CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Finding Bugs in Functional Programs using Theorem Provers

Master's thesis in Computer science and engineering

Thaleia Tsioka

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Finding Bugs in Functional Programs using Theorem Provers

Thaleia Tsioka

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Finding Bugs in Functional Programs using Theorem Provers
Thaleia Tsioka

Supervisor: Koen Claessen, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Finding Bugs in Functional Programs using Theorem Provers
Thaleia Tsioka
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis investigates whether automated first-order theorem provers can be efficiently used to discover bugs in Haskell programs by extracting counterexamples to incorrect program properties. To enable counterexample extraction, we present an alternative translation from Haskell programs to first-order logic that reformulates programs to purely equational first-order theories and properties to existential conjectures. The translation to the proposed encoding builds on the pre-existing Haskell $\rightarrow$ TIP $\rightarrow$ TPTP translation pipeline by introducing a sequence of transformations that preserve the semantics of the original program with the goal of making the resulting theories better suited for equational theorem provers.

The approach is evaluated using the theorem provers Twee, E and Vampire on a collection of Haskell benchmarks. The results show that provers are capable of successfully extracting counterexamples to incorrect Haskell properties, and that the proposed encoding does in fact improve their performance compared with the one produced by the pre-existing translation. Among the evaluated provers, Twee consistently produced the strongest results on the proposed encoding, while E achieved the best overall performance when combined with an appropriate lexicographic path ordering that approximates goal-directed proof search.

The evaluation identifies weak goal direction as the principal practical limitation of theorem-prover counterexample extraction and the resulting time inefficiency, as its main disadvantage in comparison to property-based testing. However, our work also demonstrates that theorem-prover counterexample extraction offers capabilities which property-based testing doesn't, including the successful extraction of counterexamples to higher-order properties, ultimately making it a promising direction for future research on automated bug finding in functional programs.

Acknowledgements

Here, you can say thank you to your supervisor(s), company advisors and other people that supported you during your project.

Thaleia Tsioka, Gothenburg, 2026-06-29

Contents

1

Introduction

Haskell, as a purely functional programming language, is particularly well suited to formal reasoning about program behaviour. Its core features such as side-effect absence, immutability and static typing makes programs written in Haskell behave as mathematical functions, and thus subject to mathematical reasoning. As a result, Haskell programs provide a natural target for formal verification using automated reasoning tools.

Previous research has focused primarily on program verification. Following the prominent method of program testing, which is property-based testing, the existing approaches have managed to translate programs and their properties into appropriate logical forms and then give them to Automated Theorem Provers with the goal of them proving that the properties hold universally for all program inputs. While this direction has achieved promising results [1], it only addresses the one side of the problem of program verification: the one where the input program is indeed correct for the specified property. However, it is just as important to be able to verify that a program is incorrect and does not always satisfy a given property, as in practice, more often than not, programs contain bugs.

This opposite direction has been explored primarily by property-based testing frameworks, such as QuickCheck [2] and SmallCheck/Lazy SmallCheck [3]. These tools attempt to disprove a program property by searching for a counterexample, i.e. an input that falsifies the property. QuickCheck performs this search by randomly generating test inputs, while SmallCheck and Lazy SmallCheck systematically explore bounded portions of the input space. If a counterexample is found, it is returned to the programmer. While these approaches are both lightweight and practical, they still remain testing techniques rather than formal reasoning methods. The main limitation is that the failure to discover a counterexample provides no information about behaviour outside the explored portion of the input space, and thus cannot be a sufficient correctness guarantee. Counterexamples that occur only for rare, highly specific, or sufficiently large inputs may therefore remain undiscovered.

So, on the one hand, Haskell programs are suited for formal reasoning and there is existing evidence suggesting that Automated Theorem provers are indeed able to formally prove them correct. On the other hand, it is sometimes more practical to be able to tell that a program is incorrect but the existing methods do not use for-

mal approaches. This makes us naturally wonder "Can Automated Theorem Provers also be used to Discover Haskell Program Bugs?". This thesis aims to answer this question.

2

Background

Automated Theorem Provers cannot directly reason about source programs written in high-level programming languages like Haskell. Consequently, in order to be able to utilize them in our thesis, the programs and their properties must first be translated into an appropriate logical representation. Since, as mentioned, provers have already been used to reason about Haskell programs, there is pre-existing infrastructure providing the foundation this thesis builds on, redirecting it towards our novel goal of counterexample extraction.

2.1 First-Order Logic in Automated Theorem Proving

First-order logic is a logical formalism commonly used in automated reasoning. In first-order logic, formulas are constructed using variables, function symbols, predicates, logical connectives and quantifiers. Unlike higher-order logics, quantification ranges only over individual domain elements and does not extend to functions or predicates.

First-Order Theorem Provers, such as the three used in the scope of this thesis Twee [4], E [5] and Vampire [6], are the provers which operate on first-order logic encodings. Given a collection of logical formulas representing assumptions together with a conjecture, the prover attempts to determine whether the conjecture logically follows from the assumptions. Modern theorem provers typically perform proof search using saturation-based techniques, where new logical consequences are repeatedly derived from existing formulas until either a contradiction is found or no further inferences can be produced, at which point the logical theory is considered saturated. Depending on the prover, saturation may be achieved through different procedures such as resolution and superposition (Vampire/E), or rewriting and equational reasoning (Twee). Although these procedures are complete for first-order logic, meaning that any valid statement is, in principle, guaranteed to eventually be derived (so, assuming there exists a counterexample, then it will eventually be found), the search space can become extremely large in practice. Consequently, prover performance often depends heavily on search heuristics and the structure of the logical encoding.

2.2 Inductionless Induction

One could reasonably wonder how first-order reasoning can be considered a promising approach for deriving counterexamples to properties of functional programs, when reasoning about recursive structures typically requires induction and induction can not be represented in first-order logic.

This apparent contradiction has been long recognized in automated reasoning and thus there have been several attempts to solve it. The one most relevant to our work is the idea of inductionless induction [7]. Normally, the verification of a property $Prop$ for a logical theory T describing a program, is expressed as

$$T \vdash Prop,$$

where the proof system $\vdash$ must, in the general case, be capable of inductive reasoning. Since induction is notoriously hard to automate, inductionless induction proposes a different perspective. Rather than doing the proof directly, we can first transform the original theory T and property $Prop$ into corresponding first-order encodings T^* and $Prop^*$, thus reformulating the problem into finding a model M such that

$$M \models T^* \wedge Prop^*.$$

Here $\models$ denotes first-order satisfiability, thus this transformation eliminates the need for a proof system capable of performing induction, allowing reasoning to proceed through ordinary first-order inference instead.

Consider the recursive Haskell definition of the function `reverse` that reverses a list:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

and its (correct) property `prop`:

```
prop = reverse (reverse xs) == xs
```

Typically, the verification of this property would be expressed as

$$T \vdash \forall xs.\texttt{reverse}(\texttt{reverse}(xs)) = xs$$

where T contains the recursive definitions of reverse and append.

This property cannot be verified by unfolding the recursive definitions a finite number of times, since xs may be a list of any size. Thus, the proof system $\vdash$ must be able to perform structural induction on xs, proving the property first for the empty list [] and then showing that if it holds for a list xs it will also hold for a list $x : xs$.

Inductionless induction proposes that instead of reasoning directly about the property using induction, the program and the property should first be transformed into

corresponding first-order encodings. For this example, the transformed theory T^* could be:

```
reverse(nil) = nil
reverse(cons(X, Xs)) = append(reverse(Xs), cons(X, nil))
append(nil, Y) = Y
append(cons(X, Xs), Y) = cons(X, append(Xs, Y))
```

and the transformed property $Prop^*$

$$\forall X.\text{reverse}(\text{reverse}(X)) = X$$

Thus the original verification problem

$$T \vdash \forall xs.\texttt{reverse}(\texttt{reverse}(xs)) = xs$$

is replaced by the first-order satisfiability problem

$$M \models T^* \wedge Prop^*$$

which no longer requires induction and where ordinary first-order inference is used to determine whether a model satisfying the transformed theory and property exists instead.

In spite of how promising it might look, inductionless induction is shown to be unsuccessful in practice, as finding a model that satisfies a first order theory is, in the general case, at least as hard as the original problem i.e. using a proof system that automates induction.

However, observe that the objective of this thesis is the exact opposite of what inductionless induction was originally proposed for. Rather than attempting to establish that a property is correct, we attempt to establish that it is incorrect by finding a counterexample. But a takeaway from inductionless induction that we can apply in our case is that reasoning about recursive functional programs can be transferred to reasoning about suitably constructed equivalent first-order encodings. That makes the real challenge not be the design of new proof procedures (that can automate induction) but the construction of first-order encodings T^* and $Prop^*$ that faithfully capture the semantics of the original program and property. Once such encodings have been obtained, then existing theorem provers are strong enough to produce a counterexample even though they cannot perform inductive reasoning.

This conclusion also lets us clarify the claim of completeness made in the previous section. Suppose that the original Haskell property admits a counterexample. Then there exist concrete Haskell values $a, b, c, ...$, such that

$$Prop(a, b, c, ...) = false.$$

Assume now that the generated first-order encoding faithfully captures the semantics of the original Haskell program and property. Let x^* denote the first-order term corresponding to a Haskell value x. Then the existence of the above counterexample

implies that the corresponding first-order statement is derivable from the encoded theory, i.e.

$$T^* \vdash Prop(a^*, b^*, c^*) = false.$$

or, by existential introduction,

$$T^* \vdash \exists X, Y, Z, ...Prop(X, Y, Z, ...) = false.$$

Thus, whenever a genuine counterexample exists, there exists a corresponding first-order witness represented within the encoding.

As discussed in the previous section, theorem provers are also complete, meaning that every consequence of the generated theory is eventually derivable. Combining these two notions of completeness, we can see that under the assumption that our method produces a faithful encoding, the entire counterexample-search process becomes complete: whenever a counterexample exists in the original Haskell program, then a corresponding first order witness also exists in the theory, and is guaranteed to eventually be discovered by the theorem prover.

However this guarantee has a primarily theoretical value. Saturation-based theorem provers may need to explore extremely large search spaces before deriving the relevant witness, and thus the time required to discover a counterexample may be large enough for the search to be considered a practical failure.

2.3 TPTP

Although first-order logic provides the formal basis for automated theorem proving, provers cannot operate directly on abstract logical formulas, thus the logical problems must be expressed in some concrete syntax that the prover can parse and process. One of the most commonly used standardisations for this purpose is TPTP (Thousands of Problems for Theorem Provers), a language and benchmark infrastructure designed to facilitate interoperability between automated reasoning systems.

