 3

Logic

Phileas logic is based on first order logic and extended with type polymorphism. Below we list the differences between first order logic and Phileas logic.

- No existential quantification.
- Terms have types, which can be polymorphic.
- It includes equality ($\equiv$) as a primitive predicate and implication ($\Rightarrow$) as the only logical connective symbol.
- Includes structural induction. Explained in detail in section 4.
- Includes extensional equality of functions.
- Includes some axioms specific to the Phi language semantics. These axioms are defined in figure 3.3.

We believe that any proof generated by Phileas could be expressed by composing these axioms. On the other hand, Phileas will not be able to prove all theorems provable from these axioms since the search is not exhaustive.

A *property* is a formula in first order logic that expresses something that we want Phileas to prove. Figure 3.1 inductively defines a property. It should be noted that even though the grammar allows for much more complex properties, Phileas is focused on properties of the following form:

$$\forall a_1 \ldots a_n.\ e_1 \equiv e_2 \Rightarrow \ldots \Rightarrow e_{m-1} \equiv e_m$$

$$\frac{\Gamma \vdash a :: \tau \qquad \Gamma \vdash b :: \tau}{\Gamma \vdash a \equiv b :: \mathit{Property}} \qquad\qquad \frac{\Gamma, x :: \tau \vdash P :: \mathit{Property}}{\Gamma \vdash \forall x.\ P :: \mathit{Property}}$$

$$\frac{\Gamma \vdash P :: \mathit{Property} \qquad \Gamma \vdash Q :: \mathit{Property}}{\Gamma \vdash P \Rightarrow Q :: \mathit{Property}}$$

Figure 3.1: Property inductive definition.

Figures 3.2 and 3.3 show the axioms of the logic. Lower case letters refer to expressions and variables. P, Q, R to properties. K to a constructor.

$$\textsc{Reflexivity}\quad \frac{}{a \equiv a} \qquad \textsc{Symmetry}\quad \frac{a \equiv b}{b \equiv a} \qquad \textsc{Transitivity}\quad \frac{a \equiv b \qquad b \equiv c}{a \equiv c}$$

$$\textsc{Substitution}\quad \frac{a \equiv b \qquad x \equiv y}{a[y/x] \equiv b}$$

$$\textsc{Antecedent intro}\quad \frac{Q}{P \Rightarrow Q} \qquad \textsc{Antecedent commut}\quad \frac{P \Rightarrow (Q \Rightarrow R)}{Q \Rightarrow (P \Rightarrow R)} \qquad \textsc{Antecedent elim}\quad \frac{P \Rightarrow P}{P}$$

$$\textsc{Absurd}\quad \frac{}{\bot \Rightarrow P} \qquad \textsc{Imply transitivity}\quad \frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R} \qquad \textsc{Modus ponens}\quad \frac{P \qquad P \Rightarrow Q}{Q}$$

$$\textsc{Generalization}\quad \frac{v \notin \Gamma \qquad \Gamma, v :: \tau \vdash P(v)}{\Gamma \vdash \forall x :: \tau.\ P(x)} \qquad \textsc{Instantiation}\quad \frac{\Gamma \vdash e :: \tau \qquad \forall v :: \tau.\ P(v)}{\Gamma \vdash P(e)}$$

Figure 3.2: Axiomatic deduction rules.

Case match

$$\frac{\begin{array}{c} K, K_1,\ \dots\ K_c \text{ are constructors} \\ K \equiv K_i \qquad K \not\equiv K_1\ \dots\ K \not\equiv K_{i-1} \\ \texttt{case}\ K\ a_1\ \dots\ a_n\ \texttt{of}\ \{\ K_1\ x_{1,1}\ \dots\ x_{1,m_1} \to e_1;\ \dots\ ;\ K_c\ x_{c,1}\ \dots\ x_{c,m_c} \to e_c;\ \_ \to d\ \} \end{array}}{\texttt{case}\ K\ a_1\ \dots\ a_n\ \texttt{of}\ \{\ \texttt{...}\ \} \equiv e_i[a_1/x_{i,1},\ \dots\ , a_n/x_{i,n}]}$$

Case default

$$\frac{\begin{array}{c} K, K_1,\ \dots\ K_c \text{ are constructors} \\ K \not\equiv K_1\ \dots\ K \not\equiv K_c \\ \texttt{case}\ K\ a_1\ \dots\ a_n\ \texttt{of}\ \{\ K_1\ x_{1,1}\ \dots\ x_{1,m_1} \to e_1;\ \dots\ ;\ K_c\ x_{c,1}\ \dots\ x_{c,m_c} \to e_c;\ \_ \to d\ \} \end{array}}{\texttt{case}\ K\ a_1\ \dots\ a_n\ \texttt{of}\ \{\ \texttt{...}\ \} \equiv d}$$

Definition

$$\frac{f \overset{def}{=} e \in Definitions}{f \equiv e}$$

Extensionality

$$\frac{\forall x. f\ x \equiv g\ x}{f \equiv g}$$

Beta equivalence

$$\frac{}{(\lambda x.\ e)\ a \equiv e[a/x]}$$

Constructor intro

$$\frac{K \text{ is constructor} \qquad a_1 \equiv b_1\ \dots\ a_n\ \equiv\ b_n}{K\ a_1\ \dots\ a_n \equiv K\ b_1\ \dots\ b_n}$$

Constructor elim

$$\frac{K \text{ is constructor} \qquad K\ a_1\ \dots\ a_n \equiv K\ b_1\ \dots\ b_n}{a_1 \equiv b_1\ \dots\ a_n\ \equiv\ b_n}$$

Constructor absurd

$$\frac{K_{1,2} \text{ are constructors} \qquad K_1 \not\equiv K_2 \qquad K_1\ a_1\ \dots\ a_n \equiv K_2\ b_1\ \dots\ b_m}{\bot}$$

Structural induction

$$\frac{\begin{array}{c} \Gamma \vdash e :: \mathrm{T} \\ \mathrm{T} \text{ is a data type} \\ \{\ K_1,\ \dots\ , K_n\ \} = ConstructorsOf(\varphi) \\ \mathcal{C}(K_1)\ \dots\ \mathcal{C}(K_n) \end{array}}{\Gamma \vdash P(e)}$$

$$\text{where } \mathcal{C}(K) = \forall (K\ a_1\ \dots\ a_n) :: \mathrm{T}.\ \left(\bigwedge_{\substack{i \in 1 \dots j \\ a_i :: \mathrm{T}}} P(a_i) \right) \Rightarrow P(K\ a_1\ \dots\ a_n)$$

The symbol $\bigwedge$ is syntax sugar for an aggregation of $\Rightarrow$ with right associativity.

Figure 3.3: Part 2. Axiomatic deduction rules.

Chapter 4

Structural induction

This section describes and gives some theoretical background to structural induction in the context of inductively defined data types [18].

Mathematical induction is a proof technique that can be used to prove properties on the natural numbers. This technique consists in two parts. (1) Proving that it holds for 0, and (2) proving that it holds for $n+1$ assuming that it holds for n. If these two parts can be proved, then the principle of mathematical induction asserts that this property holds for all natural numbers. As an inference rule:

$$\textsc{Mathematical Induction}\quad \frac{P(0) \qquad \forall n.\ P(n) \Rightarrow P(n+1)}{\forall m.\ P(m)}$$

In order to proof theorems on Phi, and any other pure functional language for that matter, we need a more general principle called structural induction, which is the backbone of any nontrivial proof on a recursive function.

First we will glance over *algebraic data types* (ADTs or data types in short). ADTs are structures that are defined by specifying its possible shapes in terms of *data constructors* (constructors in short) and the types of their children. A type can have one or more constructors and a constructor can have zero or more children. The identifier of the algebraic data type is also called type constructor. For example, the data type `Bool` has two constructors with no children, `False` and `True`.

In mathematical notation an ADT definition has the form:

$$\texttt{data}\ T\ p_1\ ...\ p_n = K_1\ t_{1,1}\ ...\ t_{1,a_1} \mid ... \mid K_m\ t_{m,1}\ ...\ t_{m,a_m}$$

On the left hand side there is T, the type constructor being defined. A type constructor can have $0 \leq n$ polymorphic type variables, p_i, which are in scope in the definition of the data constructors. We say that a type is polymorphic if it has at least one polymorphic type variable ($1 \leq n$). On the right hand side there is the definition of each of the constructors K_i. Phi does not support the empty type, therefore there must be at least one constructor defined for every type ($1 \leq m$). Each of the constructors can have a different arity, represented by $0 \leq a_i$. Finally $t_{i,j}$ are the type arguments of the constructors.

