DEPARTMENT OF COMPUTER SCIENCE

GenEx: Automatically Proving Properties of Functional Programs

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Thursday 4th May, 2023

Abstract

This thesis presents GenEx, a system for automatically proving properties of functional programs. These properties are expressed as equations between program terms. For example, one property that GenEx can formally prove is `map id [x1, x2, x3]` $\doteq$ `[x1, x2, x3]`.

These properties can also be seen as specifications for how some function should behave. By formally proving that various properties are satisfied, we can verify that the program behaves as expected. If GenEx fails to prove some specification, this indicates that there may be a bug in the function.

GenEx is a saturation-based theorem prover, using a novel set of inference rules. These inference rules are a variation of the superposition calculus, with the addition of symbolic execution.

In this project, we focus on the underlying proof system. We will develop the metatheory of the proof system. In particular, we will show that the proof system is refutation sound and conditionally terminating. This conditional termination is a unique aspect of GenEx, as the pure superposition calculus is not guaranteed to terminate.

In addition, we will extend the proof system with a procedure for case splitting on the value of expressions. This will allow GenEx to prove a larger variety of properties. We will also prove that this procedure works correctly. Finally, we will propose a new rule that could be added to the set of inference rules. We will justify why this rule is useful. Then, we will consider whether the proof system would retain the same soundness and termination properties with the addition of this rule.

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. Steven Ramsay.

Chapter 1

Introduction

In this thesis, we present GenEx, a system for automatically proving properties of functional programs. These properties are expressed as equations between terms. For example, a property that GenEx is able to prove is `map id [x1, x2, x3]` $\doteq$ `[x1, x2, x3]`.

Programmers often use comments to describe the purpose of some function and how the function should behave. One way they do this is by listing some examples of input-output pairs for the function.

The name GenEx, short for Generic Examples, alludes to the use case of the system. The purpose of GenEx is to formally prove properties. These properties can also be seen as examples describing how some function should behave. Properties abstract away specific input-output pairs, instead describing some general behaviour that a function should satisfy. Therefore, we call these generic examples.

GenEx is also able to report when it fails to prove some property. Since these properties serve as examples of how a function should behave, failure to prove a property indicates that there may be a bug in the function.

There exists an implementation of GenEx for Haskell. However, as the main contributions of this thesis are proving properties of the underlying proof system and extending it to allow for case splitting on the values of expressions, we will be considering GenEx as a theoretical system. Therefore, when we refer to the "programmer" throughout this thesis, this is just some human who is using the underlying proof system to prove a property.

1.1 Examples

In this section, we give some examples of properties that GenEx can and cannot prove. For now, we will refrain from describing in full detail the exact process that GenEx follows in attempting to prove some property, instead showing the process informally. Further examples listing the entire process with details of the inference rules used are provided in Section 3.7.

To use GenEx, the programmer specifies some goal equation that they wish to prove. They may also optionally specify some hypotheses. Hypotheses are statements that the proof system will assume to be true. Each hypothesis must be a Horn clause, i.e. of the form $(\ell_1 \doteq r_1, \cdots, \ell_n \doteq r_n) \Rightarrow (s \doteq t)$. Note that the antecedent of the implication may be trivial: Horn clauses of the form $\top \Rightarrow s \doteq t$ are abbreviated to simply $s \doteq t$.

1.1.1 Example Property

Suppose we have written a `map` function in Haskell:

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

We want to ensure that our function behaves correctly. One example of its expected behaviour that we may provide is the following:

$$\texttt{map id [x1, x2, x3]} \doteq \texttt{[x1, x2, x3]}$$

GenEx is able to automatically prove that this property holds.

Informally, GenEx uses symbolic execution to reduce some clause when possible. When symbolic execution is not possible because the value of some expression is not known, it uses equations from the

hypotheses to rewrite the expression. In this example, GenEx simply uses symbolic execution to reduce `map id [x1, x2, x3]` to `[x1, x2, x3]`.

1.1.2 Usage of Hypotheses

The property shown in Section 1.1.1 holds regardless of what the values of `x1`, `x2` and `x3` are. However, some properties will only hold if the variables involved fulfil certain conditions. An example of such a property is:

$$\texttt{filter p [x1, x2, x3]} = \texttt{[x1, x3]}$$

This property holds only if the following statements are true:

$$\texttt{p x1} \doteq \texttt{True} \tag{1.1}$$

$$\texttt{p x2} \doteq \texttt{False} \tag{1.2}$$

$$\texttt{p x3} \doteq \texttt{True} \tag{1.3}$$

GenEx allows programmers to specify some hypotheses, which are statements that the proof system will assume to be true. If the programmer specifies (1.1), (1.2) and (1.3) as hypotheses, GenEx is then able to prove the property.

1.1.3 False Properties

Suppose we have some property that is not true. For example, say we made a mistake while writing the map function:

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

In this case, the property `map id [x1, x2, x3]` $\doteq$ `[x1, x2, x3]` does not hold. GenEx is able to automatically deduce that this property does not hold for this function.

1.1.4 Termination

Suppose we have a function that takes the head of a list and returns a singleton list of this element:

```
headList [] = []
headList (x:xs) = [x]
```

We also define a function that checks whether a list is empty:

```
empty [] = True
empty (x:xs) = False
```

The following property should hold:

$$\texttt{empty (headList xs)} \doteq \texttt{empty xs} \tag{1.4}$$

If the programmer provides a hypothesis stating the structure of `xs`, we can prove this property. For example, given the following hypothesis:

$$\texttt{xs} \doteq \texttt{[x1, x2, x3]}$$

We are able to rewrite `xs` to `[x1, x2, x3]` and prove that (1.4) holds by symbolically executing `empty (headList [x1, x2, x3])`.

However, consider the situation where the programmer instead provides the following hypothesis:

$$\texttt{xs} \doteq \texttt{map id xs} \tag{1.5}$$

This hypothesis is valid, as it holds for any list `xs`. However, if we naively apply this hypothesis whenever it is possible to, the process will never terminate.

We could rewrite any occurrence of `xs` within `empty (headList xs)` $\doteq$ `empty xs` to `map id xs`. This gives the following equation:

$$\texttt{empty (headList (map id xs))} \doteq \texttt{empty (map id xs)}$$

This rewriting has not revealed anything about the structure of `xs`, so we are still unable to use symbolic execution. The only option is to use the hypothesis (1.5) to rewrite `map id xs` to `xs`.

$$\texttt{empty (headList xs)} \doteq \texttt{empty xs}$$

Note that this is our original goal equation. The above process would simply repeat infinitely.

This demonstrates that if we naively apply the hypotheses provided by the programmer, we may encounter situations where the proof system does not terminate. We would like to avoid the possibility of GenEx running forever while trying to prove some property. If the system has no guarantee of termination, there is no guarantee that GenEx is able to report whether a property is true or false.

GenEx restricts the situations in which hypotheses can be applied to rewrite terms. The author's supervisor conjectured that these restrictions will guarantee termination of the system, provided that certain conditions are fulfilled. One condition is that the hypotheses provided by the programmer are indeed true. The second condition is that every term used in the goal equation and hypotheses is terminating (cannot be reduced infinitely). It has been noted that it is very rare for Haskell programmers to write non-terminating functions [16]. Therefore, we believe it is reasonable that GenEx offers guarantees only for terminating terms.

Intuitively, if we restrict rewriting such that terms must become closer to termination (i.e. more evaluated) after a rewrite, we will avoid the situation described above. `map id xs` is not closer to termination than `xs`, and this results in the infinite loop. However, there are situations in which we want to rewrite from a variable to a more complex term. Consider the property (1.4) again:

$$\texttt{empty (headList xs)} \doteq \texttt{empty xs}$$

We cannot symbolically execute `empty (headList xs)` because this requires pattern matching on the structure of `xs`. If we rewrite `xs` to `(y:ys)`, symbolic execution can proceed. Therefore, we allow `xs` to be rewritten to `(y:ys)`.

We treat variables like `xs` as standing in for values (i.e. fully evaluated terms). Clearly, `(y:ys)` is not a more evaluated term than `xs`, as `xs` is already a value, but this is a valid rewrite in GenEx. This demonstrates that proving the termination of GenEx is not straightforward. One of the main objectives of this project is to formalise the metatheory of the proof system sufficiently, such that proving the conditional termination of the system is possible.

1.1.5 Case Splitting

Let us return to the property shown in Section 1.1.4:

$$\texttt{empty (headList xs)} \doteq \texttt{empty xs} \tag{1.6}$$

If the programmer provides no hypotheses about `xs`, then the system is unable to make progress. It cannot use symbolic execution, since it does not know the structure of `xs`, and it has no hypotheses to use to rewrite `xs`. However, if the system were able to split on the value of `xs`, it would be able to prove this property.

Suppose `xs` is either `[]` or `(y:ys)` for some `y` and `ys`. This can be expressed as the clause $\texttt{xs} \doteq \texttt{[]} \vee \texttt{xs} \doteq \texttt{(y:ys)}$. Note that this is not a Horn clause, so this cannot be used as a hypothesis by GenEx. However, let us consider what would happen if GenEx could split on the two possible values of `xs`.

If $\texttt{xs} \doteq \texttt{[]}$, then we can rewrite `xs` with `[]`. This gives us the following equation:

$$\texttt{empty (headList [])} \doteq \texttt{empty []} \tag{1.7}$$

We are able to symbolically execute the left hand side of (1.7), showing that both sides of the equation are equal.

$$\texttt{empty []} \doteq \texttt{empty []}$$

Now suppose $\texttt{xs} \doteq \texttt{(y:ys)}$. Again, we rewrite `xs`, giving the following formula:

$$\texttt{empty (headList (y:ys))} \doteq \texttt{empty (y:ys)} \tag{1.8}$$

Then, we symbolically execute both sides of (1.8), eventually deriving the following equation:

$$\texttt{False} \doteq \texttt{False}$$

Since we are able to prove the property in both cases, it must be that the property holds for all lists `xs`.

This example shows that the addition of case splitting on the values of expressions would allow GenEx to prove more types of properties. Therefore, another objective of the project is to add a procedure for case splitting to GenEx.

1.2 Contributions

The author was provided with the proof system and its semantics. In addition, the author's supervisor had conjectured that the proof system is refutation sound and conditionally terminating.

The contributions of this project are:

- Description of the syntax and semantics of a small functional language, and proofs of various properties about the language
- Proof that the system is refutation sound
- Proof that the system terminates, provided certain conditions are fulfilled
- A method for case splitting on the values of expressions and a proof that this method works correctly
- Proposal of a new rule for the proof system and discussion of whether the new rule would affect the soundness and termination properties of the system
- Discussion on possible areas of future research into GenEx.

1.3 Outline

We begin in Chapter 2, by discussing some related works.

In Chapter 3, we start by presenting the small functional language that we will be using throughout the rest of the thesis. We then introduce the proof system itself. This includes the decision problem that the system aims to solve, the inference rules and our new method for case splitting. We also show several examples of how GenEx proves properties.

Then, in Chapter 4, we prove that the system is refutation sound and conditionally terminating.

In Chapter 5, we present an additional rule that could be added to the set of inference rules. We also discuss whether the proof system retains the same soundness and termination properties when this rule is added.

Finally, we conclude in Chapter 6 by discussing possible areas of future research.

Chapter 2

Background

In this chapter, we provide some background on each aspect of GenEx, and discuss some related works.

2.1 Property-Based Testing

In property-based testing, the aim is to show that some property of the program holds for all possible input values. This is used to verify that the program is behaving correctly. These properties can also be used as a form of documentation for the code [8].

QuickCheck [8] is perhaps the most famous system for property-based testing. QuickCheck takes some property specified by the programmer, and randomly generates a large number of input values. It checks if the property holds for each set of input values. If QuickCheck finds a counterexample to the property, it will shrink this counterexample as much as possible, producing a minimal counterexample. This minimal counterexample will then be outputted [10].

QuickCheck is designed to be lightweight and simple to use [8]. However, the downside to random testing is it cannot guarantee some property is correct. It is possible that the provided property is false, but on some specific run QuickCheck fails to find a counterexample. This is particularly problematic for properties which are in the form of an implication. QuickCheck attempts to generate test cases in which the antecedent of the implication holds. If the conditions are very restrictive, it may successfully generate only a few test cases [8].

In contrast, if GenEx is able to prove some property, it is guaranteed that this property holds for all combinations of input values that can be represented by the symbolic input. Consider the property listed in Section 1.1.1: `map id [x1, x2, x3]` $\doteq$ `[x1, x2, x3]`. GenEx is able to prove this property; this means that this property is guaranteed to hold for all lists of length 3. In addition, GenEx is designed to prove properties in the form of implications (it aims to prove that the hypotheses imply the goal), regardless of how restrictive the conditions are.

Although the properties provided to QuickCheck and GenEx are of a similar form, these two systems are essentially orthogonal in operation. QuickCheck is able to "prove" a very large variety of properties, as it automatically splits on the values of expressions during the process of its random test generation [8]. However, it cannot guarantee that some property is correct. In contrast, GenEx formally proves properties, but it is able to prove a comparatively small variety of properties, since it requires the programmer to explicitly provide case splits.

An example that demonstrates the differences between QuickCheck and GenEx is the property `map id xs` $\doteq$ `xs`.

GenEx will not be able to progress at all unless the programmer provides more information about the structure of `xs`. It cannot split on the possible values of `xs` unless the programmer provides the hypothesis `xs` $\doteq$ `[]` $\vee$ `xs` $\doteq$ `(y:ys)`. In fact, even with this hypothesis, GenEx will not successfully prove this property. Formally proving this property would require induction. This is discussed further in Sections 2.4 and 6.2.1.

In contrast, QuickCheck will have no issues generating test cases for this property. Since it generates random inputs rather than trying to prove the property formally, it can simply generate a large number of different lists. This property indeed holds, so QuickCheck will report that it found no counterexamples.

Another area of research within property-based testing is exploiting the polymorphism of functions. Bernardy et al. showed that, for certain polymorphic properties, it is possible to construct one monomorphic

instance such that showing correctness of that one instance is sufficient to show correctness of all possible instances [5].

GenEx does not consider the polymorphism of functions. This means that our proof system does not use polymorphism to shortcut any proofs. However, we believe that there is an advantage to the way that GenEx operates: it should also work for dynamically typed languages, where the type of some function is not necessarily known at compilation time.

2.2 Superposition Calculi

GenEx formally proves properties using a method known as saturation-based theorem proving [2]. Saturation-based theorem provers aim to prove that some set of clauses is unsatisfiable. They begin by assuming that each clause in this set is true, and aim to derive a contradiction. They successively apply some set of inference rules to the set of known clauses. Application of inference rules to known clauses deduces new clauses, which can then be added to the set of known clauses. This process stops when the set of known clauses is saturated. Saturation occurs when any new clause that can be deduced is a duplicate of some existing clause. If the saturated set of clauses contains the empty clause (a contradiction), this indicates the original set of clauses is unsatisfiable.

In GenEx, we use a novel set of inference rules. These inference rules are a variation on the superposition calculus [1]. The superposition calculus is a set of inference rules for the purpose of proving clauses in first-order logic.

[9, p.386] presents a superposition calculus for ground Horn clauses. We reproduce the rules here, omitting some of the side conditions in the interests of simplicity:

$$\ell \succ r \frac{\Gamma' \Rightarrow \ell \doteq r \quad \Gamma \Rightarrow s[\ell] \doteq t}{\Gamma', \Gamma \Rightarrow s[r] \doteq t} \text{ (Right Superposition)}$$

$$\ell \succ r \frac{\Gamma' \Rightarrow \ell \doteq r \quad \Gamma, s[\ell] \doteq t \Rightarrow \Delta}{\Gamma', \Gamma, s[r] \doteq t \Rightarrow \Delta} \text{ (Left Superposition)}$$

$$\frac{\Gamma, s \doteq s \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \text{ (Equality Resolution)}$$

We will present the steps that a saturation-based theorem prover would take to prove some goal equation, using this set of inference rules. Suppose we want to prove the equation $s \doteq t$ with the hypotheses $\ell \doteq t$ and $\ell \doteq s$. For simplicity, we will assume $s \succ l$ and $l \succ t$ under the ordering $\succ$ used in this calculus.

The proof system begins with a set of known clauses. This set contains the hypotheses. In addition, the goal clause $s \doteq t$ is negated, giving the clause $s \doteq t \Rightarrow \bot$, which is also added to the set of known clauses. We negate the goal equation since we are aiming to prove the unsatisfiability of some set of clauses. If we can show the unsatisfiability of this set of clauses by deducing the empty clause $\top \Rightarrow \bot$, this is equivalent to showing that the hypotheses imply the satisfiability of the original goal equation.

We can then make the following steps:

- The (Left Superposition) rule can be applied to $s \doteq \ell$ and the negated goal clause $s \doteq t \Rightarrow \bot$, resulting in a new clause $\ell \doteq t \Rightarrow \bot$.
- By (Right Superposition) with $\ell \doteq t$, we can rewrite ℓ to t in $s \doteq \ell$, giving the clause $s \doteq t$.
- By (Left Superposition) on $\ell \doteq t$ and $\ell \doteq t \Rightarrow \bot$, we get $t \doteq t \Rightarrow \bot$.
- By (Equality Resolution) on $t \doteq t \Rightarrow \bot$ we derive the empty clause $\top \Rightarrow \bot$.

