Zeno:
A tool for the fully automated verification of functional program properties

Final Report

William Sonnex (ws506)
MEng Computing 4

Department of Computing, Imperial College London

Supervisor: Professor Sophia Drossopoulou

June 2010

Abstract

Statically verifying properties of imperative programs is a very developed field with many different techniques and tools. The automated verification of functional programs however is not so developed. Work in this area often relies on the proof to be created by the programmer, with the tool simply checking its correctness. This requires considerable work on the part of the user, as well as a strong knowledge of logic and proof specification .

In this report we present 'Zeno', a tool for statically verifying properties of functional programs without any input from the programmer. Programs, and the properties we wish to prove, are expressed in Function Logic, a language we have created for this purpose, and theorems of which can then be proven or disproven by our tool. The method for these proofs, and the fundamental rules underlying it, are also described here. In addition we give an encoding of the functional language Haskell into Function Logic, which we have also implemented in Zeno.

What sets Zeno apart from existing methods is its ability to infer intermediary lemmas necessary for a proof. In particular we provide an automated proof of the idempotence of list reversal, using only the definition of the reversal function, with an interesting lemma proven in the process.

Acknowledgements

First and foremost I would like to thank my supervisor, Professor Sophia Drossopoulou. Without her continuous support and knowledgeable advice this project would not have been possible.

My great thanks also to Professor Susan Eisenbach, for her general help and guidance. Tristan Allwood, for being my mentor on all things Haskell and functional. Dr. Krysia Broda, for her invaluable course on automated reasoning which provided the foundation of this project. Thanks also to Dr. Steffen van Bakel for his course on type systems, which helped me understand the theoretical foundations of functional languages.

Finally I would like to thank Francesca Bellei, for helping me pick the name of my tool, and for her constant support and encouragement throughout the year.

Contents

Chapter 1

Introduction

Software bugs are estimated to cost the global economy over $100 billion dollars every year. Such bugs could be described as programs violating a certain property in a certain situation, and while the situation was obviously not anticipated, the property might well have been. It is often the case that the developers will know the properties that their program is supposed to fulfil, and lack only the tools to prove that they will always be upheld.

There are many existing tools for checking properties of functional programs by enumerating inputs to these properties, examples being QuickCheck[10] and SmallCheck[32]. These tools do not conclusively prove a property, only that a property holds for a certain finite set of values. Zeno, on the other hand, seeks to give a mathematically sound proof that a property can never be violated, for any input it might be given. This approach is known as *static* property checking, as opposed to the *dynamic* property checking employed by tools such as QuickCheck.

It is unfortunate however that full static verification is doomed to incompleteness from the start. Proving non-trivial properties of Turing complete language such as Haskell has been shown to be undecidable by Rice's theorem[31], so this is a field that can never be entirely solved.

A simple example function, given in Figure 1.1, defines the equality over lists. There are certain properties we now may wish to prove about this equality function, to make sure it behaves as an equality should. Examples of such properties are given in Figure 1.2.

Figure 1.1 Haskell equality function over lists

```
listEq :: Eq a => [a] -> [a] -> Bool
listEq [] [] = True
listEq (x : xs) (y : ys) = (y == x) && (listEq xs ys)
listEq _ _ = False
```

Figure 1.2 Useful properties of the equality of lists

$\forall xs^{[a]} :$ `listEq xs xs`	Reflexivity
$\forall xs^{[a]}.\forall ys^{[a]} :$ `listEq xs ys` $\Longleftrightarrow$ `listEq ys xs`	Symmetry
$\forall xs^{[a]}.\forall ys^{[a]}.\forall zs^{[a]} :$ `listEq xs ys` $\wedge$ `listEq ys zs` $\Rightarrow$ `listEq xs zs`	Transitivity

1.1 Contributions

In this report we detail the following three contributions.

Function Logic
A formal logic for describing functional programs alongside properties we wish to prove about them, along with a set of rules for proving or disproving these properties.

Function Logic Tableau
A formal method that structures the application of the rules of Function Logic in such a way that a human or computer can algorithmically construct a proof or disproof of a property. An advantage of Function Logic Tableau over existing proof methods is the ability to infer intermediary lemmas necessary to complete a larger proof.

Zeno
A tool implementing Function Logic Tableau which is able to discover and output proofs or disproofs of reasonably complex properties about functional programs. In particular Zeno is able to prove the idempotence of the reverse function over lists without any information other than the function definition, a proof which requires three human provided lemmas when performed in existing tools. More than that, it provides us with another property of list reversal that it discovers during the proof (this proof is evaluated fully in Section 7.1.1). Zeno also

provides an encoding of Haskell into Function Logic, so that Haskell program properties may be solved using the tool.

1.2 Overview of the Report

The report is organised as follows. In Chapter 2 we give a background to program verification in general. In Chapter 3 we describe what features of functional languages, and specifically Haskell, we address, giving a precise formal definition of the functional language we operate over, called Haskell-Core. In Chapter 4 we formalise Function Logic, the underlying representation that Zeno solves theorems in, and the basis of our approach, along with a formal encoding of Haskell-Core into Function Logic. In Chapter 5 we describe the specific method in which Zeno utilizes Function Logic, called Function Logic Tableau. In Chapter 6 we outline the implementation of Zeno and describe how one uses the tool. In Chapter 7 we evaluate the success of Zeno in proving properties, along with some examples of properties it was able to prove, and some it couldn't. In Chapter 8 we explain various ways in which Zeno and Function Logic could be extended to provide more functionality. We finish the report with some concluding remarks in Chapter 9.

Chapter 2

Background

In this chapter we give a brief introduction to program verification in general, along with a description of existing work that has been done in this field.

2.1 Hoare logic

The most widely used method for specifying program properties to be verified is **Hoare logic**, invented by C.A.R. Hoare in 1969[21], the central feature of which is the **Hoare triple**:

$$\{P\}\ C\ \{Q\}$$

Here, P and Q are formulas in formal logic[1] about the program state and C is a command that can be run within the program. The formula P, known as the *precondition*, is an assertion about the state the system will be in *before* C is executed. The formula Q, known as the *postcondition*, is an assertion about the state the system will be in *after* C is executed. In this way we can describe, in formal logic, the effect that a command running within a program should have. A common naming convention is to say that C requires P and ensures Q. See Figure 2.1 for examples of Hoare triples, the last of which is an example of an *invariant*, a property which does not change upon execution of a command (so $Q \Rightarrow P$).

[1]Usually first-order logic with equality and arithmetic.

Figure 2.1 Example Hoare triples

$$\{x \geqslant 0\}\ x := x + 1\ \{x > 0\}$$

$$\{x > y\}\ \mathit{if}\ z > 0\ \mathit{then}\ x := x + z\ \mathit{else}\ x := x - z\ \mathit{endif}\ \{x > y\}$$

A system which verifies Hoare logic must prove that if the precondition P holds then the execution of the command C will make the postcondition Q hold. Such a system might also check that any time the command C is run the precondition P must hold.