In this thesis the inputs given to the three theorem provers are untyped first-order TPTP translations of the respective Haskell programs and their associated properties. Although, as explained, TPTP acts as a common interface between theorem provers, different provers may still impose additional restrictions on the structure of the logical encoding they accept. An example in our case is Twee, which operates specifically on equational Horn clauses and will not accept a logical encoding containing a non-Horn clause. A Horn clause is a disjunctive clause with at most one positive, i.e. unnegated, literal. The other two provers used, E and Vampire, impose no such restrictions.

While the TPTP language also offers typed variants, we chose to use untyped first-order TPTP. This choice was made because the counterexample-search process does not require explicit type information at this level. We will see in the following sec-

tion that type information is still present on the TIP level and the TIP → TPTP translation performs a transformation which ensures that objects originating from different Haskell-level data types cannot interact with one another (monomorphisation). For objects originating from the same data type, the only information that the TPTP encoding needs preserved is their equality/ disequality relationships. We leave the details of how this information is represented in our encoding for later sections, but note that we made sure it is preserved.

2.4 TIP

One can see that directly translating Haskell to TPTP, especially when more than one different formats of TPTP is required, is a challenge that solving on our own could take away from the time spent on the actual contribution of this thesis. Here is where the most central piece of pre-existing infrastructure becomes relevant. TIP (Tools for Inductive Provers) takes a dual role, providing both a translation from Haskell to a consistent intermediate logical representation and a collection of tools for transforming this representation into different formats of TPTP based on a variety of options. Thus we chose to have the transformation of a Haskell program to a prover's input file not be done in a single step, but rather by the transformation pipeline

$$\text{Haskell} \rightarrow \text{TIP} \rightarrow \text{TPTP} \rightarrow \text{ATP}.$$

The TIP language is based on SMT-LIB, extending it with common functional programming features including recursive function definitions, algebraic data types, pattern matching, polymorphism and higher order functions. Thus the TIP translation of a Haskell program is able to preserve its entire semantics.

The TIP toolchain offers tools for transformations such as monomorphisation, which replaces polymorphic definitions with monomorphic instances, defunctionalisation for eliminating higher order functions and pattern match elimination for simplifying recursive function definitions. Thus it allows us to end up with a TPTP file that matches the requirements and the capabilities of the target prover.

It is important to make a distinction: Some transformations, like defunctionalisation, are required for the resulting TPTP file's compatibility with the provers which, since they are first-order, do not support quantification over higher-order terms like functions. Some others, however, are needed to ensure the semantic correctness of the prover's output, the most important of them being monomorphisation. As explained we chose untyped TPTP as the format of the input files given to the provers. As a result, directly translating a polymorphic function into a single function symbol shared by every type instance may allow interactions between different type objects that would be impossible in the original typed program. Thus, although polymorphic function symbols are allowed in a first-order logical encoding, monomorphisation is needed to resolve this issue by replacing polymorphic definitions with separate monomorphic instances, ensuring that type correctness is preserved throughout the proof.

In order to get a better understanding of the TIP framewrork, and especially its role as an intermediate representation, consider the Haskell program shown below:

```
module CFG5 where

import Tip

data E = E :+: E | E :*: E | EX | EY
  deriving (Eq,Ord,Show)


data Tok = C | D | X | Y | Plus | Mul
  deriving (Eq,Ord,Show)

lin :: E -> [Tok]
lin (a :+: b) = linTerm a ++ [Plus] ++ linTerm b
lin (a :*: b) = lin a ++ [Mul] ++ lin b
lin EX        = [X]
lin EY        = [Y]

linTerm :: E -> [Tok]
linTerm (a :*: b) = [C] ++ lin (a :+: b) ++ [D]
linTerm e         = lin e

assoc :: E -> E
assoc ((a :+: b) :+: c) = assoc (a :+: (b :+: c))
assoc (a :+: b) = assoc a :+: assoc b
assoc (a :*: b) = assoc a :*: assoc b
assoc a         = a

prop_unambig u v = lin u === lin v ==> assoc u === assoc v
```

This program defines the expression type `E` which can be either a variable `EX`/ `EY` or an addition `:+:` / multiplication `:*:` between expressions. It also defines the data type `Tok` which is the alphabet of tokens `X`, `Y`, `Plus`, `Mul`, `C`, `D` used to "print" an expression. `lin` converts an expression into a list of tokens. When it prints an addition where one of the operands is a multiplicative term it adds the delimeters `C` and `D` around it, which act as parentheses. `assoc` normalises addition associativity by rewriting left-nested sums to their corresponding right-nested form recursively. The defined property `prop_unambig` claims that the tokenization `lin` is unambiguous up to associativity of `:+:`. In other words if two expressions print to exactly the same token list, then after normalizing addition associativity with assoc, they must be the same expression.

The property uses the special operators `===` and `==>`, provided by the TIP library. Unlike ordinary Haskell operators such as `==`, these represent equality and impli-

cation in the target logical theory rather than executable Haskell functions. They allow the TIP frontend to recognise the definition they are in as a logical property and translate it into the conjecture of the resulting theory.

When translated to TIP using the standard pre-existing translation, the following logical encoding is produced:

```
(declare-datatype
  list :source |[]|
  (par (a)
    ((nil :source |[]|)
        (cons :source |:| (head a) (tail (list a))))))
(declare-datatype
  Tok :source Tok
  ((C :source C) (D :source D) (X :source X) (Y :source Y)
   (Plus :source Plus) (Mul :source Mul)))
(declare-datatype
  E :source E
  ((|:+:| :source |:+:| (|proj1-:+:| E) (|proj2-:+:| E))
   (|:*:| :source |:*:| (|proj1-:*:| E) (|proj2-:*:| E))
   (EX :source EX) (EY :source EY)))
(define-fun-rec
  assoc :keep :source assoc
  ((x E)) E
  (match x
    (((|:+:| y c)
      (match y
        (((|:+:| a b) (assoc (|:+:| a (|:+:| b c))))
         (_ (|:+:| (assoc y) (assoc c))))))
     ((|:*:| a2 b2) (|:*:| (assoc a2) (assoc b2)))
     (_ x))))
(define-fun-rec
  append :source ++
  (par (a) (((x (list a)) (y (list a))) (list a)))
  (match x
    ((nil y)
     ((cons z xs) (cons z (append xs y))))))
(define-funs-rec
  ((linTerm :keep :source linTerm
    ((x E)) (list Tok))
   (lin :keep :source lin
    ((x E)) (list Tok)))
  ((match x
     (((|:*:| a b)
       (append (cons C (_ nil Tok))
         (append (lin (|:+:| a b)) (cons D (_ nil Tok))))))
```

```
      (_ (lin x))))
   (match x
     (((|:+:| a b)
       (append (linTerm a)
            (append (cons Plus (_ nil Tok)) (linTerm b))))
      ((|:*:| a3 b2)
       (append (lin a3)
            (append (cons Mul (_ nil Tok)) (lin b2))))
      (EX (cons X (_ nil Tok)))
      (EY (cons Y (_ nil Tok)))))))
(prove
  :source prop_unambig
  (forall ((u E) (v E))
    (=> (= (lin u) (lin v)) (= (assoc u) (assoc v)))))
```

First of all, we can see that the structure of the original Haskell program is well preserved in the TIP encoding. The algebraic data type definitions `E, Tok` become explicit datatype declarations and the recursive function definitions `lin, linTerm, assoc` become recursive logical definitions.

Second, observe that TIP also introduces the logical definitions for the corresponding built-in Haskell constructs (`list` datatype, `++` function) that are not explicitly defined in the source program itself.

Finally, the QuickCheck property is translated into a universally quantified logical formula and is marked by the special keyword `prove` which denotes the theory's conjecture. The conjecture's current format expressing "for all inputs the property holds" mirrors exactly the original purpose of the TIP framework which is the formal verification of program properties.

There is no point in including here the full corresponding TPTP encoding produced by the TIP $\rightarrow$ TPTP translation. Unlike the TIP language, which is still somewhat high-level, and closely resembles the structure of a functional programming language, TPTP is a purely first-order logical encoding. Consequently, many of the high-level programming abstractions which are still easily understandable on the TIP level are flattened into low-level constructs on the TPTP level, making it significantly less readable. This should be expected, since the TPTP format is made to cater to the needs of theorem provers and not the programmer's. This is another reason why TIP, as an intermediate step between Haskell and first-order logic, is useful and thus hopefully the choice of implementing our alternative method on the TIP level seems intuitive.

It might be useful, however, to see how the conjecture looks on the TPTP level, as this is the goal of the theorem prover:

```
fof(goal, conjecture,
```

```
  ![U, V]: (lin(U) = lin(V) => assoc(U) = assoc(V))).
```

The symbol `!` is how the universal quantifier ∀ is represented in TPTP. The special keyword `conjecture` is what indicates to the prover that this is the property it needs to try to establish. There is a correspondence between TIP's "prove" definition and TPTP's "conjecture" axiom.

3

The Haskell to TIP Translation

We decided not to interfere with TIP's pre-existing machinery directly, but rather to first apply the standard Haskell → TIP translation unchanged and, after the original TIP encoding has been produced, apply our sequence of additional transformations on it. The purpose of these transformations is to redirect the encoding from its original verification goal towards our new goal of counterexample extraction.

What one of these transformations needs to be, has hopefully been made apparent by our previous discussion: the conjecture must be changed. In the current (standard) TIP encoding, the Haskell program's property has become a universally quantified conjecture stating that "for all inputs the property holds". We need to reformulate it into some convenient representation of "there exists an input so that the property does not hold". Before presenting the exact way we chose to formalise this sentence, we first need to explain our translation's main design decision which determines the structure of this and every other transformation.

Remember that even though we tested our method on three different provers, the main target prover of this thesis was from the beginning Twee, which is an equational theorem prover. Twee's proof search is based primarily on rewriting and completion procedures operating on equations. Thus, we made the hypothesis that if Twee's input, i.e. the TPTP encoding produced by the translation pipeline, is purely equational, then the proof search will naturally align with the structure of the logical theory which hopefully allows Twee to derive witnesses faster and more efficiently. Consequently, every transformation our method does on the TIP encoding is to ensure that after it is finalized and translated to TPTP, the resulting TPTP encoding has the following property: it exclusively consists of axioms expressed as equations between the function symbols of the theory.

For this to be achieved, every construct of the TIP theory should be either a function definition or a data type declaration. Datatype declarations produce the constant symbols (which can be also thought as nullary function symbols) in the TPTP theory, and function definitions produce the non-nullary function symbols and the equational axioms describing the function's behaviour. Therefore, we must identify the logical constructs that TIP treats as built-in operators, an thus its logical framework interprets directly, and redefine them appropriately with explicit functions.