Our set of known clauses now consists of the following clauses:

$$\ell \doteq t$$
$$\ell \doteq s$$
$$s \doteq t \Rightarrow \bot$$
$$\ell \doteq t \Rightarrow \bot$$
$$s \doteq t$$
$$t \doteq t \Rightarrow \bot$$
$$\top \Rightarrow \bot$$

Note that any clause that can be deduced will be a duplicate of a clause in the set of known clauses. Therefore, this set of clauses is saturated. The saturated set contains the empty clause $\top \Rightarrow \bot$, indicating that the original set of clauses is unsatisfiable.

The pure superposition calculus, and modified versions of it, have been used in many automated theorem provers, including Vampire [13] and the E theorem prover [14]. These theorem provers usually operate on clauses in first-order logic [13, 14], whilst our proof system operates only on ground (without variables) Horn clauses.

The superposition calculus is refutation complete for Horn clauses [1]. This means that the calculus is able to derive a contradiction from any starting set of unsatisfiable Horn clauses [2].

The calculus we use in GenEx is a variation on the superposition calculus, with a stricter ordering on terms and additional rules to allow for symbolic execution. The consequence of these modifications is that our proof system is terminating, provided that any hypotheses are valid and every term used in the initial set of clauses is terminating. In contrast, the pure superposition calculus is not guaranteed to terminate: the saturation procedure may continue infinitely [9]. However, GenEx is not refutation complete.

2.3 Symbolic Execution

In normal execution, a program is run with some specific input values. King [12] describes symbolic execution as the process of executing some program with symbols, rather than values, as input. These symbols represent a range of input values. The existing operators of the language are adapted to handle symbols as well as values [12]. This allows one execution of the program to cover multiple possible inputs that can be represented by the input symbols [3].

As described in [3], when some branch in the program is encountered by a symbolic execution engine, the engine creates multiple new states to deal with the possible paths that execution could take. Each of these states will continue execution independently. Each state represents some control flow path, and the engine keeps track of which assumptions were made along each path. Each path is verified to check whether some set of inputs could cause execution to take that path. Each path is also checked for whether some property of interest is violated [3].

Baldoni et al. [3] note that exhaustive symbolic execution is sound and complete. This requires that all possible execution states are explored. However, exhaustive symbolic execution is likely infeasible for many real applications due to an issue known as path explosion. Path explosion refers to the fact that the number of different states may increase exponentially with the number of branches encountered in the program [3].

Due to this issue, symbolic execution engines will often make tradeoffs for the sake of performance. For example, they will often sacrifice soundness or completeness [3].

GenEx aims to resolve the path explosion problem by requiring the programmer to specify which expressions they would like the system to split on, and which values they would like the system to consider for each expression.

If some symbolic execution engine encounters an expression of the form `if (x1 <= x2) then ... else ...`, it creates two execution states: one to follow the `then` branch and the other to follow the `else` branch. GenEx would not perform this case split unless the programmer specified the hypothesis `x1 <= x2` $\doteq$ `True` $\vee$ `x1 <= x2` $\doteq$ `False`.

This requirement does not affect the soundness of the proof system. In addition, GenEx requires no bookkeeping of the assumptions made in each instance: these are implicitly maintained during the splitting procedure.

However, the tradeoff that has been made is that GenEx is will get "stuck" where other symbolic executions engines may not, if progress depends on knowing the value of some expression that it has no information about. This is because GenEx will not case split on the value of an expression automatically.

2.4 Inductive Theorem Proving

Induction is a technique that can be used to formally prove that some property holds. Induction involves assuming that the property holds of "smaller" inputs, and using this assumption (which is known as the induction hypothesis) to prove that the property holds of a larger input [7].

Induction is often required to formally prove properties where the structure of the input is not explicitly known, such as `map id xs` $\doteq$ `xs`. Even with the addition of case splitting, GenEx cannot prove this property. Intuitively, the length of the input list could be infinite, and it is not possible for the programmer

to specify an infinite number of case splits. Informally, an induction scheme could first prove that the property holds of the empty list `[]`. It could then show that if the property holds for some list `xs`, it will also hold for the list `(x:xs)`. This amounts to proving that the property holds for lists of any possible length.

Induction is a powerful technique, but it can also be very difficult and computationally expensive. As Bundy [7] describes, induction techniques can result in extremely large or even infinite search spaces, since there are multiple induction rules that can be applied at each step. This means that careful consideration of heuristics in navigating the search space is required [7].

In addition, any induction principle is incomplete, meaning that no form of induction can prove all possible formulas [7]. As a consequence, there is no solution that can solve all problems.

Zeno [15] is an automated theorem prover for properties of Haskell programs. If Zeno is given a property with some input `x`, where the structure of `x` is unknown, it will try to prove the property by induction. It will create a branch for each constructor of the datatype of `x`. In each branch, any relevant induction hypotheses are assumed. For example, if one branch requires the property to be proved for `(y:ys)`, Zeno will assume in this branch that the property holds of `ys`. If the property can be proved in all branches, this shows that the property holds of the original input `x` [15].

Another approach to induction is cyclic proofs [6, 11]. This involves applying some set of rules to create a derivation tree, and trying to find cycles in the resulting derivation tree. These cycles are formed when a statement of the same form as the goal is seen in the derivation tree [11]. It is possible for cycles to be unsound, e.g. if the cycle links the goal back to itself [6]. Therefore, cycles must be checked for soundness. Informally, for a cycle to be valid, there must be some decrease (for example, in the input of the property) within the cycle. This check can be very expensive.

GenEx does not currently have any form of induction. The purpose of GenEx is to be a lightweight tool for verifying the correctness of program behaviour. The generic examples that GenEx is designed to prove do not require induction. GenEx is able to cope well with problems where the structure of the input is known. This fits with the philosophy that a generic example describes how a function should behave for some general set of inputs. Therefore, the addition of induction would increase the complexity and overhead of the system, whilst offering limited practical benefits for our situation. However, there are interesting theoretical aspects that may arise from combining GenEx with induction schemes. There is further discussion of this possibility in Section 6.2.1.

Chapter 3

Proof System

In this chapter, we begin by formalising the functional language that we will be using throughout this thesis. We present some properties of this language that will be helpful in later proofs.

Then, we formally describe the decision problem that the proof system aims to solve. We also describe the case splitting procedure that we added to GenEx.

We conclude the chapter by discussing the inference rules that the proof system uses. We also provide some examples of how the proof system proves properties.

3.1 Functional Language

We consider a small functional language, which we will call GFL, for GenEx Functional Language. We will derive the symbolic execution dynamics from this language.

3.1.1 Syntax

We begin by discussing the syntax of GFL.

Variables We assume we have a denumerable set of logical variables x, y, z, etc. These represent arbitrary terms that we are reasoning about. In particular, they are used for symbolic input terms. For example, consider the property `filter p [x1, x2, x3]` $\doteq$ `[x1, x3]`. In this case, `p`, `x1`, `x2` and `x3` are logical variables.

Names We assume a denumerable set of term variables which we refer to as *names* to avoid contention with logical variables. These are ranged over by a. Names are most often used for bound variables in the program.

Constants We consider two kinds of constant symbols:

- *Programming language constants.* The constants of the programming language will be uninterpreted by the logic, with their behaviour instead described by the transition relation of each program. We further distinguish a subset of the language constants as *constructors*, for which we reserve the metavariable k. For example, we have the `[]` and `(:)` constructors (the latter of which may also be used infix) for lists.

- *Symbolic projections.* For each constructor k, we assume a *symbolic projection* k_i^{-1} and we also assume a symbolic projection for abstractions λ^{-1}.

  The k_i^{-1} projection projects out the i^{th} argument of some constructor k, i.e. $k_i^{-1}\,(k\,s_0 \cdots s_n) = s_i$. Note that k_i^{-1} is different from any projection that already exists in the programming language.

  The λ^{-1} projection projects out the body of some abstraction, i.e. $\lambda^{-1}\,(\lambda a.s) = s$. In some situations, we will specify the specific abstraction to be matched against, e.g. λ_a^{-1}.

  These projections are used in the definition of extended variables.

Definition 3.1.1 (Extended Variables)**.** An *extended variable* is an expression v of the following grammar:

$$v ::= x \mid k_i^{-1}\, v \mid \lambda^{-1}\, v$$

Any extended variable v contains exactly one logical variable x, and is said to be *rooted* at x.

Extended variables are treated as atomic by symbolic execution. During symbolic execution, GenEx cannot look 'inside' an extended variable to see what the result of the projection would be.

The purpose of extended variables is to reveal more about the structure of some logical variable. For example, the equation $\texttt{xs} \doteq \texttt{(:)}\,((\texttt{:})_1^{-1}\,\texttt{xs})\,((\texttt{:})_2^{-1}\,\texttt{xs})$ tells us that `xs` is a non-empty list. Suppose that symbolic execution cannot proceed because it requires pattern matching on the structure of `xs`. By rewriting `xs` to $\texttt{(:)}\,((\texttt{:})_1^{-1}\,\texttt{xs})\,((\texttt{:})_2^{-1}\,\texttt{xs})$, the structure of `xs` has been revealed, so symbolic execution can progress. An example of this situation is shown in Section 3.7.2.

Definition 3.1.2 (Terms)**.** Terms, typically s, t and other lowercase letters from the end of the alphabet, are as follows:

$$\begin{array}{rrcl} \text{(Terms)} & s, t & ::= & a \mid v \mid k \mid s\,t \mid \lambda a.s \mid \mathsf{case}\ s\ \mathsf{of}\ alts \\ \text{(Alts)} & alts & ::= & \epsilon \mid k\,\overline{a} \to s,\ alts \end{array}$$

A term can be some name a, an extended variable v, a constructor k, a function application $s\,t$, an abstraction $\lambda a.s$ or a pattern matching case statement $\mathsf{case}\ s\ \mathsf{of}\ alts$.

We use the notation $\overline{k\,\overline{a} \to t}$ as shorthand for $k_1\,\overline{a_1} \to t_1, \cdots, k_n\,\overline{a_n} \to t_n$.

The lambda abstraction $\lambda a.s$ binds a in s. We define $\mathsf{FV}(t)$ to be the set of all logical variables in the term t. We take the notion of α-equivalence as usual.

Λ_Σ is the set of all terms t with $\mathsf{FV}(t) \subseteq \Sigma$. This definition will be useful when we come to prove the conditional termination of the proof system in Section 4.2.

A term is *concrete* just if it does not contain any logical variable. The set of all concrete terms is Λ_0.

Definition 3.1.3. We define programs in the following way.

$$\text{(Programs)} \quad P \quad ::= \quad \epsilon \mid a = s; P$$

Note that we will also interpret this list of program statements as a set.

Any Haskell functions used throughout this thesis can be translated into GFL. For example, consider the `map` function we defined in Section 1.1.1:

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

We can define this `map` function as a GFL program statement in the following way:

```
map = λf. λys. case ys of
                    [] → []
                    (:) x xs → (:) (f x) (map f xs)
```

For the rest of this thesis, we will use the Haskell version of functions and omit their translations into GFL.

Definition 3.1.4 (One-hole Contexts)**.** We write $s[]$ for a one-hole context and $s[t]$ for the term obtained by replacing the unique hole variable $\Box$ with the term t.

$$\begin{array}{rrcl} \text{(One-hole contexts)} & C[] & ::= & \Box \mid C[]\,s \mid s\,C[] \mid \lambda a.C[] \mid \mathsf{case}\ C[]\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \\ & & & \mid \mathsf{case}\ s\ \mathsf{of}\ A[] \\ & A[] & ::= & k\,\overline{a} \to C[],\ alts \mid k\,\overline{a} \to s,\ A[] \end{array}$$

A consequence of this definition is that the hole cannot occur inside an extended variable.

One-hole contexts are used to express that some term $s[t]$ has t as a subterm.

3.1.2 Semantics

We fix a program P for the remainder of this thesis. We write $s \triangleright^* t$ to denote that s reduces to t in finitely many steps using the dynamics described in Figure 3.1. We write $s \overset{*}{=} t$ to denote that there is a common reduct u such that $s \triangleright^* u$ and $t \triangleright^* u$.

$$\frac{}{(\lambda a.s)\, t \triangleright s[t/a]}\ (\mathsf{Beta}) \qquad \frac{s \triangleright s'}{s\, t \triangleright s'\, t}\ (\mathsf{AppL}) \qquad \frac{t \triangleright t'}{s\, t \triangleright s\, t'}\ (\mathsf{AppR})$$

$$\frac{s \triangleright s'}{\lambda a.s \triangleright \lambda a.s'}\ (\mathsf{Abs})$$

$$\frac{||\overline{s}|| = ||\overline{a_i}||}{\mathsf{case}\ k_i\ \overline{s}\ \mathsf{of}\ \overline{k\, \overline{a} \to t} \triangleright t_i[\overline{s}/\overline{a}]}\ (\mathsf{Case1}) \qquad \frac{s \triangleright s'}{\mathsf{case}\ s\ \mathsf{of}\ alts \triangleright \mathsf{case}\ s'\ \mathsf{of}\ alts}\ (\mathsf{Case2})$$

$$\frac{t_i \triangleright t_i'}{\mathsf{case}\ s\ \mathsf{of}\ \overline{k\, \overline{a} \to t} \triangleright \mathsf{case}\ s\ \mathsf{of}\ k_1\, \overline{a_1} \to t_1, \cdots, k_i\, \overline{a_i} \to t_i', \cdots, k_n\, \overline{a_n} \to t_n}\ (\mathsf{Case3})$$

$$\frac{a = s \in P}{a \triangleright s}\ (\mathsf{Prog})$$

Figure 3.1: Dynamics of GFL

The reduction strategy GFL uses is essentially full β-reduction, with any redex able to make a step at any time. The rule (Prog) states that any program statement is a valid reduction step that can be made.

We now provide some examples of how terms are reduced using $\triangleright$. In the following examples, we will use the $(:)$ constructor for lists infix. We also underline the redex that is to be reduced at each step.

$$\begin{array}{lll}
 & \mathsf{case}\ \underline{((\lambda x.x)\,(y:ys))}\ \mathsf{of}\ [] \to [], (z:zs) \to (\lambda x.x)\, z & \\
\triangleright & \mathsf{case}\ \underline{x[(y:ys)/x]}\ \mathsf{of}\ [] \to [], (z:zs) \to (\lambda x.x)\, z & \\
 & = \mathsf{case}\ (y:ys)\ \mathsf{of}\ [] \to [], (z:zs) \to \underline{(\lambda x.x)\, z} & (\mathsf{Case2})\ \text{and}\ (\mathsf{Beta}) \\
\triangleright & \underline{\mathsf{case}\ (y:ys)\ \mathsf{of}\ [] \to [], (z:zs) \to z} & (\mathsf{Case3}) \\
\triangleright & z[y/z] = y & (\mathsf{Case1})
\end{array}$$

$$\begin{array}{lll}
 & (\lambda x.(x\,x))\underline{((\lambda y.y)\, z)} & \\
\triangleright & (\lambda x.(x\,x))(y[z/y]) = \underline{(\lambda x.(x\,x))\, z} & (\mathsf{AppR})\ \text{and}\ (\mathsf{Beta}) \\
\triangleright & (x\,x)[z/x] = z\, z & (\mathsf{Beta})
\end{array}$$

Assumption 3.1.5. We assume that GFL is confluent.

The confluence of full β-reduction is already known [4]. Since our reduction strategy $\triangleright$ is essentially full β-reduction, we believe the proof of confluence for GFL would not be particularly interesting (and it would be time-consuming).

Definition 3.1.6 (Termination)**.** We say that a concrete term $s \in \Lambda_0$ is *terminating* just if $\triangleright$ admits no infinite chain starting from s.

Normal Forms We write the set of normal forms that are also concrete terms (i.e. contains no logical variables) as N_0.

Examples of normal forms include the terms $(\lambda x.x)$ and `[2,3]` (using the Haskell shorthand for lists). Note that we consider a term with free names to be concrete, as long as it has no logical variables within it. For example, a term variable a is a concrete term.

3.1.3 Useful Lemmas

The following lemmas will be helpful in upcoming proofs. We have deferred the proofs of these lemmas to Appendix A.

First, we define a reduction relation $\triangleright_\alpha$ on $A[]$, the one-hole contexts for $alts$. This is listed in Figure 3.2. This relation is used only in the proofs of the following lemmas.

Lemma 3.1.1. *If* $alts_1 \triangleright_\alpha alts_2$ *then* $\mathsf{case}\ s\ \mathsf{of}\ alts_1 \triangleright \mathsf{case}\ s\ \mathsf{of}\ alts_2$

The following lemma formalises that any subterm can make a step at any time.