Figure 2.2 Real world Hoare logic in Spec#

```
public void incrementX()
  requires x >= 0; \\ Precondition
  ensures x > 0; \\ Postcondition
{
  x = x + 1; \\ Method body
}
```

2.2 Types of verification

The two main types of program verification are *static* and *dynamic*. Furthermore, it is my opinion that both of these types then have two subtypes, which I will call *automated* and *manual*.

Figure 2.3 Examples of verification types

	Static	**Dynamic**
Automated	Boogie	QuickCheck
Manual	Isabelle/HOL	Unit Testing

2.2.1 Dynamic verification

Dynamic verification refers to checking properties of a program against a set of example data. Preconditions of a method are checked against its arguments and the program state when it is called and postconditions are checked against its return value and the program state when the method returns. Dynamic verification is largely simple to implement, though issues can arise in conditions that crash or fail to terminate, and so break a program that might otherwise have been correct. The main shortcoming of this method is that it is only capable of falsifying properties, rather than proving them. It certainly allows one to test that properties hold for many different input values, but never for all of them[2].

Manual dynamic verification is when the user manually enters any test data to be used to test the program's properties. Unit testing frameworks, such as JUnit for the Java language are the most prolific form of manual dynamic verification. The programmer creates *unit tests* which will run certain commands on the program, testing the outcome to see if it matches given properties. In addition, the *assert* statement found in many imperative languages, which will check a property as and when the code is actually run, is also a form of manual dynamic verification, which is known as *run-time* verififcation.

A tool performing automated dynamic verification will test program properties against an algorithmically generated set of example data. This allows for a much greater range of values to be tested with much less effort from the programmer. A good example of this is the Haskell framework **QuickCheck**[10], discussed in Section 2.7, which will test a boolean function against a series of randomly generated inputs to see if it ever returns `False`. Automated dynamic verification is very difficult for any programs with nondeterministic external input or mutable global state, as it is very difficult to automatically simulate such an interaction, or generate such a state.

2.2.2 Static verification

The second type of verification is *static* verification. This consists of proving properties of a program for all possible instances and inputs, and is substantially more difficult than dynamic checking. Indeed, Rice's theorem[31] states that for any non-trivial property of partial functions, there is no general and effective method to decide whether an algorithm computes a partial function with that property. This means that full static verification of a Turing Complete language is an undecidable problem, and so can never be completely solved. It is also known as *compile-time* verification, as it does not involve the actual execution of the program code.

Manual static verification is when the proofs of any properties to be checked are written by the programmer and the tool merely proves their validity. One such manual static verification tool that exists for functional languages in Isabelle/HOL[36]. This tools allows one to express functional programs in a form of ML and then construct formal proofs of their properties, which

[2]If your properties can take an infinite number of input values as ours can.

Isabelle will check for correctness. Isabelle/HOL is discussed more fully, and with examples, in Section 2.9.1.

Automated static verification is when the properties to be verified are expressed in simple formulas and the verifier must infer the proof of these formulas for itself. This is the type of verification done by the tool **Boogie**[5], which the **Spec#**[6] language uses for its static verification. One can also translate their properties and programs from any imperative language into the intermediary language BoogiePL, which can then be statically verified by the Boogie tool.

2.2.3 Hybrid verification

One solution to the undecidability of static verification versus the incompleteness of dynamic verification is to combine the two. Checking properties statically wherever possible and checking them dynamically otherwise. One version of this approach is called "Hybrid type checking"[3] by Cormac Flanagan[17], and refers to the combination of automated static verification and run-time manual dynamic verification. This is the approach of many verification systems for imperative lanugages, including Spec#.

2.3 Implications of verifying a functional language

2.3.1 Simpler Contracts

Pure functional programming is programming without side-effects. The only inputs that need to be considered are the parameters to the function, and the only effect of the function is to produce its output. Unlike imperative programming there can be no reading or writing of global state. A function in a purely functional program is a mathematical mapping between types, nothing more.

2.3.2 Recursive Types

Recursivly typed variables can have infinitely many different values, and in a language with lazy evaluation such as Haskell may actually be infinite in size at runtime. Any property of a function that takes a recursive type as an argument must be proven for all of these infinite possible inputs. Looking at

[3]The usage of "type checking" here refers to the verification of properties expressed through refinement types.

the example Haskell code given in Figure 2.4 we can see that the datatype `Nat` represents the countable infinity of the natural numbers, and that `add` can therefore accept an infinite number of different arguments.

Figure 2.4 Haskell definition of the natural numbers and their addition

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = Succ (add x y)
```

It is also the case that *pointers* or *references* in imperative languages can allow for potentially infinite constructs. The difference however, is that it is acceptable that a static verification tool for an imperative language not to be able to reason about the contents[4] of any references an object may hold. Most properties you would wish to prove only concern the internal state of the object itself, and the effects of any methods that mutate it. In functional programming though, all but the most trivial properties must be proven over recursive data structures, and it would be unacceptable to have a tool incapable of dealing with them.

One method of reasoning about recursive data structures is with *structural induction*. Structural induction is the application of proof by induction applied to recursive data structures. Using `Nat` from Figure 2.4 as an example, and attempting to prove the arbitrary property $P^{\wp(Nat)}$: If we have a proof of this property for `Zero` (so $P(Zero)$), and by assuming a proof for an arbitrary `n` of type `Nat` we can find a proof of `(Succ n)` (so $\forall n^{Nat}.P(n) \Rightarrow P(Succ(n))$), then we have the proof for all of the natural numbers (so $\forall n^{Nat}.P(n)$). This is expressed formally with the sentence of second-order logic in Figure 2.5.

Figure 2.5 Axiom of structural induction over **Nat**

$$\forall P^{\wp(Nat)} \cdot P(Zero) \wedge (\forall n^{Nat} \cdot P(n) \Rightarrow P(Succ(n))) \Rightarrow \forall n^{Nat} \cdot P(n)$$

[4]Many such tools can verify whether a pointer references a valid location, but not what the data at the location is.

2.3.3 Higher Order Functions

In functional programming an argument to a function can itself be another function. Any function that takes another function as an argument in this way is known as a *higher-order* function. It is generally accepted that the best way to write Haskell code is using higher-order functions, and so we should endeavour to be able to verify properties about function-typed arguments. As an example take the function $f^{(Int \to Int) \to Int}$ that we wish to verify will always return a positive value if its argument function always returns a positive value, as formally specified in Figure 2.6.

Figure 2.6 Simple property of the f function

$$\forall g^{Int \to Int} \cdot (\forall i^{Int} \cdot g(i) > 0) \Rightarrow f(g) > 0$$

2.3.4 Polymorphism and Type Classes

Another feature of the Haskell type system is *polymorphism*, allowing us to quantify over the type of all types, also known as the type *Set*. Polymorphic types are represented by lower-case letters in Haskell type definitions. There is an implicit `forall` statement at the beginning of every such polymorphic type but it is not incorrect to state it explicitly. Indeed, the following three types for the map function are equivalent, with the first two in Haskell and the third in formal logic:

1. `map :: (a -> b) -> [a] -> [b]`
2. `map :: forall a b . (a -> b) -> [a] -> [b]`
3. $map^{\forall a^{Set} b^{Set} : (a \to b) \to [a] \to [b]}$

There are also polymorphic types, ones which include one or more polymorphic arguments in their type declaration. One common example is the type of lists `[a]`, which is polymorphic in the type of its members.

Haskell's *type classes* bring further complication to polymorphism. These are subsets of *Set* as defined by a set of operations that can be performed on them. A simple example is the type class `Eq` of all types that can be equated, that is to say they have an equality operation `(==) :: a -> a -> Bool` defined for them. This type class is shown in Figure 2.7, along with an

Figure 2.7 The **Eq** type class

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

instance Eq Nat where
  Zero == Zero = True
  Zero == (Succ _) = False
  (Succ _) == Zero = False
  (Succ x) == (Succ y) = x == y
```

instance of `Eq` for the type `Nat` which was given in Figure 2.4. The use of `instance` in this way effectively adds `Nat` to the set `Eq`.