3.1 Equality

In the Background section we avoided including the full TPTP encoding produced by the standard Haskell → TIP → TPTP translation of our example, but we did show that the encoding's conjecture is

```
fof(goal, conjecture,
  ![U, V]: (lin(U) = lin(V) => assoc(U) = assoc(V))).
```

Let's observe another one of the encoding's generated axioms

```
fof(axiom, axiom,
  ![C, A, B]: assoc(z(z(A, B), C)) = assoc(z(A, z(B, C)))).
```

which corresponds to the following case of the source program's `assoc` definition

```
assoc ((a :+: b) :+: c) = assoc (a :+: (b :+: c))
```

We can observe that in both the conjecture and the axiom the equality symbol is used, however it serves two different purposes.

`lin(U) = lin(V)` (same for `assoc(U) = assoc(V)`) is a logical formula whose truth depends on the values assigned to the variables `U` and `V`, and can evaluate to true or false. In the first-order theory represented by the TPTP encoding it acts as an atomic proposition.

In contrast, `assoc(z(z(A, B), C)) = assoc(z(A, z(B, C)))` is a definitional equation which during the proof search acts as a rewrite rule, meaning that it informs the prover that whenever it encounters the term `assoc(z(z(A, B), C))` it may replace it with the term `assoc(z(A, z(B, C)))` and vice versa.

While logical equality, i.e. equality used in expressions like `lin(U) = lin(V)` is perfectly valid in a first-order encoding, its presence in the encoding makes it not satisfy the goal property we stated earlier. Consequently, we need a way to transform propositions such as `lin(U) = lin(V)` into ordinary terms of the theory. Once they are successfully represented as terms, equality tests can appear themselves as arguments of other function symbols, just like the function symbol `z` is an argument of `assoc` in `assoc(z(z(A, B), C))`.

To achieve this, we introduce into the TIP theory a polymorphic boolean function `eq`, which takes two objects of the same data type and returns true if they are equal and false otherwise, essentially encoding logical equality but as an ordinary function, just like `assoc`, `linTerm` and `lin`. In TIP syntax `eq`'s definition is

```
(declare-fun eq (par (a) ((a a) Bool)))
```

After introducing `eq` the TIP theory's conjecture becomes

```
(prove
  :source prop_unambig
  (forall ((u E) (v E))
    (=> ((_ eq (list Tok)) (lin u) (lin v))
    ((_ eq E) (assoc u) (assoc v)))))
```

where `(_ eq (list Tok))` and `(_ eq E)` is how TIP represents a polymorphic function's application to specific type instances.

After translating TIP to TPTP, `eq` is in no way distinct to the other function symbols of the theory, and is thus treated in the exact same way. The TPTP theory's conjecture is now

```
fof(goal, conjecture,
  ![U, V]: (eq(lin(U), lin(V)) => eq(assoc(U), assoc(V)))).
```

3.2 Logical Connectives

The next construct we need to add explicit definition for already becomes apparent by looking at the current state of the conjecture of the TIP and TPTP theories of our running example. Logical implication, i.e. `=>` is used in the TIP encoding without a definition, as we can see by inspecting the full TIP encoding we showed earlier. It is interpreted directly by the framework, and survives unchanged the TIP → TPTP translation, making the TPTP theory's conjecture still not have the purely equational shape we aim for. Like before with equality, logical implication in expressions like `eq(lin(U), lin(V)) => eq(assoc(U), assoc(V))` makes them logical formulas rather than terms, thus unable to participate in rewriting.

We resolve this issue in a similar way, by introducing a boolean function `impl` in the TIP theory, whose definition in TIP syntax is

```
(define-fun
  impl
  ((p Bool) (q Bool)) Bool
  (match p
    ((true q)
     (false true))))
```

following the truth table of logical implication: if the antecedent is true then the expression evaluates with the same value as the consequent, else if the antecedent is false then the expression evaluates to true. Thus the TIP theory's conjecture becomes

```
(prove
  :source prop_unambig
  (forall ((u E) (v E))
```

```
    (impl ((_ eq (list Tok)) (lin u) (lin v))
    ((_ eq E) (assoc u) (assoc v))))
```

After translating TIP to TPTP, we can see that first of all the conjecture has the intended format

```
fof(goal, conjecture,
  ![U, V]: impl(eq(lin(U), lin(V)), eq(assoc(U), assoc(V)))).
```

and second, `impl` does no longer have a "hidden" built-in definition like `=>`, but is accompanied by the following defining equational axioms produced by the translation of `impl`'s TIP definition:

```
fof(axiom, axiom,
  ![Q]: impl(true, Q) = Q).

fof(axiom, axiom,
  ![Q]: impl(false, Q) = true).
```

Implication is now too represented by an ordinary function symbol and can participate in rewriting just like any other function symbol in the encoding.

The same idea would be applied to the logical connective representing negation, i.e. $\sim$, which can also appear in TIP/TPTP conjectures. In such a case we would equivalently introduce a boolean `not` function in the TIP theory that followed the truth table of logical negation:

```
(define-fun
  impl
  ((p Bool)) Bool
  (match p
    ((true false)
     (false true))))
```

3.3 Existential Conjecture

We have now defined all the necessary components to finally present the way in which we modify the TIP theory's conjecture in order to represent the goal of counterexample extraction. We discussed previously that the conjecture should be a convenient formalisation of the sentence "there exists an input for which the property does not hold".

At this point, logical equality has already been replaced by the `eq` function, and equality tests are represented as ordinary terms of the theory. In order to express that a property evaluates to false, the property itself must also be represented as a term so it can be passed as an argument to the `eq` function. Thus, rather than embedding the property directly inside the conjecture, we decided to define it in two

separate steps in the TIP theory.

The first step is to introduce it as an ordinary boolean function of the theory. For our running example the property's function definition looks like this:

```
(define-fun
  prop_unambig
  ((u E) (v E)) Bool
  (impl ((_ eq (list Tok)) (lin u) (lin v))
    ((_ eq E) (assoc u) (assoc v))))
```

This definition introduces a dedicated function symbol for the property, here `prop_unambig`, whose truth value corresponds to the truth value of the property. Being represented by a function symbol is also how the property can be used as an argument to other functions like `eq`.

Therefore, all that's left to do in the second step is to express the conjecture as the statement that the property evaluates to false. Since equality is represented by `eq`, this is encoded as an equality between the property and the boolean value false. For our example this leads to the following finalised conjecture format:

```
(prove
  (exists ((u E) (v E))
    (= ((_ eq Bool) (prop_unambig u v) false) true)))
```

After the translation, the resulting TPTP encoding will contain the corresponding property definition:

```
fof(axiom, axiom,
  ![U,V]:
    prop_unambig(U,V) =
      impl(eq(lin(U), lin(V)),
           eq(assoc(U), assoc(V)))).
```

together with the conjecture:

```
fof(goal, conjecture,
  ?[U,V]:
    eq(prop_unambig(U,V), false) = true).
```

The symbol `?` represents the existential quantifier ∃ in TPTP. Note that the finalised conjecture has the purely equational form that motivated the transformations applied.

Observe that now, the structure of the conjecture becomes completely independent of the particular benchmark it belongs to. The specific property information is contained exclusively within the definition of the corresponding property function, while the conjecture itself always has the form

$$\exists x.\mathrm{eq}(\mathrm{prop}(x), \mathrm{false}) = \mathrm{true}$$

both in the TIP and in the TPTP encodings.

3.4 Guard Elimination

The elimination of logical operators we described earlier is not by itself sufficient to produce a purely equational TPTP theory.

Consider once again the function `assoc` from our running example. In the Haskell source file `assoc` is defined by pattern matching.

```
assoc :: E -> E
assoc ((a :+: b) :+: c) = assoc (a :+: (b :+: c))
assoc (a :+: b) = assoc a :+: assoc b
assoc (a :*: b) = assoc a :*: assoc b
assoc a         = a
```

Pattern matching function definitions in Haskell are interpreted top-to-bottom, meaning that a particular case only gets selected if none of the preceding cases can apply.

Such functions, when translated to TPTP with the current pipeline (even after applying all the previous transformations) can produce guarded axioms. This happens because the first-order theory, doesn't preserve the top-to-bottom priority rule on its axiom set, meaning that during the proof search, if more than one axioms can be applied to rewrite a clause, then the choice is based on prover heuristics and not the axiom's order. Thus, guards are placed in order to ensure that no more than one equation is applicable for a certain clause, and that equation matches the corresponding Haskell branch that would have matched the Haskell equivalent form of the clause.

In the specific case of `assoc`, its defining axioms in the produced encoding are:

```
fof(axiom, axiom,
  ![X]:
    (X != z(proj1(X), proj2(X)) =>
       (X != x2(proj12(X), proj22(X)) => assoc(X) = X))).

fof(axiom, axiom,
  ![Y, C]:
    (Y != z(proj1(Y), proj2(Y)) =>
       assoc(z(Y, C)) = z(assoc(Y), assoc(C)))).

fof(axiom, axiom,
  ![C, A, B]:
    assoc(z(z(A, B), C)) = assoc(z(A, z(B, C)))).
```

```
fof(axiom, axiom,
  ![A2, B2]:
    assoc(x2(A2, B2)) = x2(assoc(A2), assoc(B2))).
```

The first two axioms are guarded by disequality conditions, thus making the encoding still not be purely equational. So, to make the encoding purely equational we should find what is the origin of the guards and eliminate it.