Lemma 3.1.2. *If* $\ell \triangleright r$ *then* $C[\ell] \triangleright C[r]$ *and if* $\ell \triangleright r$ *then* $A[\ell] \triangleright A[r]$

$$\frac{s \rhd t}{k\,\overline{a} \to s,\ alts \rhd_\alpha k\,\overline{a} \to t,\ alts}\ (\mathsf{Alt1}) \qquad \frac{alts_1 \rhd_\alpha alts_2}{k\,\overline{a} \to s,\ alts_1 \rhd_\alpha k\,\overline{a} \to s,\ alts_2}\ (\mathsf{Alt2})$$

Figure 3.2: Definition of $\rhd_\alpha$

Corollary 3.1.3. *If $\ell \stackrel{*}{=} r$ then $s[\ell] \stackrel{*}{=} s[r]$*

Proof. Suppose $\ell \stackrel{*}{=} r$.

This means there is some term v such that $\ell \rhd^* v$ and $r \rhd^* v$. Need to show there is some term u such that $s[\ell] \rhd^* u$ and $s[r] \rhd^* u$.

By Lemma 3.1.2, $s[\ell] \rhd^* s[v]$ and $s[r] \rhd^* s[v]$, so we have $s[\ell] \stackrel{*}{=} s[r]$, as required. □

3.2 Satisfiability

Before we formalise the decision problem that GenEx aims to solve, we define the types of equations and clauses that GenEx deals with. We also define what it means for some clause to be satisfiable.

Definition 3.2.1 (Formulas)**.** Atomic formulas are (unoriented) *equations* $s \doteq t$ between terms in Λ_Σ, considered as a multiset $\{s, t\}$. Its *subjects* are $\{s, t\}$.

We also refer to atomic formulas as *unit equations*.

Definition 3.2.2 (Clauses)**.** A *clause* $\Pi \Rightarrow \Delta$ is a pair of multisets of equations. Its *subjects* are the union of the subjects of each atomic formula. A *Horn clause* is a clause in which Δ has size at most 1.

For trivial Horn clauses of the form $\top \Rightarrow \ell \doteq r$, we abbreviate this to simply $\ell \doteq r$.

We use $\top \Rightarrow \bot$ to denote the empty clause.

Every clause has some number of logical variables within it. If a clause is satisfiable, it should be possible to substitute some concrete terms for these logical variables such that the clause becomes true.

Firstly, we formalise this idea of substituting concrete terms in for logical variables.

Definition 3.2.3 (Interpretation modulo the theory of a program)**.** An interpretation I consists of an assignment of a concrete normal form $I(x) \in \mathrm{N}_0$ to each logical variable x. An interpretation induces a partial meaning function on terms:

$$\begin{array}{rcll}
[\![a]\!] & = & a & \\
[\![k]\!] & = & k & \\
[\![s\,t]\!] & = & [\![s]\!][\![t]\!] & \\
[\![\lambda a.s]\!] & = & \lambda a.[\![s]\!] & \\
[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] & = & \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]} & \\
[\![x]\!] & = & I(x) & \\
[\![\lambda^{-1}\,v]\!] & = & t[a/b] & \text{if } [\![v]\!] = \lambda b.t \text{ and } a \text{ is not free in } t \\
[\![k_i^{-1}\,v]\!] & = & t_i & \text{if } [\![v]\!] = k\,t_1 \cdots t_n \text{ and } i \in [1, n]
\end{array}$$

By definition, when defined $[\![s]\!] \in \Lambda_0$.

We define $[\![s[]]\!]$ for some one-hole context $s[]$ to be $[\![s]\!]$ with the hole $\Box$ left in its original place.

The use of the symbolic projections adds constraints that must be satisfied for a term to be defined.

We will sometimes specify the interpretation that was used to interpret some term, e.g. $[\![s]\!]_I$.

This allows us to define the situations in which some clause is satisfied.

Definition 3.2.4 (Satisfaction)**.** We define satisfaction as:

- $I \models s \doteq t$ just if $[\![s]\!]$ and $[\![t]\!]$ are defined and $[\![s]\!] \stackrel{*}{=} [\![t]\!]$, and
- $I \models \Pi \Rightarrow \Delta$ just if (i) the meaning of all constituent terms is defined, and (ii) each equation in Δ is satisfied by I whenever each element of Π is.

We say that a set of clauses S is *satisfiable* just if there is an interpretation I that satisfies each of the clauses. In this case, we write $I \models S$.

We now present some examples of satisfaction.

Under the interpretation $I = \{\texttt{x1} \mapsto 1, \texttt{x2} \mapsto 2\}$, we have $I \models \texttt{x1 <= x2} \doteq \texttt{True}$. `1 <= 2` will be symbolically executed to `True`, so `x1 <= x2` and `True` have some common reduct (i.e. `True`). Under the interpretation $I = \{\texttt{x1} \mapsto 3, \texttt{x2} \mapsto 2\}$, we have $I \not\models \texttt{x1 <= x2} \doteq \texttt{True}$. `3 <= 2` will be symbolically executed to `False`, so `x1 <= x2` and `True` have no common reduct.

We have $I \models \texttt{x1 <= x2} \doteq \texttt{False} \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True}$ for any interpretation I. Clearly, if we know `x1 <= x2` is false, then it must be that `x2 <= x1`. This means that for any interpretation I such that $I \models \texttt{x1 <= x2} \doteq \texttt{False}$, we know $I \models \texttt{x2 <= x1} \doteq \texttt{True}$.

3.2.1 Useful Lemmas

We now present another set of lemmas that will be helpful in upcoming proofs. Again, we have deferred the proofs of these lemmas to Appendix A.

Lemma 3.2.1. *If $s \triangleright t$ and $[\![s]\!]$ defined, then $[\![t]\!]$ defined.*

Lemma 3.2.2. *If $s \triangleright t$ then $[\![s]\!] \triangleright [\![t]\!]$*

3.3 Decision Problem

The goal of GenEx is to prove that some set of Horn clauses is unsatisfiable. In this section, we formalise the decision problem that GenEx aims to solve.

In order to use GenEx, the programmer must specify a *goal* equation that they would like to prove. They may also optionally specify some *hypotheses*, which are a set of Horn clauses that they would like the proof system to assume to be true.

The decision problem we are interested in is as follows. Fix a program. Then given the hypotheses C and the unoriented goal equation $s \doteq t$, decide if every interpretation I that satisfies C also satisfies $s \doteq t$. In such cases we write $C \models s \doteq t$.

This is equivalent to checking the unsatisfiability of $C \wedge s \neq t$, which is equivalent to checking the unsatisfiability of the Horn clauses $C \wedge (s \doteq t \Rightarrow \bot)$.

Consider the example shown in Section 1.1.2. In that example, we briefly considered how GenEx would prove the property `filter p [x1, x2, x3]` $\doteq$ `[x1, x3]` with the following assumptions:

$$\texttt{p x1} \doteq \texttt{True} \tag{3.1}$$

$$\texttt{p x2} \doteq \texttt{False} \tag{3.2}$$

$$\texttt{p x3} \doteq \texttt{True} \tag{3.3}$$

In this case, our hypotheses C is the set containing (3.1), (3.2) and (3.3), and our unoriented goal equation is `filter p [x1, x2, x3]` $\doteq$ `[x1, x3]`.

Theorem 3.3.1. *$C \models s \doteq t \iff C \wedge (s \doteq t \Rightarrow \bot)$ unsatisfiable.*

Proof. We prove each direction separately.

($\Rightarrow$) Suppose that for all interpretations I, if $I \models C$ then $I \models s \doteq t$. We need to show $C \wedge (s \doteq t \Rightarrow \bot)$ is unsatisfiable.

We will prove this by contradiction. Suppose $C \wedge (s \doteq t \Rightarrow \bot)$ is satisfiable. This means there exists some interpretation I' such that $I' \models C \wedge (s \doteq t \Rightarrow \bot)$. Hence, we know $I' \models s \doteq t \Rightarrow \bot$. But $I' \models C$, so we also know that $I' \models s \doteq t$. Thus, we can deduce $I \models \bot$, a contradiction.

($\Leftarrow$) Suppose that $C \wedge (s \doteq t \Rightarrow \bot)$ is unsatisfiable. This means that there exists no interpretation I such that $I \models C \wedge (s \doteq t \Rightarrow \bot)$.

Need to show that for all interpretations I, if $I \models C$ then $I \models s \doteq t$.

Let us assume for some interpretation I' that $I' \models C$. Consider the set $C \wedge (s \doteq t \Rightarrow \bot)$ again. This set is unsatisfiable. Since $I' \models C$, this means $I' \not\models s \doteq t \Rightarrow \bot$ as it is the only remaining clause in the set $C \wedge (s \doteq t \Rightarrow \bot)$. Thus, it must be that $I' \models s \doteq t$. □

3.4 Case Splitting

Some equations can only be proved if GenEx is able to case split on the value of some expression. This involves considering the possible values that some expression could take, and verifying if the property holds for each possible value. For example, if GenEx encounters a term of the form `if (x1 <= x2) then ... else ...` and it knows nothing about the values of `x1` and `x2`, a procedure for case splitting would allow it to split on whether `x1 <= x2` is true or false.

One of the contributions of this project is adding a procedure for case splitting to the proof system. If the programmer wishes for the proof system to split on the value of some expression, they must specify the values that the expression can take.

These case splits are specified in the form of disjunctive clauses. For example, one such clause is $\texttt{x} \doteq \texttt{[]} \vee \texttt{x} \doteq \texttt{(y:ys)}$. These disjunctive clauses are added to the set of hypotheses. However, as we mentioned in Section 3.3, GenEx only operates on Horn clauses.

To deal with this, each disjunctive clause results in multiple instances of the decision problem. Within each instance, the hypotheses contain only Horn clauses. The process for converting the disjunctive clauses into multiple instances is as follows:

1. Order the disjunctive clauses D in some arbitrary way. Label the clauses $D_1, \cdots, D_n$.
2. Create a set S containing all of the clauses D. Create the singleton set of sets $\Sigma_0 = \{S\}$.
3. Iterate through the sets D_i from $i = 1$ to n. For each D_i:
   - Iterate through the sets $S_j \in \Sigma_{i-1}$. For each set S_j, create m copies of S_j where m is the number of disjuncts in D_i. In each copy, replace D_i by a different disjunct in D_i.
   - When this process has been completed for all sets $S_j \in \Sigma_{i-1}$, we have a new collection of sets. Call this new collection of sets Σ_i.

Each Σ_i is some collection of sets, where each set contains:

- A single literal from D_j, for $j \in [1, i]$
- The disjunctive clause D_j, for $j \in [i+1, n]$

The set Σ_n contains all possible sets that contain one literal from each clause D_i. Note that this process is equivalent to taking the Cartesian product of all clauses D, if each clause is viewed as a set of literals.

We must show that this process indeed works correctly, i.e. that the original set of clauses is unsatisfiable if and only if all of the instances are unsatisfiable.

Firstly, we prove that the original set of disjunctive clauses is satisfiable if and only if there is some resulting instance which is satisfiable.

Theorem 3.4.1. *$I \models D$ iff $\exists S \in \Sigma_n. I \models S$.*

Proof. We will prove this using a loop invariant. The loop invariant is: at the beginning of iteration i, $I \models D$ iff $\exists S \in \Sigma_{i-1}. I \models S$

Initialisation: Need to show that before the first iteration of the algorithm, the loop invariant holds. Σ_0 contains only one set S, which contains all of the disjunctive clauses. By definition, S is the same as D so the invariant holds trivially.

Maintenance: Suppose the invariant holds at the start of iteration i. Need to show the invariant holds at the start of iteration $i+1$.

Since the invariant holds at the start of iteration i, we know $I \models D$ iff $\exists C \in \Sigma_{i-1}. I \models C$.

At the start of iteration $i+1$, we have just finished converting clause D_i. Need to show $I \models D$ iff $\exists S \in \Sigma_i. I \models S$. We will prove each direction separately.

For the 'only if' direction, suppose $I \models D$. Need to show $\exists S \in \Sigma_i. I \models S$.

We know $\exists C \in \Sigma_{i-1}. I \models C$. During iteration i, m copies of C were created, each replaced by a different disjunct in D_i.

We know $I \models D$ so there is at least one literal d in D_i such that $I \models d$. Then, there will be some set $S \in \Sigma_i$ which is the same as C but with D_i replaced by d. We know $I \models d$ and $I \models s$ for all other clauses $s \in S_j$ because $s \in C_k$. Thus, we know $I \models S$.

For the 'if' direction, suppose $\exists S \in \Sigma_i. I \models S$. Let s_k for $k \in [1, n]$ be the clauses in S. For all $k \leq i$, s_k is some literal from D_k. Since $I \models s_k$, we know $I \models D_k$ as D_k is a disjunctive clause.

For all $k > i$, s_k is D_k itself, so we know $I \models D_k$ trivially.

Termination: When the algorithm terminates, we have Σ_n, which contains a collection of sets, each with one literal from each disjunctive clause. We have that $I \models D$ iff $\exists S \in \Sigma_n . I \models S$, as required. □

This allows us to prove that the original set of disjunctive clauses is unsatisfiable if and only if all the resulting instances are unsatisfiable.

Corollary 3.4.2. *D is unsatisfiable $\iff$ All $S \in \Sigma_n$ are unsatisfiable.*

Proof. We will prove each direction separately.

($\Rightarrow$) Suppose D is unsatisfiable. This means there is no I such that $I \models D$. By Theorem 3.4.1, this means that, for all I, there is no $S \in \Sigma_n$ such that $I \models S$ and so all $S \in \Sigma_n$ are unsatisfiable.

($\Leftarrow$) Suppose all $S \in \Sigma_n$ are unsatisfiable. This means that for all I, there is no $S \in \Sigma_n$ such that $I \models S$. Then, by Theorem 3.4.1, for all I, $I \not\models D$ so D is unsatisfiable. □

We can then add the other hypotheses and the negated goal statement to each set in Σ_n. Now each set represents a different instance of the decision problem. Each instance must be solved separately. By Theorem 3.3.1 and Corollary 3.4.2, the original set of clauses (including the disjunctive clauses) is unsatisfiable if and only if all of the instances are unsatisfiable.

Consider the following Haskell function that sorts a tuple of two integers:

```
sort (x1, x2) = if x1 <= x2 then (x1, x2) else (x2, x1)
```

We also define a corresponding function that checks whether a tuple is sorted:

```
sorted (x1, x2) = x1 <= x2
```

We want to prove that the following property will always be true:

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True}$$

We need to check that that the statement holds regardless of which of `x1` and `x2` is greater. This means we must specify the possible values that `x1 <= x2` could take. Our set of clauses consists of the following:

$$\texttt{x1 <= x2} \doteq \texttt{True} \vee \texttt{x1 <= x2} \doteq \texttt{False} \tag{3.4}$$

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True} \Rightarrow \bot \tag{3.5}$$

As described above, this results in two instances of the decision problem that must be solved. For the first instance, we must show the unsatisfiability of the clauses:

$$\texttt{x1 <= x2} \doteq \texttt{True}$$

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True} \Rightarrow \bot$$

For the second instance, we must show the unsatisfiability of the clauses:

$$\texttt{x1 <= x2} \doteq \texttt{False}$$

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True} \Rightarrow \bot$$

If we are able to prove the unsatisfiability of both instances, this shows that the original set of clauses ((3.4) and (3.5)) is unsatisfiable.

Detailed examples of case splitting are provided in Sections 3.7.3 and 3.7.4.

By requiring case splits to be explicitly stated by the programmer, we aim to avoid the path explosion problem that many symbolic execution engines suffer from [3]. An additional advantage of this method is that it does not require GenEx to know the type of some expression. This means that GenEx should work for both statically and dynamically typed languages.

$$\ell \triangleright r \frac{\Pi,\ \ell \doteq t \Rightarrow \Delta}{\Pi,\ r \doteq t \Rightarrow \Delta}\ (\mathsf{RedL}) \qquad \ell \triangleright r \frac{\ell \doteq t}{r \doteq t}\ (\mathsf{RedR})$$

$$\frac{\Pi,\ s \doteq s \Rightarrow \Delta}{\Pi \Rightarrow \Delta}\ (\mathsf{ResE}) \qquad i \in [1,n] \frac{k\, s_1 \cdots s_n \doteq x}{s_i \doteq k_i^{-1}\, x}\ (\mathsf{EtaK}) \qquad \frac{\lambda a.s \doteq x}{s \doteq \lambda_a^{-1}\, x}\ (\mathsf{EtaB})$$

$$\ell \succ r \frac{\ell \doteq r \quad \Pi, s[\ell] \doteq t \Rightarrow \Delta}{\Pi, s[r] \doteq t \Rightarrow \Delta}\ (\mathsf{SupL}) \qquad \ell \succ r \frac{\ell \doteq r \quad s[\ell] \doteq t}{s[r] \doteq t}\ (\mathsf{SupR})$$

Figure 3.3: Superposition Calculus

3.5 Ordering

In this section, we define a partial order on terms. This partial order will be used in the inference rules described in the next section.

Definition 3.5.1 (Value Patterns)**.** A *value pattern* is a term of the following grammar, where we let s range over N_0:

$$p ::= v \mid k\, p_1 \cdots p_n \mid \lambda a.p \mid s$$

A pattern is called a *proper pattern* if it is not an extended variable.

Note that, when defined, $[\![p]\!] \in \mathrm{N}_0$, because an interpretation maps all logical variables to normal forms.