There are many type classes for which certain implicit rules exist over the functions they are defined for. A good example of this is the type class `Monoid`, for monoidal types (shown in Figure 2.9). One such rule for monoids is that `mempty` is the identity element. This means that if we give `mempty` as one of the arguments to `mappend`, the function will return the other argument unchanged. This property is formally expressed in Figure 2.8.

Figure 2.8 Identity of monoidal types

$$\forall M^{Monoid}.\forall m^{M} : mappend\ x\ mempty = mappend\ mempty\ x = x$$

Figure 2.9 The **Monoid** type class

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

This property is expressed informally in the Haskell documentation, but at no point is it actually formally verified for any `Monoids`. Any tool that is able to statically reason about type class functions might be able to officially verify this property, and the informal type class properties of the rest of the Haskell standard libraries.

2.4 Automated theorem proving based verification

Most program verification tools that use Hoare logic style pre and post conditions perform static verififcation by encoding the program and any conditions as a logic and using an automated theorem prover to find a proof of this theorem, or to generate a counter-example if the theorem is not valid.

Given a Hoare triple $\{P\}\ C\ \{Q\}$ we can then attempt to find a proof that: The encoding of the program C into logic, given as $[[C]]$, means that the precondition implies the postcondition, over all possible inputs. So the encoding $[[\{P\}\ C\ \{Q\}]]$ becomes $\forall \vec{x} : [[C]] \Rightarrow (P \Rightarrow Q)$, or the equivalent $\forall \vec{x} : (P \wedge [[C]]) \Rightarrow Q$, where $\vec{x}$ represents all free variables in the formula. An example of this is given in Figure 2.10. Note that as variables in logic are immutable we must create the new variable x_2 to represent the value of x after assignment in this example.

Figure 2.10 Encoding Hoare triples to logic

$$\{x \geqslant 0\}\ x := x + 1\ \{x > 0\}$$

gives

$$\forall x . \forall x_2 : ((x \geqslant 0) \wedge (x_2 = x + 1)) \Rightarrow (x_2 > 0)$$

2.4.1 Satisfiability Modulo Theories

The most common type of automated theorem prover that is used to statically verify programs is a **Satisfiability Modulo Theories** or **SMT** solver. Instead of finding a generalised proof of the validity of a formula, an SMT solver instead attempts to search for an assignment to all its variables, such that the formula becomes true; it attempts to find a model which satisfies the formula.

This proof search is augmented by the use of *background theories*, which give it the general rules needed to solve problems quickly. Examples of which are: The theory of integers, the theory of real numbers, and theories relating to data structures such as arrays and lists.

As an SMT solver only attempts to satisfy a formula this means that it will only find one model for our encoded Hoare triple, instead of proving it is correct for all models. It is the case however that if the negation of a formula is unsatisfiable, then the formula is valid. So we can negate the formula of our encoded Hoare triple, giving $\neg(P \wedge [[C]] \Rightarrow Q)$, or equivalently $P \wedge [[C]] \wedge \neg Q$. If this now cannot be satisfied by the SMT solver then the properties are valid. Conversely, if the SMT solver is able to find a model that satisfies this negated formula, then this model is a counter-example to the properties we are trying to prove.

SMT-LIB

The standardized format for expressing SMT formulas is **SMT-LIB**, as described in the "The SMT-LIB Standard"[34] documentation. This format can be recognized by all of the major SMT solvers such as Yices[14] and Z3[12]. In fact, the speed at which a solver can, for various SMT-LIB files, give a model or declare them unsatisfiable, forms the basis of an annual competition between SMT solvers[5]. In Figure 2.11 we have an example of such an encoding.

Figure 2.11 Encoding a property into SMT-LIB

$$\{x \geqslant 0\}\ x := x + 1\ \{x > 0\}$$

encodes to

$$(x \geqslant 0) \wedge (x_2 = x + 1) \wedge \neg(x_2 > 0)$$

which gives the following SMT-LIB code

```
(benchmark hoare
  :extrafuns ((x Int) (x2 Int))
  :assumption (>= x 0)
  :assumption (= x2 (+ x 1))
  :formula (not (> x2 0))
)
```

Now, if we feed this into the SMT solver Z3[12] it informs us that the formula is unsatisfiable, so this Hoare triple is valid. However, if we were to alter the post-condition to one that is invalid, say $x_2 > 1$, Z3 is able to find the model `x -> 0; x2 -> 1`, giving us the exact counter-example for our now invalid properties.

[5]http://smt-comp.org

Unsuitability of SMT for functional program verification

The SMT-LIB standard does not on its own admit the definition of recursive datatypes, but there are extensions for various SMT solvers to support them. Unfortunately, the only way we can reason with recursive functions in classical logic is with quantifiers, and SMT solving becomes undecidable in the presence of quantifiers.

One example of such reasoning is in Figure 2.12, which shows the definition of the addition function for natural numbers, implemented in SMT-LIB with the `:datatypes` extension for the Z3 solver. However the usage of this definition in any formula will render its satisfiability unknown.

Figure 2.12 Recursive datatypes in Z3

```
(benchmark addition
  :datatypes ((Nat Zero (Succ (pred Nat))))
  :extrafuns ((add Nat Nat Nat))
  :assumption (forall (?x Nat) (= (add Zero ?x) ?x))
  :assumption (forall (?x Nat) (?y Nat)
    (= (add (Succ (pred ?x)) ?y) (Succ (pred (add ?x ?y)))))
)
```

2.4.2 Tableau methods

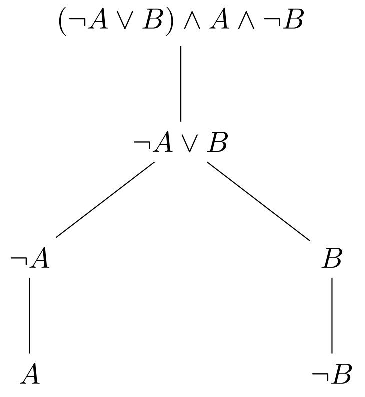
A popular method for automated theorem proving is with the use of tableaux. A tableau is a tree structure that represents the proof of a logical theorem, such that this proof could be generated in a systematic way by a computer algorithm. Tableaux are created by placing the theorem we are trying to prove at the root node and then generating new branches beneath it with a set of formal rules. If a node at a descendent branch satisfies a certain property then it is said to “close”, meaning that a proof has been found down this branch. If every branch of a tree closes then the entire tree is closed and the proof was successful.