It is worth noting that the only way to build something of type T is by applying one of its constructors to some arguments of the proper types. A consequence of this is expressed in the following theorem.

Theorem 4.0.1 *Every value of a data type T can be uniquely expressed with one of its constructors in its head.*

$$\forall e :: T\ \exists! K \in T, \exists a_1,\ \dots\ , a_n\ \textit{such that}\ e = K\ a_1\ \dots\ a_n$$

Here $K \in T$ means that K is one of the constructors of T.

A value of some data type T can be viewed as a tree where the root is a constructor of T, other nodes are constructors (not necessarily of T) and leaves are either constant constructors or constructors which children have a function type. This is the key concept of structural induction since it gives a sense of finite size for any value of a data type. To denote this we introduce the immediate child relation $\prec_c$. We say that a is related to b if a is an immediate child of b.

$$x \prec_c K\ a_1\ \dots\ a_n \text{ if } \exists i.\ x = a_i$$

Because we are only considering finite structures, it should be obvious that $\prec_c$ is a *well-founded relation*. Remember that a relation is well-founded if there is no infinite descending chain $\dots \prec a_1 \prec a_0$.

Now we introduce the principle of *well-founded induction*, a more general principle than mathematical induction which is not restricted to natural numbers, instead, it is applicable to any set with a well-founded relation. This principle as an inference rule is as follows.

$$\textsc{Well-founded induction}\quad \frac{\forall a \in A.\ (\forall x \in A.\ x \prec a \Rightarrow P(x)) \Rightarrow P(a)}{\forall a \in A.\ P(a)}$$

We will use this as a starting point to informally derive the principle of structural induction applied to data types. First, let us change the order relation $\prec$ to the immediate child relation $\prec_c$ and the membership relation $a \in A$ to the typing relation $a :: T$, where T is a data type. We have:

$$\frac{\forall a :: T.\ ((\forall x :: T.\ x \prec_c a \Rightarrow P(x)) \Rightarrow P(a)}{\forall a :: T.\ P(a)}$$

Now, let us focus on the condition above the horizontal line. Because of theorem 4.0.1 we can rewrite a as a constructor application $K\ a_1\ \dots\ a_n$. Since the data type T can have more than one constructor, it is convenient to abstract it as a parameter.

$$\mathcal{C}(K) = \forall (K\ a_1\ \dots\ a_n :: T).\ ((\forall x :: T.\ x \prec_c K\ a_1\ \dots\ a_n \Rightarrow P(x)) \Rightarrow P(K\ a_1\ \dots\ a_n)$$

Continue by centering the attention to the induction hypotheses, namely $\forall x :: T.\ x \prec_c K\ a_1\ \dots\ a_n \Rightarrow P(x)$. By the definition of the immediate child relation $\prec_c$, there will be an induction hypothesis for each a_i that is of type T. We can write this as a constrained conjunction:

$$\mathcal{C}(K) = \forall (K\ a_1\ \dots\ a_n) :: T.\ \left(\bigwedge_{\substack{i \in 1 \dots j \\ a_i :: T}} P(a_i) \right) \Rightarrow P(K\ a_1\ \dots\ a_n)$$

Finally, with the help of the previous abstraction, we can define the axiom of structural induction:

$$\textsc{Structural Induction}\quad \frac{\begin{array}{c}\Gamma \vdash e :: T \\ T \text{ is a data type} \\ \{\ K_1,\ \ldots\ , K_n\ \} = ConstructorsOf(T) \\ \mathcal{C}(K_1)\ \ldots\ \mathcal{C}(K_n)\end{array}}{\Gamma \vdash P(e)}$$

It can be pedagogical to observe that mathematical induction is just an special case of structural induction. Consider the inductive definition of natural numbers as a Phi data type:

```
data Nat = Zero | Suc Nat
```

Then we have that $\mathcal{C}(Zero) = P(Zero)$ and $P(Suc) = \forall Suc\ n.\ P(n) \Rightarrow P(Suc\ n)$, which are exactly the two conditions needed to prove P using mathematical induction.

Limitation. It is worth remarking that the immediate child relation $\prec_c$ prevents us from using the induction hypothesis on deeper children. This is a limitation in mutually recursive data types such as n-ary trees. Consider the following definition:

```
data Tree x = Node x (List (Tree x))
```

The children of $Node$ are of types x and $List\ (Tree\ x)$. Since none of them is exactly $Tree\ x$, when applying induction to this type no induction hypotheses will be available.

Chapter 5

A guided proof

This section contains a simple proof that should help the reader to understand the basic skeleton of a proof with structural induction.

First, we define natural numbers and lists as data types. We avoid polymorphism of lists in order to ease the readability of the code and proof.

```
data Nat = Zero | Suc Nat
data List = Nil | Cons Nat List

length :: List ↦ Nat
length = λl. case l of
  Nil → Zero
  Cons a as → Suc (length as)

(+) :: Nat ↦ Nat ↦ Nat
(+) = λa. λb.
  case a of
    Zero -> b
    Suc a + b = Suc (a + b)

(++) :: List ↦ List ↦ List
(++) = λa. λb.
  case a of
    Nil -> b
    Cons a as ++ b = Cons a (as ++ b)
```

Definition. A monoid is a set with an associative operation and an identity element.

Definition. A monoid homomorphism is a function $f :: A \mapsto B$ such that $A\langle\star, i_A\rangle$, $B\langle\diamond, i_B\rangle$ are monoids and:

- $f\ i_A \equiv i_B$
- $\forall x, y \in A.\ f\ (x \star y) \equiv f\ x \diamond f\ y$

Postulate. Naturals form a monoid with + and *Zero*.

Postulate. Lists form a monoid with ++ (the concatenate operator) and *Nil*.

Theorem 5.0.1 *The function length is a monoid homomorphism from List to Nat.*

Proof The proof consists of two parts:

1. $length\ Nil \equiv Zero$
2. $\forall a, b :: List.\ length\ (a{+}{+}b) \equiv length\ a + length\ b$

The first part is very simple and it will be done by only applying the axioms. The second part is more complex and it will be done in a pen and paper style.

Proof of 1. $length\ Nil \equiv Zero$

Applying the DEFINITION axiom to *length* and SUBSTITUTION gives:

$$\begin{array}{lcl} (\lambda l.\ \texttt{case}\ l\ \texttt{of} & & \\ \quad Nil \to Zero & \equiv & Zero \\ \quad Cons\ a\ as \to Suc\ (length\ as))\ a & & \end{array}$$

Applying BETA REDUCTION gives:

$$\begin{array}{lcl} \texttt{case}\ Nil\ \texttt{of} & & \\ \quad Nil \to Zero & \equiv & Zero \\ \quad Cons\ a\ as \to Suc\ (length\ as) & & \end{array}$$

Applying CASE MATCH gives $Zero \equiv Zero$ which is closed using REFLEXIVITY.

Proof of 2. $\forall a, b :: List.\ length\ (a{+}{+}b) \equiv length\ a + length\ b$

In order to make this part of the proof less verbose and more readable we will not unfold function definitions and we will implicitly apply beta reduction and pattern matching simplifications.

We begin by applying induction on a, which has type *List*. By the axiom of structural induction, we have to prove the property for each constructor. Thus, we have a base case for *Nil* and an inductive case with an induction hypothesis for *Cons*.

- **Base case**:

  Prove: $length\ (Nil{+}{+}b) \equiv length\ Nil + length\ b$

  By using the definitions of ++ and + it simplifies to $length\ b \equiv length\ b$, which is proved by REFLEXIVITY.