Definition 3.5.2 (Ordering of terms)**.** We order terms $s \succ t$ just if either of the following is true:

- s is not a pattern and t is a pattern
- s is a extended variable and t is a proper pattern

The latter of these allows for the eta-expansion of extended variables, e.g. $v \succ k\,(k_1^{-1}\, v) \cdots (k_n^{-1}\, v)$ in order to reveal their inner structure.

The terms $(\lambda x.x)$, $(\lambda x.(x\,x))(\lambda y.y)$, $(y : ys)$ and `True` are all examples of value patterns. Examples of terms that are not patterns include $\lambda x.(x\,y)$ and `((head xs) : xs)`.

There are many pairs of terms that cannot be oriented by this ordering. Notably, any two terms that are not patterns are not orientable. Suppose we have two functions `append` and `append'` which are known to be equivalent. `append ys zs` and `append' ys zs` are both patterns, and thus are not orientable. It will become clear in Section 3.6 that this means the proof system cannot rewrite one to the other. We discuss the consequences of this in Section 3.7.2.

3.6 Superposition Calculus

As described in Section 3.3, the programmer specifies some goal clause $s \doteq t$ and some hypotheses C. The proof system is a set of inference rules designed to prove the unsatisfiability of the Horn clauses $C \wedge (s \doteq t \Rightarrow \bot)$. These rules are shown in Figure 3.3.

The proof system begins with a set of known clauses containing the hypotheses and the negated goal clause. The premise of each inference rule is some clause in the set of known clauses. Application of a rule deduces a new clause, which is added to the set of known clauses.

If at some point the empty clause $\top \Rightarrow \bot$ can be deduced, then this indicates unsatisfiability of the original clauses. If the empty clause cannot be deduced, then this process will stop if the set of known clauses is saturated. This means that any new clauses that can be deduced are duplicates of existing clauses.

Definition 3.6.1 (Saturation)**.** Given a set of clauses S, define *the saturation of* S, written $\mathsf{Sat}(S)$, as the least set of clauses closed under the rules of Figure 3.3.

Rules (SupL) and (SupR) are similar to the pure superposition calculus [1]. They deduce a new clause by rewriting some subterm within an existing clause. A subterm ℓ may only be rewritten to a smaller (by Definition 3.5.2) term r, and only if the set of known clauses contains the equation $\ell \doteq r$.

Rules (RedL) and (RedR) symbolically execute some subject of a formula to deduce a new clause.

(RedL) and (SupR) can only be applied when the (main) premise is a non-trivial implication (i.e. the left hand side contains at least one equation). Conversely, (RedR) and (SupR) can only be applied to to trivial Horn clauses of the form $T \Rightarrow \ell \doteq r$.

The (EtaK) rule is used to project out the arguments of a constructor. This is useful for eta-expanding some logical variable to reveal its structure. An example of this is provided in Section 3.7.2. The (EtaB) rule is used to project out the body of some abstraction. There is discussion on the usage of this rule in Section 6.2.2.

If there is a trivially true equation $s \doteq s$ within some clause, this formula can be removed using the (ResE) rule. This is important for deducing the empty clause $\top \Rightarrow \bot$.

3.7 Examples

3.7.1 Usage of (RedL) and (SupL)

Consider the following Haskell function:

```
filter p [] = []
filter p (x:xs) = if p x then x : (filter p xs) else filter p xs
```

Suppose we have the following hypotheses:

$$\texttt{p x1} \doteq \texttt{True} \tag{3.6}$$

$$\texttt{p x2} \doteq \texttt{False} \tag{3.7}$$

$$\texttt{p x3} \doteq \texttt{True} \tag{3.8}$$

We want to prove the following property:

$$\texttt{filter p [x1, x2, x3]} \doteq \texttt{[x1, x3]}$$

We negate our goal clause and aim to obtain a contradiction.

$$\texttt{filter p [x1, x2, x3]} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.9}$$

As described in Section 3.3, our set of known clauses now consists of clauses (3.6), (3.7), (3.8) and (3.9). We begin by using (RedL) on (3.9) to symbolically execute `filter p [x1, x2, x3]` and obtain a new clause.

$$\texttt{if p x1 then x1 : (filter p [x2, x3]) else filter p [x2, x3]} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.10}$$

Now our set of known clauses also includes (3.10). At this point, we cannot continue applying (RedL) without knowing the value of `p x1`. We have the hypothesis (3.6) stating `p x1` $\doteq$ `True`. We have `p x1` $\succ$ `True` because the first is not a pattern whilst the latter is (it is a constructor). This means we can use (SupL) with (3.6) and (3.10) to rewrite the expression `p x1`.

$$\texttt{if True then x1 : (filter p [x2, x3]) else filter p [x2, x3]} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.11}$$

Now we can continue using (RedL) to symbolically execute the left-hand side of the formula in (3.11).

$$\texttt{x1 : filter p [x2, x3]} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.12}$$

We use (RedL) again on (3.12).

$$\texttt{x1 : (if p x2 then x2 : (filter p [x3]) else filter p [x3])} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.13}$$

We can use (SupL) on (3.7) and (3.13) to replace `p x2` in (3.13), and then (RedL) on the resulting clause. We would then follow the same process for `p x3`. We will not list the resulting clauses because the process is essentially repeated application of what has already been shown. Eventually, we derive the following clause:

$$\texttt{[x1, x3]} \doteq \texttt{[x1, x3]} \Rightarrow \bot \tag{3.14}$$

Now we can use (ResE) on (3.14) to derive a contradiction.

$$\top \Rightarrow \bot$$

3.7.2 Usage of (EtaK)

Suppose we have two different functions to `append` two lists. One recurses on the first list and the other the second list. An example of two such functions is:

```
append [] zs = zs
append (y:ys) zs = y : (append ys zs)

append' ys [] = ys
append' ys (z:zs) = append' (append ys [z]) zs
```

We also have a function which checks whether a list is empty:

```
empty [] = True
empty (x:xs) = False
```

We want to prove `empty (append' (y:ys) zs)` $\doteq$ `False`. Suppose we have proven separately that `append` and `append'` are equivalent. We could add the following formula to our hypotheses:

$$\texttt{append (y:ys) zs} \doteq \texttt{append' (y:ys) zs}$$

However, this is would not be helpful for progressing the proof. Both `append (y:ys) zs` and `append' (y:ys) zs` are not patterns. This means that they are not comparable by our definition of $\succ$. The proof system would not be able to directly rewrite one with the other using (SupL) or (SupR).

Instead, we can add the following assumptions, stating that both are equal to the same logical variable `x`:

$$\texttt{append (y:ys) zs} \doteq \texttt{x} \tag{3.15}$$

$$\texttt{append' (y:ys) zs} \doteq \texttt{x} \tag{3.16}$$

Again, we negate our goal clause and aim to obtain a contradiction.

$$\texttt{empty (append' (y:ys) zs)} \doteq \texttt{False} \Rightarrow \bot \tag{3.17}$$

We cannot use (RedL) on (3.17), because the structure of `zs` is not known. Thus, the only option is to use (SupL) with (3.16) to replace `append' (y:ys) zs` in (3.17) with `x`. This is possible because `x` is a pattern whilst `append' (y:ys) zs` is not, so we have `append' (y:ys) zs` $\succ$ `x`. Then, we have the new clause:

$$\texttt{empty x} \doteq \texttt{False} \Rightarrow \bot \tag{3.18}$$

At this point, we cannot apply (RedL) to (3.18) because we know nothing about the structure of `x`. We need to deduce more about the structure of `x`. We can do this by rewriting `x` with its eta-expanded version.

To begin the process of eta-expanding `x`, we first use (RedR) on (3.15),

$$\texttt{y : (append ys zs)} \doteq \texttt{x} \tag{3.19}$$

Then we are able to use (EtaK) on (3.19) to project out the two arguments of `x`

$$\texttt{y} \doteq (\texttt{:})_1^{-1}\,\texttt{x} \tag{3.20}$$

$$\texttt{append ys zs} \doteq (\texttt{:})_2^{-1}\,\texttt{x} \tag{3.21}$$

Now we are able to eta-expand `x` by using (SupR) on (3.21) and (3.19). Note that we are unable to rewrite `y` to $(\texttt{:})_1^{-1}\,\texttt{x}$ because they are both extended variables, and thus not comparable by Definition 3.5.2. However, this does not matter in this situation because we simply need to know whether `x` of the form `[]` or $((\texttt{:})\ldots)$.

$$\texttt{x} \doteq \texttt{y : } ((\texttt{:})_2^{-1}\,\texttt{x}) \tag{3.22}$$

Let us return to (3.18). We now have `x` $\succ$ `y : ` $((\texttt{:})_2^{-1}\,\texttt{x})$, as `x` is an extended variable and `y : ` $((\texttt{:})_2^{-1}\,\texttt{x})$ is a proper pattern. This means we can use (SupL) to rewrite `x`.

$$\texttt{empty (y : } ((\texttt{:})_2^{-1}\,\texttt{x})) \doteq \texttt{False} \Rightarrow \bot \tag{3.23}$$

Now we are able to use (RedL) on (3.23).

$$\texttt{False} \doteq \texttt{False} \Rightarrow \bot \tag{3.24}$$

Finally, we use (ResE) on (3.24) to deduce the empty clause.

$$\top \Rightarrow \bot$$

3.7.3 Example of Case Splitting

Consider the following Haskell function that sorts a tuple of two integers:

```
sort (x1, x2) = if x1 <= x2 then (x1, x2) else (x2, x1)
```

We also define a corresponding function that checks whether a tuple is sorted:

```
sorted (x1, x2) = x1 <= x2
```

We want to prove that the following property will always be true:

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True}$$

We need to check that that the statement holds regardless of which of `x1` and `x2` is greater. This means we must specify the possible values that `x1 <= x2` could take:

$$\texttt{x1 <= x2} \doteq \texttt{True} \vee \texttt{x1 <= x2} \doteq \texttt{False} \tag{3.25}$$

We also need to specify that when `x1 <= x2` is false, `x2 <= x1` is true. The proof system is unable to deduce this by itself. There is further discussion of this limitation in Section 6.2.4.

$$\texttt{x1 <= x2} \doteq \texttt{False} \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True} \tag{3.26}$$

As described in Section 3.4, this results in two instances of the decision problem. Each instance contains a different literal from (3.25) in its hypotheses.

Instance 1

Hypotheses:

$$\texttt{x1 <= x2} \doteq \texttt{True} \tag{3.27}$$

$$\texttt{x1 <= x2} \doteq \texttt{False} \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True} \tag{3.28}$$

As always, we negate our goal clause.

$$\texttt{sorted (sort (x1, x2)} \doteq \texttt{True} \Rightarrow \bot \tag{3.29}$$

We begin by using (RedL) to reduce (3.29).

$$\texttt{sorted (if x1 <= x2 then (x1, x2) else (x2, x1))} \doteq \texttt{True} \Rightarrow \bot \tag{3.30}$$

We have $\texttt{x1 <= x2} \succ \texttt{True}$ as the first is not a pattern whilst the latter is. This means we can use (SupL) on (3.27) to rewrite `x1 <= x2` in (3.30).

$$\texttt{sorted (if True then (x1, x2) else (x2, x1))} \doteq \texttt{True} \Rightarrow \bot$$

Now we are able to use (RedL) twice, each time on the last deduced clause.

$$\texttt{sorted (x1, x2)} \doteq \texttt{True} \Rightarrow \bot$$

$$\texttt{x1 <= x2} \doteq \texttt{True} \Rightarrow \bot \tag{3.31}$$

We use (SupL) with (3.27) again, this time to rewrite `x1 <= x2` in (3.31).

$$\texttt{True} \doteq \texttt{True} \Rightarrow \bot \tag{3.32}$$

Finally, we use (ResE) on (3.32) to deduce the empty clause.

$$\top \Rightarrow \bot$$

Instance 2

Hypotheses:

$$\texttt{x1 <= x2} \doteq \texttt{False} \tag{3.33}$$

$$\texttt{x1 <= x2} \doteq \texttt{False} \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True} \tag{3.34}$$

We negate our goal clause:

$$\texttt{sorted (sort (x1, x2))} \doteq \texttt{True} \Rightarrow \bot \tag{3.35}$$

We will require the equation `x2 <= x1` $\doteq$ `True` later in the proof, so we first deduce this. As described in Definition 3.2.2, this is shorthand for the Horn clause $\top \Rightarrow$ `x2 <= x1` $\doteq$ `True`. We can apply (SupL) to (3.33) and (3.34).

$$\texttt{False} \doteq \texttt{False} \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True} \tag{3.36}$$

Using (ResE) on (3.36), we can deduce our required clause.

$$\top \Rightarrow \texttt{x2 <= x1} \doteq \texttt{True} \tag{3.37}$$

Now we turn to our negated goal clause, (3.35). We use (RedL) to obtain the following clause:

$$\texttt{sorted (if x1 <= x2 then (x1, x2) else (x2, x1))} \doteq \texttt{True} \Rightarrow \bot \tag{3.38}$$

We have `x1 <= x2` $\succ$ `False` so we can use (SupL) with (3.33) to rewrite `x1 <= x2` in (3.38).

$$\texttt{sorted (if False then (x1, x2) else (x2, x1))} \doteq \texttt{True} \Rightarrow \bot$$

For the following two clauses, each is the result of using (RedL) on the last deduced clause.

$$\texttt{sorted (x2, x1)} \doteq \texttt{True} \Rightarrow \bot$$

$$\texttt{if x2 <= x1 then True else False} \doteq \texttt{True} \Rightarrow \bot \tag{3.39}$$

Since we deduced (3.37) earlier, we are able to use (SupL) to rewrite `x2 <= x1` in (3.39).

$$\texttt{True} \doteq \texttt{True} \Rightarrow \bot \tag{3.40}$$

We use (ResE) on (3.40) to deduce the empty clause.

$$\top \Rightarrow \bot \tag{3.41}$$

We have shown that both instances are unsatisfiable. This proves that the original set of clauses ((3.25), (3.26), and the negated goal clause) is unsatisfiable.

3.7.4 Another Example of Case Splitting

Consider the `append` and `empty` functions from Section 3.7.2. Suppose we want to prove the following property:

$$\texttt{empty (append xs [])} \doteq \texttt{empty xs}$$

To check that this holds for all possible values of `xs`, we specify its possible values as a hypothesis:

$$\texttt{xs} \doteq \texttt{[]} \vee \texttt{xs} \doteq \texttt{(y:ys)}$$

Instance 1

Hypotheses:

$$\texttt{xs} \doteq \texttt{[]} \tag{3.42}$$

Negated goal clause:

$$\texttt{empty (append xs [])} \doteq \texttt{empty xs} \Rightarrow \bot \tag{3.43}$$

We begin by using (SupL) with (3.42) to rewrite any occurrences of `xs` within (3.43). First, we rewrite the occurrence on the left hand side of the equation:

$$\texttt{empty (append [] [])} \doteq \texttt{empty xs} \Rightarrow \bot \tag{3.44}$$

Since formulas are unoriented, we can use (SupL) to replace `xs` on the right-hand side of (3.44) (by simply swapping the sides when applying the rule).

$$\texttt{empty (append [] [])} \doteq \texttt{empty []} \Rightarrow \bot \tag{3.45}$$

Now we can use (RedL) to continue symbolically execute `empty (append [] [])` in (3.45).

$$\texttt{empty []} \doteq \texttt{empty []} \Rightarrow \bot \tag{3.46}$$

We deduce the empty clause by using (ResE) on (3.46).

$$\top \Rightarrow \bot$$

Instance 2

Hypotheses:

$$\texttt{xs} \doteq \texttt{(y:ys)} \tag{3.47}$$

Negated goal clause:

$$\texttt{empty (append xs [])} \doteq \texttt{empty xs} \Rightarrow \bot \tag{3.48}$$

As before, we use (SupL) twice with (3.47) to replace occurrences of `xs` on both sides of the equation in (3.48).

$$\texttt{empty (append (y:ys) [])} \doteq \texttt{empty (y:ys)} \Rightarrow \bot$$

For the following three clauses, each is the result of using (RedL) on the last deduced clause.

$$\texttt{empty (y : append ys [])} \doteq \texttt{empty (y:ys)} \Rightarrow \bot$$

$$\texttt{False} \doteq \texttt{empty (y:ys)} \Rightarrow \bot$$

$$\texttt{False} \doteq \texttt{False} \Rightarrow \bot \tag{3.49}$$

Now we are able to use (ResE) on (3.49) to derive a contradiction.

$$\top \Rightarrow \bot \tag{3.50}$$

Chapter 4

Properties of the Proof System

In this chapter, we cover two of the main contributions of this thesis. We will show that the proof system is refutation sound and conditionally terminating.

4.1 Refutation Soundness

So far, we have been assuming that any time the system is able to derive a contradiction, this means the original set of clauses is unsatisfiable. This property is called refutation soundness. In this section, we prove that this property indeed holds for our proof system.

Firstly, we prove the following theorem, which states that if the system is provided with a set of satisfiable clauses, any new clause it deduces is also satisfiable.

Theorem 4.1.1 (Model Preservation)**.** *If* $I \models S$ *then* $I \models \mathsf{Sat}(S)$.