As an example we will describe the construction of a Semantic Tableau[30], used for proving the unsatisfiability of a propositional formula. This method uses as its branching rules the inference rules of classical propositional logic. Each node contains a single propositional formula which is true at that node. All the formulas in the parents of a node are also true at that node. A branch closes if it contains both a node with A, and one with $\neg A$, for any formula A, as we have then found a contradiction.

We start with the formula we are trying to prove unsatisfiable at the root node: $(\neg A \vee B) \wedge A \wedge \neg B$. We then apply the $(\wedge)$ inference rule to derive a single new branch node with $\neg A \vee B$. We then apply the $(\vee)$ inference rule which generates two new branches, one for $\neg A$ and one for B. Each branch represents a different possibility that could be true in above the formula. Down the left branch we can use $(\wedge)$ again to get the node A, which means we can immediately close the branch as it contains both A and $\neg A$. Down the right branch we can use the $(\wedge)$ rule on the head node to get $\neg B$, which means we can also close this branch. So now we have a tableau in which every branch is closed, meaning we have proven the original formula unsatisfiable.

This example is given fully in Figure 2.13.

Figure 2.13 Example of a Semantic Tableau



2.5 Verification of imperative programs

While verifying functional programs is still an undeveloped field, the verification of imperative programs is currently very popular. There is a large market for verifying of critical pieces of software, and most of the world's software is written in imperative code.

There are many languages and tools for expressing and verifying imperative program properties, both statically and dynamically. Examples include ESC/Modula3, ESC/Java, .NET Code Contracts, Boogie, and Spec#. However the approach of these are all very similar, so I will only go into one in detail.

2.5.1 Spec#

One popular language for research into program contract verification is Microsoft's Spec#[6], a superset of the C# language extended with various constructs for expressing logical properties. In Figure 2.2 we can see an example of pre and post conditions on the method `incrementX` using the *requires* and *ensures* keywords respectively.

In Figure 2.14 we see an example of Spec# that with an object invariant stating that its field `x` should always be positive. In this case the Spec# compiler is able to statically verify that this invariant will always hold.

Figure 2.14 Spec# example

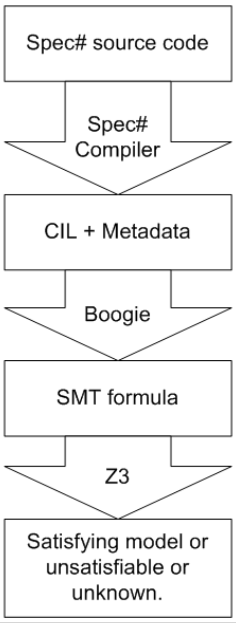
```
using System;
namespace N
{
  class C
  {
    int x = 1;

    invariant x > 0;

    public void AddToX(int y)
      requires y >= 0;
    {
      x = x + y;
    }
  }
}
```

However, if we modify the precondition of `AddToX` to `requires y >= -1`, or remove the precondition entirely, then we are informed by the Spec# compiler that this method might break the object invariant. So the verifier is able to statically detect the potential violation of the invariant.

Spec# Architecture

Spec# utilizes Flanagan's hybrid verification (see Section 2.2.3), generating an application with dynamic runtime checking after attempting to statically verify any properties. The compiler uses the tool **Boogie**[5], also developed by Microsoft, to perform its static verification. Indeed, Boogie will verify any program compiled to Microsoft's .NET executable bytecode (CIL), where properties you wish to verify are expressed as metadata within the bytecode. See Figure 2.15 for a diagram of this process.

Figure 2.15 Spec# architecture



2.6 Dana Xu's ESC/Haskell

In her PhD thesis "Static Contract Checking for Haskell"[39], Dana Xu outlines a method for the static verification of Hoare logic style pre and post conditions for Haskell functions, called Extended Static Checking for Haskell, or ESC/Haskell. Her contracts are written in the same style as in Haskell function type assignment, but instead of the type there is a property about that the value given at that type position. This property is of the form `{ x | P }`, where `x` is a variable name given to that argument and `P` is a boolean property about it. More specifically `P` is any Haskell expression of type `Bool`.

In Figure 2.16 we have an example taken from her paper[39] of an increment function with a contract that states it must take a positive value and return a value greater than the input:

Figure 2.16 Example contract in ESC/Haskell

```
{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}
inc :: Int -> Int
inc x = x + 1
```

2.6.1 Verifying ESC/Haskell

Xu's method for the verification of her ESC/Haskell is as follows. Contracts for functions and the functions themselves are translated from Haskell into her own functional language $\mathcal{H}$ in such a way that every execution that would lead to a contract violation is replaced by the term `BAD`. An algorithm is then used to simplify the $\mathcal{H}$-expression in an attempt to eliminate all unreachable paths. This algorithm will unroll function calls where possible in order to remove them. If the term `BAD` no longer exists in the expression then the contract is verified, or *syntactically safe*. Any path which returns `BAD` without relying on the result of a function call represents a violation of the contract and by following the values along that path we can construct this counter-example. Paths that both contain a function call and return `BAD` have an unknown result. See Figure 2.17 for a flowchart of this process.

One important result of Xu's thesis is the soundness and completeness of the formalization of her contracts into the $\mathcal{H}$ language. Given the functional expression e and the contract t, she uses the statement $e \in t$ to mean either e diverges or it is a crash-free expression satisfying the contract t. She defines $e \triangleright t$ as the projection of e and t into her language $\mathcal{H}$ and proves:

$$(e \triangleright t) \text{ is crash-free} \iff e \in t$$

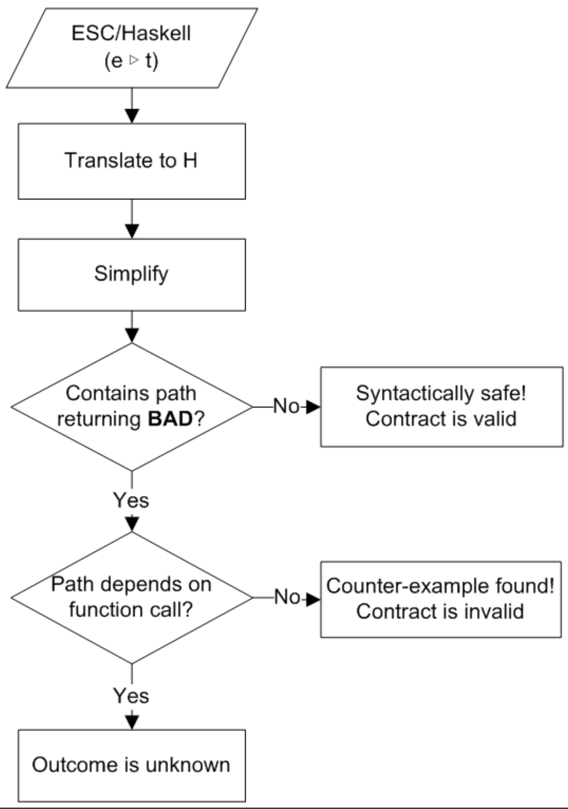
Where *crash-free* refers to the inability to reach the term `BAD`.