We can trace the origin of the guards back to the wildcard patterns appearing in the TIP definition of `assoc`:

```
(define-fun-rec
  assoc :keep :source assoc
  ((x E)) E
  (match x
    ((((|:+:| y c)
      (match y
        ((((|:+:| a b) (assoc (|:+:| a (|:+:| b c))))
         (_ (|:+:| (assoc y) (assoc c))))))
     ((|:*:| a2 b2) (|:*:| (assoc a2) (assoc b2)))
     (_ x))))
```

A wildcard (_) branch is chosen in all the cases where none of the previous branches applied. In TIP the top-to-bottom semantics of pattern matching are preserved, so the "catch-all" does not create collisions since it will apply only after all previous branches have failed to match. For example, in the outer match expression, the wildcard branch

```
(_ x)
```

should only be selected when `x` is neither of the form `(|:+:| y c)` or `(|:*:| a2 b2)`, thus producing the semantically equivalent guarded TPTP axiom

```
fof(axiom, axiom,
  ![X]:
    (X != z(proj1(X), proj2(X)) =>
       (X != x2(proj12(X), proj22(X)) => assoc(X) = X))).
```

Thus, to eliminate the guarded axioms from the TPTP theory, we must eliminate the wildcards from the TIP function definitions. Hence, one more transformation our alternative translation Haskell → TIP makes, is making function definition constructor-complete, meaning there is a separate branch for each constructor of the function's input data type. For `assoc`, the corresponding constructor complete definition is

```
(define-fun-rec
  assoc :keep :source assoc
  ((x E)) E
  (match x
```

```
    (((|:+:| y c)
      (match y
        (((|:+:| a b) (assoc (|:+:| a (|:+:| b c))))
         ((|:*:| z x2) (|:+:| (assoc y) (assoc c)))
         (EX (|:+:| (assoc y) (assoc c)))
         (EY (|:+:| (assoc y) (assoc c))))))
     ((|:*:| a2 b2) (|:*:| (assoc a2) (assoc b2)))
     (EX x)
     (EY x))))
```

which after the TIP → TPTP translation produces the unguarded, equational axioms

```
fof(axiom, axiom,
  ![C, A, B]: assoc(z(z(A, B), C)) = assoc(z(A, z(B, C)))).
fof(axiom, axiom,
  ![C, Z, X2]:
    assoc(z(x2(Z, X2), C)) = z(assoc(x2(Z, X2)), assoc(C))).
fof(axiom, axiom, ![C]: assoc(z(eX, C)) = z(assoc(eX), assoc(C))).
fof(axiom, axiom, ![C]: assoc(z(eY, C)) = z(assoc(eY), assoc(C))).
fof(axiom, axiom,
  ![A2, B2]: assoc(x2(A2, B2)) = x2(assoc(A2), assoc(B2))).
fof(axiom, axiom, assoc(eX) = eX).
fof(axiom, axiom, assoc(eY) = eY).
```

3.5 Equality Axioms

In the equality section we introduced the polymorphic function `eq` and we used it to replace logical equality throughout the theory. But the declaration

```
(declare-fun eq (par (a) ((a a) Bool)))
```

simply introduces a new function symbol `eq` in the theory without giving it any semantics. The theory has still no knowledge of when two objects are equal or not equal. For this to change, the introduction of `eq` should be accompanied by a collection of axioms defining the behaviour of equality for objects of a certain datatype, for every datatype on which it is used.

At a minimum, every TIP theory must contain such axioms for the boolean data type since the uniform way of representing the conjecture makes a boolean equality comparison between the property and the value false. Additional equality axioms should be introduced only for the datatypes that actually appear as arguments of `eq` in the theory. For the running example, all of the theory's data types (`Bool`, `list`, `Tok`, `E`) require their respective axioms, but that is not true for every benchmark.

To avoid explicitly defining same-constructor equality axioms for every constructor,

we can first introduce a polymorphic axiom representing `eq`'s reflexivity property:

```
(assert
  (par (a)
    (forall ((x a))
      (= ((_ eq a) x x) true))))
```

We can then distinguish four separate cases on how we introduce the `eq` axioms for a specific data type. In the general case, let's suppose that the data type has both non-recursive and recursive constructors, as for the example's `E` data type which has the non-recursive constructors `EX`, `EY` and the resursive constructors `|:+:|`, `|:*:|`.

The first case is when both arguments of `eq` are constructed using non-recursive constructors. Since we already have reflexivity, we only need the cross-constructor axioms that infer that two objects constructed by two different constructors are not equal. For each pair of such constructors we need two axioms asserting the two directions of the same disequality, since `eq` is an ordinary function symbol that is not inherently symmetric. We could also add a polymorphic symmetry axiom but we don't, for reasons that will be explained later. For the example's `E` data type the according axioms are:

```
(assert (= ((_ eq E) EX EY) false))
(assert (= ((_ eq E) EY EX) false))
```

The second case is when one argument of `eq` is constructed using a non-recursive constructor and the other using a recursive constructor. Since two values built by different constructors can never be equal, equality immediately evaluates to false. As before, both directions must be introduced explicitly. For the example's `E` data type the according axioms are:

```
(assert
  (forall ((x E) (y E)) (= ((_ eq E) EX (|:+:| x y)) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) (|:+:| x y) EX) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) EX (|:*:| x y)) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) (|:*:| x y) EX) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) EY (|:+:| x y)) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) (|:+:| x y) EY) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) EY (|:*:| x y)) false)))
(assert
  (forall ((x E) (y E)) (= ((_ eq E) (|:*:| x y) EY) false)))
```

The third case is when both arguments of `eq` are constructed using recursive con-

structors, but two different ones. Then the same as the previous case holds. For the example's `E` data type the according axioms are:

```
(assert
  (forall ((x E) (y E) (z E) (x2 E))
    (= ((_ eq E) (|:+:| x y) (|:*:| z x2)) false)))
(assert
  (forall ((x E) (y E) (z E) (x2 E))
    (= ((_ eq E) (|:*:| x y) (|:+:| z x2)) false)))
```

The fourth and most interesting case is when both arguments of `eq` are constructed using the same recursive constructor. In this case, equality must be propagated recursively to the constructor arguments. More specifically, two values constructed by the same constructor are equal exactly when all their corresponding fields are equal. To express this behaviour while keeping the translated TPTP encoding purely equational, we introduce a boolean conjunction function

```
(define-fun
  and
  ((p Bool) (q Bool)) Bool
  (match p
    ((true q)
     (false false))))
```

We can then define equality between values built using the same recursive constructor as the conjunction of the equality comparisons between their corresponding fields. For the example's `E` data type the according axioms are:

```
(assert
  (forall ((x E) (y E) (z E) (x2 E))
    (= ((_ eq E) (|:+:| x y) (|:+:| z x2))
       (and ((_ eq E) x z)
            ((_ eq E) y x2)))))

(assert
  (forall ((x E) (y E) (z E) (x2 E))
    (= ((_ eq E) (|:*:| x y) (|:*:| z x2))
       (and ((_ eq E) x z)
            ((_ eq E) y x2)))))
```

Thus, equality between recursive constructor values is reduced to equality between their immediate subterms.

These axioms give back to `eq` the structural behaviour expected from equality over algebraic datatypes that was lost when we replaced built-in equality, while still representing that behaviour entirely through ordinary function symbols and equational axioms.

4

The TIP to TPTP Translation

We have already explained in the Background section that our actual contribution, in the sense of a new translation format, happens in the previous step of the pipeline, i.e. the Haskell → TIP part. The current TIP → TPTP stage uses the transformation tools offered by the TIP framework, and our task is merely to select the collection of TIP passes that produces a TPTP theory most suitable for a successful witness extraction.

4.0.1 Monomorphisation

As discussed in the Background section, the TPTP encodings used throughout this thesis are untyped. Thus, it is necessary for monomorphisation to be one of the passes applied during the TIP → TPTP translation. Monomorphisation replaces each definitional axiom of a polymorphic function with a collection of monomorphic instances of the same axiom. Thus, rather than generating a single function symbol shared by all types, the translation produces a distinct function symbol for each data type on which the polymorphic function is applied to in the theory.

This behaviour can be observed in every benchmark since for each of them, the corresponding TIP theory will contain at least one polymorphic function, the equality function `eq`. Due to monomorphisation, the produced TPTP encoding will contain as many distinct `eqX` function symbols as the data types on which equality is used. For a certain data type, the TPTP translation of the TIP equality axioms we introduced in the previous section will share the same function symbol. For our example, equality over the `list` data type is represented by the function symbol `eq`:

```
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq(cons(X, Y), cons(Z, X2)) = and(eq2(X, Z), eq(Y, X2))).
fof(axiom, axiom, ![X, Y]: eq(nil, cons(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq(cons(X, Y), nil) = false).
```

equality over the `Tok` data type is represented by the function symbol `eq2`:

```
fof(axiom, axiom, eq2(c, d) = false).
fof(axiom, axiom, eq2(c, x) = false).
fof(axiom, axiom, eq2(c, y) = false).
```

```
fof(axiom, axiom, eq2(c, plus) = false).
fof(axiom, axiom, eq2(c, mul) = false).
...
fof(axiom, axiom, eq2(mul, c) = false).
fof(axiom, axiom, eq2(mul, d) = false).
fof(axiom, axiom, eq2(mul, x) = false).
fof(axiom, axiom, eq2(mul, y) = false).
fof(axiom, axiom, eq2(mul, plus) = false).
```

equality over the `E` data type is represented by the function symbol `eq3`:

```
fof(axiom, axiom, eq3(eX, eY) = false).
fof(axiom, axiom, ![X, Y]: eq3(eX, z(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq3(eX, x2(X, Y)) = false).

fof(axiom, axiom, eq3(eY, eX) = false).
fof(axiom, axiom, ![X, Y]: eq3(eY, z(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq3(eY, x2(X, Y)) = false).

fof(axiom, axiom,
  ![X, Y, Z, X2]: eq3(x2(X, Y), z(Z, X2)) = false).
fof(axiom, axiom, ![X, Y]: eq3(x2(X, Y), eX) = false).
fof(axiom, axiom, ![X, Y]: eq3(x2(X, Y), eY) = false).
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq3(x2(X, Y), x2(Z, X2)) = and(eq3(X, Z), eq3(Y, X2))).

fof(axiom, axiom,
  ![X, Y, Z, X2]: eq3(z(X, Y), x2(Z, X2)) = false).
fof(axiom, axiom, ![X, Y]: eq3(z(X, Y), eX) = false).
fof(axiom, axiom, ![X, Y]: eq3(z(X, Y), eY) = false).
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq3(z(X, Y), z(Z, X2)) = and(eq3(X, Z), eq3(Y, X2))).
```

and equality over the `Bool` data type is represented by the function symbol `eq4`:

```
fof(axiom, axiom, eq4(false, true) = false).
fof(axiom, axiom, eq4(true, false) = false).
```

An important detail is that monomorphisation does not generate instances naively, for every type present in the theory. A new instance of a polymorphic function is introduced only for those types on which the polymorphic symbol is actually used. Thus, the number of generated instances depends on the structure of the particular benchmark.

This can be observed in the equality axioms of the `list` datatype in our running

example. Although lists are polymorphic, in this specific benchmark they are only instantiated with elements of type `Tok`. Consequently, the equality axiom for non-empty lists compares the list heads specifically using `eq2`, the equality function corresponding to Tok, and the list tails using `eq`, the equality function corresponding to lists. There are no other versions of this axiom produced in the theory, comparing list elements with a different `eq` variant, as they are not needed.

Monomorphisation also applies to the polymorphic reflexivity axiom introduced in the TIP theory, making it produce a collection of axioms representing the reflexivity of each distinct instance of equality in the theory:

```
fof(axiom, axiom, ![X]: eq(X,X) = true).
fof(axiom, axiom, ![X]: eq2(X,X) = true).
fof(axiom, axiom, ![X]: eq3(X,X) = true).
fof(axiom, axiom, ![X]: eq4(X,X) = true).
```

In the Inductionless Induction subsection we showed that the success of the entire counterexample-search process depends on the produced first-order theory faithfully capturing the semantics of the original Haskell program. Monomorphisation is a crucial component of the faithfullness of an untyped encoding. Without it, objects originating from different types could interact through the same equality function, potentially allowing derivations that would never occur in the original typed Haskell program. By generating separate monomorphic instances, monomorphisation prevents such interactions and therefore preserves the intended semantics of the source program throughout the proof search.