Proof. Membership in $\mathsf{Sat}(S)$ is determined by application of the rules in Figure 3.3 with a starting set of S. Since we are assuming $I \models S$, it suffices to perform induction over the rules in Figure 3.3 to show $I \models \mathsf{Sat}(S)$.

CASE(RedL). Suppose $I \models \Pi,\ \ell \doteq t \Rightarrow \Delta$ and $\ell \triangleright r$. We need to show $I \models \Pi,\ r \doteq t \Rightarrow \Delta$.

Let us assume $I \models \Pi,\ r \doteq t$. We need to derive $I \models \Delta$.

We know $I \models \Pi$ and $I \models r \doteq t$. From the latter, we know $[\![r]\!]$ and $[\![t]\!]$ are defined. We also know $I \models \Pi,\ \ell \doteq t \Rightarrow \Delta$ so, by definition, $[\![\ell]\!]$ defined. $\ell \triangleright r$ so, by Lemma 3.2.2, $[\![\ell]\!] \triangleright [\![r]\!]$.

We also know $[\![r]\!] \overset{*}{=} [\![t]\!]$ so, by Confluence, they have some common reduct v. We have $[\![\ell]\!] \triangleright [\![r]\!] \triangleright^* v$ and $[\![t]\!] \triangleright^* v$ so $[\![\ell]\!] \overset{*}{=} [\![t]\!]$. By definition, we have $I \models \Pi,\ \ell \doteq t$ so we have $I \models \Delta$, as required.

CASE(RedR). Suppose $I \models \ell \doteq t$ and $\ell \triangleright r$. Need to show $I \models r \doteq t$.

We have that $[\![\ell]\!]$ defined and $\ell \triangleright r$ so $[\![r]\!]$ defined by Lemma 3.2.1. By definition, $[\![t]\!]$ defined, and $[\![\ell]\!]$ and $[\![t]\!]$ have a common reduct, v.

$[\![\ell]\!] \triangleright^* v$ and $[\![\ell]\!] \triangleright [\![r]\!]$ (by Lemma 3.2.2) so by Confluence, $[\![r]\!]$ and v have some common reduct w. Now we have $[\![r]\!] \triangleright^* w$ and $[\![t]\!] \triangleright^* v \triangleright^* w$, so $[\![r]\!] \overset{*}{=} [\![t]\!]$ and $I \models r \doteq t$.

CASE(ResE). Suppose $I \models \Pi,\ s \doteq s \Rightarrow \Delta$. Need to show $I \models \Pi \Rightarrow \Delta$.

From $I \models \Pi,\ s \doteq s \Rightarrow \Delta$ we know $[\![s]\!]$ defined.

Let us assume $I \models \Pi$. Now we need to show $I \models \Delta$. We know $s \overset{*}{=} s$ trivially. Thus, we have $I \models s \doteq s$. This gives $I \models \Pi,\ s \doteq s$ so we have $I \models \Delta$, as required.

CASE(EtaK). Suppose $I \models k\, s_1 \cdots s_n \doteq x$. Need to show $I \models s_i \doteq k_i^{-1}\, x$ for some $i \in [1, n]$.

From $I \models k\, s_1 \cdots s_n \doteq x$ we know $k\, [\![s_1]\!] \cdots [\![s_n]\!]$ defined and $[\![x]\!]$ defined. From $k\, [\![s_1]\!] \cdots [\![s_n]\!]$ defined we know $[\![s_i]\!]$ defined for all $i \in [1, n]$.

We know $[\![x]\!] \overset{*}{=} k\, [\![s_1]\!] \cdots [\![s_n]\!]$, so it must be that $x = k\, s'_1 \cdots s'_n$ for some $s'_1, \cdots, s'_n$. Thus, $[\![x]\!] = k\, [\![s'_1]\!] \cdots [\![s'_n]\!]$ so we have $[\![k_i^{-1}\, x]\!] = [\![s'_i]\!]$ defined for all $i \in [1, n]$. For a term of the form $k\, s_1 \cdots s_n$, any reduction may only occur inside some s_i. Therefore, for $k\, [\![s'_1]\!] \cdots [\![s'_n]\!] \overset{*}{=} k\, [\![s_1]\!] \cdots [\![s_n]\!]$ it must be that $[\![s'_i]\!] \overset{*}{=} [\![s_i]\!]$.

Thus, we have $I \models s_i \doteq k_i^{-1}\, x$, as required.

CASE(EtaB). Suppose $I \models \lambda a.s \doteq x$. Need to show $I \models s \doteq \lambda_a^{-1}\, x$.

From $I \models \lambda a.s \doteq x$ we know $\lambda a.[\![s]\!]$ and $[\![x]\!]$ defined.

We know $\lambda a.[\![s]\!] \overset{*}{=} [\![x]\!]$, so it must be that $x = \lambda a.s'$ for some s'. Then $[\![x]\!] = \lambda a.[\![s']\!]$ so $[\![\lambda_a^{-1}\, x]\!] = [\![s']\!]$ defined. For $\lambda a.[\![s]\!] \overset{*}{=} \lambda a.[\![s']\!]$ it must be that $[\![s]\!] \overset{*}{=} [\![s']\!]$. Thus, we have $[\![s]\!] \overset{*}{=} [\![\lambda_a^{-1}\, x]\!]$ and consequently $I \models s \doteq \lambda_a^{-1}\, x$.

CASE(SupL). Suppose $I \models \ell \doteq r$ and $I \models \Pi,\, s[\ell] \doteq t \Rightarrow \Delta$. Need to show $I \models \Pi,\, s[r] \doteq t \Rightarrow \Delta$.

Let us assume $I \models \Pi,\, s[r] \doteq t$. Now we have to derive $I \models \Delta$.

From $I \models \Pi,\, s[r] \doteq t$ we know $[\![s[r]]\!]$ and $[\![t]\!]$ defined. From $I \models \Pi,\, s[\ell] \doteq t \Rightarrow \Delta$ we know $[\![s[\ell]]\!]$ defined. By definition, $[\![s[\ell]]\!] = [\![s]\!][[\![\ell]\!]]$ and $[\![s[r]]\!] = [\![s]\!][[\![r]\!]]$. Since $I \models \ell \doteq r$, we know $[\![\ell]\!] \overset{*}{=} [\![r]\!]$. By Corollary 3.1.3, we have that $[\![s]\!][[\![\ell]\!]]$ and $[\![s]\!][[\![r]\!]]$ have some common reduct u.

From $I \models \Pi,\, s[r] \doteq t$ we know $[\![s[r]]\!]$ and $[\![t]\!]$ have some common reduct v. $[\![s[r]]\!]$ reduces to both v and u so we know they have some common reduct w. Now we have $[\![s[\ell]]\!] \rhd^* u \rhd^* w$ and $[\![t]\!] \rhd^* v \rhd^* w$ so $[\![s[\ell]]\!] \overset{*}{=} [\![t]\!]$ and $I \models s[\ell] \doteq t$.

CASE(SupR). Suppose $I \models \ell \doteq r$ and $I \models s[\ell] \doteq t$. Need to show $I \models s[r] \doteq t$.

From $I \models s[\ell] \doteq t$ we know $[\![s[\ell]]\!]$ and $[\![t]\!]$ defined. From $I \models \ell \doteq r$ we know $[\![r]\!]$ defined. $[\![s[\ell]]\!] = [\![s]\!][[\![\ell]\!]]$ so $[\![s[]]\!]$ defined. Therefore, $[\![s]\!][[\![r]\!]]$ defined (as the hole cannot occur inside an extended variable).

$[\![s[\ell]]\!]$ and $[\![t]\!]$ have some common reduct v. $[\![\ell]\!] \overset{*}{=} [\![r]\!]$ so by Corollary 3.1.3, $[\![s]\!][[\![\ell]\!]]$ and $[\![s]\!][[\![r]\!]]$ have some common reduct u.

$[\![s[\ell]]\!] \rhd^* u$ and $[\![s[\ell]]\!] \rhd^* v$ so by Confluence, u and v have some common reduct w. Now we have $[\![s[r]]\!] \rhd^* u \rhd^* w$ and $[\![t]\!] \rhd^* v \rhd^* w$ so $[\![s[r]]\!] \overset{*}{=} [\![t]\!]$ and $I \models s[r] \doteq t$. □

Now we are able to prove that the proof system is refutation sound.

Corollary 4.1.2 (Refutation Soundness)**.** *If* $\mathsf{Sat}(S)$ *is inconsistent then* S *is unsatisfiable.*

Proof. This statement follows from the contrapositive of Theorem 4.1.1. □

4.2 Termination

In this section, we show that the process of trying to prove any property will terminate, given certain conditions are fulfilled. The conditions are:

- Every term used in the hypotheses and the goal equation is terminating
- The hypotheses provided by the programmer are satisfiable.

4.2.1 Property of $\succ$

We show that if $s \succ t$, s is indeed "larger" than t, in the sense that s can reduce to t.

Lemma 4.2.1. *If* $s \succ t$ *and* $I \models s \doteq t$ *then* $[\![s]\!] \rhd^* [\![t]\!]$.

Proof. Let us assume $s \succ t$ and $I \models s \doteq t$. We have to show $[\![s]\!] \rhd^* [\![t]\!]$.

From $I \models s \doteq t$ we know $[\![s]\!]$ and $[\![t]\!]$ are defined and have some common reduct v.

From $s \succ t$ we obtain two possible cases. Either s is not a pattern and t is a pattern, or s is an extended variable and t is a proper pattern. We proceed by case analysis.

First, suppose s is not a pattern and t is a pattern. We know that $[\![s]\!]$ and $[\![t]\!]$ have some common reduct v. $[\![t]\!]$ has no proper reducts (as noted in Definition 3.5.1). Thus, it must be that v is $[\![t]\!]$ and we have $[\![s]\!] \rhd^* [\![t]\!]$, as required.

Now suppose s is an extended variable and t is a proper pattern. $[\![s]\!]$ and $[\![t]\!]$ have some common reduct. Both $[\![s]\!]$ and $[\![t]\!]$ have no proper reducts, so it must be that $[\![s]\!]$ and $[\![t]\!]$ are the same term. Then, we know $[\![s]\!] \rhd^* [\![t]\!]$ trivially. □

$$t \in T \,\frac{}{t \in \mathsf{Cl}(T)}\ (\mathsf{Base}) \qquad s \triangleright t \,\frac{s \in \mathsf{Cl}(T)}{t \in \mathsf{Cl}(T)}\ (\mathsf{Step})$$

$$i \in [1,n] \,\frac{k\, s_1 \cdots s_n \in \mathsf{Cl}(T)}{s_i \in \mathsf{Cl}(T)}\ (\mathsf{Constr}) \qquad \frac{\lambda a.s \in \mathsf{Cl}(T)}{s \in \mathsf{Cl}(T)}\ (\mathsf{Lam})$$

Figure 4.1: Rules for membership in $\mathsf{Cl}(T)$

4.2.2 Closure

Definition 4.2.1 (Closure)**.** We define the *closure* $\mathsf{Cl}(T) \subseteq \Lambda_0$ of a set of concrete terms $T \subseteq \Lambda_0$ as the least set satisfying the rules shown in Figure 4.1.

We will show that, for any set T where all the elements are terminating, $\mathsf{Cl}(T)$ is finite. To do this, we define the following sets:

$$T^* = \{t \mid \exists s \in T.\, s \triangleright^* t\}$$
$$T^*_< = \{t \mid \exists s \in T^*.\, t \text{ is a subterm of } s\}$$

Firstly, we show that $\mathsf{Cl}(T)$ is a subset of $T^*_<$.

Lemma 4.2.2. $\forall t \in \mathsf{Cl}(T).\, t \in T^*_<$

Proof. By induction on the derivation of $t \in \mathsf{Cl}(T)$

CASE(Base). Suppose $t \in \mathsf{Cl}(T)$ by (Base). Need to show $t \in T^*_<$.
We know $t \in T$. Then, by definition, $t \in T^*$. Thus, $t \in T^*_<$ since t is a subterm of itself.

CASE(Step). Suppose $t \in \mathsf{Cl}(T)$ by (Step). We know there is some s such that $s \in \mathsf{Cl}(T)$ and $s \triangleright t$. Assume, by induction hypothesis, that $s \in T^*_<$. This means there is some term $v \in T^*$ such that s is a subterm of v. There are two possibilities:

- s is v. Then, we have $s \in T^*$. By definition, $t \in T^*$ and so $t \in T^*_<$, as required.
- s is a strict subterm of v. Then we know $v = w[s]$ for some context w. Since $s \triangleright t$, we know $w[s] \triangleright w[t]$ by Lemma 3.1.2. $w[s] \in T^*$ so $w[t] \in T^*$. t is a subterm of $w[t]$ so $t \in T^*_<$, as required.

CASE(Constr). Suppose $t \in \mathsf{Cl}(T)$ by (Constr). We know t is some s_i such that $k\, s_1 \cdots s_n \in \mathsf{Cl}(T)$ and $i \in [1,n]$. Assume, by induction hypothesis, that $k\, s_1 \cdots s_n \in T^*_<$. This means there is some term $v \in T^*$ such that $k\, s_1 \cdots s_n$ is a subterm of v. But s_i is also a subterm of v, so we know $s_i \in T^*_<$.

CASE(Lam). Similar to **CASE(Constr)**. □

Corollary 4.2.3. *If every term in a given set T is terminating, then $\mathsf{Cl}(T)$ is finite.*

Proof. By Lemma 4.2.2, $\mathsf{Cl}(T) \subseteq T^*_<$.
$T^*_<$ must be finite since:

- Every term in T is terminating, so T^* must be finite
- Every term in T^* has a finite number of subterms.

Thus, $\mathsf{Cl}(T)$ is also finite. □

4.2.3 Inverse Image

Given a set of concrete terms $T \subseteq \Lambda_0$, define the inverse image of meaning by:

$$[\![T]\!]^{-1} = \{s \in \Lambda_\Sigma \mid [\![s]\!] \in T\}$$

For some interpretation I, the set $[\![T]\!]^{-1}$ is the set of all terms s with $\mathsf{FV}(s) \subseteq \Sigma$ (where Σ is finite and fixed), such that $[\![s]\!]_I$ is the same as some term in T. We will show that if T is finite, then $[\![T]\!]^{-1}$ is also finite.

In order to prove this, we also define a similar set for a single clause $s \in \Lambda_0$:

$$[\![s]\!]^{-1} = \{t \in \Lambda_\Sigma \mid [\![t]\!] = s\}$$

One may expect that $[\![-]\!]^{-1}$ can be defined simply as:

- $[\![a]\!]^{-1} = \{a\}$
- $[\![k]\!]^{-1} = \{k\}$
- $[\![s\,t]\!]^{-1} = \{u\,v \mid u \in [\![s]\!]^{-1} \wedge v \in [\![t]\!]^{-1}\}$
- $[\![\lambda a.s]\!]^{-1} = \{\lambda a.u \mid u \in [\![s]\!]^{-1}\}$
- $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]^{-1} = \{\mathsf{case}\ u\ \mathsf{of}\ \overline{k\,\overline{a} \to t'} \mid u \in [\![s]\!]^{-1} \wedge \forall i \in [1,n].\, t'_i \in [\![t]\!]^{-1}\}$

However, this does not cover all the possible ways that an interpretation can result in some concrete term s. For example, under the interpretation $I = \{x \mapsto \lambda a.(s_1\,s_2)\}$, we have $[\![\lambda^{-1}\,x]\!] = s_1\,s_2$. This is not covered by the above definition. Therefore, in order to define $[\![s]\!]^{-1}$, we also have to consider all the possible extended variables that could be interpreted to s.

The following lemma and corollary show that, for any term s, the set of extended variables that can be interpreted to s is finite. In fact, they prove the stronger statement that the number of extended variables defined under any interpretation is finite.

Lemma 4.2.4. *$\forall u$. The set $\{v \mid v$ is an extended variable rooted at x and $[\![v]\!][x \mapsto u]$ is defined$\}$ is finite.*

Proof. By induction on the structure of u. In the interests of brevity, let

$$S(u) = \{v \mid v \text{ is an extended variable rooted at } x \text{ and } [\![v]\!][x \mapsto u] \text{ is defined}\}$$

Suppose u is of the form a, where a is some term variable. $[\![k_i^{-1}\,a]\!]$ and $[\![\lambda^{-1}\,a]\!]$ are not defined. Thus, for $[\![v]\!][x \mapsto a]$ to be defined, it must be that v is some $x \in \Sigma$. Σ is finite so there are finite possible v such that $[\![v]\!][x \mapsto a]$ is defined. Thus, $S(a)$ is finite.

Suppose u is of the form k, where k is some constructor. $[\![k_i^{-1}\,k]\!]$ and $[\![\lambda^{-1}\,k]\!]$ are not defined, so again it must be that v is some $x \in \Sigma$. Σ is finite, so $S(k)$ is finite.

Suppose u is of the form $s\,t$. By the induction hypothesis, $S(s)$ and $S(t)$ are finite. This means that s must have a finite number of constructors and abstractions inside it. If this were not the case, $S(s)$ would be infinite as we could just keep projecting out parts of s to get more valid terms. The same argument can be made for t.