Xu's theorem is very useful, it shows us that given a sound and complete simplification algorithm her entire verification process is sound and complete. Obviously by Rice's theorem a complete simplification algorithm is unobtainable but if one could identify a case in which this algorithm must be incomplete then we have found a case in which all static verification must be incomplete, by the Turing Completeness of $\mathcal{H}$.

One important side-note is that Xu's engine itself does not deal with arithmetic inequality. She instead uses the SMT solver Simplify[6] to handle verification problems relating to the comparison of integers and real numbers.

[6]Previously a popular SMT solver, Simplify has recently been superceeded by solvers such as Z3

Figure 2.17 ESC/Haskell flowchart



ESC/Haskell (e ▹ t)
Translate to H
Simplify
Contains path returning **BAD**?
No
Syntactically safe! Contract is valid
Yes
Path depends on function call?
No
Counter-example found! Contract is invalid
Yes
Outcome is unknown

2.7 QuickCheck

For the automated dynamic checking of Haskell code there is the popular tool **QuickCheck**[10]. Given a function that returns type `Bool`, QuickCheck will generate multiple random inputs for that function and report if any caused the function to return `False`.

In order to generate random data for the function its input must be of a type that is an instance of the type class `Arbitrary`, which specifies how to generate random values for that type. Most Haskell built in types have predefined instances of `Arbitrary` defined by the QuickCheck developers.

2.8 Jean Goubault-Larrecq's Sequent Calculus

Gerhard Gentzen developed the first sequent calculi **LK** and **LJ** in 1934[18], to study the technique of natural deduction on classical and intuitionistic logic respectively. It, and the theories surrounding it, admit automatable techniques for deciding the validity of logical theories.

More recently however, Jean Goubault Larrecq has extended this calculus with equality, free constructors and, importantly, structural induction[19]. He goes on to describe a tableux method by which one may soundly construct proofs of formulas in his calculus. This method, along with the theory behind it, could potentially be adapted as a proof method for functional languages.

2.9 Manual static verification

While automated theorem proving for higher-order logic is an incredibly complex task, it is much easier to simply check the proofs that humans have created. That is to say you express every logical rule and the structure in which it was used, along with any intermediary lemmas, and the proof assistant will verify that this is indeed a valid proof.

2.9.1 Isabelle/HOL

Isabelle/HOL[36] is **HOL** (Higher Order Logic) running on top of the proof assistant Isabelle. Isabelle itself is a tool that allows one to formally check proofs such that you know them to be sound, and remove any possibility of human error. Using Isabelle/HOL we can specify functional programs, using a form of ML, and develop proofs of their properties.

In Figure 2.18 we give an example of the proof of a function property in Isabelle/HOL, specifically that the length of one list appended to another is the sum of the length of each list. Isabelle/HOL has a reasonably powerful automated theorem prover built in, as after we have specified we want to perform proof by structural induction on the variable `xs` (using `apply ( induct_tac xs)`) it can complete the rest of the proof for us (using `apply (auto)`).

Figure 2.18 List application length proof in Isabelle

```
theory Length
imports Datatype
begin

datatype list
  = Empty                              ("[]")
  | Cons nat list                      (infixr "#" 64)

primrec app :: "list => list => list"  (infixr "++" 65)
where
  "app [] ys = ys" |
  "app (x # xs) ys = x # (app xs ys)"

primrec length :: "list => nat"
where
  "length [] = 0" |
  "length (x # xs) = Suc (length xs)"

lemma lengthApp [simp]:
    "length (xs ++ ys) = (length xs) + (length ys)"
  apply (induct_tac xs)
  apply (auto)
done
```

Haskabelle

Haskabelle[37] is a tool for the conversion of Haskell source code into the higher-order logic syntax used by Isabelle/HOL. Having converted your program you can then use Isabelle to prove any properties you wish about it.

The shortcoming of Haskabelle however is that is requires users to know how to use Isabelle, in addition to the obvious inconvenience of converting and moving between two different sets of source code. In addition Haskabelle can only check existing proofs, rather than inferring proofs for itself, which is further effort for the programmer, especially with proofs needing to be rewritten if one changes a small part of the source code.

2.10 Summary of existing verification methods

- **Satisfiability modulo theories** based methods, although the most popular technique for imperative program verification, have shortcomings with respect to functional programming which may be impossible to overcome. Namely their inability to easily reason over recursive datatypes, which are one of the most central features of functional languages. Therefore, this project would either have to employ a different strategy, or focus on the enhancement of an existing SMT solver.

- **Dana Xu's ESC/Haskell** gives a good theoretical basis for statically verifying functional code. Unfortunately, the strength of her method rests on her simplification algorithm. It is at this stage that the problem of function calls, and hence recursion in general is dealt with by an unrolling technique. It is my opinion that a method that focuses on structural induction should instead be used.
- **QuickCheck** is a very useful method for disproving properties of Haskell programs, but can never be really trusted to ensure that they are correct. If a technique such as this is to be used it should be in tandem with one that can positively verify programs, so that properties can be proved either way.
- **Proof checkers** such as Isabelle/HOL provide a method by which human intelligence can be utilized in program verification. However these methods rely almost entirely on the ability of the user to construct such proofs themselves, something which developers may be neither willing nor able to do. One useful compromise might be to use these methods to augment an automated approach, with a tool possibly querying the user should there exist a proof step that it cannot solve.

Chapter 3

Haskell

In this chapter we start by describing formally the subset of Haskell that is addressed by our tool (Section 3.1), referred to as Haskell-Core (**HC**) (Section 3.2), along with its typing and operational semantics. We finish by discussing the extent to which Zeno tackles non-terminating programs and infinite data structures (Section 3.3).

3.1 Restrictions

It is important to note that we have not addressed all of the Haskell language in our tool. Our restrictions are on the *types* of the functions our tool can address, which are limited to function types and user-defined datatypes. If you wish to prove a property of a function there can be no polymorphism or primitives in its type, or anywhere in inside the function definition. In Figure 3.1 we give examples of function types that our valid or invalid in our method, where `Nat` is a user defined datatype.

Figure 3.1 Valid and invalid function types

`id :: a -> a`	**Invalid** (polymorphism)
`(+) :: Num a => a -> a -> a`	**Invalid** (polymorphism)
`factorial :: Int -> Int`	**Invalid** (primitive `Int` type)
`add :: Nat -> Nat -> Nat`	**Valid**
`fixNat :: (Nat -> Nat) -> Nat`	**Valid**

3.1.1 Why no primitive types?

Our approach deals with inductive proofs of recursive functions over recursive datatypes. Primitively types and functions over them are very inefficient when represented recursively. The unsigned integers, which we can represent as we did the natural numbers (Figure 2.4), are given as an example in Figure 3.2.