- **Inductive case**:

  Prove: $length\ ((Cons\ x\ xs){+}{+}b) \equiv length\ (Cons\ x\ xs) + length\ b$

  Assuming: $length\ (xs{+}{+}b) \equiv length\ xs + length\ b$

  Apply a chain of transformations:

$$\begin{array}{rr} length\ ((Cons\ x\ xs){+}{+}b) \equiv length\ (Cons\ x\ xs) + length\ b & \text{Definition of ++} \\ length\ (Cons\ x\ (xs{+}{+}b) \equiv length\ (Cons\ x\ xs) + length\ b & \text{Definition of } length \\ Suc\ (length\ (xs{+}{+}b)) \equiv Suc\ (length\ xs) + length\ b & \text{Definition of +} \\ Suc\ (length\ (xs{+}{+}b)) \equiv Suc\ (length\ xs + length\ b) & \text{Congruence on } Suc \\ length\ (xs{+}{+}b) \equiv length\ xs + length\ b & \text{Apply induction hypothesis} \end{array}$$

■

As an appetizer, we show the proof that Phileas generates for this theorem. Notice that not all the steps that we mentioned are in the proof. This is because Phileas tries be succinct in its proofs to improve readability. More information on this can be found in section 6.3.

```
Induction on (a1 :: List a)
  Nil
    Reflexivity ... ≡ ...
  Cons
    Congruence on Suc
        Close using hypothesis ... ≡ ...
```

Phileas tags some proof steps with additional information. For this example we have removed it (and substitute it with `...`) to avoid obfuscating the proof.

For more information on how proofs are presented to the user, refer to section 9.

Chapter 6

Search algorithm

The search algorithm tries to construct a valid proof for a goal property using a guided backtracking algorithm. In this section we describe the components of the algorithm.

A proof is a tree composed of *proof steps*. Think of a proof step as a composition of axiomatic inference rules defined in 3.2 and 3.3. Section 6.2 describes each of the proof steps and give arguments to support their soundness with respect to the axioms. It is convenient to express proofs in terms of proof steps instead of application of axioms because they operate on a higher level and the resulting proofs are more lucid.

The application of a proof step to some property will have one of the outcomes listed below:

- **Failed**. The prover was unable to close this branch of the proof.
- **Success**. The application of the proof step succeeded and it found a valid proof.
- **Absurd**. A contradiction was found in the goal equality. This result is internally used to prune the search tree, see section 6.1.

Before trying to apply a proof step, the goal property is simplified using a series of rules described in section 6.3.

6.1 Subproof aggregation

As mentioned before, a proof step may convert the current property into a number of subgoals, or, at a given point more than one proof step may be applied. Because of this, we need to define some methods that aggregate subproof results into a new result according to the context of the aggregation.

- **Disjunction**. It corresponds to a choice. For example, when several proof steps are applicable. It sequentially tries each subproof and results in:
  - **Failed** if all subgoals fail or if there were no subgoals.
  - **Success** if it finds a valid subproof. The remaining subproofs are not checked.
  - **Absurd** if a subproof results in an absurd. The remaining subproofs are not checked.

- **Strong conjunction**. It corresponds to an *if and only if* condition. It sequentially tries each subproof and results in:
  - **Failed** if at least one subproof failed.
  - **Success** if all subproofs succeeded or if there were no subgoals.
  - **Absurd** if a subproof results in an absurd. The remaining subproofs are not checked.
- **Weak conjunction**. It corresponds to an implication. It sequentially tries each subproof and results in:
  - **Failed** if at least one subproof failed or returned an absurd.
  - **Success** if all subproofs succeeded or if there were no subgoals.
  - **Absurd**. It never returns an absurd.

Note that nested application of this aggregation methods corresponds to a backtracking search with pruning in DFS order.

6.2 Proof steps

6.2.1 Reflexivity

Trivial proof step that closes the proof using the REFLEXIVITY axiom when possible, i.e. it closes the proof when both sides of the goal are syntactically equal.

6.2.2 Congruence

An important property of constructors is:

$$K\ a_1\ \dots\ a_n \equiv K\ b_1\ \dots\ b_n \Leftrightarrow a_1 \equiv b_1 \wedge \dots \wedge a_n \equiv b_n$$

$\boxed{\Rightarrow}$ follows from CONSTRUCTOR ELIM and $\boxed{\Leftarrow}$ follows from CONSTRUCTOR INTRO.

The congruence proof step consists in reducing a goal equality of the form $K\ a_1\ \dots\ a_n \equiv K\ b_1\ \dots\ b_n$ into a several subgoals $a_1 \equiv b_1\ \dots\ a_n \equiv b_n$, which are aggregated using using strong conjunction.

Constructor application preserves the structure of its components, thus, it can be thought as a homomorphism and equality as an associated congruence relation, hence the name of the proof step.

6.2.3 Induction

This proof step is based on the principle of structural induction. First, it inspects the property and uses a heuristic to find feasible induction candidates. In plain words, an induction candidate is an expression that has a type defined by a data type definition and it is blocking the progress of the proof. Candidates are precisely defined in figure 6.1. Then, for each candidate, it starts the corresponding proof by induction. These proofs are aggregated by disjunction.

A proof by induction will have a subproof for each constructor as described in section 4, and these proofs are aggregated using weak conjunction. For reference, we copy the part of the axiom relevant to one constructor:

$$\mathcal{C}(K) = \forall (K\ a_1\ ...\ a_n) :: T.\ \left(\bigwedge_{\substack{i \in 1...j \\ a_i :: T}} P(a_i) \right) \Rightarrow P(K\ a_1\ ...\ a_n)$$

The variables a_1... a_n are fresh variables that may themselves become induction candidates in the subproof. In order to avoid a loop by infinitely applying induction on generated variables we introduce the concept of *variable history*.

A history is a multiset of *events* that indicate the origins of a variable. The prover will use this information to prune the search tree and avoid infinite loops. History events are generated when the prover applies induction to a candidate. There are two types of candidates and each of them have their own event:

- **Constructor expansion**. When there is an equality in the property of the form $K\ b_1\ ...\ b_n \equiv b$, the expression a becomes a candidate. When the prover applies induction to a candidate of this kind, an *expansion event* is generated. An expansion event is tagged and identified with the type of b.
- **Case induction**. When there is an expression in a case that is not a constructor application, and thus, prevents case simplification, it becomes a case induction candidate. During the compilation process, Phileas tags each case expression with a unique id called *case id*. Then, when the prover applies induction to this kind of candidate, it generates a case induction event. A case induction event is tagged and identified with the corresponding case id.

We define the *history of an expression* to be the union variable histories that appear in it.

Then, when a fresh variable is created during an induction step, it inherits the history of the candidate expression that caused the induction. Additionally the prover adds the newly generated event to the new variable.

Finally, the history of an expression is used by the candidate search heuristic using the following criteria:

If e is a potential candidate but it would generate an event that has already happened n times in its history and $n \geq k$, then e is discarded.

Note that k is a parameter that regulates this restriction. Through experimental research we have found that $k = 2$ is a good default value. It can be modified by the user through a command flag as described in section 7.2.

$$
\begin{aligned}
&\textit{canidatesPred}\ (P \Rightarrow Q) = \textit{candidatesPred}\ P \cup \textit{candidatesPred}\ Q \\
&\textit{canidatesPred}\ (K\ a_1\ \dots\ a_n \equiv b) = \{b\} \\
&\textit{canidatesPred}\ (a \equiv K\ b_1\ \dots\ b_n) = \{a\} \\
&\textit{canidatesPred}\ (a \equiv b) = \textit{candidatesExpr}\ a \cup \textit{candidatesExpr}\ b \\
&\textit{canidatesPred}\ (\forall x.\ P) = \emptyset \\
\\
&\textit{canidatesExpr}\ (f\ x) = \textit{candidatesExpr}\ f \\
&\textit{canidatesExpr}\ (\texttt{case[id]}\ e\ \texttt{of}\ \{\ \dots\ \}) = \{(id, e)\} \cup \textit{candidatesExpr}\ e \\
&\textit{canidatesExpr}\ (\lambda\texttt{@}t.\ e) = \textit{candidatesExpr}\ e \\
&\textit{canidatesExpr}\ (e\ \texttt{@}\ t) = \textit{candidatesExpr}\ e \\
&\textit{canidatesExpr}\ \textit{otherwise} = \emptyset
\end{aligned}
$$

Figure 6.1: Induction candidates.

6.2.4 Use hypothesis

It searches for an antecedent syntactically equal to the goal (applying symmetry if necessary), if it succeeds, it closes the proof. This proof step is usually used to apply some induction hypothesis although it can also use antecedents provided by the user in the original goal.

Note that Phileas does not try to derive new lemmas from the current hypotheses nor it tries to instantiate universally quantified hypotheses.

The proof of soundness follows from the axioms ANTECEDENT COMMUT, ANTECEDENT ELIM and SYMMETRY.