$s\,t$ cannot have more constructors and abstractions than the number in s plus the number in t. Thus, it must be that the term $s\,t$ has a finite number of constructors and abstractions in it. This means that $S(s\,t)$ is finite, since there are a finite number of valid projections of $s\,t$ that can be made.

Suppose u is of the form $\lambda a.s$. By induction hypothesis, $S(s)$ is finite. This means that s has a finite number of constructors and abstractions. $\lambda a.s$ has exactly one more abstraction than s, so $\lambda a.s$ must also have a finite number of constructors and abstractions. Thus, $S(\lambda a.s)$ is finite.

Suppose u is of the form $\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}$. $[\![k^{-1}\,u]\!]$ and $[\![\lambda^{-1}\,u]\!]$ are not defined, so it must be that v is some logical variable $x \in \Sigma$. Σ is finite so $S(u)$ is finite. □

Corollary 4.2.5. $\forall I.\, \{v \mid v \text{ is an extended variable rooted at } x \text{ and } [\![v]\!]_I \text{ is defined}\}$ *is finite.*

Proof. Any extended variable v will only have only logical variable x within it. For any interpretation I, x will be mapped to some term t. Any projections in v are unaffected by interpretations. Therefore, the set $\{v \mid v \text{ is rooted at } x \text{ and } [\![v]\!]_I \text{ is defined}\}$ is in fact equivalent to the set $\{v \mid v \text{ is rooted at } x \text{ and } [\![v]\!][x \mapsto t] \text{ is defined}\}$, which is finite by Lemma 4.2.4. □

Now we are able to show that $[\![s]\!]^{-1}$ is finite for any concrete term s.

Lemma 4.2.6. $\forall s \in \Lambda_0.\, [\![s]\!]^{-1}$ *is finite*

Proof. By induction on the structure of s. As a reminder, the set $[\![s]\!]^{-1}$ is:

$$[\![s]\!]^{-1} = \{t \in \Lambda_\Sigma \mid [\![t]\!] = s\}$$

Suppose s is of the form a, where a is some term variable. For $[\![t]\!] = a$, it must be that t is some extended variable v. v must be rooted at some logical variable $x \in \Sigma$. By Corollary 4.2.5, $\{v \mid v \text{ is rooted at } x \text{ and } [\![v]\!]_I \text{ is defined}\}$ is finite for any interpretation I. The set $\{v \mid v \text{ rooted at } x \text{ and } [\![v]\!] = s\}$ is a subset of this set, so it is also finite. There are finitely many $x \in \Sigma$, so there is a finite number of $v \in \Lambda_\Sigma$ such that $[\![v]\!] = s$. Thus, $[\![s]\!]^{-1}$ is finite.

Suppose s is of the form k, where k is some constructor. The argument is similar to that of the case when s is of the form a.

Suppose s is of the form $u_1\, u_2$. There are two possibilities for terms $t \in \Lambda_\Sigma$ such that $[\![t]\!] = u_1\, u_2$:

- t is of the form $t_1\, t_2$ such that $[\![t_1\, t_2]\!] = u_1\, u_2$. Then, it must be that $[\![t_1]\!] = u_1$ and $[\![t_2]\!] = u_2$, i.e. $t_1 \in [\![u_1]\!]^{-1}$ and $t_2 \in [\![u_2]\!]^{-1}$. By induction hypothesis, $[\![u_1]\!]^{-1}$ and $[\![u_2]\!]^{-1}$ are finite.

  The size of the set $\{t_1\, t_2 \mid t_1 \in [\![u_1]\!]^{-1} \wedge t_2 \in [\![u_2]\!]^{-1}\}$ is the size of $[\![u_1]\!]^{-1}$ multiplied by the size of $[\![u_2]\!]^{-1}$, so it is also finite.

- t is some extended variable v. v must be rooted at some logical variable $x \in \Sigma$. By Corollary 4.2.5, $\{v \mid v \text{ is rooted at } x \text{ and } [\![v]\!]_I \text{ is defined}\}$ is finite. The set $\{v \mid v \text{ rooted at } x \text{ and } [\![v]\!] = u_1\, u_2\}$ is a subset of this set, so it is also finite. There are finitely many $x \in \Sigma$, so there is a finite number of $v \in \Lambda_\Sigma$ such that $[\![v]\!] = u_1\, u_2$.

In both cases, the set of possible terms t is finite. Therefore, the set $\{t \in \Lambda_\Sigma \mid [\![t]\!] = s\}$ must be finite.

Suppose s is of the form $\lambda a.u$. Again, there are two possibilities:

- t is of the form $\lambda a.t'$ such that $\lambda a.[\![t']\!] = \lambda a.u$. Then, it must be that $t' \in [\![u]\!]^{-1}$. By induction hypothesis, $[\![u]\!]^{-1}$ is finite.

  The size of the set $\{\lambda a.t' \mid t' \in [\![u]\!]^{-1}\}$ is the same as the size of $[\![u]\!]^{-1}$, so it is also finite.

- t is some extended variable v. Then, the argument follows as before.

Suppose s is of the form $\mathsf{case}\ u\ \mathsf{of}\ \overline{k\, \overline{a} \to w}$. Again, there are two possibilities:

- t is of the form $\mathsf{case}\ u'\ \mathsf{of}\ \overline{k\, \overline{a} \to w'}$ such that $\mathsf{case}\ [\![u']\!]\ \mathsf{of}\ \overline{k\, \overline{a} \to [\![w']\!]} = \mathsf{case}\ u\ \mathsf{of}\ \overline{k\, \overline{a} \to w}$. Then, it must be that $u' \in [\![u]\!]^{-1}$ and $w'_i \in [\![w_i]\!]^{-1}$ for all $i \in [1, n]$. By induction hypothesis, $[\![u]\!]^{-1}$ and $[\![w_i]\!]^{-1}$ finite for all $i \in [1, n]$. The size of $\{\mathsf{case}\ u'\ \mathsf{of}\ \overline{k\, \overline{a} \to w'} \mid u' \in [\![u]\!]^{-1} \wedge \forall i \in [1, n].\, w'_i \in [\![w_i]\!]^{-1}\}$ is equal to the size of $[\![u]\!]^{-1}$ multiplied by the size of $[\![w_i]\!]^{-1}$ for each $i \in [1, n]$. All of these sets are finite, so the size of $\{\mathsf{case}\ u'\ \mathsf{of}\ \overline{k\, \overline{a} \to w'} \mid u' \in [\![u]\!]^{-1} \wedge \forall i \in [1, n].\, w'_i \in [\![w_i]\!]^{-1}\}$ is also finite.

- t is some extended variable v. Then, the argument follows as before.

□

Using the above lemma, we can prove that for any finite set $T \subseteq \Lambda_0$, $[\![T]\!]^{-1}$ is finite.

Lemma 4.2.7. *If $T \subseteq \Lambda_0$ is finite, then $[\![T]\!]^{-1}$ is finite.*

Proof. Every term in T is in Λ_0. Then, by Lemma 4.2.6, $[\![s]\!]^{-1}$ is finite for all $s \in T$. As there are finitely many terms in T, $[\![T]\!]^{-1}$ is finite. □

4.2.4 Termination Theorem

We are now finally able to prove the main theorem of this section. In the rest of this section, we assume that the programmer has provided the proof system with some (possibly empty) set of hypotheses C and a goal equation $s \doteq t$, so that the initial set of clauses S is $C \wedge (s \doteq t \Rightarrow \bot)$.

Lemma 4.2.8. *Let S be a set of clauses with subjects T. Suppose $I \models C$, where C is the set of hypotheses. Every term in the subjects of $\mathsf{Sat}(S)$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.*

Proof. Let Σ be the set of logical variables present in T. $\mathsf{Cl}([\![T]\!])$ contains the set $[\![T]\!]$ by definition. Since $\mathsf{Cl}([\![T]\!])$ contains $[\![T]\!]$, the set $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1} = \{s \in \Lambda_\Sigma \mid [\![s]\!] \in \mathsf{Cl}([\![T]\!])\}$ contains the set $\{s \in \Lambda_\Sigma \mid [\![s]\!] \in [\![T]\!]\}$. T is clearly contained in this set, so every term in the subjects of S belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

Then, to show that every term in the subjects of $\mathsf{Sat}(S)$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, it suffices to perform induction over the rules shown in Figure 3.3.

CASE(RedL). Suppose every term in the subjects of Π, $\ell \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. This means $[\![v]\!] \in \mathsf{Cl}([\![T]\!])$ for all terms v in the subjects of Π, $\ell \doteq t \Rightarrow \Delta$. Need to show every term in the subjects of Π, $r \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

We know $\ell \triangleright r$. $[\![\ell]\!] \in \mathsf{Cl}([\![T]\!])$ and $[\![\ell]\!] \triangleright [\![r]\!]$ by Lemma 3.2.2. Thus, we have $[\![r]\!] \in \mathsf{Cl}([\![T]\!])$ by (Step). By definition, $r \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Every other term in the subjects of Π, $r \doteq t \Rightarrow \Delta$ is in also in the subjects of Π, $\ell \doteq t \Rightarrow \Delta$. Thus, every term in the subjects of Π, $r \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, as required.

CASE(RedR). Similar to **CASE(RedL)**.

CASE(ResE). Suppose every term in the subjects of Π, $s \doteq s \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. This means $[\![v]\!] \in \mathsf{Cl}([\![T]\!])$ for all terms v in the subjects of Π, $s \doteq s \Rightarrow \Delta$. Need to show every term in the subjects of $\Pi \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

Every term in the subjects of $\Pi \Rightarrow \Delta$ is also in the subjects of Π, $s \doteq s \Rightarrow \Delta$. Thus, all terms in the subjects of $\Pi \Rightarrow \Delta$ belong to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

CASE(EtaK). Suppose every term in the subjects of $k\, s_1 \cdots s_n \doteq x$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Need to show every term in the subjects of $s_i \doteq k_i^{-1}\, x$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, for some $i \in [1, n]$.

We know $k\, s_1 \cdots s_n \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, so we have $k\, [\![s_1]\!] \cdots [\![s_n]\!] \in \mathsf{Cl}([\![T]\!])$. $[\![s_i]\!] \in \mathsf{Cl}([\![T]\!])$ by (Constr). Then, we know $s_i \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

We know $I \models C$. By Theorem 4.1.1 we have that $I \models \mathsf{Sat}(C)$. We know $k\, s_1 \cdots s_n \doteq x \in \mathsf{Sat}(C)$ because the negated goal clause is not a unit equation and so could not have been used as a premise for deducing a unit equation. Then, we know $I \models k\, s_1 \cdots s_n \doteq x$.

From this, we obtain $k\, [\![s_1]\!] \cdots [\![s_n]\!] \overset{*}{=} [\![x]\!]$. It must be that $x = k\, s_1' \cdots s_n'$ for some $s_1, \cdots, s_n$. We know $x \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, so $[\![x]\!] \in \mathsf{Cl}([\![T]\!])$. Then we have $[\![s_i']\!] \in \mathsf{Cl}([\![T]\!])$ so $s_i \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. $k_i^{-1}\, x = s_i'$ so we have $k_i^{-1}\, x \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

Thus, every term in the subjects of $s_i \doteq k_i^{-1}\, x$ belongs to $[\![\mathsf{Cl}(T)]\!]^{-1}$.

CASE(EtaB). Suppose every term in the subjects of $\lambda a.s \doteq x$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Need to show every term in the subjects of $s \doteq \lambda_a^{-1}\, x$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

We know $\lambda a.s \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ so we have $\lambda a.[\![s]\!] \in \mathsf{Cl}([\![T]\!])$. We have $[\![s]\!] \in \mathsf{Cl}([\![T]\!])$ by (Lam). Then, by definition, we have $s \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

We know $I \models \mathsf{Sat}(C)$ and $s \doteq \lambda_a^{-1}\, x \in \mathsf{Sat}(C)$ (by a similar argument to the (EtaK) case) so we have $I \models s \doteq \lambda_a^{-1}\, x$. Then, we know $[\![\lambda_a^{-1}\, x]\!]$ defined. It must be that $x = \lambda a.s'$ and $\lambda_a^{-1}\, x = s'$. We know $x \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ so $[\![x]\!] \in \mathsf{Cl}([\![T]\!])$. Then, $[\![s']\!] \in \mathsf{Cl}([\![T]\!])$ by (Lam), so $s' \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Thus, we have $\lambda_a^{-1} \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, so every term in the subjects of $s \doteq \lambda_a^{-1}\, x$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

CASE(SupL). Suppose every term in the subjects of $\ell \doteq r$ and Π, $s[\ell] \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. This means $[\![\ell]\!], [\![r]\!] \in \mathsf{Cl}([\![T]\!])$ and $[\![v]\!] \in \mathsf{Cl}([\![T]\!])$ for all v in the subjects of Π, $s[\ell] \doteq t \Rightarrow \Delta$. Need to show every term in the subjects of Π, $s[r] \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}(T)]\!]^{-1}$.

$\ell \doteq r$ and $\ell \succ r$ so $[\![\ell]\!] \triangleright^* [\![r]\!]$ by Lemma 4.2.1. $[\![\ell]\!] \in \mathsf{Cl}([\![T]\!])$ so $[\![r]\!] \in \mathsf{Cl}([\![T]\!])$ by (Step). Then $r \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Every other term in the subjects of Π, $s[r] \doteq t \Rightarrow \Delta$ is in also in the subjects of Π, $s[\ell] \doteq t \Rightarrow \Delta$. Thus, every term in the subjects of Π, $s[r] \doteq t \Rightarrow \Delta$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, as required.

CASE(SupR). Similar to SupL. □

We extend the definition of the meaning function to a set T by stipulating that $[\![T]\!]$ is defined and equal to $\{[\![t]\!] \mid t \in T\}$ just if each element of the latter is defined.

Theorem 4.2.9. *Let S be a set of clauses with subjects T. Suppose $I \models C$, where C is the set of hypotheses, and every term in $[\![T]\!]$ is terminating. Then $\mathsf{Sat}(S)$ is finite.*

Proof. By Corollary 4.2.3, $\mathsf{Cl}([\![T]\!])$ is finite. Then, by Lemma 4.2.7, $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ is finite. By Lemma 4.2.8, every term in the subjects of $\mathsf{Sat}(S)$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, so $\mathsf{Sat}(S)$ contains a finite number of distinct terms. From a finite number of distinct terms, a finite number of unique atomic formulas can be created. From a finite number of atomic formulas, a finite number of unique Horn clauses can be created. By definition, $\mathsf{Sat}(S)$ has no duplicated terms so it will be a subset of all these possible Horn clauses. Thus, $\mathsf{Sat}(S)$ must be finite. □

Recall that the process of trying to prove some property will terminate if the set of known clauses is saturated, i.e. the set of known clauses is $\mathsf{Sat}(S)$. Theorem 4.2.9 tells us that $\mathsf{Sat}(S)$ is finite. This means that the proof process is guaranteed to terminate, regardless of whether S is satisfiable or not, provided the aforementioned conditions are fulfilled.

Chapter 5

Extending the Calculus

5.1 The (Convert) Rule

By our partial order $\succ$, any two terms which are both not patterns cannot be oriented. As we showed in Section 3.7.2, this means that hypotheses of the form $\ell \doteq r$, where ℓ and r are both not patterns, cannot be used by the proof system to rewrite ℓ to r or vice versa. Instead, we must split this into two hypotheses:

$$\ell \doteq x$$

$$r \doteq x$$

where x is some logical variable.

We would first need to rewrite, for example, ℓ to x. Then, we are able to eta-expand x and rewrite x to its eta-expanded version.

This process of converting one hypothesis into two hypotheses may become tedious if there are many such equations that must be converted. In addition, this requirement is not intuitive for a programmer wanting to use GenEx. The system would easier to use if the programmer could write the hypothesis $\ell \doteq r$, with the system then converting it to the two required hypotheses behind the scenes.

For this, we would need to add a new rule to our inference rules. This new rule, (Convert), is shown in Figure 5.1. Note that we add the subscript ℓ, r to the logical variable to ensure that the variable used is fresh.

Of course, we are interested in the soundness and termination properties of the extended proof system. We will have to make some changes to our existing results.

To illustrate why these changes have to be made, we consider what happens when we try to prove refutation soundness for this new rule. As a reminder, in our proof of refutation soundness we first proved the following theorem:

Theorem 4.1.1 (Model Preservation)**.** *If* $I \models S$ *then* $I \models \mathsf{Sat}(S)$.

Suppose $I \models \ell \doteq r$ and ℓ, r are not patterns. We need to show $I \models \ell \doteq x_{\ell,r}$ and $I \models r \doteq x_{\ell,r}$. This means we must show $[\![x_{\ell,r}]\!]$ defined and $[\![\ell]\!] \overset{*}{=} [\![x_{\ell,r}]\!]$, $[\![r]\!] \overset{*}{=} [\![x_{\ell,r}]\!]$. Unfortunately, our assumptions tell us nothing about $[\![x_{\ell,r}]\!]$, so we are unable to prove our goals. We require an extended interpretation to deal with the fresh variable that we have introduced.