It is unfortunate that our running example cannot demonstrate the significance of the monomorphisation pass. In this particular case, the generated theory never produces a comparison between values that originate from different types, thus the absence of monomorphisation would not make the counterexample unsound. However we need to understand that this is merely a coincidence and should not be expected in general.

Consider this very simple example Haskell program

```
import Prelude (Eq,Ord,Show(..),(.),iterate,(!!),return,Bool(..))
import Tip

data Nat = S Nat | Z deriving (Eq,Ord)

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

rotate :: Nat -> [a] -> [a]
rotate Z     xs     = xs
rotate _     []     = []
```

```
rotate (S n) (x:xs) = rotate n (xs ++ [x])

prop  n xs = xs === rotate n (xs :: [Nat])
```

This program defines the natural numbers and the functions append (`++`) and `rotate` on lists. `rotate` takes a natural number and a list as arguments and rotates the list to the left by the number of positions specified by the natural number. The property `prop` states that any rotation of any list leaves the list unchanged.

Suppose that we translate this Haskell program to TIP using our translation and then produce two different TPTP encodings, one using the monomorphisation pass during the TIP → TPTP translation:

```
fof(axiom, axiom, ![Y]: x(nil, Y) = Y).
fof(axiom, axiom,
  ![Y, Z, Xs]: x(cons(Z, Xs), Y) = cons(Z, x(Xs, Y))).

fof(axiom, axiom, ![Z]: rotate(s(Z), nil) = nil).
fof(axiom, axiom,
  ![Z, X2, X3]:
    rotate(s(Z), cons(X2, X3)) = rotate(Z, x(X3, cons(X2, nil)))).
fof(axiom, axiom, ![Y]: rotate(z, Y) = Y).

fof(axiom, axiom,
  ![X, Y]: prop(X, Y) = eq(Y, rotate(X, Y))).

fof(axiom, axiom, eq3(false, true) = false).
fof(axiom, axiom, eq3(true, false) = false).

fof(axiom, axiom, ![X, Y]: eq2(s(X), s(Y)) = eq2(X, Y)).
fof(axiom, axiom, ![X]: eq2(s(X), z) = false).
fof(axiom, axiom, ![X]: eq2(z, s(X)) = false).

fof(axiom, axiom, ![X]: eq(X, X) = true).
fof(axiom, axiom, ![X]: eq2(X, X) = true).
fof(axiom, axiom, ![X]: eq3(X, X) = true).

fof(axiom, axiom, ![Q]: and(true, Q) = Q).
fof(axiom, axiom, ![Q]: and(false, Q) = false).
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq(cons(X, Y), cons(Z, X2)) = and(eq2(X, Z), eq(Y, X2))).

fof(axiom, axiom, ![X, Y]: eq(nil, cons(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq(cons(X, Y), nil) = false).

fof(goal, conjecture,
```

```
  ?[X, Y]: eq3(prop(X, Y), false) = true).
```

and one without using the monomorphisation pass during the TIP → TPTP translation:

```
fof(axiom, axiom, ![Y]: x(nil, Y) = Y).
fof(axiom, axiom,
  ![Y, Z, Xs]: x(cons(Z, Xs), Y) = cons(Z, x(Xs, Y))).

fof(axiom, axiom, ![Z]: rotate(s(Z), nil) = nil).
fof(axiom, axiom,
  ![Z, X2, X3]:
    rotate(s(Z), cons(X2, X3)) = rotate(Z, x(X3, cons(X2, nil)))).
fof(axiom, axiom, ![Y]: rotate(z, Y) = Y).

fof(axiom, axiom,
  ![X, Y]: prop(X, Y) = eq(Y, rotate(X, Y))).

fof(axiom, axiom, eq(true, false) = false).
fof(axiom, axiom, eq(false, true) = false).

fof(axiom, axiom, ![X]: eq(X, X) = true).

fof(axiom, axiom, ![Q]: and(true, Q) = Q).
fof(axiom, axiom, ![Q]: and(false, Q) = false).

fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq(cons(X, Y), cons(Z, X2)) = and(eq(X, Z), eq(Y, X2))).
fof(axiom, axiom, ![X, Y]: eq(nil, cons(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq(cons(X, Y), nil) = false).

fof(axiom, axiom, ![X, Y]: eq(s(X), s(Y)) = eq(X, Y)).
fof(axiom, axiom, ![X]: eq(s(X), z) = false).
fof(axiom, axiom, ![X]: eq(z, s(X)) = false).

fof(goal, conjecture,
  ?[X, Y]: eq(prop(X, Y), false) = true).
```

When running twee on the first, monomorphised file, it produces the witness

```
X = s(z)
Y = cons(z, cons(s(X), nil))
```

together with the following proof:

```
Proof:
  eq3(prop(s(z), cons(z, cons(s(X), nil))), false)
```

```
= { by axiom 7 (axiom) }
  eq3(eq(cons(z, cons(s(X), nil)), rotate(s(z), cons(z, cons(s(X), nil)))), false)
= { by axiom 9 (axiom) }
  eq3(eq(cons(z, cons(s(X), nil)), rotate(z, x(cons(s(X), nil), cons(z, nil)))), fa
= { by axiom 4 (axiom) }
  eq3(eq(cons(z, cons(s(X), nil)), x(cons(s(X), nil), cons(z, nil))), false)
= { by axiom 8 (axiom) }
  eq3(eq(cons(z, cons(s(X), nil)), cons(s(X), x(nil, cons(z, nil)))), false)
= { by axiom 2 (axiom) }
  eq3(eq(cons(z, cons(s(X), nil)), cons(s(X), cons(z, nil))), false)
= { by axiom 10 (axiom) }
  eq3(and(eq2(z, s(X)), eq(cons(s(X), nil), cons(z, nil))), false)
= { by axiom 10 (axiom) }
  eq3(and(eq2(z, s(X)), and(eq2(s(X), z), eq(nil, nil))), false)
= { by axiom 5 (axiom) }
  eq3(and(eq2(z, s(X)), and(eq2(s(X), z), true)), false)
= { by axiom 6 (axiom) }
  eq3(and(false, and(eq2(s(X), z), true)), false)
= { by axiom 1 (axiom) }
  eq3(false, false)
= { by axiom 3 (axiom) }
  true
```

When running twee on the second, non-monomorphised file, it produces the witness

```
X = s(z)
Y = cons(true, cons(false, nil))
```

together with the following proof

```
Proof:
  eq(prop(s(z), cons(true, cons(false, nil))), false)
= { by axiom 7 (axiom) }
  eq(eq(cons(true, cons(false, nil)), rotate(s(z), cons(true, cons(false, nil)))),
= { by axiom 8 (axiom) }
  eq(eq(cons(true, cons(false, nil)), rotate(z, x(cons(false, nil), cons(true, nil
= { by axiom 3 (axiom) }
  eq(eq(cons(true, cons(false, nil)), x(cons(false, nil), cons(true, nil))), false
= { by axiom 6 (axiom) }
  eq(eq(cons(true, cons(false, nil)), cons(false, x(nil, cons(true, nil)))), false
= { by axiom 2 (axiom) }
  eq(eq(cons(true, cons(false, nil)), cons(false, cons(true, nil))), false)
= { by axiom 9 (axiom) }
  eq(and(eq(true, false), eq(cons(false, nil), cons(true, nil))), false)
= { by axiom 9 (axiom) }
  eq(and(eq(true, false), and(eq(false, true), eq(nil, nil))), false)
= { by axiom 4 (axiom) }
  eq(and(eq(true, false), and(eq(false, true), true)), false)
```

```
= { by axiom 5 (axiom) }
  eq(and(false, and(eq(false, true), true)), false)
= { by axiom 1 (axiom) }
  eq(false, false)
= { by axiom 4 (axiom) }
  true
```

Observe that the witness returned by the non-monomorphised encoding is not valid in the original Haskell program, which explicitly requires the second argument of `rotate` to have type `[Nat]`. We can pinpoint the exact point in the proof where it diverges from the semantics of the original program. It is the application of the cons-cons list equality axiom right here

```
  eq(and(eq(true, false), eq(cons(false, nil), cons(true, nil))), false)
= { by axiom 9 (axiom) }
  eq(and(eq(true, false), and(eq(false, true), eq(nil, nil))), false)
```

This happens, because the relevant axiom of the non-monomorphised theory

```
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq(cons(X, Y), cons(Z, X2)) = and(eq(X, Z), eq(Y, X2))).
```

uses the same equality symbol `eq`, which is shared by every datatype in the theory. So, when it compares the heads of two lists through the subterm `eq(X, Z)`, there is no indication that both `X` and `Z` should be values of type `Nat`. Consequently, as we can see in the proof above, Twee is allowed to rewrite the term `and(eq(false, true), eq(nil, nil))` as `eq(cons(false, nil), cons(true, nil))` using the aforementioned axiom, and thus instantiate lists with boolean values which would be impossible in the original Haskell program were the annotation `xs :: [Nat]` ensures that all elements of the compared lists should be of type `Nat`. Thus, it is then to be expected that the witness returned is a witness only for the untyped first-order theory and not for the typed original program.

Twees run on the monomorphised theory would never have allowed this rewrite to happen. The corresponding axiom

```
fof(axiom, axiom,
  ![X, Y, Z, X2]:
    eq(cons(X, Y), cons(Z, X2)) = and(eq2(X, Z), eq(Y, X2))).
```

uses the symbol `eq2` to compare the heads of the two lists, which is generated by the translation specifically for values of type `Nat`.

5

Counterexample Extraction

We have now presented our complete translation process from Haskell programs to first-order theories. The question that remains to be answered is whether these theories actually allow theorem provers to discover counterexamples to the properties of the corresponding Haskell programs.

As discussed in the Inductionless Induction subsection, assuming that the produced first-order theory faithfully captures the semantics of the original Haskell program, the existence of a counterexample implies the existence of a first order witness in the encoding. Since the theorem provers used in this thesis are complete for first-order reasoning, they are in principle guaranteed to eventually discover such a witness.

But for a method of bug finding to be practical, the completeness guarantee is not sufficient, if the actual extraction takes several minutes, hours or even more. Consequently, what we examine in this section is not whether counterexamples are theoretically derivable from the encoding, which we have already argued to be true, but whether the provers used are capable of deriving them within a practically useful amount of time. For that, we decided to execute each benchmark with each prover under a fixed time limit of 120 seconds. If the prover did not manage to terminate within the limit then the extraction was considered unsuccessful. Since all three provers are run on the same encoding, the one produced by our translation, the differences in performance reflect the compatibility of each prover with our encoding.