Figure 3.2 Unsigned integers as recursive Haskell datatypes

```
data UInt = Zero | Succ UInt

Zero + y = y
(Succ x) + y = Succ (x + y)
```

If we were to use this definition and calculate `1000 + x` somewhere in a function it would require up to one thousand function calls, and in our proof method would require over a one thousand step proof.

Signed integers and floating point numbers are even more complex. Especially so when you consider fixed precision types, such as `Int`, which are not recursive and in fact have a finite number of different values.

For these reasons, and to reduce the complexity of our approach in general, we are not addressing Haskell's primitive types.

3.1.2 Why no polymorphism?

Polymorphically typed variables are interesting as they cannot case analysed or have induction performed on them. They represent an arbitrary type that we know nothing about. We believe however that the properties we are trying to prove, such as the idempotence of list reversal, are just as complex when proven over a non-polymorphic version of a type.

In our proofs over lists (Section 7.1) we have used lists of natural numbers as a specific non-polymorphic case. For these proofs we could replace the type `Nat` with any other in the type definition of the list and the proof would be identical.

We would endeavour to add support for polymorphism in a future version of our tool, but for now we have left it out to simplify our research.

3.2 Haskell-Core (HC)

The full definition of Haskell[24] is large and intricate; designed to be written and read by humans with many of its intricacies largely redundant from an operational point of view. To parse and analyse Haskell would be a large engineering task in itself and mostly tangental to the aims of our project.

The Glasgow Haskell Compiler[1] (normally abbreviated GHC), operates by compiling Haskell down to a much more primitive syntax, called GHC-core. This represents the "pure" functional program that underlies your Haskell code. Our tool will therefore use GHC to parse the Haskell code it is given and utilize the generated GHC-core format for its purposes.

As GHC-core is part of a real world tool it still has some unnecessary complexity from a formalisation point of view, in addition to features which are outside the scope of our project, so we will not use a direct representation when formalising our method. We have instead used a smaller syntax which we call Haskell-Core (**HC**).

3.2.1 HC Syntax

Haskell-Core is defined over a set of variables $\mathcal{V} = \{x, y, ...\}$, functions $\mathcal{F} = \{f, g, ...\}$, and constructors $\mathcal{K} = \{K, J, ...\}$. The set $\mathcal{S} = \mathcal{V} \cup \mathcal{K} \cup \mathcal{F}$ is the set of all symbols, members of which are referred to with the letter s. An **HC** program (P) consists of a list of bindings of function names to expressions (E), a set of type definitions (Δ), and a global type environment (Γ). This is formalised with the grammar in Figure 3.3. The description of the type definitions and environment is given in Section 3.2.2.

[1]http://haskell.org/ghc

Figure 3.3 Syntax of Haskell-Core (**HC**)

Program

P	$::=$	$\langle\ (f = E)^*, \Delta, \Gamma\ \rangle$	

Expression

E	$::=$	s	Symbol
	$\vert$	$(E_1 \cdot E_2)$	Application
	$\vert$	$\lambda x.E$	Abstraction
	$\vert$	$let\ f = E_1\ in\ E_2$	Definition
	$\vert$	$case\ E\ of\ (\kappa_i \rightarrow E_i)^*$	Pattern matching

Constructor term

κ	$::=$	$(\kappa \cdot x)$	Constructor term application
	$\vert$	K	Constructor

One noteworthly omission from GHC-core in **HC** is that of literals: constant strings and numbers one finds in most programs, like `3`, an integer literal, and `"Hello World"`, a string literal. These are members of primitive datatypes, that have no natural representation as a recursive datatype, and so we they have not been addressed in our project (see Section 3.1.1).

Another feature left out is the usage of types as expressions, in order to bind the type of a polymorphic function. As an example, the expression `id 3` in Haskell is actually `id Int 3`, as the polymorphic type `a` for `id :: a -> a` must first be bound to `Int` before it can be given an `Int` argument. These are a feature of polymorphism which is not covered by our project (see Section 3.1.2).

3.2.2 HC Typing

Types in **HC** are ranged over by simple type names $\mathcal{T} = \{A, B, ...\}$ and defined over T, which is either a simple type or arrow/function type, as defined in Figure 3.4.

Figure 3.4 Types in **HC**

T	$::=$	$(T_1 \to T_2)$	Arrow/Function Type
	$\mid$	$\mathcal{T}$	Simple Type

The typing of an **HC** program is given over type definitions Δ and a type environment Γ.

Type definitions (Δ)

The type definitions give the constructors for each type variable.

$$\Delta \in (\mathcal{T} \to \wp(\mathcal{K}))$$

Type environment (Γ)

The type environment defines the type of all local symbols.

$$\Gamma \in (\mathcal{S} \to T)$$

HC Expression Typing Rules

The rules in Figure 3.5 define the type of an **HC** expression under its program's type definitions and type environment. $\Delta, \Gamma \vdash E : T$ means the expression E has type T under the type defintions in Δ and type environment Γ.

Figure 3.5 Rules for typing **HC** expressions

$$\frac{T = (\Gamma s)}{\Delta, \Gamma \vdash s : T} \qquad \frac{\Delta, \Gamma \vdash E_1 : (T_1 \to T_2) \qquad \Delta, \Gamma \vdash E_2 : T_1}{\Delta, \Gamma \vdash (E_1 \cdot E_2) : T_2}$$

$$\frac{\Delta, \Gamma \cup \{x \mapsto T_1\} \vdash E : T_2}{\Delta, \Gamma \vdash (\lambda x.E) : (T_1 \to T_2)} \qquad \frac{\Delta, \Gamma \vdash f : T_1 \qquad \Delta, (\Gamma \cup \{f \mapsto T_1\}) \vdash E_2 : T_2}{\Delta, \Gamma \vdash (let\ f = E_1\ in\ E_2) : T_2}$$

$$\frac{\begin{array}{c}\forall(\kappa_i \to E_i) \in C : \\ [\Delta, (\Gamma \cup (typecons\ (\Delta(leftmost\ \kappa_i))\ \kappa_i)) \vdash E_i : T]\end{array}}{\Delta, \Gamma \vdash (case\ E\ of\ C) : T}$$

The function *leftmost* returns the inner left-most expression of an application expression and is defined in Figure 3.6.

Figure 3.6 Definition of *leftmost*

$$leftmost : E \to E$$

$$\begin{array}{lcl} leftmost\ E & = & E \\ leftmost\ (E_1 \cdot E_2) & = & leftmost\ E_1 \end{array}$$

The function *typecons* returns the typing of every variable in the constructor term κ, by recursing on the type of the constructor (T). This is defined in Figure 3.7.

Figure 3.7 Definition of *typecons*

$$typecons : \kappa \to T \to \Gamma$$

$$\begin{array}{lcl} typecons\ K\ T & = & \emptyset \\ typecons\ (\kappa \cdot x)\ (T_1 \to T_2) & = & (typecons\ (\kappa, T_2)) \cup \{x \mapsto T_1\} \end{array}$$

3.2.3 Operational semantics of HC

The execution of **HC** programs is defined by the **HC** reduction operation on expressions in a program P:

$$\overset{P}{\rightarrow}_{\mathbf{HC}} \subseteq E \times E$$

The inference rules defining the behaviour of $\overset{P}{\rightarrow}_{\mathbf{HC}}$ are given in Figure 3.8. See the earlier Figure 3.6 for the definition of *leftmost*.