This extended interpretation must map each $x_{\ell,r}$ to some concrete term. There is no obvious "best" choice for what this mapping should be. One possibility would be to map $x_{\ell,r}$ to $[\![\ell]\!]$. However, this has the potentially undesirable side effect that interpretations are no longer guaranteed to map logical variables to normal forms. This means that value patterns (Definition 3.5.1) are no longer guaranteed to be interpreted to normal forms. The consequence of this is that Lemma 4.2.1 would no longer hold.

$$x_{\ell,r} \text{ fresh} \quad \frac{\ell \doteq r \quad \ell, r \text{ not patterns}}{\ell \doteq x_{\ell,r}, r \doteq x_{\ell,r}} \ (\mathsf{Convert})$$

Figure 5.1: Additional (Convert) rule

Another option is to map every $x_{\ell,r}$ to the normal form of $[\![\ell]\!]$, if it exists. We use the notation $[\![\ell]\!]_N$ to denote the normal form of $[\![\ell]\!]$. All other types of terms are interpreted in the same way as before. The benefit of this option is that interpretations still map each logical variable to a normal form, so Lemma 4.2.1 still holds. Therefore, we will be using this option for the remainder of this chapter. To summarise, we define an extended interpretation in the following way:

Definition 5.1.1 (Extended Interpretation)**.** An extended interpretation is an interpretation (as described in Definition 3.2.3) that also maps each $x_{\ell,r}$ to $[\![\ell]\!]_N$. An extended interpretation induces a partial meaning function on terms:

$$
\begin{array}{rcll}
[\![a]\!] & = & a & \\
[\![k]\!] & = & k & \\
[\![s\,t]\!] & = & [\![s]\!][\![t]\!] & \\
[\![\lambda a.s]\!] & = & \lambda a.[\![s]\!] & \\
[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] & = & \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]} & \\
[\![x]\!] & = & I(x) & \\
[\![x_{\ell,r}]\!] & = & [\![\ell]\!]_N & \text{if } [\![\ell]\!] \text{ has a normal form} \\
[\![\lambda^{-1}\,v]\!] & = & t[a/b] & \text{if } [\![v]\!] = \lambda b.t \text{ and } a \text{ is not free in } t \\
[\![k_i^{-1}\,v]\!] & = & t_i & \text{if } [\![v]\!] = k\,t_1 \cdots t_n \text{ and } i \in [1,n]
\end{array}
$$

5.2 Refutation Soundness

Even with our extended interpretation, we still encounter a problem when we try to prove Theorem 4.1.1 for the (Convert) rule. Knowing $I \models \ell \doteq r$ tells us that $[\![\ell]\!]$ and $[\![r]\!]$ are defined, but nothing about whether they terminate. $[\![x_{\ell,r}]\!]$ exists only if $[\![\ell]\!]$ (and by consequence, $[\![r]\!]$) is terminating.

Intuitively, a logical variable represents some normal form. Clearly, it would be unsound to show $I \models \ell \doteq x_{\ell,r}$ if $[\![\ell]\!]$ does not have a normal form.

This means we must add a condition to this theorem. Let T be the subjects of S. The refutation soundness result is now conditional on all terms in $[\![T]\!]$ terminating. This is a weaker result than the one for the original proof system. However, this condition is in line with the design of the system; note that the termination result for the original proof system has the same condition. As we noted in Chapter 1, we believe that this is a reasonable condition, since it is very rare for Haskell programmers to write non-terminating functions [16].

To summarise, our new soundness result is as follows:

Theorem 5.2.1. *Let S be a set of clauses with subjects T. If $I \models S$ and every term in $[\![T]\!]$ is terminating, then $I \models \mathsf{Sat}(S)$.*

Proof. The cases we showed previously in Section 4.1 still hold, as the other rules do not introduce any new logical variables. This means we simply have to show that the theorem holds for the new rule (Convert).

CASE(Convert). Suppose $I \models \ell \doteq r$, where ℓ and r are both not patterns. Need to show $I \models \ell \doteq x_{\ell,r}$ and $I \models r \doteq x_{\ell,r}$.

From $I \models \ell \doteq r$ we know $[\![\ell]\!]$, $[\![r]\!]$ defined and $[\![\ell]\!] \stackrel{*}{=} [\![r]\!]$. $[\![\ell]\!]$ is terminating so by definition, $[\![x_{\ell,r}]\!] = [\![\ell]\!]_N$. $[\![\ell]\!] \triangleright^* [\![\ell]\!]_N$ so, by Lemma 3.2.1, $[\![\ell]\!]_N$ is defined.

From $[\![\ell]\!] \triangleright^* [\![\ell]\!]_N$, we know $[\![\ell]\!] \stackrel{*}{=} [\![x_{\ell,r}]\!]$, so we have $I \models \ell \doteq x_{\ell,r}$.

We know $[\![\ell]\!]$ and $[\![r]\!]$ have some common reduct v. We have $[\![\ell]\!] \triangleright^* v$ and $[\![\ell]\!] \triangleright^* [\![\ell]\!]_N$ so, by Confluence, v and $[\![\ell]\!]_N$ have some common reduct. But $[\![\ell]\!]_N$ has no proper reducts so it must be that $v \triangleright^* [\![\ell]\!]_N$. Then we have $[\![r]\!] \triangleright^* v \triangleright^* [\![\ell]\!]_N$ so $[\![r]\!] \stackrel{*}{=} [\![x_{\ell,r}]\!]$. Thus, we know $I \models r \doteq x_{\ell,r}$ □

Corollary 5.2.2 (Refutation Soundness)**.** *Let S be a set of clauses with subjects T. Suppose every term in $[\![T]\!]$ is terminating. If $\mathsf{Sat}(S)$ is inconsistent then S is unsatisfiable.*

Proof. This follows from the contrapositive of Theorem 5.2.1. □

5.3 Termination

Since we interpret each $x_{\ell,r}$ to $[\![\ell]\!]_N$, a normal form, value patterns are still guaranteed to be interpreted to normal forms. This means that Lemma 4.2.1 still holds. Lemma 4.2.2, and consequently Corollary 4.2.3, still hold as they do not rely on any interpretation function.

Consider the following lemma, which is used in the proof of termination.

Lemma 4.2.8. *Let S be a set of clauses with subjects T. Suppose $I \models C$, where C is the set of hypotheses. Every term in the subjects of $\mathsf{Sat}(S)$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.*

Proof. All of the previous cases shown in Section 4.2 still hold. We have to show that the lemma holds for (Convert).

CASE(Convert). Suppose every term in the subjects of $\ell \doteq r$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. Need to show every term in the subjects of $\ell \doteq x_{\ell,r}$ and $r \doteq x_{\ell,r}$ belongs to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$.

By definition, $[\![x_{\ell,r}]\!] = [\![\ell]\!]_N$. We know $\ell \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ so $[\![\ell]\!] \in \mathsf{Cl}([\![T]\!])$. Then, we know $[\![\ell]\!]_N \in \mathsf{Cl}([\![T]\!])$ by (Step). We have $[\![x_{\ell,r}]\!] \in \mathsf{Cl}([\![T]\!])$ so $x_{\ell,r} \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. We already know ℓ and r belong to $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. □

The case for (Convert) is subtly different from all the previous cases. Note that the definition of $[\![T]\!]^{-1}$ is:

$$[\![T]\!]^{-1} = \{s \in \Lambda_\Sigma \mid [\![s]\!] \in T\}$$

For all of the other rules, we were able to assume that Σ is fixed throughout the proof, since none of them introduce a fresh logical variable. However, application of the (Convert) rule introduces a new logical variable to the system.

When we note that $\ell \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$, this $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ is some set $\{s \in \Lambda_\Sigma \mid [\![s]\!] \in T\}$. We are able to deduce that $x_{\ell,r} \in [\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$. However, $x_{\ell,r}$ is a fresh variable that was introduced by the application of (Convert), so it is not in Σ. Therefore, the set $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ has become $\{s \in \Lambda_{\Sigma'} \mid [\![s]\!] \in T\}$, where $\Sigma' = \Sigma \cup \{x_{\ell,r}\}$. If the (Convert) rule can be applied infinitely, this set of logical variables Σ could become infinite. This means that the set $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ would become infinitely large.

To illustrate the consequences of this, let us consider the termination theorem:

Theorem 4.2.9. *Let S be a set of clauses with subjects T. Suppose $I \models C$, where C is the set of hypotheses, and every term in $[\![T]\!]$ is terminating. Then $\mathsf{Sat}(S)$ is finite.*

In the proof for this theorem, we noted that $\mathsf{Cl}([\![T]\!])$ is finite by Corollary 4.2.3. We then used Lemma 4.2.7 to deduce that $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ is finite. The proof of Lemma 4.2.7 relies on the set of logical variables Σ being finite. As we have described, this may no longer be the case. $\mathsf{Cl}([\![T]\!])$ will still be finite, since every $x_{\ell,r}$ is interpreted to a term that is already in $\mathsf{Cl}([\![T]\!])$. However, the set of logical variables used in $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ may become infinite, thus causing $[\![\mathsf{Cl}([\![T]\!])]\!]^{-1}$ to become infinitely large. Then, we are no longer guaranteed by Lemma 4.2.8 that there are finitely many terms in the subjects of $\mathsf{Sat}(S)$. This means that $\mathsf{Sat}(S)$ may no longer be finite, and the proof process may not terminate.

It is currently not clear if it is indeed possible to apply the (Convert) rule infinitely, or if the ordering $\succ$ would prevent this. If there is some limit to the number of times that (Convert) can be applied in the saturation process, then our previous termination proof will still hold. Otherwise, our current proof will need to be modified. There may be some modification or weakening that can be made to the termination theorem. This is an area that future research could be dedicated to.

Chapter 6

Conclusion

6.1 Summary

We began in Chapter 3 by introducing the syntax and semantics of the GFL language. We formalised the decision problem that GenEx aims to solve.

We then presented one of the main contributions of this thesis: extending GenEx with a procedure for case splitting on the value of expressions. This procedure involves creating multiple instances of the decision problem for each case split. We showed that proving unsatisfiability of all instances amounts to proving unsatisfiability of the original set of clauses.

We concluded the chapter by presenting the proof system that is used in GenEx. We showed the inference rules used in the system, and some examples of properties that GenEx can prove.

In Chapter 4, we covered two of the main properties of the proof system. The first property is that the proof system is refutation sound. This means that if it is able to deduce a contradiction, it must be that the original set of clauses is unsatisfiable. This is important for guaranteeing that any property GenEx reports to be true is indeed true.

The second property is that the proof system terminates, provided that any terms used in the hypotheses and goal equation are terminating and the hypotheses are satisfiable. This is a particular novelty of GenEx, as this property is not true of the pure superposition calculus [9].

Finally, in Chapter 5 we presented an additional rule, (Convert), that could be added to the set of inference rules. This rule would improve the ease of using GenEx.

We considered the soundness and termination properties of the extended proof system. Refutation soundness still holds for the extended proof system, but with an additional condition that any terms used in the hypotheses and goal statement must be terminating. We noted that, although this is a weaker result than for the original proof system, this condition is in line with the design of the system. In addition, we showed that our current termination result may not hold for the extended proof system. This is because it may be possible to apply the (Convert) rule an infinite number of times throughout the proof process. We concluded that there is further research required to determine if the extended proof system has the same conditional termination property as the original proof system.

6.2 Future Work

In this section, we discuss some other possible directions for future research into GenEx.

6.2.1 Induction

A potential area of future research is adding induction to the proof system. Many interesting properties can only be formally proven with the addition of inductive reasoning.

Consider the following property, which GenEx is able to prove with a combination of (SupL) and (RedL):

$$\texttt{map id [x1, x2, x3]} \doteq \texttt{[x1, x2, x3]}$$

However, even with the addition of case splitting, GenEx cannot prove the stronger property where the structure of the input list is unknown:

$$\texttt{map id xs} \doteq \texttt{xs}$$

Suppose we attempt to prove this property. We would state the case split hypothesis:

$$\texttt{xs} \doteq \texttt{[]} \vee \texttt{xs} \doteq \texttt{(y:ys)}$$

We negate the goal clause:

$$\texttt{map id xs} \doteq \texttt{xs} \Rightarrow \bot \tag{6.1}$$

In the first instance, where $\texttt{xs} \doteq \texttt{[]}$, a contradiction can be derived. However, we will encounter a problem in the instance where $\texttt{xs} \doteq \texttt{(y:ys)}$.

We begin by using (SupL) with $\texttt{xs} \doteq \texttt{(y:ys)}$ to replace `xs` in (6.1).

$$\texttt{map id (y:ys)} \doteq \texttt{(y:ys)} \Rightarrow \bot \tag{6.2}$$

Then, we use (RedL) to symbolically execute `map id (y:ys)` in (6.2).

$$\texttt{id y : map id ys} \doteq \texttt{(y:ys)} \Rightarrow \bot \tag{6.3}$$

Using (RedL) again to reduce `id y` in (6.3), we derive the following clause:

$$\texttt{y : map id ys} \doteq \texttt{(y:ys)} \Rightarrow \bot$$

Now there are no further rules we can apply. We cannot use (RedL) to symbolically execute `map id ys` because we do not know the structure of `ys`. Of course, we could add another hypothesis:

$$\texttt{ys} \doteq \texttt{[]} \vee \texttt{ys} \doteq \texttt{(z:zs)}$$

This would allow us to rewrite `ys` using (SupL). However, we would eventually reach a situation where we need to know the structure of `zs`. This process would repeat infinitely. Therefore, with the tools currently available to us, we cannot prove this property.

However, we are guaranteed that this `ys` is smaller than `xs` (in length, rather than by our ordering $\succ$), because `xs` is `(y:ys)`. We would need some induction principle to leverage this fact to prove the property.

There are various existing induction principles that could be applied to our proof system. One such approach is cyclic proofs. As we discussed in Chapter 2, cyclic proofs generate a proof tree and try to find cycles within it [6, 11]. These cycles must be checked to ensure that they are sound (i.e. some decrease happens within the cycle). This check can be very expensive.

The side condition for the (SupL) and (SupR) rules means that these rules will only rewrite some subterm to a smaller (by Definition 3.5.2) subterm. In addition, the (RedL) and (RedR) rules rewrite some subterm ℓ to r only if $\ell \triangleright r$. This guarantees that r is smaller than ℓ, in the sense that is closer to termination.

Therefore, combining GenEx and cyclic proofs may eliminate the requirement to check cycles. Instead, we would only apply an inductive assumption if some rewrite rule has been applied to the goal, as this provides a guarantee that the goal has decreased (made a step towards termination).

This approach may be more restrictive than existing cyclic proof systems, as we would only allow the induction hypothesis to be used after some rewrite has occurred on the goal. However, it has the potential to be much more efficient, if it can skip the potentially expensive soundness check.

6.2.2 Possible Removal of (EtaB) Rule

Currently, it is not clear whether the (EtaB) rule is ever useful in the process of proving some property.

Consider a situation where we have an equation of the form $\lambda a.s \doteq x$. If s is not a pattern, then $\lambda a.s$ is not a pattern and so we have $\lambda a.s \succ x$ by Definition 3.5.2. This means we cannot rewrite x to $\lambda a.s$ using (SupL) or (SupR).

Using (EtaK), we deduce the equation $s \doteq \lambda_a^{-1}\, x$. We have $s \succ \lambda_a^{-1}\, x$ so we can use (SupR) on $s \doteq \lambda_a^{-1}\, x$ and $\lambda a.s \doteq x$ to deduce the equation $\lambda a.(\lambda_a^{-1}\, x) \doteq x$. Now we have $x \succ \lambda a.(\lambda_a^{-1}\, x)$ so we could rewrite x to $\lambda a.(\lambda_a^{-1}\, x)$ in some other clause. However, it is not clear if there is any situation in which this would be helpful for progressing the proof. By rewriting x to $\lambda a.(\lambda_a^{-1}\, x)$, we are only revealing that x is some function; we have no information about the body of the function.

To illustrate this more concretely, suppose we want to prove the following property:

$$\texttt{filter p [x1, x2, x3]} \doteq \texttt{[x1, x2, x3]}$$

We provide the following hypotheses:

$$\texttt{p} \doteq \texttt{\textbackslash x -> (x < y)} \tag{6.4}$$

$$\texttt{x1 < y} \doteq \texttt{True} \tag{6.5}$$

$$\texttt{x2 < y} \doteq \texttt{True} \tag{6.6}$$

$$\texttt{x3 < y} \doteq \texttt{True} \tag{6.7}$$

We negate the goal clause:

$$\texttt{filter p [x1, x2, x3]} \doteq \texttt{[x1, x2, x3]} \Rightarrow \bot \tag{6.8}$$

We will omit the subscript for λ^{-1} as we will only deal with one abstraction. `\x -> (x < y)` is not a pattern as `x < y` is not in N_0. This means we have `(x < y)` $\succ$ `p`. Applying (EtaB) to (6.4), we get the following clause:

$$\texttt{x < y} \doteq \lambda^{-1}\,\texttt{p} \tag{6.9}$$

`(x < y)` $\succ \lambda^{-1}$ `p` so we can use (SupR) with (6.9) to rewrite `x < y` in (6.4).

$$\texttt{p} \doteq \texttt{\textbackslash x.(}\lambda^{-1}\,\texttt{p)} \tag{6.10}$$

Now we can use (SupL) with (6.10) to rewrite `p` to `\x.(`λ^{-1} `p)` in (6.8).

$$\texttt{filter (\textbackslash x.(}\lambda^{-1}\,\texttt{p)) [x1, x2, x3]} \doteq \texttt{[x1, x2, x3]} \Rightarrow \bot \tag{6.11}$$

However, this rewrite will not allow us to prove this property. We do not know what λ^{-1} `p` is, and we cannot deduce anything more about it from our known clauses. We will never reach a point where we can apply the assumptions (6.5), (6.6) and (6.7).