For the example of the previous sections, all three provers run with their standard settings terminate successfully. Twee is the fastest, requiring approximately 5 seconds and producing the output

```
Goal 1 (goal): eq4(prop_unambig(X, Y), false) = true.
The goal is true when:
  X = x2(eX, x2(X, Y))
  Y = x2(x2(eX, X), Y)
```

closely followed by E which requires approximately 6 seconds and produces the output

```
# SZS answers Tuple [[x2(eY,x2(eY,eY)), x2(x2(eY,eY),eY)]|_]
# Proof found!
```

and then by Vampire which requires 43 seconds, which is significantly longer, and answers

```
SZS answers Tuple [X1,X0.[x2(x2(eY,X0),X1),x2(eY,x2(X0,X1))]|_]
```

The results of this example are in fact representative of the results obtained across the entire benchmark suite. Overall, Twee has the highest number of successful witness extractions, and in the cases where another prover also succeeded, Twee was typically faster. E had a similar success rate, as it managed to extract a witness for almost every benchmark that Twee did. However, E was consistently slower, sometimes by only a few seconds, like in the example, and other times by a substantially larger margin. Vampire displayed a vastly different behaviour. In most cases, as in the example, it was the slowest prover, requiring significantly more time than Twee and E or even failing to terminate within the time limit even when both of the other two provers succeeded. However, there was still a small number of cases, where the opposite behaviour was displayed. In these cases, Vampire extracted a witness surprisingly fast, sometimes almost immediately, while both Twee and E failed to terminate. Together, these results indicate that Twee and E have similar proof search mechanisms, which are generally more compatible with the encodings produced by our translation, while Vampire's distinct mechanisms are not that compatible.

An interesting observation that is displayed by the witnesses the three provers found for the example encoding, is that sometimes a prover will not produce a ground term but a term containing variables. These terms describe an entire family of terms that are falsifying inputs for the property rather than simply giving a single concrete one. From the perspective of debugging, such witnesses can be preferred as they are more informative than ground ones. A ground witness simply demonstrates that a bug exists, whereas a parametrised witness may reveal a general pattern of inputs that trigger the bug. This can make it easier for a programmer to identify the underlying cause of the incorrect behaviour.

Twee's output is actually even more informative as, for every successful run, besides showing the falsifying witness it also includes the full equational proof leading to said witness. This is especially useful for debugging as it provedes the full rewrite derivation of how the property evaluates to false on that witness. Thus the programmer can identify the precise point at which the property fails.

It is important to know that the prover cannot be specifically instructed to search for a parametrised witness. The goal given to the prover is simply to find a witness that satisfies the conjecture. Whether the returned witness is ground or symbolic depends entirely on the state of the proof search at the moment the conjecture is established. To understand why such witnesses are produced, recall that the prover constructs new clauses by repeatedly applying its inference rules to previously derived clauses. Eventually, it may derive a clause that is sufficient to satisfy the conjecture. No matter whether that clause still contains variables or not, the prover has no incentive to continue the search and derive additional clauses that would in-

stantiate those variables further. From the prover's perspective, since the conjecture has been proven, the required task has already been completed.

One limitation of the current implementation is that the extracted witnesses are reported using the symbols of the generated first-order encoding rather than the original Haskell program. Although in most cases it is relatively easy to interpret these symbols as their Haskell counterparts, in some others it can be trickier. For instance, in the example the constructors `|:+:|` and `|:*:|` are represented in the encoding by the symbols `x2` and `z` respectively, as the function symbols in a TPTP encoding can not have non-letter identifiers. But this is a rare case, and in general function and constructor names are preserved by the translation pipeline and can thus be interpreted back to its origins directly. It could be useful, however, in the future to implement a post-processing step that automatically translates witnesses back into Haskell syntax.

We recognise that the 120 second time limit used throughout the experiments is a very arbitrary way of determining success, as for sufficiently difficult properties or sufficiently rare counterexamples, even several minutes, hours, or longer could still constitute a successful result. In fact, we came across benchmarks whose counterexamples correspond to solutions of computationally hard problems, aiming to take advantage of this method's completeness to try and solve these problems. For example, consider a Haskell program which encodes a graph and whose property states that the graph is not 3-colourable. Then a counterexample for this property would be precisely a valid 3-colouring of the graph, which depending on the specific graph can be very difficult to find. For this problem, our method extracting a "counterexample" could be considered a significant success even if the prover required a substantial amount of time. Problems like this, however, are not the objective of this thesis. Rather, we are more interested in the discovery of ordinary programming bugs of the kind for which frameworks such as QuickCheck are commonly used. Having in mind that when QuickCheck succeeds, it does so almost immediately, a response time on the order of minutes could already make one doubt the practicality of the method. Since our experiments showed that our method uncovers bugs for which QuickCheck fails, falsely reporting that all tests passed, a small compromise in time is not prohibitive, but still it isn't practical to increase the limit much further.

In fact QuickCheck does not find a counterexample for the property of the recurring example program. Running QuickCheck with a simple generator

```
instance Arbitrary E where
  arbitrary = sized gen where
    gen 0 = elements [EX, EY]
    gen n =
      oneof
        [ pure EX
        , pure EY
        , (:+:) <$> gen (n `div` 2) <*> gen (n `div` 2)
        , (:*:) <$> gen (n `div` 2) <*> gen (n `div` 2)
```

```
        ]
```

produces

```
ghci> quickCheck prop_unambig
+++ OK, passed 100 tests; 695 discarded.
```

Thus, QuickCheck reports the property, which was shown false by our method, as correct since none of its tests falsified it.

The main reason QuickCheck fails in this case is the property's specific format, and specifically the fact that it has an antecedent. That makes most tests get discarded before the consequent is even evaluated, thus significantly lowering the probability of finding a counterexample. This is one of the situations in which our method of counterexample extraction can offer a significant advantage over random testing, thus making accepting an increase in the extraction time seem reasonable.

One could argue that a more carefully designed `Arbitrary` instance tailored specifically to this property might reduce the number of discarded tests and thus increase the likelihood of discovering a counterexample. That is consistent with QuickCheck's literature, which emphasises the importance of custom generators for difficult properties. However, in order to do a fair comparison, we should consider the standard configurations for each tool. We also can not assume that in the general case the programmer will know how to manually engineer the input distributions to make random testing concentrate on the part of the input space relevant to the property.

It is important to clarify that the TPTP theory's conjecture does not act as a direct prompt for the proof search, but as a success condition. An existential conjecture such as the ones we are concerned with, specifies what information must eventually be derived does not make the prover explicitly search for inputs that satisfy it. The prover begins its proof search repeatedly deriving logical consequences of the theory according to its inference rules and heuristics. During this process, it periodically checks whether the information derived so far is sufficient to establish the conjecture. If it is, the proof terminates and a witness is returned. If it isn't the proof search continues.

This distinction is important for understanding the limitations of the method, especially when it fails for benchmarks where random generation succeeds by doing a small number of tests. A reasonable misconception is that the prover starts from the conjecture and therefore focuses its effort on derivations that appear relevant to proving it. But the reality is that provers offer weak goal direction, meaning that the search is only slightly guided by the conjecture. Due to that, they may spend substantial amount of time deriving consequences that are ultimately unrelated to the goal property. This constitutes the main struggle of our method.

Although the overall counterexample-search process is, as explained, complete, the time required to discover a witness can vary dramatically between benchmarks and

this variation does not necessarily correlate with how rare or how deeply hidden the actual bug is in the original Haskell program. Instead, it is largely determined by whether the prover's heuristics choose to prioritise the derivations that ultimately uncover the witness.

This can be experimentally confirmed by two observations that may seem counterintuitive to someone that assumes that the conjecture of the encoding is the determining factor of the derivations that get prioritised in the proof search.

First, adding extra axioms to a theory, even when they are completely irrelevant to both the conjecture and the rest of the theory, consistently delays proof termination. This indicates that the prover still spends a non-negligible part of its search effort deriving consequences from those axioms despite them not contributing at all to the proof goal.

Second, and more surprising, we encountered benchmarks for which the provers failed to terminate on the standard produced encoding, and still failed to terminate even after adding an axiom that explicitly reveals the counterexample. Intuitively we would expect such an axiom to make the proof trivial, and the prover to terminate immediately. However, since the heuristics of the prover made it fail on the standard run, meaning that they likely made it exhaust its time on irrelevant derivations, then the same heuristics made it ignore the revealing axiom in the second run thus making it fail again. This demonstrates that the failures are not caused by lack of information required to find the witness but by the inability of guiding the proof search towards that information.

Taken together, these observations suggest that improving goal direction is likely to be the most important step towards making theorem-prover-based counterexample extraction a practical bug-finding technique.

6

Lexicographic Path Ordering

In the previous section we identified weak goal direction as the primary limitation of the counterexample-extraction process. We thus need to explore alternative methods of guiding the prover towards making the derivations that are actually meaningful for the conjecture.

Although Twee achieved the strongest overall performance in our initial experiments where we run all three provers with their standard settings, it doesn't offer any options for guiding the proof search. So, although its performance is good, it doesn't have much potential for improvement. On the other hand E provides a mechanism that can constitute a form of artificial goal direction, and that is Lexicographic Path Ordering (LPO).

LPO is normally used to orient equations and control rewriting. It requires the user to specify a precedence ordering on the function symbols of the theory. This ordering influences which terms are considered larger and therefore affects the equations and derivations that the prover prefers during proof search. Consequently, although LPO was not originally designed as a goal-direction mechanism, it can be employed to bias the search towards derivations involving particular symbols.

This observation provides another concrete reason why our proposed equational encoding is especially suited for the counterexample-extraction procedure. Recall that one of the main transformations made in our alternative Haskell $\rightarrow$ TIP translation was the replacing of logical constructs such as equality and implication by ordinary function symbols. Another one was the representation of the property itself as an ordinary function definition. These transformations resulted in both the logical constructs and the property having their respective representative function symbols in the TPTP encoding as well as in the TIP encoding. If we hadn't implemented them, then we wouldn't be able to have the logical constructs or the property participate in the symbol ordering and thus be prioritised through LPO. This would significantly limit how helpful LPO can be and how much it actually decreases the proof search time, as these are exactly the symbols that need to get prioritised in order for that to happen.

Since the motivation is to imitate goal direction, the precedent order should be determined by the structure of the conjecture. Recall that every conjecture produced

by our translation has the form

$$\exists x.\mathrm{eq}(\mathrm{prop}(x), \mathrm{false}) = \mathrm{true}.$$

Consequently, the symbols that appear closest to the conjecture should receive the highest priority. The specific equality function symbol used in the conjecture should therefore be placed first, followed by the property function symbol, and then by the symbols appearing in the property's defining equation. The remaining symbols are ordered according to their distance from the conjecture in the dependency graph of the theory.