Note that $\overset{P}{\rightarrow}_{\mathbf{HC}}$ can be applied anywhere inside an expression rather than just on the outermost expression. So the E in *case E of C* could be reduced before reducing the rest of the *case* expression.

Figure 3.8 Definition of the operational semantics of **HC**

$$\frac{}{((\lambda x.E_1)\cdot E_2) \overset{P}{\rightarrow}_{\mathbf{HC}} E_1[E_2/x]} \qquad \frac{}{let\ f = E_1\ in\ E_2 \overset{P}{\rightarrow}_{\mathbf{HC}} E_2[E_1/f]}$$

$$\frac{(\kappa \rightarrow E_2) \in C \qquad (leftmost\ E_1) = (leftmost\ \kappa)}{case\ E_1\ of\ C \overset{P}{\rightarrow}_{\mathbf{HC}} case\ E_1\ of\ \{\kappa \rightarrow E_2\}}$$

$$\frac{}{case\ (E_1 \cdot E_2)\ of\ \{(\kappa \cdot x) \rightarrow E_3\} \overset{P}{\rightarrow}_{\mathbf{HC}} case\ E_1\ of\ \{\kappa_2 \rightarrow E_3[E_2/x]\}}$$

$$\frac{}{case\ K\ of\ \{K \rightarrow E\} \overset{P}{\rightarrow}_{\mathbf{HC}} E} \qquad \frac{(x = E) \in P}{x \overset{P}{\rightarrow}_{\mathbf{HC}} E}$$

If our representation of **HC** is correct it it implies that if P is the equivalent **HC** program of a Haskell program compiled to GHC-core, and $E_1 \overset{P}{\twoheadrightarrow}_{\mathbf{HC}} E_2$ and there is no E_3 such that $E_2 \overset{P}{\rightarrow}_{\mathbf{HC}} E_3$ then the in the Haskell program for P the execution of E_1 will give the result E_2. Where $E_1 \overset{P}{\twoheadrightarrow}_{\mathbf{HC}} E_2$ means E_1 reduces to E_2 under P in zero or more steps.

3.2.4 HC Examples

In Figure 3.9 we give the Haskell function for addition of the natural (Peano) numbers, along with their definition as a recursive datatype. In Figure 3.10 we have the **HC** program for this Haskell code.

Figure 3.9 Haskell definition of natural numbers and their addition

```
data Nat = Zero | Succ Nat

add Zero y = y
add (Succ x) y = Succ (add x y)
```

Figure 3.10 HC definition of the natural numbers and their addition

$$E_{add} = \lambda x.\lambda y.case\ x\ of\ \{\ Zero \to y,\ Succ\ x' \to Succ\ (add\ x'\ y)\ \}$$

$$\Delta_{add} = \{\ Nat \mapsto (Zero, Succ)\ \}$$

$$\begin{aligned}\Gamma_{add} = \{\ & add \mapsto (Nat \to (Nat \to Nat)),\\ & Zero \mapsto Nat,\\ & Succ \mapsto (Nat \to Nat)\ \}\end{aligned}$$

$$P_{add} = \langle\ \{\ add = E_{add}\ \}, \Delta_{add}, \Gamma_{add}\ \rangle$$

3.3 Non-termination and infinite values

As Haskell is a Turing Complete language we can express functions that will not terminate for certain arguments. Hence **HC** can have expressions that do not converge with respect to its operational semantics. Non-termination is a complex issue in itself so in the work detailed here we have given a method that is only defined for functions that always converge. Note that when a function throws an error it is in fact converging to an error value, so this is not an issue.

Haskell also allows one to create infinitely large values by allowing the definition of a value to refer to itself. As an example, Figure 3.11 defines the infinitely large natural number `infinity`.

Figure 3.11 Infinity

```
infinity = let x = Succ x in x
```

Structural induction, as used by our tool, is no longer sound for these "recursive" values, and deals only with finitely large variables. This is because structural induction is a specific form of well-founded induction, using the well-founded ordering of definite sub-terms (see Figure 4.7). However, this ordering is no longer well-founded in the presence of infinite data structures. Taking as an example the code in 3.12, a set containing the values a and b does not have a minimal element, as each member is a sub-term of the other.

Figure 3.12 Mutually recursive data structures

```
data D = K D | C D

a = K b
b = C a
```

Chapter 4

Function Logic

Function Logic (**FL**) is our attempt to create a logic into which functional programs could be translated, and any properties we wish to prove about them be expressed.

Detailed in this section is the syntax of Function Logic (Section 4.1) along with its semantics (Section 4.2) and typing (Section 4.3), along with some examples of formulas in **FL** (Section 4.4). We then give a quick definition of sub-terms in **FL** (Section 4.6) and use this in a description of well-formedness for **FL** formulas (Section 4.7).

We follow on by explaining the necessity of constrained equality in the conditions of **FL** sentences (Section 4.5). We then give the formal inference rules by which one can prove a property to be true in Function Logic (Section 4.8), and then give two example derivations created using these rules, one of a proof and the other of a disproof (Section 4.9). We finish off by describing a complete encoding of Haskell-Core into Function Logic (Section 4.10).

Given a Haskell Program H and a property we wish to prove about it φ_γ, Zeno will first translate H into an **FL** formula Φ_H before then applying the **FL** rules in a systematic way in order to try and prove that Φ_H implies φ_γ. This will show whether the definition of the Haskell program implies the given property to be true.

4.1 FL Syntax

Function Logic is the underlying representation that our tool attempts to solve formulas in. It is defined over a set of variables $\mathcal{V} = \{x, y, ...\}$, functions

$\mathcal{F} = \{f, g, ...\}$, and constructors $\mathcal{K} = \{K, J, ...\}$. The set $\mathcal{S} = \mathcal{V} \cup \mathcal{K} \cup \mathcal{F}$ is the set of all symbols, members of which are referred to with the letter s. This is exactly the same as in the definition of **HC** (Section 3.2.1). The full syntax of **FL** is described in Figure 4.1. Note that we will often abbreviate $\tau_1 = \tau_2 \leftarrow \emptyset$ to $\tau_1 = \tau_2$. Note also that bracketing is often simplified and term application symbols removed, as they would be in a real functional language, so $((f \cdot (g \cdot x)) \cdot y) \cdot z)$ might be given as $f\ (g\ x)\ y\ z$.