It is possible that the (EtaB) rule can simply be removed. However, more research is required to determine if there are any situations in which (EtaB) is required to progress a proof.

6.2.3 Redundancy

So far, we have been discussing GenEx as a theoretical system. This has allowed us to ignore issues of efficiency. However, in an implementation of GenEx, it would be important to optimise the proof system to ensure that it is lightweight.

One way to improve the efficiency would be to remove any redundant clauses as the saturation process progresses. Note that in the example shown in Section 3.7.1, each newly deduced clause essentially replaces the clause that was used as the main premise in its deduction. In this example, we never have to refer to a clause after it has been used as a main premise, so those clauses could have been deleted.

However, there may be situations in which a previously used clause will be required again later in the proof process. Therefore, there is further research required to determine the specific circumstances in which an application of (RedL), (RedR), (SupL) or (SupR) results in a new clause that can replace the main premise that was used.

6.2.4 Deducing Related Equations

In Section 3.7.3 we noted that GenEx is unable to deduce the equation `x2 <= x1` $\doteq$ `True` from the equation `x1 <= x2` $\doteq$ `False`, unless the hypothesis `x1 <= x2` $\doteq$ `False` $\Rightarrow$ `x2 <= x1` $\doteq$ `True` is provided. In general, the proof system is cannot deduce clauses through any means other than the inference rules. For example, this would also be true of deducing `x1 /= x2` $\doteq$ `True` from `x1 == x2` $\doteq$ `False`.

To a programmer, it may not be intuitive that these related clauses must be specified manually. In addition, if there are many such clauses, this process will become tedious. Therefore, a possible area of research is into techniques that would allow these to be automatically deduced.

Bibliography

[1] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In *Proceedings of the Tenth International Conference on Automated Deduction*, CADE-10, page 427–441, Berlin, Heidelberg, 1990. Springer-Verlag.

[2] Leo Bachmair, Harald Ganzinger, David McAllester, and Christopher Lynch. Chapter 2 - resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 19–99. North-Holland, Amsterdam, 2001. URL: https://www.sciencedirect.com/science/article/pii/B9780444508133500047, doi:https://doi.org/10.1016/B978-044450813-3/50004-7.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018. doi:10.1145/3182657.

[4] Hendrik Pieter Barendregt. Chapter 3 - reduction. In *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*, pages 50–75. Elsevier, 1984. URL: https://www.sciencedirect.com/science/article/pii/B9780444875082500113, doi:https://doi.org/10.1016/B978-0-444-87508-2.50011-3.

[5] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, page 125–144, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-11957-6_8.

[6] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 350–367, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Alan Bundy. Chapter 13 - the automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 845–911. North-Holland, Amsterdam, 2001. URL: https://www.sciencedirect.com/science/article/pii/B9780444508133500151, doi:https://doi.org/10.1016/B978-044450813-3/50015-1.

[8] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, sep 2000. doi:10.1145/357766.351266.

[9] Jieh Hsiang, Christopher Lynch, and Michael Rusinowitch. Paramodulation-based theorem proving. *Handbook of automated reasoning*, 1:371, 2001.

[10] John Hughes. QuickCheck testing for fun and profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, page 1–32, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-69611-7_1.

[11] Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. CycleQ: An efficient basis for cyclic equational reasoning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 395–409, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523731.

[12] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[14] Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15, 09 2002.

[15] Will Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 407–421, 03 2012. doi:10.1007/978-3-642-28756-5_28.

[16] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. *SIGPLAN Not.*, 49(9):269–282, aug 2014. doi:10.1145/2692915.2628161.

Appendix A

Proofs for Lemmas in Chapter 3

Lemma 3.1.1. *If* $alts_1 \triangleright_\alpha alts_2$ *then* $\mathsf{case}\ s\ \mathsf{of}\ alts_1 \triangleright \mathsf{case}\ s\ \mathsf{of}\ alts_2$

Proof. By induction on the derivation of $alts_1 \triangleright_\alpha alts_2$.

CASE(Alt1). Suppose $k\,\overline{a} \to s, alts \triangleright_\alpha k\,\overline{a} \to t, alts$. We know $s \triangleright t$. We have $\mathsf{case}\ u\ \mathsf{of}\ k\,\overline{a} \to s, alts \triangleright \mathsf{case}\ u\ \mathsf{of}\ k\,\overline{a} \to t, alts$ by (Case3).

CASE(Alt2). Suppose $k\,\overline{a} \to s, alts_1 \triangleright_\alpha k\,\overline{a} \to s, alts_2$. We know $alts_1 \triangleright_\alpha alts_2$. By induction hypothesis on this, we know $\mathsf{case}\ u\ \mathsf{of}\ alts_1 \triangleright \mathsf{case}\ u\ \mathsf{of}\ alts_2$. By Inversion, it must be that:

- $alts_1$ is $\overline{k\,\overline{a} \to t}$
- $alts_2$ is $k_1\,\overline{a_1} \to t_1, \cdots, k_i\,\overline{a_i} \to t'_i, \cdots, k_n\,\overline{a_n} \to t_n$
- $t_i \triangleright t'_i$

Then, we have $\mathsf{case}\ u\ \mathsf{of}\ k\,\overline{a} \to s, \overline{k\,\overline{a} \to t} \triangleright \mathsf{case}\ u\ \mathsf{of}\ k\,\overline{a} \to s, k_1\,\overline{a_1} \to t_1, \cdots, k_i\,\overline{a_i} \to t'_i, \cdots, k_n\,\overline{a_n} \to t_n$ by (Case3).

□

Lemma 3.1.2. *If* $\ell \triangleright r$ *then* $C[\ell] \triangleright C[r]$ *and if* $\ell \triangleright r$ *then* $A[\ell] \triangleright A[r]$

Proof. By simultaneous induction on the structure of $C[]$ and $A[]$.
Let us assume that $\ell \triangleright r$.

Suppose $C[]$ is of the form $\Box$. $\Box[\ell] = \ell$ and $\Box[r] = r$. We already know $\ell \triangleright r$, as required.

Suppose $C[]$ is of the form $s[]\,t$. Need to show $s[\ell]\,t \triangleright s[r]\,t$. By induction hypothesis on $s[]$, we know $s[\ell] \triangleright s[r]$. Then we have $s[\ell]\,t \triangleright s[r]\,t$ by (AppL).

Suppose $C[]$ is of the form $s\,t[]$. The argument is similar to the one for $s[]\,t$.

Suppose $C[]$ is of the form $\lambda a.s[]$. Need to show $\lambda a.(s[\ell]) \triangleright \lambda a.(s[r])$. By induction hypothesis on $s[]$, we know $s[\ell] \triangleright s[r]$. Then we have $\lambda a.(s[\ell]) \triangleright \lambda a.(s[r])$ by (Abs).

Suppose $C[]$ is of the form $\mathsf{case}\ s[]\ \mathsf{of}\ alts$. Need to show $\mathsf{case}\ s[\ell]\ \mathsf{of}\ alts \triangleright \mathsf{case}\ s[r]\ \mathsf{of}\ alts$. By induction hypothesis on $s[]$, we know $s[\ell] \triangleright s[r]$. Then we have $\mathsf{case}\ s[\ell]\ \mathsf{of}\ alts \triangleright \mathsf{case}\ s[r]\ \mathsf{of}\ alts$ by (Case2).

Suppose $A[]$ is of the form $k\,\overline{a} \to C[], alts$. Need to show $k\,\overline{a} \to C[\ell], alts \triangleright_\alpha k\,\overline{a} \to C[r], alts$. By induction hypothesis on $C[]$, we know $C[\ell] \triangleright C[r]$. Then we have $k\,\overline{a} \to C[\ell], alts \triangleright_\alpha k\,\overline{a} \to C[r], alts$ by (Alt1).

Suppose $A[]$ is of the form $k\,\overline{a} \to s,\ A'[]$. Need to show $k\,\overline{a} \to s,\ A'[\ell] \rhd_\alpha k\,\overline{a} \to s,\ A'[r]$. By induction hypothesis, $A'[\ell] \rhd_\alpha A'[r]$. Then we have $k\,\overline{a} \to s,\ A'[\ell] \rhd_\alpha k\,\overline{a} \to s,\ A'[r]$ by (Alt2). □

Lemma 3.2.1. *If $s \rhd t$ and $[\![s]\!]$ defined, then $[\![t]\!]$ defined.*

Proof. By induction on the derivation of $s \rhd t$.

CASE(Beta). Suppose $(\lambda a.s)\,t \rhd s[t/a]$ and $[\![(\lambda a.s)\,t]\!]$ defined. Need to show $[\![s[t/a]]\!]$ defined.
We know $[\![(\lambda a.s)\,t]\!] = (\lambda a.[\![s]\!])\,[\![t]\!]$, so we know $[\![s]\!]$ and $[\![t]\!]$ defined. Thus, we know $[\![s[t/a]]\!] = [\![s]\!][[\![t]\!]/a]$ defined.

CASE(AppL). Suppose $s\,t \rhd s'\,t$ and $[\![s\,t]\!]$ defined. Need to show $[\![s'\,t]\!]$ defined.
$[\![s\,t]\!] = [\![s]\!]\,[\![t]\!]$ so $[\![s]\!]$ and $[\![t]\!]$ defined. We know $s \rhd s'$. By induction hypothesis on $s \rhd s'$ and $[\![s]\!]$ defined, it must be that $[\![s']\!]$ defined. Thus, $[\![s'\,t]\!] = [\![s']\!]\,[\![t]\!]$ defined.

Case(AppR). Similar to **CASE(AppL)**.

CASE(Abs). Suppose $\lambda a.s \rhd \lambda a.s'$ and $[\![\lambda a.s]\!]$ defined. Need to show $[\![\lambda a.s']\!]$ defined.
$[\![\lambda a.s]\!] = \lambda a.[\![s]\!]$ so $[\![s]\!]$ defined. We know $s \rhd s'$. By induction hypothesis on $s \rhd s'$ and $[\![s]\!]$ defined, it must be that $[\![s']\!]$ defined. Thus, $[\![\lambda a.s']\!] = \lambda a.[\![s']\!]$ defined.

CASE(Case1). Suppose $\mathsf{case}\ k_i\,\overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \rhd t_i[\overline{s}/\overline{a}]$ and $[\![\mathsf{case}\ k_i\,\overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]$ defined. Need to show $[\![t_i[\overline{s}/\overline{a}]]\!]$ defined.
$[\![\mathsf{case}\ k_i\,\overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![k_i\,\overline{s}]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ so we know $[\![t_i]\!]$ defined for all i. In addition, $[\![k_i\,\overline{s}]\!] = [\![k_i]\!]\,\overline{[\![s]\!]}$ so $\overline{[\![s]\!]}$ defined. Thus, $[\![t_i[\overline{s}/\overline{a}]]\!] = [\![t_i]\!][\overline{[\![s]\!]}/\overline{a}]$ defined.

CASE(Case2). Suppose $\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \rhd \mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}$ and $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]$ defined. Need to show $[\![\mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]$ defined.
$[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ so $[\![s]\!]$ and $\overline{[\![t]\!]}$ defined. We know $s \rhd s'$. By induction hypothesis on $s \rhd s'$ and $[\![s]\!]$ defined, it must be that $[\![s']\!]$ defined. Thus, $[\![\mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s']\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ defined, as required.

CASE(Case3). Suppose $\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \rhd \mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}$, where $t_j = t'_j$ for all $j \neq i$, and $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]$ defined. Need to show $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}]\!]$ defined.
$[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ so $[\![s]\!]$ and $[\![t_j]\!]$ defined for all j. Thus, we know $[\![t'_j]\!]$ defined for all $j \neq i$.
We know $t_i \rhd t'_i$. By induction hypothesis on $t_i \rhd t'_i$ and $[\![t_i]\!]$ defined, it must be that $[\![t'_i]\!]$ defined. This gives us $\overline{[\![t']\!]}$ defined. Thus, $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t']\!]}$ defined, as required.

CASE(Prog). Suppose $a \rhd s$ and $[\![a]\!]$ defined. We know $a = s \in P$. Need to show $[\![s]\!]$ defined. s has no logical variables or projections in it as it is a term in the program definition, so $[\![s]\!]$ is always defined. □

Lemma 3.2.2. *If $s \rhd t$ then $[\![s]\!] \rhd [\![t]\!]$*

Proof. By induction on the derivation of $s \rhd t$.

CASE(Beta). Suppose $(\lambda a.s)\,t \rhd s[t/a]$. Need to show $[\![(\lambda a.s)\,t]\!] \rhd [\![s[t/a]]\!]$.
We know $[\![(\lambda a.s)\,t]\!] = (\lambda a.[\![s]\!])[\![t]\!]$ and $[\![s[t/a]]\!] = [\![s]\!][[\![t]\!]/a]$. We have $(\lambda a.[\![s]\!])[\![t]\!] \rhd [\![s]\!][[\![t]\!]/a]$ by (Beta).

CASE(AppL). Suppose $s\,t \rhd s'\,t$. Need to show $[\![s\,t]\!] \rhd [\![s'\,t]\!]$.
We know $s \rhd s'$. By definition, $[\![s\,t]\!] = [\![s]\!]\,[\![t]\!]$ and $[\![s'\,t]\!] = [\![s']\!]\,[\![t]\!]$. By induction hypothesis on $s \rhd s'$, we have $[\![s]\!] \rhd [\![s']\!]$. Then, we have $[\![s]\!]\,[\![t]\!] \rhd [\![s']\!]\,[\![t]\!]$ by (AppL).

CASE(AppR). Similar to **CASE(AppL)**.

CASE(Abs). Suppose $\lambda a.s \rhd \lambda a.s'$. Need to show $[\![\lambda a.s]\!] \rhd [\![\lambda a.s']\!]$.
We know $s \rhd s'$. By definition, $[\![\lambda a.s]\!] = \lambda a.[\![s]\!]$ and $[\![\lambda a.s']\!] = \lambda a.[\![s']\!]$. By induction hypothesis on $s \rhd s'$, we have $[\![s]\!] \rhd [\![s']\!]$. Then, we have $\lambda a.[\![s]\!] \rhd \lambda a.[\![s']\!]$ by (Abs).

CASE(Case1). Suppose $\mathsf{case}\ k_i\ \overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \triangleright t_i[\overline{s}/\overline{a}]$. Need to show $[\![\mathsf{case}\ k_i\ \overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] \triangleright [\![t_i[\overline{s}/\overline{a}]]\!]$.

We know $[\![\mathsf{case}\ k_i\ \overline{s}\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![k_i\ \overline{s}]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$. We also have $[\![k_i\ \overline{s}]\!] = [\![k_i]\!]\ [\![\overline{s}]\!]$ and $[\![t_i[\overline{s}/\overline{a}]]\!] = [\![t_i]\!][[\![\overline{s}]\!]/\overline{a}]$. Then, we have $\mathsf{case}\ [\![k_i]\!]\ [\![\overline{s}]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]} \triangleright [\![t_i]\!][[\![\overline{s}]\!]/\overline{a}]$ by (Case1).

CASE(Case2). Suppose $\mathsf{case}\ s\ \mathsf{of}\ alts \triangleright \mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}$. Need to show $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] \triangleright [\![\mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!]$.

We know that $s \triangleright s'$. By definition, $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ and $[\![\mathsf{case}\ s'\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s']\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$. By induction hypothesis on $s \triangleright s'$ we have $[\![s]\!] \triangleright [\![s']\!]$. Then, we have $\mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]} \triangleright \mathsf{case}\ [\![s']\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ by (Case2).

CASE(Case3). Suppose $\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t} \triangleright \mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}$ where $t_j = t'_j$ for all $j \neq i$. Need to show $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] \triangleright [\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}]\!]$.

We know $t_i \triangleright t'_i$. By definition, $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t]\!]}$ and $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}]\!] = \mathsf{case}\ [\![s]\!]\ \mathsf{of}\ \overline{k\,\overline{a} \to [\![t']\!]}$. By induction hypothesis on $t_i \triangleright t'_i$, we have $[\![t_i]\!] \triangleright [\![t'_i]\!]$. In addition, we have $[\![t_j]\!] = [\![t'_j]\!]$ for all $j \neq i$. Thus, we have $[\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t}]\!] \triangleright [\![\mathsf{case}\ s\ \mathsf{of}\ \overline{k\,\overline{a} \to t'}]\!]$ by (Case3).

CASE(Prog). Suppose $a \triangleright s$. We know $a = s \in P$. Need to show $[\![a]\!] \triangleright [\![s]\!]$. We have $[\![a]\!] = a$ as it is simply a name. s has no logical variables inside as it is a term in the program definition, so $[\![s]\!] = s$. We already know $a \triangleright s$. □