6.2.5 Extensional equality

Our logic includes the axiom of EXTENSIONAL EQUALITY, therefore two functions are considered to be equal if they are extensionally equal, that is, if they exhibit the same behaviour when applied to the same argument. More concretely $f \equiv g \Leftrightarrow \forall x.f\ x \equiv g\ x$.

This proof step, checks for the type of both sides of the goal equality[1]. When this type is a function type $a \mapsto b$, a fresh variable of type a is created and applied to both sides.

This proof of soundness follows from the axioms of EXTENSIONAL EQUALITY and GENERALIZATION.

6.2.6 Factor application

As previously discussed, in a pure functional language, functions can be treated as normal mathematical functions. A basic property that satisfy all mathematical functions is that every element in its domain has a unique image. From this we can derive the following property:

$$a_1 \equiv b_1 \Rightarrow\ \dots \Rightarrow\ a_n \equiv b_n \Rightarrow f\ a_1\ \dots\ a_n \equiv f\ b_1\ \dots\ b_n$$

[1] Both sides will always have the same type.

Similarly to congruence, this proof step simplifies the goal into a number of subproofs. However, since the condition is weaker, it aggregates its subproofs using weak conjunction.

6.2.7 Absurd in hypotheses

A constructor contradiction is an equality of the form

$$K_1\ a_1\ ...\ a_n \equiv K_2\ b_1\ ...\ b_m \qquad K_1 \not\equiv K_2$$

where K_1, K_2 are constructors.

When a constructor contradiction is found in one of the antecedents of the current goal, the goal is closed.

The proof of soundness follows from the axioms of Antecedent commut, Constructor absurd and Absurd.

6.2.8 Absurd in goal

If all the hypotheses are trivially true, and the goal contains a constructor contradiction, results in an absurd.

We say that a property is trivially true when it can be proved using only Reflexivity.

6.3 Simplification rules

There are two types of simplification rules that work on different levels:

- **Expression** level. A syntactic transformation on an expression that is congruent with respect to the equality relation $\equiv$.
- **Property** level. A transformation on a property. The resulting property will only be provable by the axioms if the original property was provable as well.

Phileas applies a set simplification rules before every proof step. Simplifications are not explicit in the resulting proof. However, we believe that by not including simplifications in the proofs, they become less verbose and equally valuable, since simplifications should always be obvious to the human eye.

Simplification rules are applied repeatedly until none is applicable.

6.3.1 Apply definition

Consists in replacing a defined variable by its definition. Trying to blindly apply this rule would cause an infinite loop for recursive functions. Because of that, it is important to apply it only when the definition of the function is necessary to bring the proof forward.

It replaces a variable f by its definition in the following cases:

- When the root expression is a defined variable f.
- When the root expression is an application $f\ a$ and f is a defined variable.
- When the root expression is a `case` f `of { ... }` and f is a defined variable.
- When the root expression is a `case` $f\ a$ `of { ... }` and f is a defined variable.

By root we mean the topmost expression of both sides of an equality. For instance, consider the property $f\ x \equiv Cons\ a\ as$. Assuming f, x, a, as are defined variables and $Cons$ is a constructor, only f would be replaced by its definition.

Its soundness follows from the Definition and Substitution axioms.

6.3.2 Beta reduction

Consists in reducing a lambda function application by using the Beta equivalence axiom from left to right. Applicable to any subexpression.

6.3.3 Pattern matching

Consists in reducing a case expression by using the axioms {Case match} and Case default from left to right. Applicable to any subexpression.

6.3.4 Case factoring

Consider a case expression of the form

$$
\begin{array}{l}
\texttt{case}\ e\ \texttt{of} \\
\qquad p_1 \to K\ a_1 \\
\qquad \ldots \\
\qquad p_n \to K\ a_n \\
\qquad \_ \to K\ d
\end{array}
$$

Where K is a constructor that expects exactly one argument and it is applied to each of the alternative expressions. Then by doing case analysis[2] on e and applying Case match and Case default we can prove that it is equivalent to

$$
\begin{array}{l}
K\ (\texttt{case}\ e\ \texttt{of} \\
\qquad p_1 \to a_1 \\
\qquad \ldots \\
\qquad p_n \to a_n \\
\qquad \_ \to d)
\end{array}
$$

This simplification is applicable to any subexpression.

6.3.5 Universal variable instantiation

Consists in applying the axiom of Instantiation. That is, it substitutes a universally quantified variable for a fresh variable of the appropriate type. This is applied to both the goal and the hypotheses.

[2]By case analysis we refer to induction without the need of induction hypotheses.

Note that when this is applied to an antecedent it is being weakened and the whole property may become unprovable. For example, the following property:

$$\forall a\ b.\ (\forall x\ y.\ x \equiv y) \Rightarrow a \equiv b$$

is simplified to:

$$x' \equiv y' \Rightarrow a' \equiv b'$$

where x', y', a', b' are fresh variables. After this simplification the antecedent has become weaker and property is no longer true. As mentioned in section 3, properties of this form are not the focus of this project.

6.3.6 Unify variable synonyms

The goal of this simplification is to remove variable synonyms from the set of hypotheses. We consider a variable synonym an equality of the form $v_1 \equiv v_2$, where v_1 and v_2 are variables.

Suppose the set of variable synonyms in the hypotheses is $S = \{v_1 \equiv v_2,\ \dots, v_{n-1} \equiv v_n\}$. Then consider an undirected graph $G = \langle V, E \rangle$, where $V = \{v_1,\ \dots, v_n\}$ and $E = \{\ (a, b) \mid\ \ a \equiv b \in S \lor b \equiv a \in S\}$. Let $\sim$ be a relation where $a \sim b$ iff a and b belong to the same component in G. It follows from the definition of a connected component that $\sim$ is an equivalence relation. Note that two variables are transitively identical if they are related. Then, for every equivalence class we pick a representative r_i. Finally, for each r_i and for each variable v_j we substitute all instances of v_i by r_i if $r_i \sim v_j$.

Part IV

System description

Chapter 7

Phileas user guide

This section is intended to be a short user guide for Phileas. It guides the reader through the process of installation and usage of the prover.

7.1 Installation

The source code is freely available at https://gitlab.com/janmasrovira/phileas.

The easiest way to compile is using the Stack[1] build tool. Stack is a modern tool in charge of downloading the correct version of the Haskell compiler and all the necessary libraries to build the project. Using Stack is the recommended method since it allows for reproducible builds by automatically installing the same compiler version and dependencies that were used during the development process. As an additional resource, we suggest watching the screencast linked in figure 7.1.

```
git clone git@gitlab.com:janmasrovira/phileas.git
cd phileas
stack setup -- downloads the compiler (GHC)
stack test  -- compiles Phileas and runs the test suite
```

7.2 Usage

Once you have Phileas installed. The first step is to write a Haskell module containing the data type and functions definitions needed by your target theorem. Phileas supports all Haskell syntax, however, there is the restriction that the source code of every definition must be available at compile time. A common way to do it is to add the `NoImplicitPrelude` pragma to the top of the file and write all the definitions in the same module.

Figure 1 shows an example module containing the inductive definition of the natural numbers and an order relation between them. It also includes the property $\forall a\ b :: Nat.\ a \leq b \Rightarrow b \leq a \Rightarrow a \equiv b$. Note that we imported a module called `PhileasPrelude`. This module is shipped with Phileas and exports the necessary operators to build properties, namely ≡, ⇒ and ∀.

[1] https://docs.haskellstack.org/en/stable/README/

By observing the types of theses operators below we can see that they match the grammar described in section 3. The ⇒ operator has right associativity, so in the common use case parentheses will not be needed. The ∀ operator is hardly ever needed since it is implicitly added for each argument of a property. So `f a = a ≡ a` is equivalent to `f = (∀) (\a -> a ≡ a)`.

```
(∀) :: (a → Property) → Property
(≡) :: a → a → Property
(⇒) :: Property → Property → Property
```

The next step is invoking Phileas to prove the properties defined in the module.

```
./Phileas -i Properties.hs --include Path/To/PhileasPreludeDir
```

The `-i` flag is used to pass the input file. The `--include` path is used to extend the include path. The file PhileasPrelude.hs must be in a directory in the include path.