For our running example this produces the precedence order:

```
eq4 > prop_unambig > impl > eq > eq3 > lin > assoc > and >
linTerm > append > eq2 > z > x2 > cons > true > false >
nil > plus > mul > c > d > x > y > eX > eY
```

Running E again on the example's TPTP encoding but this time using the specified LPO order produces the same witness as the standard run did but terminates in approximately 0.5 seconds showing a significant improvement, and actually being even faster than Twee was.

Again, the behaviour observed on this example is representative of the general results of the entire benchmark suite. The appropriate LPO order improved E's performance in every case, besides the cases where both the standard run and the LPO run failed to terminate, for which we can't know whether the use of LPO helped or not. In the cases that E using LPO terminated, it ended up having a lower execution time than Twee did.

Seeing how much a relatively indirect mechanism of goal direction improved the results of our experiments makes the main argument of the previous section even stronger as we can imagine how much more efficient the counterexample extraction process would be if the provers offered explicit goal direction.

Note that an inappropriate LPO ordering can have the exact opposite effect and substantially increase the proof-search time. Determining a good precedence is often non-straightforward because most symbols simultaneously play multiple roles within the theory. Consequently, there is often no single obviously correct ordering. Therefore, when the appropriate ordering is unclear, it is better to explore multiple candidate orderings than to select one arbitrarily.

It is worth mentioning that Vampire also supports a format of LPO-based term orderings, but that specific format isn't useful for our counterexample-extraction procedure. Unlike E, Vampire does not allow the user to explicitly state a complete precedence ordering for the symbols of the theory. Instead, it provides a collection of predefined precedence schemes based on properties such as the order in which symbols appear in the input, symbol arity, or symbol frequency. These schemes are ultimately unrelated to our line of work. Since our use of LPO aims to imitate

goal direction, the precedence of a symbol should be determined by its relevance to the conjecture and this information can not be expressed through such schemes. Thus, since its format of LPO can not improve its performance, Vampire remains the weakest of the three provers based on successful counterexample extraction from the theories generated by our translation.

7

Evaluation

7.1 Comparison

Up to this point we have only concerned ourselves with the proposed equational encoding. The results were encouraging, especially for Twee and E and we actually identified advantages that arose directly from the equational structure of the encoding. Most notably, the encoding can allow logical constructs and properties themselves to participate in rewriting and enables the use of LPO as a mechanism for approximating goal direction.

However, it is important to remember that the decision that purely equational theories would be the best representation for counterexample extraction was ultimately just a hypothesis, mainly motivated by the characteristics of Twee. So, even though the proposed encoding performs well, allowing provers to extract counterexamples for most of our benchmarks, does not automatically imply that it is the best possible encoding for counterexample extraction.

In the Background section we described the pre-existing Haskell → TIP → TPTP translation, which is made for program verification. So, using that as a baseline, we could had simply changed the conjecture from the verification goal "for all inputs the property holds" to the counterexample goal "there exists an input so that the property doesn't hold" while keeping everything else as it was. Remember Section 3 where we introduced the transformations we made in TIP. Now, we are removing all these transformations except for the reformulation of the conjecture itself. This allows us to isolate the effect of the equational encoding and conclude whether it was actually beneficial.

The standard TIP encoding, as can be seen in the code provided in the Background section, contains wildcard branches which as explained produce guarded clauses in the translated TPTP encoding. In the general case, some of them will be non-Horn. Thus, Twee will not be able to be used for most of the benchmarks, and therefore it can not participate in the comparison of the encodings.

On the standard encoding, Vampire consistently performs at least as well as on the equational encoding and often significantly better. More specifically, in the benchmarks which we mentioned that on the equational encoding it terminates sur-

prisingly fast, it retains its performance, and for the other cases it often improves by a margin. E, didn't produce the same results. It performed consistently worse than it did for the equational encoding. In a very few cases, it had some surprisingly quick results but for these Vampire was still faster, so it is clear that the encoding benefits Vampire.

Once again, our running example's results are representative of the general case. Running Vampire on its standard encoding terminates in approximately 2 seconds

```
% SZS answers Tuple
    [X1,X0.[x2(x2(eX,X0),z(eX,X1)),x2(eX,x2(X0,z(eX,X1)))]|_]
    for CFG5_prop_unambig_fof
% Success in time 2.277 s
```

which is a big improvement from the equational's 43 seconds. Runnig E fails to terminate within the 120 second time limit

```
# Failure: Resource limit exceeded (time)
# SZS status ResourceOut
eprover: CPU time limit exceeded, terminating
119.14s user 1.25s system 99% cpu 2:00.58 total
```

which is an obvious regression, especially since all three provers had successfully terminated for the equational encoding, and specifically E only needed 6 seconds, which across our benchmarks is a relatively very fast time.

Thus, if we wanted to pair each prover with the encoding that we expect it to perform better for, then for E the objective choice would still be the equational. And this choice is only further strengthened when LPO is taken into account. Remeber that the standard encoding does not contain explicit function symbols representing the property, equality, or the logical connectives. As a consequence, these symbols cannot participate in the precedence ordering and therefore cannot be prioritised through LPO. Since these are precisely the symbols that consistently receive the highest priority in the most effective LPO orderings for the equational encoding, as they are exactly the ones contained in the conjecture clause, the sense of artificial goal direction that was previously given is now substantially weakened.

Of course, LPO can still be applied, as the standard encoding still contains function symbols which we can order appropriately, but the effect has been consistently weaker, which was, as explained, expected.

For the example, the precedence ordering produced by the dependency graph of its standard encoding is

```
lin > assoc > linTerm > append >
proj1 > proj2 > proj12 > proj22 > head > tail >
z > x2 > cons > nil > c > d > x > y > plus > mul > eX > eY
```

Using this ordering, E terminates in approximately 22 seconds. While this is a significant improvement over the non-terminating standard run, it remains substantially slower than Vampire's approximately 2-second execution time on the same encoding. In contrast, for the equational encoding, LPO reduced E's time from approximately 6 seconds to 0.5 seconds, making it faster than both Twee and Vampire, which we can now see is also the best overall performance across provers and encodings.

Therefore our equational encoding seems to be the one having the best chance for successfully extracting a counterexample for a Haskell program property. Especially, when the prover is either Twee or E, from which Twee's absence of LPO makes E to be the most preferable.

7.2 Limitations

In the introduction, we stated that one of the limitations of property-based testing tools is that, even when they fail to find a counterexample, they do not provide any guarantee that none exists and thus that the property is correct.

To be fair, we should admit that our method exhibits the same limitation, although for a very different reason. Assuming that the produced first-order encoding faithfully captures the semantics of the original Haskell program and property, and assuming that the theorem prover is complete, then a correct property would in theory eventually be established by saturating the theory without deriving any witness that falsifies it. However, having a prover continue its proof search until saturation is practically impossible, even for relatively small theories.

The lack of strong goal direction only makes this limitation more apparent. When a prover fails to terminate within a reasonable amount of time (or even within any amount of time), provides little to no information about whether the property is actually correct. The prover may be spending its effort on derivations unrelated to the conjecture, and not have yet explored the path that would lead to a witness, or it may have already exhausted all relative paths and being on its way to saturation. Consequently, the absence of a counterexample after a long execution does not make us increasingly confident that the property is correct. It only indicates that the prover has not yet managed to derive a witness.

8

Conclusion

The experiments conducted throughout this thesis allow us to give a positive answer to the question posed in the Introduction: Theorem provers can be used to find bugs in Functional programs.

As the inductionless induction argument had already theoretically suggested, the production of a faithful first-order encoding of a program and its property allows us to overcome the need for a proof system capable of inductive reasoning and instead extract witnesses corresponding to property counterexamples using ordinary first-order theorem provers. The success of our work can therefore be considered a confirmation of this idea.

Thus, the central question shifts from "can provers reason about recursive functional programs?" to "can theorem provers reason about recursive functional programs efficiently?". And now, the efficiency isn't determined by the strength of the proof system, but by the prover's ability to direct its efforts towards derivations that actually contribute to the proof goal.

The work presented in this thesis can be summed up as an attempt to mitigate this issue from the encoding side of the encodingprover combination. The proposed equational encoding, together with the suggested use of LPO, was designed to give the prover the best possible opportunity to discover witnesses efficiently. However, there is only a limited amount of actions we can take before we have to actually steer our efforts on the prover part.

To encourage further investigation of this issue, the benchmark suite and generated encodings produced during this thesis have been contributed to the TPTP benchmark infrastructure. Since modern theorem provers are routinely evaluated on this infrastructure, we hope that our contribution will provide additional motivation for the automated reasoning community to develop proof-search techniques that provide stronger goal direction in the context of counterexample extraction.

9

Future Work

It has now been made clear that speed is not a strong point of our method in comparison to mainstream bug-finding tools and mainly QuickCheck. It is also a fact that there is little that can be done on our end to substantially improve this situation. One direction we could explore is to develop techniques for producing more effective LPO orderings than we do now, where we follow the conjecture's dependency graph in the TPTP encoding. However, anything we do is secondary to the main fix, which is adding stronger goal-direction mechanisms to the provers themselves and is in the hands of the people who develop and maintain them.

Thus, if we want to make our method appealing, we should give the programmer a reason to prefer it against testing tools even if that would mean a compromise on time. That can only happen if our method is able to extract counterexamples for property classes where testing tools are inherently ineffective.

The main example used throughout this thesis already revealed one such category: properties containing antecedents. Such properties are indeed harder for testing tools, in fact we saw that QuickCheck failed to find a counterexample for the example benchmark while our method succeeded. However, in this case, there are some things that can be done to help QuickCheck succeed. Since the reason for the failure is that because of the antecedent, a large proportion of generated tests is discarded before the property can even be evaluated, a more carefully engineered generator and Arbitrary instance can bias said tests towards the input space that satisfies the antecedent, thus helping the counterexample search. We can not, therefore, claim this as a class of properties where our method can succeed while QuickCheck definitely fails.

Thus, an actual big win for our method would be the ability to extract counterexamples for properties for which testing isn't just ineffective, but inapplicable. One such direction is higher-order properties, i.e. properties that quantify over functions.

During the final stages of this thesis, we began investigating the problem of higher-order counterexample extraction and obtained some encouraging first results. Before describing this first attempt, it is important to understand why higher-order properties require special treatment in our method. At first thought, one might expect the existing translation pipeline to already handle them successfully, since the TIP

framework provides transformations such as defunctionalisation, which eliminate higher-order constructs and thus still produce a corresponding first-order TPTP encoding.