The result of the previous command will be:

```
Phileas has found 1 properties to prove.

partialOrderAntisimmetry: The property was proven true.

1 out of 1 theorems were proved. 0 could not be proved.
```

Then, if we want Phileas to show us the proof we need to pass the `--show-proof` flag. Invoking Phileas again with this flag gives us the proof.

```
Induction on (a :: Nat)
  Zero: Induction on (b :: Nat)
          Zero: Reflexivity True ≡ True
          Suc: Contradiction in hypothesis: False ≡ True
  Suc: Induction on (b :: Nat)
        Zero: Contradiction in hypothesis: False ≡ True
        Suc: Induction on (var.94 :: Nat)
                Zero: Contradiction in hypothesis: False ≡ True
                Suc: Induction on (case var.98 of
                                    Zero → False
                                    Suc b → $c==$ var.100 b :: Bool)
                      True: Reflexivity True ≡ True
                      False: Induction on (var.98 :: Nat)
                                Zero: Contradiction in hypothesis: False ≡ True
                                Suc: Contradiction in hypothesis: False ≡ True
```

Phileas supports a number of flags that change its behaviour. We list the full list of flags below in two categories:

Relevant to the user:

- `-h, --help`: Shows a help text describing the basic usage and the description of each flag.
- `-i, --input-file`: To give the input file.

- `-t, --timeout`: A time limit for each property in seconds.
- `-l, --include`: To extend the include path. Each path must be separated with a `:`.
- `--phi`: Pretty prints the resulting Phi code from compiling the input file. Phileas proofs are based on the Phi code so inspecting it can certainly help the user understanding the proof.
- `--max-induction-depth`: Expects a natural argument. It is a flag that is used to relax the history restriction described in 6.2.3. The default behaviour corresponds to `max-induction-depth=2`.

Relevant to the developer:

- `--ghc-core`: Pretty prints the resulting GHC-Core from compiling the input file.
- `--trace`: During the search it prints the state of the prover at every step. The state includes the current goal and induction candidates.
- `--show-id`: When printing Phi code, it shows the id of each variable next to it. If variable `f` has id equal to 4, it prints `f!4`. it also shows the case ids.
- `--show-type`: When printing Phi code, it shows the type of each variable next to it. If variable `n` has type `Nat` it prints `n :: Nat`.
- `--show-cons-type`: Same as `--show-type` but for constructors.
- `--show-case-type`: Same as `--show-type` but for case expressions.
- `--show-history`: When printing Phi code, it shows the history of each variable next to it. For example `v{(I2, 1)}` denotes that var `v` has been generated due to induction on case expression with $id = 2$.

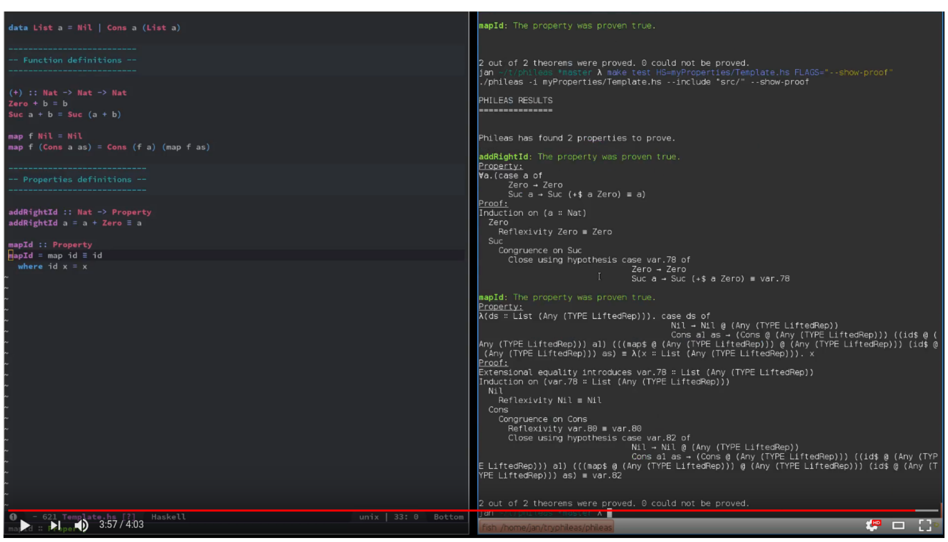

Figure 7.1: A video-guide through the first steps in Phileas. https://www.youtube.com/watch?v=hzFzKS_6qpQ

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
import PhileasPrelude

data Nat = Zero
         | Suc Nat

class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool

instance Eq Nat where
  Zero == Zero = True
  Suc a == Suc b = a == b
  _ == _ = False

instance Ord Nat where
  Zero <= _ = True
  Suc a <= Suc b = a <= b
  _ <= _ = False

partialOrderAntisimmetry :: Nat -> Nat -> Property
partialOrderAntisimmetry a b =
    a <= b ≡ True ⇒ b <= a ≡ True ⇒ a == b ≡ True
```

Listing 1: Contents of Properties.hs. An example Haskell source file with a property defined.

Chapter 8

Pipeline

Phileas pipeline consists of the following steps:

1. Use the GHC as a library to parse Haskell source code and compile it to an intermediate typed language named GHC-Core.
2. Compile GHC-Core into Phi.
3. Search and try to prove all properties in the module.
4. Report the results.

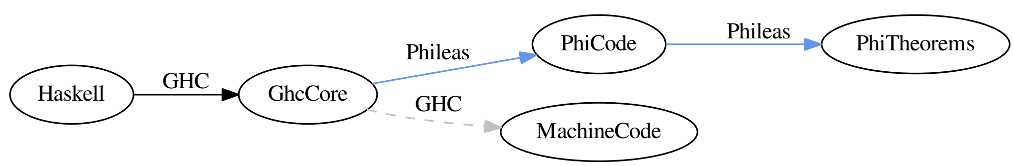

Figure 8.1: Pipeline graph.

Chapter 9

Proof output format

The proofs presented to the user must give enough information to allow them to follow the proof while not being too verbose. Finding a balance is certainly a challenge. We have tried to find a reasonable compromise by tagging each proof step with relevant information and taking advantage of indentation to express the tree structure of a proof. Below we describe the information attached to each proof step and its output schema.

- **Reflexivity** is tagged with the equality used.

  Schema:

  ```
  Reflexivity a ≡ a
  ```

- **Congruence** is tagged with the relevant constructor.

  Schema:

  ```
  Congruence on K
    <Subproof 1>
    <Subproof 2>
    ...
  ```

- **Induction** is tagged with the expression where it applies induction and its type. Each subproof is tagged with the corresponding constructor.

  Schema:

  ```
  Induction on e :: Type
    K1
     <Subproof 1>
    K2
     <Subproof 2>
    ...
  ```

- **Use hypothesis** is tagged with the hypothesis that was used to close the proof.

  Schema:

  ```
  Use hypothesis a ≡ b
  ```

- **Extensional equality** is tagged with the variable that introduces.

  Schema:

```
Extensional equality introduces newVar :: Type
<Subproof>
```

Note that the subproof does not need to be indented since it always have an only child.

- **Factor application** is tagged with the relevant expression.

  Schema:

  ```
  Factor application on f
    <Subproof 1>
    <Subproof 2>
    ...
  ```

- **Absurd in hypotheses** is tagged with the equality in the hypotheses that caused the absurd.

  Schema:

  ```
  Contradiction in hypothesis: K1 args1 ≡ K2 args2
  ```

- **Absurd in goal**. This is never part of a successful proof.

Chapter 10

Presentation

Phileas does extensive use of ANSI compatible terminals to enrich the format of its output. We believe this results in a far more enjoyable user experience. In this section we show some screen captures showcasing Phileas output.

```
semigroupEitherAssociative: The property was proven true.
Property:
∀a1 b1 c.(case a1 of
            Left ds → case b1 of
                        Left ds1 → c
                        Right ipv → b1
            Right ipv → case a1 of
                          Left ds → c
                          Right ipv1 → a1 ≡ case a1 of
                                              Left ds → case b1 of
                                                          Left ds1 → c
                                                          Right ipv → b1
                                              Right ipv → a1)
Proof:
Induction on (a1 ∷ Either a b)
  Right
    Reflexivity Right var.3279 ≡ Right var.3279
  Left
    Induction on (b1 ∷ Either a b)
      Right
        Reflexivity Right var.3283 ≡ Right var.3283
      Left
        Reflexivity c ≡ c
```

Figure 10.1: Output of a successful proof.

```
semigroupEndofunctionAssociative: The property was proven true.

semigroupListAssociative: The property was proven true.

semigroupNatPlusAssociative: The property was proven true.

appStateCompose: The prover was unable to reach a conclusion.

arrowFunctionProp3: The prover was unable to reach a conclusion.

appListCompose: The prover run out of time.
```

Figure 10.2: Presentation of the 3 different kinds of results.

```
insert$ ∷ Nat ↦ ([Nat] ↦ [Nat])
insert$ = λ(a ∷ Nat). λ(ds ∷ [Nat]). case ds of
                                        [] → (: @ Nat) a ([] @ Nat)
                                        : x xs → case <=$ a x of
                                                   False → (: @ Nat) x (insert? a xs)
                                                   True → (: @ Nat) a ((: @ Nat) x xs)
```

Figure 10.3: Coloured Phi code as outputted by the Phileas pretty printer.

```
$fFunctorEither [InlPrag=INLINE (sat-args=0)]
  :: forall l. Functor (Either l)
[GblId[DFunId(nt)], Arity=2, Caf=NoCafRefs]
$fFunctorEither
  = $cfmap4
    `cast` (forall (l :: <*>_N). Sym (N:Functor[0] <Either l>_N)
            :: Coercible (forall l a b. (a -> b) -> Either l a -> Either l b)
                         (forall l. Functor (Either l)))
```

Figure 10.4: Coloured GHC-core code as outputted by Phileas using the GHC pretty printer.

Part V

Results

Chapter 11

Overview

The results section consist in the evaluation of Phileas in three different test suites and the comparison to the theorem provers Zeno and IsaPlanner.

- **Zeno's** test suite (41 theorems): It is composed of the examples found in Zeno's package. This test suite compares Zeno and Phileas. We ran all tests on both provers.
- **IsaPlanner's** test suite (86 theorems): It is composed of the examples found in IsaPlanner's website. We ran all tests in Zeno and Phileas. We did not run IsaPlanner, instead we rely on the results reported by its authors.
- **Phileas** test suite (72 theorems): Our own test suite based on type class properties. We tried to run Zeno on this test suite and we encountered some problems.

Prover	Zeno suite (41)		Isa suite (86)		Phileas suite (72)	
Isa Induction	-		37	43%	-	
Isa Rippling	-		47	54.6%	-	
Zeno	39	95.1%	75	87.2%	4	5.6%
Phileas	29	70.7%	68	79.1%	68	94.4%

Table 11.1: Overview of the results displaying the number of proved theorems and the success rate.

11.1 Comparison to Zeno

Zeno [5] and Phileas are both automatic theorem provers that target to inductively proof equalities between Haskell expressions and use the GHC compiler internally. They are both based on a backtracking algorithm that tries to simplify the proof using different proof steps including induction. Therefore, it is reasonable to compare them in detail and pinpoint their main advantages and shortcomings.

Below we list a number of noteworthy facts about Zeno.
Positive aspects:

1. Is able to generate proofs for Isabelle [7], a robust proof assistant developed at the Cambridge University. If the proofs are verified by Isabelle, there is a strong guarantee that they are correct.

2. Has a built-in counterexample search engine, making it capable of disproving some theorems.

Negative aspects:

1. Its internal language has a flawed type system and crashes in many theorems involving type polymorphism.

2. It can give false negatives. We have found true theorems were Zeno returns an incorrect counterexample.

3. Zeno was released in 2011. It is currently unmaintained and no longer compiles with modern versions of GHC, which makes its installation impractical. In order to test it we had to get GHC-7.0.1, released in November 2010. A full explanation is given in section 11.1.1.

11.1.1 How we got Zeno

Since the results reported in Zeno's paper [5] do not match what we were able to reproduce, we believe that it is relevant to explain how we installed Zeno.

We download Zeno's source code from Hackage[1, 2]. The latest version is 0.2.0.1 and it was uploaded on 28 April 2011. The authors claim that it supports GHC 6 and 7 but do not specify the minor version of the compiler. We believed it was a good guess to use the first stable version of GHC 7, which is GHC 7.0.1, released on 16 November 2010. Another issue is that the Cabal file[3] does not specify version boundaries for the dependencies, which caused GHC 7.0.1 to crash when trying to compile a modern version of the `text` library. For this reason we needed to manually specify an older version. The last problem was that compiler flags were wrong. We followed the solution suggested by Zeno's author in an online forum[4]. Figure 11.1 links a video of the installation process.

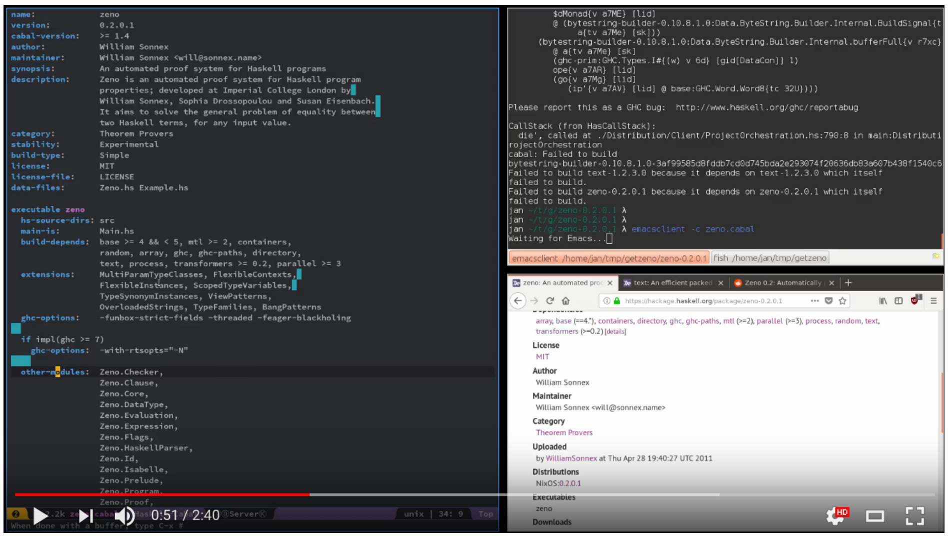

Figure 11.1: Video showing the installation process of Zeno in our computer. https://www.youtube.com/watch?v=AZwFjV27RGY

11.2 Comparison to IsaPlanner

IsaPlanner [6] is a plugin, developed at the University of Edinburgh, for Isabelle [7]. It offers a framework for automatic inductive theorem proving. It is based on rippling [19], a heuristic used to guide rewrite rules application. It also includes a lemma discovery heuristic [20].

What sets IsaPlanner apart from Zeno and Phileas is the fact that it supports interactive proving, thus, in some occasions, the user can guide the proof. However, in this project we only focus on the automatic part.

The project seems to be unmaintained since some links in its homepage[5] no longer work and last activity found in the official repository [6] is from February 2016. Also, it does not support the latest Isabelle version (Isabelle 2017). For these reasons, we have not tried to manually test it ourselves, instead, we rely on the results reported by the authors[7].

[1] Link to Zeno's package page: https://hackage.haskell.org/package/zeno

[2] Hackage is Haskell community's central repository for open source software.

[3] The Cabal file contains information needed to build the package such as compiler flags, dependencies and so on.

[4] https://www.reddit.com/r/haskell/comments/gzm0u/zeno_02_automatically_prove_haskell_program/c1rqjbf/

[5] http://dream.inf.ed.ac.uk/projects/isaplanner/

[6] https://github.com/TheoryMine/IsaPlanner

[7] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/

Chapter 12

Zeno's test suite

This test suite is composed of 41 properties about natural numbers (12), lists of natural numbers (27) and binary trees (2).

The results are the following:

Prover	Naturals		Lists		Trees	
Zeno	12	100%	27	100%	0	0%
Phileas	7	58.3%	20	74.1%	2	100%

Table 12.1: Results of the Zeno test suite.

Zeno clearly has a leg up on Phileas in this test suite. It successfully proves all theorems on natural numbers and lists. However, it times out (on a 20 seconds limit) on both tree properties while Phileas can proof them in under 0.2 seconds.