Like for the first-order properties, we will see the issue of the standard translation in practice through an example. Consider the Haskell program only consisting of the property

```
prop f p xs =
map f (filter p xs)
    ===
filter p (map f xs)
```

which states that the ordinary Haskell `map` and `filter` functions commute. This property is false. A real life counterexample would be: `f` the function that defines the successor on Natural Numbers, `p` a predicate that denotes the even Natural Numbers and `xs` a list of natural numbers, say `[Z, S Z]`. Then

```
map successor (filter even [Z, S Z]]) =
map successor [Z] =
[S Z]
```

while

```
filter even (map successor [Z, S Z]) =
filter even [S(Z), S(S Z)] =
[S(S(Z))]
```

We obviously do not expect a first-order theorem prover to return a counterexample in the form of complete function definitions. However, the behaviour of a function on a particular finite set of inputs can be described using ordinary first-order facts. Thus, we can expect from a prover to be able to derive a witness describing only the function values that are relevant to the counterexample. For the example property, such a witness could have the form

```
xs = [Z, S(Z)]
f Z = S Z
f (S Z) = S(S Z)
p Z = True
p (S Z) = False
```

Notice that these equations do not fully define either `f` or `p`. They only describe their behaviour on the inputs that are required to falsify the property. This observation will become important later in our discussion.

The prover can even find a smaller counterexample like

```
xs = [Z]
f Z = S Z
```

```
p Z = True
p (S Z) = False
```

When translated through the standard pipeline it produces the following TPTP theory

```
fof(axiom, axiom, ![X, X2]: head(cons(X, X2)) = X).
fof(axiom, axiom, ![X, X2]: tail(cons(X, X2)) = X2).
fof(axiom, axiom, ![X, X2]: nil != cons(X, X2)).
fof(axiom, axiom, ![F]: map(F, nil) = nil).
fof(axiom, axiom,
  ![F, Y, Xs]: map(F, cons(Y, Xs)) = cons(apply1(F, Y), map(F, Xs))).
fof(axiom, axiom, ![P]: filter(P, nil) = nil).
fof(axiom, axiom,
  ![P, Y, Xs]:
    (apply12(P, Y) =>
       filter(P, cons(Y, Xs)) = cons(Y, filter(P, Xs)))).
fof(axiom, axiom,
  ![P, Y, Xs]:
    (~apply12(P, Y) => filter(P, cons(Y, Xs)) = filter(P, Xs))).
fof(goal, conjecture,
  ?[F, P, Xs]: map(F, filter(P, Xs)) != filter(P, map(F, Xs))).
```

where the higher-order functions `f` and `p` are represented using application operators `apply1` and `apply12` respectively. One could think this is sufficient since the behaviour required for a counterexample can indeed be expressed within the first-order theory. For example the witness

```
xs = [Z]
f Z = S Z
p Z = True
p (S Z) = False
```

corresponds in the translated theory to the first-order equations

```
xs = cons(z, nil)
apply1(F, z) = s(z)
apply12(P, z) = true
apply12(P, s(z)) = true
```

which are sufficient to falsify the property. Thus, the problem is not that the translation loses the information required for a counterexample to be found. On the contrary, the corresponding first-order theory is still satisfiable and still contains models representing the desired function behaviour. The problem actually appears in the conjecture itself:

```
fof(goal, conjecture,
  ?[F, P, Xs]: map(F, filter(P, Xs)) != filter(P, map(F, Xs))).
```

where `F` and `P` are still represented as ordinary first-order variables. The provers we use in this thesis report witnesses by providing substitutions for existentially quantified variables. For first-order inputs this was sufficient, since a witness could be represented by a single term. But a higher-order witness is not a term but a collection of function evaluations. For example, the intended witness for `P` is not a single object, but the behaviour `z` $\mapsto$ `true`, `s(z)` $\mapsto$ `false`. There is no first-order term whose structure can directly encode this finite mapping.

To make higher-order witnesses reconstructible, functions must be represented in the generated first-order theory as ordinary first-order terms. Achieving this properly would require modifications to the TIP $\rightarrow$ TPTP translation itself, so that the translated theory contains explicit representations of functions that can be returned as part of a witness.

However, since the work presented here is only a first exploration of higher-order counterexample extraction, we do not modify the translation pipeline yet. Instead, we apply the necessary transformations directly at the Haskell level, replacing higher-order values with equivalent first-order representations before the translation process even begins. Thus, from the perspective of the translation pipeline, the arguments of the property are now ordinary first-order values, which our existing alternative Haskell $\rightarrow$ TIP $\rightarrow$ TPTP translation can already handle without any modifications.

For the example, that means that we introduce a datatype for functions over natural numbers:

```
data Nat = Zero | Suc Nat

data NatFun a =
    Answer a (NatFun a)
```

together with an application operator:

```
apply :: NatFun a -> Nat -> a
apply (Answer y _) Zero     = y
apply (Answer _ f) (Suc n) = apply f n
```

The intuition behind this representation is that a function over natural numbers can be viewed as a collection of answers, one for each possible input. The constructor Answer stores the value that the function should return for Zero, while its second argument contains the information required to determine the values for all successor inputs. The apply function acts as the evaluator of such a representation. Evaluating apply f Zero returns the first stored answer, and evaluating apply f (Suc n) recursively evaluates the remainder of the representation. Thus, the goal has been achieved, as function values are no longer higher-order objects but first-order values constructed from the constructor Answer.

The original property can then be rewritten as:

```
prop f p xs =
    map (apply f) (filter (apply p) xs)
        ===
    filter (apply p) (map (apply f) xs)
```

Translating the new property with our alternative translation pipeline produces the TPTP file

```
fof(axiom, axiom, ![Q]: and(true, Q) = Q).
fof(axiom, axiom, ![Q]: and(false, Q) = false).

fof(axiom, axiom,
  ![X, Y, Z]:
    prop(X, Y, Z) =
      eq(
        map(lam(X), filter(lam2(Y), Z)),
        filter(lam3(Y), map(lam4(X), Z)))).
fof(axiom, axiom, ![F]: map(F, nil) = nil).
fof(axiom, axiom,
  ![F, Y, Xs]: map(F, cons(Y, Xs)) = cons(apply1(F, Y), map(F, Xs))).
fof(axiom, axiom, ![T, E]: ite(true, T, E) = T).
fof(axiom, axiom, ![T, E]: ite(false, T, E) = E).
fof(axiom, axiom, ![P]: filter(P, nil) = nil).
fof(axiom, axiom,
  ![P, Y, Xs]:
    filter(P, cons(Y, Xs)) =
      ite(apply12(P, Y), cons(Y, filter(P, Xs)), filter(P, Xs))).
fof(axiom, axiom, ![Z, X2]: apply(answer(Z, X2), zero) = Z).
fof(axiom, axiom,
  ![Z, X2, N]: apply(answer(Z, X2), suc(N)) = apply(X2, N)).
fof(axiom, axiom, ![Z, X2]: apply2(answer2(Z, X2), zero) = Z).
fof(axiom, axiom,
  ![Z, X2, N]: apply2(answer2(Z, X2), suc(N)) = apply2(X2, N)).
fof(axiom, axiom, eq3(false, true) = false).
fof(axiom, axiom, eq3(true, false) = false).
fof(axiom, axiom, ![X]: eq2(X, X) = true).
fof(axiom, axiom, ![X]: eq3(X, X) = true).
fof(axiom, axiom, ![X]: eq(X, X) = true).
fof(axiom, axiom,
  ![X, Y, Z, X2]: eq(cons(X, Y), cons(Z, X2)) = and(eq(Y, X2), eq2(X, Z))).
fof(axiom, axiom, ![X, Y]: eq(nil, cons(X, Y)) = false).
fof(axiom, axiom, ![X, Y]: eq(cons(X, Y), nil) = false).
fof(axiom, axiom, ![X, Y]: eq2(suc(X), suc(Y)) = eq2(X, Y)).
fof(axiom, axiom, ![X]: eq2(zero, suc(X)) = false).
fof(axiom, axiom, ![X]: eq2(suc(X), zero) = false).
fof(axiom, axiom, ![X, X2]: apply1(lam(X), X2) = apply(X, X2)).
fof(axiom, axiom, ![X, X5]: apply1(lam4(X), X5) = apply(X, X5)).
```

```
fof(axiom, axiom, ![Y, X3]: apply12(lam2(Y), X3) = apply2(Y, X3)).
fof(axiom, axiom, ![Y, X4]: apply12(lam3(Y), X4) = apply2(Y, X4)).
fof(goal, conjecture,
  ?[X, Y, Z]: eq3(prop(X, Y, Z), false) = true).
```

All three provers terminate successfully on this theory, with Vampire being the fastest. In approximately 0.1 seconds it returns the witness

```
[X1,X0.[answer(suc(zero),X0),
    answer2(false,answer2(true,X1)),cons(zero,nil)]|_]
```

which, recalling the definitions of `NatFun` and `apply` corresponds to the Haskell counterexample

```
f Zero = Suc Zero

p Zero = False
p (Suc Zero) = True

xs = [Zero]
```

The important point is that the returned values for f and p are not complete function definitions. The variables `X0` and `X1` are used as placeholders to indicate that the rest of the functions' behaviour is left unspecified. Thus, the witness only constrains the function values that are necessary for falsifying the property and leaves all remaining inputs unconstrained.

Bibliography

[1] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *Automated Deduction – CADE-24*, M. P. Bonacina, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 392–406, ISBN: 978-3-642-38574-2.

[2] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279, ISBN: 1581132026. DOI: 10.1145/351240.351266. [Online]. Available: https://doi.org/10.1145/351240.351266.

[3] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, ser. Haskell '08, Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48, ISBN: 9781605580647. DOI: 10.1145/1411286.1411292. [Online]. Available: https://doi.org/10.1145/1411286.1411292.

[4] N. Smallbone, "Twee: An equational theorem prover," in *Automated Deduction CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 1215, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 602–613, ISBN: 978-3-030-79875-8. DOI: 10.1007/978-3-030-79876-5_35. [Online]. Available: https://doi.org/10.1007/978-3-030-79876-5_35.

[5] S. Schulz, "System description: Eă1.8," in *Logic for Programming, Artificial Intelligence, and Reasoning*, K. McMillan, A. Middeldorp, and A. Voronkov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 735–743, ISBN: 978-3-642-45221-5.

[6] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35, ISBN: 978-3-642-39799-8.

[7] C.-P. Wirth, "History and future of implicit and inductionless induction: Beware the old jade and the zombie!" In *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, D. Hutter and W. Stephan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 192–203, ISBN: 978-3-540-32254-2. DOI: 10.1007/978-3-540-32254-2_12. [Online]. Available: https://doi.org/10.1007/978-3-540-32254-2_12.

A

Appendix 1