Prover	Gentle fails	Timeouts (20s)	Crashes	False negative
Zeno	0	2	0	0
Phileas	7	5	0	n/a

Table 12.2: Summary of unsuccessful tests.

Chapter 13

IsaPlanner test suite

This test suite is composed of 86 properties about natural numbers[1] (14), lists of natural numbers (71) and binary trees (1).

The following list indicates how the results were obtained.

- **Zeno***. From its authors [5].
- **Zeno**. Tested by us. Detailed information is given in section 11.1.1. In the comments we refer to these results.
- **IsaPlanner with simple induction**. From its authors[2].
- **IsaPlanner guided with rippling**. From its authors.[3].
- **Phileas**. Tested by us.

Prover	Naturals		Lists		Trees	
Zeno*	14	100%	84	96.6%	1	100%
Zeno	13	92.9%	61	85.9%	1	100%
Isa Induction	7	50%	30	42.3%	0	0%
Isa Rippling	8	57.1%	40	56.3%	1	100%
Phileas	8	57.1%	58	81.2%	1	100%

Table 13.1: Results of the Zeno test suite.

Prover	Gentle fails	Timeouts (20s)	Crashes	False negatives
Zeno	9	1	1	1
Phileas	12	7	0	n/a

Table 13.2: Summary of unsuccessful tests.

[1] One theorem has been disregarded due to being false.

[2] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/case-analysis-simp.txt

[3] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/case-analysis-extrippling.txt

During the testing process we observed one crash in Zeno due to a non-total pattern. Also, Zeno incorrectly provided a counterexample of a true theorem, namely

`member x (delete x l) ≡ False`

Where `delete` removes is a function that instances of the value in a list and `member` checks if an element is present in the list.

Phileas, although not crashing in any of the tests, experimented more timeouts in the theorems that it could not prove.

Chapter 14

Phileas test suite: type classes

We were concerned about Zeno's and IsaPlanner's test suites suffering from tunnel vision on natural numbers and lists. In fact, it is a common quirk among developers to design test suites that work well with their product but are not representative enough to weight its true value. In order to avoid this problem we set our eyes on the Haskell type system, which, in our opinion, is abstract enough to guarantee enough diversity.

Haskell's type system expressiveness allows for a number of type abstractions that benefit the composability, readability and correctness of the code. One sort of these abstractions comes in the form of *type classes*. A type class specifies a list of methods that a type in that class must implement. Usually a type class is tied to a set of properties that their methods should satisfy. For obvious reasons the compiler cannot check the validity of the implementation according to these properties. Here is were automatic theorem provers can have a key role.

We have selected some of the most ubiquitous type classes in the base package[1] and proved its required properties for different type instances.

Figure 14.1 shows the hierarchy of the analyzed classes in the Haskell type system. Figures 14.2 and 14.3 show the methods and properties that define each of the type classes.

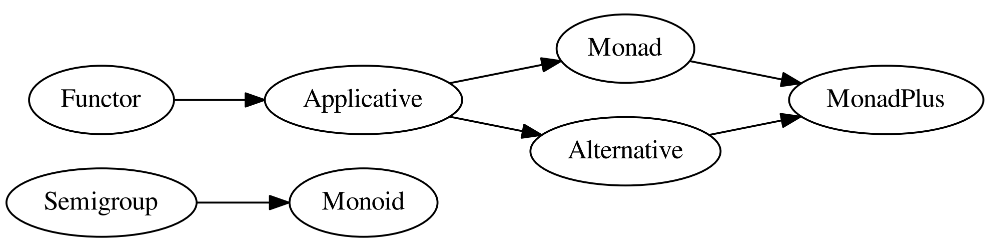

Figure 14.1: Class hierarchy. An arrow from A to B denotes that every instance of B must be also an instance of A.

[1]The base package is Haskell's standard library.

Functor

$fmap :: Functor\ f \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

Properties:

$fmap\ id \equiv id$ Identity

$\forall f\ g.\ fmap\ f \circ fmap\ g \equiv fmap\ (f \circ g)$ Composition

Applicative

$pure :: Applicative\ f \Rightarrow a \rightarrow f\ a$

$(<*>) :: Applicative\ f \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

Properties:

$\forall a.\ pure\ id <*> a \equiv a$ Identity

$\forall a\ b\ c.\ ((pure\ (\circ)\ <*> a) <*> b) <*> c = a <*> (b <*> c)$ Composition

$\forall f\ a.\ pure\ f <*> pure\ a \equiv pure\ (f\ a)$ Homomorphism

$\forall a\ b.\ a <*> pure\ b = pure\ (\lambda f.\ f\ b) <*> a$ Interchange

Monad

$return :: Monad\ m \Rightarrow a \rightarrow m\ a$

$(>>=) :: Monad\ m \Rightarrow ma \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Properties:

$\forall a\ f.\ return\ a >>= f \equiv f\ a$ Left identity

$\forall a.\ a >>= return \equiv a$ Right identity

$\forall m\ k\ h.\ (m >>= k) >>= h \equiv m >>= (\lambda x.\ k\ x{>>=}h)$ Associativity

Alternative

$empty :: Alternative\ f \Rightarrow f\ a$

$(<|>) :: Alternative f \Rightarrow f\ a \rightarrow f\ a \rightarrow f\ a$

Properties:

$\forall a\ b\ c.\ (a <|> b) <|> c \equiv a <|> (b <|> c)$ Associativity

$\forall a.\ empty <|> a \equiv a$ Left id

$\forall a.\ a <|> empty \equiv a$ Right id

Figure 14.2: Type class properties.

Monad Plus

$mzero :: MonadPlus\ m \Rightarrow m\ a$

Properties:

$\forall f.\ mzero >>= f \equiv mzero$ Left absorb

$\forall a.\ >>= f \equiv mzero$ Right absorb

Semigroup

$(<>) :: Semigroup\ s \Rightarrow s\ a \rightarrow s\ a \rightarrow s\ a$

Properties:

$\forall a\ b\ c.\ (a <> b) <> c \equiv a <> (b <> c)$ Associativity

Monoid

$mempty :: Monoid\ m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

Properties:

$\forall a.\ mempty <> a \equiv a$ Left id

$\forall a.\ a <> mempty \equiv a$ Right id

Figure 14.3: Type class properties. Part 2.

	Functor	Applicative	Monad	Alternative	MonadPlus	Semigroup	Monoid
List		1	1				
Maybe							
Either							
Binary Tree							
Reader							
State		1					
Pair							
Endofunction							
Naturals (+)							
Naturals (∗)						1	

Table 14.1: Green means all theorems were proved. A number over yellow means how many theorems from that class could not be proved. Blank implies that the type is not an instance of the class.

All tests run in under $0.2s$ except the composition property for the applicative instance of lists, which takes about 3.7s.

We tried to prove all theorems presented in this section using Zeno, unfortunately, it crashed due to a type unification error on all tests except for natural numbers. We believe there must be a bug in the prover related to type polymorphism.

Additionally, gave false counterexamples for semigroup associativity over addition and multiplication. We have found that the latter problem only occurs when the `Nat` type is inside a wrapper type as displayed in figure the below. On account of this we believe there is a bug in Zeno's counterexample finder related to types of this sort.

```
data NatPlus = NatPlus Nat
data NatMul = NatMul Nat

instance Semigroup NatPlus
  (NatPlus a) <> (NatPlus b) = NatPlus (a + b)
instance Semigroup NatMul
  (NatMul a) <> (NatMul b) = NatMul (a * b)
```

Listing 2: Wrapper types for the natural numbers. Wrapping a type is a technique that is frequently used to declare various class instances of essentially the same type. In this concrete case, since the natural numbers form a semigroup with addition and multiplication, we need a wrapper type for each instance.

14.1 Reproducibility of the results

We feel that in modern research it is essential to give detailed instructions on how to reproduce any claimed result. For this reason, we provide a video that displays the necessary commands to download and build Phileas and then run the tests described in this section. A link to the video is found in figure 14.4. It is actually an ASCII cast, so every command is selectable and can be copy-pasted by the user.

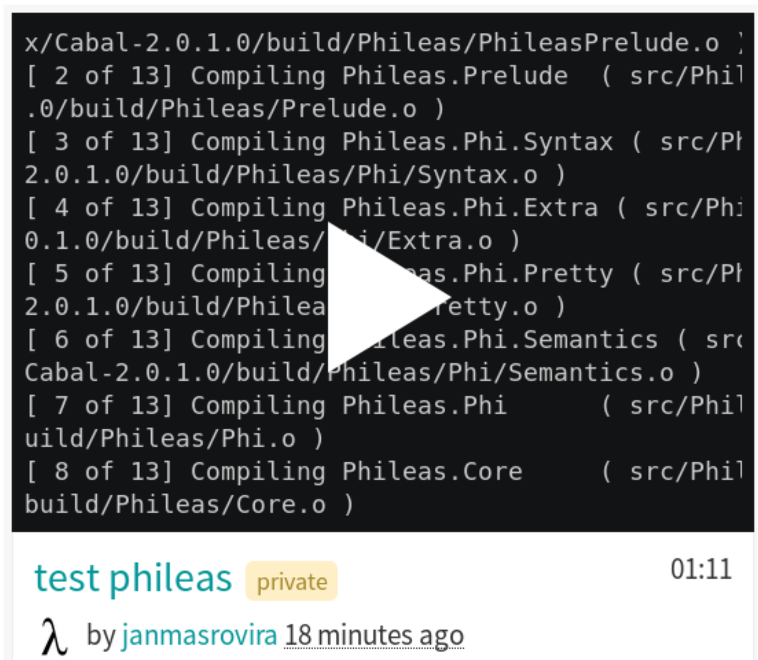

Figure 14.4: Phileas, from cloning to testing in 1 minute:
https://asciinema.org/a/9xONHjcl6nx8OBMPnIS1R7Du7

Part VI

Conclusions

Chapter 15

Perspective and accomplishments

Formally verifying programs is becoming more and more relevant in the modern world of computing. Formally verifying some property may save a lot of money in testing, moreover, preventing a bug in a critical system may save millions. On June 1996, the Ariane 5 rocket exploded just a few seconds after launch due to an overflow error that caused estimated loses of $370 million. This error could have been prevented by formal verification. Nowadays, numerous resources are being put in verifying critical software in financial systems, communication protocols, aerospace engineering and other areas.

Formal program verification is just one, albeit of critical importance, of the many use cases of theorem proving. In other areas such as logic, type theory or category theory among others, theorems are being formalized in proof assistants and the computer checked proof base is growing by the day. Perhaps, in a not so distant future, pen and paper proofs will become obsolete, or at the very least, lose some value.

Theorem proving and functional programming are two concepts that fit very well together due to the mathematical approach to programming of functional languages. Because of this, we believe that improving the ecosystem of theorem proving and functional programming, will be a well invested effort. We believe that with this project we have done our bit.

Phileas contributed to the ecosystem by:

- Being (to our knowledge) the first automatic inductive prover that uses Haskell as its starting point and that supports full type polymorphism.
- Being offered as a binary and as a library. The latter will allow other people to import Phileas to their projects and use it as part of it without the need to make an external call.
- Being released under the MIT license, a simple and very permissive open source license.
- Automatically proving a part of the type class implementation in Haskell's base package.
- Contributing to an ongoing research in formal verification of concurrent systems [10].

Chapter 16

Future work

We list a number of extensions that could be implemented in Phileas as future work.

- **Verifiable proofs**. Although our algorithm has been thoroughly tested, it has been implemented by a human, and humans make mistakes. Hence, there is a possibility that it generates an invalid proof for a false theorem. Two solutions exist to this problem:

  1. Verifying the prover: Formally verifying the solver would probably be the work of a lifetime (possibly much more) as just verifying GHC or an equivalent compiler would be a monumental task. Furthermore, we would still rely on the underlying proof assistant.
  2. Generating verifiable proofs: A more reasonable approach is to generate verifiable proofs by a proof assistant such as Isabelle, Agda or Coq. Assuming the prover assistant was infallible, then we could verify all generated proofs and discard those which are invalid.

- **Strong induction**. Phileas only supports weak induction. That is, when applying induction it only generates induction hypotheses for the immediate children. Adding support for strong induction, that is, having induction hypotheses that could be applied to any term structurally smaller than the target term, would allow Phileas to prove properties about mutually recursive types such as graphs or trees.

- **Lemma discovery heuristic**. It is often the case that properties need auxiliary lemmas that are more general than one of its subproofs. It is a challenge and a research field in itself to automatically find these lemmas. The literature on this topic should be more carefully examined and Phileas could be extended by modern techniques. We believe a state of the art heuristic would drastically improve its efficacy.

- **Support inequality proofs**. A possible extension would be to support proofs of inequality. We believe adding this feature to the prover should be reasonably easy due to the fact that some steps were already taken in this direction in the subproof aggregation methods.

- **Adding support for integers**. Adding support for primitive integers would be a big step up in terms of the strength of the prover although it is unclear the that path we should take. One possibility would be to add a *magical* axiom that relies on the correctness of some SMT solver and allow the prover to make calls to that SMT solver during the proving process. This approach would make generating verifiable proofs much more difficult as well.

- **Generation of sufficient conditions**. Sometimes an equality is only true under some conditions. As of now, if these conditions can be expressed in terms of equalities the user can provide them as antecedents in the property. An extension to Phileas would be to implement a heuristic capable of automatically finding simple sufficient conditions for an equality to hold. This would be specially valuable in the context of CDPOR.

Bibliography

[1] S. Biundo, B. Hummel, D. Hutter, and C. Walther, “The karlsruhe induction theorem proving system,” in *8th International Conference on Automated Deduction*, J. H. Siekmann, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 672–674, ISBN: 978-3-540-39861-5.

[2] A. Bouhoula, E. Kounalis, and M. Rusinowitch, “Spike, an automatic theorem prover,” in *Logic Programming and Automated Reasoning*, A. Voronkov, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 460–462, ISBN: 978-3-540-47279-7.

[3] R. M. Burstall, “Proving properties of programs by structural induction,” *The Computer Journal*, vol. 12, no. 1, pp. 41–48, 1969. DOI: 10.1093/comjnl/12.1.41. eprint: /oup/backfile/content_public/journal/comjnl/12/1/10.1093/comjnl/12.1.41/2/12-1-41.pdf. [Online]. Available: http://dx.doi.org/10.1093/comjnl/12.1.41.

[4] D. R. Musser, “On proving inductive properties of abstract data types,” in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1980, pp. 154–162.

[5] W. Sonnex, S. Drossopoulou, and S. Eisenbach, “Zeno: An automated prover for properties of recursive data structures,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Flanagan and B. König, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 407–421, ISBN: 978-3-642-28756-5.

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

[7] L. C. Paulson, “Isabelle: The next 700 theorem provers,” *CoRR*, vol. cs.LO/9301106, 1993. [Online]. Available: http://arxiv.org/abs/cs.LO/9301106.

[8] A. Bundy, A. Stevens, F. Van Harmelen, A. Ireland, and A. Smaill, “Rippling: A heuristic for guiding inductive proofs,” *Artificial intelligence*, vol. 62, no. 2, pp. 185–253, 1993.

[9] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, 2005, pp. 110–121.

[10] E. Albert, M. Gómez-Zamalloa, M. Isabel, and A. Rubio, “Constrained dynamic partial order reduction,” in *23th International Symposium on Formal Methods (FM)*, ser. Lecture Notes in Computer Science, Springer, to appear 2018.

[11] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, “Abs: A core language for abstract behavioral specification,” in *International Symposium on Formal Methods for Components and Objects*, Springer, 2010, pp. 142–164.

[12] E. Albert, M. Gómez-Zamalloa, A. Rubio, M. Sammartino, and A. Silva, "Sdn-actors: Modeling and verification of sdn programs," in *30th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, Springer, to appear 2018.

[13] A. Tolmach and T. Chevalier, *An external representation for the ghc core language (for ghc 6.10)*.

[14] H. Barendregt, W. Dekkers, and R. Statman, *Lambda calculus with types*. Cambridge University Press, 2013.

[15] A. R. Meyer, J. C. Mitchell, E. Moggi, and R. Statman, "Empty types in polymorphic lambda calculus," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87, Munich, West Germany: ACM, 1987, pp. 253–262, ISBN: 0-89791-215-2. DOI: 10.1145/41625.41648. [Online]. Available: http://doi.acm.org/10.1145/41625.41648.

[16] L. C. Paulson, "Mechanizing coinduction and corecursion in higher-order logic," *Journal of Logic and Computation*, vol. 7, no. 2, pp. 175–204, 1997.

[17] J. B. Wells, "Typability and type checking in system f are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, no. 1-3, pp. 111–156, 1999.

[18] J. Caldwell, "Structural induction principles for functional programmers," *arXiv preprint arXiv:1312.2696*, 2013.

[19] M. Johansson, L. Dixon, and A. Bundy, "Case-analysis for rippling and inductive proof," in *International Conference on Interactive Theorem Proving*, Springer, 2010, pp. 291–306.

[20] ——, "Dynamic rippling, middle-out reasoning and lemma discovery," in *Verification, Induction, Termination Analysis*, Springer, 2010, pp. 102–116.