Figure 4.1 Syntax of **FL**

Formula			
Φ	$::=$	φ^*	Set of sentences
Sentence			
φ	$::=$	$\tau_1 = \tau_2 \leftarrow X$	Conditional equality
	$\mid$	$\forall x^T.\varphi$	Quantifying a variable
Conditions			
X	$::=$	$(\tau \overset{\rightharpoonup}{=} \kappa)^*$	Pattern matching
Term			
τ	$::=$	$(\tau_1 \cdot \tau_2)$	Term application
	$\mid$	s	Variable, function or constructor
Constructor term			
κ	$::=$	$(\kappa \cdot x)$	Constructor term application
	$\mid$	K	Constructor symbol

4.2 FL Semantics

Given in Figure 4.2 is our encoding $[[\varphi]]^{FL}_{FOL}$ of **FL** sentences into first-order logic, ignoring all typing rules and equality axioms. Note that $\overset{\rightharpoonup}{=}$ is just equality that has had its right argument restricted to constructor terms. Note also that $(\leftarrow)$ represents logic implication and therefore sentences (φ's) are just clauses with exactly one positive literal, and whose only predicate is equality. $[[X]]^{FLX}_{FOL}$ is the auxiliary encoding for conditions.

Figure 4.2 Encoding of **FL** sentences into first-order logic

$$
\begin{array}{lcl}
[[\forall x^T.\varphi]]^{FL}_{FOL} & = & \forall x^T : [[\varphi]]^{FL}_{FOL} \\
[[\tau_1 = \tau_2 \leftarrow X]]^{FL}_{FOL} & = & \forall\{x \mid (\tau \overset{\rightharpoonup}{=} \kappa) \in X \wedge x \in \kappa\}.([[X]]^{FLX}_{FOL} \Rightarrow (\tau_1 = \tau_2)) \\
 & & \\
[[X_1 \cup X_2]]^{FLX}_{FOL} & = & [[X_1]]^{FLX}_{FOL} \wedge [[X_2]]^{FLX}_{FOL} \\
[[\{\ \tau \overset{\rightharpoonup}{=} \kappa\ \}]]^{FLX}_{FOL} & = & \tau = \kappa
\end{array}
$$

4.3 FL Typing

The definition of types in **FL** along with type definitions (Δ) and type environments (Γ) is the same as that in **HC**, and is detailed in Section 3.2.2.

The two rules in Figure 4.3 define the type of an **FL** term in the given type environment.

Figure 4.3 Typing terms in **FL**

$$
\frac{T = (\Gamma s)}{\Gamma \vdash s : T} \qquad \frac{\Gamma \vdash E_1 : (T_1 \to T_2) \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash (E_1 \cdot E_2) : T_2}
$$

4.4 FL Examples

Given in Figure 4.4 is the Haskell function `add` encoded in Function Logic in two equivalent ways, the first without conditions and the second with. In Figure 4.5 is the type environment and definitions for either version of `add`.

Figure 4.4 Definitions of the addition function in **FL**

$$\begin{aligned}
\Phi_{add_1} \; = \; \{ \; & \forall v^{Nat}.(add\ Zero\ v) = v, \\
& \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))\} \\
\Phi_{add_2} \; = \; \{ \; & \forall u^{Nat}.\forall v^{Nat}.(add\ u\ v) = v \leftarrow \{u \overset{\rightharpoonup}{=} Zero\}, \\
& \forall u^{Nat}.\forall v^{Nat}.(add\ u\ v) = (Succ\ (add\ u'\ v)) \leftarrow \{u \overset{\rightharpoonup}{=} (Succ\ u')\}\}
\end{aligned}$$

Figure 4.5 Typing for the addition function in **FL**

$$\begin{aligned}
\Delta_{add} \; = \; \{ \; & Nat \mapsto \{\ Zero, Succ\ \}\ \} \\
\Gamma_{add} \; = \; \{ \; & add \mapsto (Nat \to (Nat \to Nat)), \\
& Zero \mapsto Nat, \\
& Succ \mapsto (Nat \to Nat)\}
\end{aligned}$$

4.5 Why constrain equality in conditions ($\overset{\rightharpoonup}{=}$)?

This section addresses the question as to why equality at the head of a sentence is between two arbitrary terms, but in a condition is constrained to having a constructor term to its right, along with various conditions about acyclicity in the definition of well-formedness.

The reason for this is that, even though the encoding in first-order logic transforms ($\overset{\rightharpoonup}{=}$) into equality, it is actually a representation of the pattern matching found in functional languages. When we say $\tau \overset{\rightharpoonup}{=} (K\ x)$ we mean that the execution of τ converges to constructor K, where the first argument is bound to x. We are not checking whether the first argument is equal to some given x, but defining x to be the value of this argument. The value for x can therefore not be defined anywhere else in a well-formed sentence.

This restriction greatly simplifies proving properties, as every condition can be fulfilled using the case-completion or induction rules (see Section 4.8.7 and Section 4.8.8 respectively).

4.6 Subterms in FL

In Figure 4.6 we give the rules governing the operator $\subseteq_{st}$, which defines the sub-terms of a term. For example $x \subseteq_{st} (f\ (g\ x)\ (h\ y))$, and $(g\ x) \subseteq_{st} (f\ (g\ x)\ (h\ y))$ but also $(f\ (g\ x)\ (h\ y)) \subseteq_{st} (f\ (g\ x)\ (h\ y))$.

Figure 4.6 Subterms in **FL**

$$\frac{}{\tau \subseteq_{st} \tau} \qquad \frac{\tau \subseteq_{st} \tau_1}{\tau \subseteq_{st} (\tau_1 \cdot \tau_2) \qquad \tau \subseteq_{st} (\tau_2 \cdot \tau_1)}$$

In Figure 4.7 we give the definition of definite sub-terms, that is to say terms that are in the arguments of a term, but not in any of their arguments' arguments, and not the term itself. So $x \not\sqsubset_{st} x$ and $x \not\sqsubset_{st} (f\ (g\ x))$, but $x \sqsubset_{st} ((f\ x)\ y)$. It is worth noting that this relationship is the well-founded ordering using for our structural induction step in Section 4.8.8.

Figure 4.7 Subterms in **FL**

$$\frac{}{\tau_2 \sqsubset_{st} (\tau_1 \cdot \tau_2)} \qquad \frac{\tau \sqsubset_{st} \tau_1}{\tau \sqsubset_{st} (\tau_1 \cdot \tau_2)}$$

4.7 Well-formedness for FL

Not all sentences in **FL** are well-formed. The relationship $\overset{\rightharpoonup}{=}$ is analogous to pattern matching in **HC**, meaning that all variables on the right are defined by those on the left, and hence are dependent upon them. This transitive *dependence* relationship between variables cannot be cyclic at any point or an attempt at a proof using such a formula may result in an infinite loop. $X \vdash x \rhd x'$ means x *defines* x' in the set of conditions X and is defined in Figure 4.8.

Figure 4.8 Variable dependence relationship for well-formedness

$$\frac{x \subseteq_{st} \tau \qquad x' \subseteq_{st} \kappa \qquad (\tau \overset{\rightharpoonup}{=} \kappa) \in X}{X \vdash x \rhd x'} \qquad \frac{X \vdash x \rhd x' \qquad X \vdash x' \rhd x''}{X \vdash x \rhd x''}$$

A well-formed constructor term κ satisifies $wf(\kappa)$, as defined in Figure 4.9. Informally $wf(\kappa)$ means that κ does not contain any variable more than once, so all the variables of a constructor term should be unique. The reason for this is that when we say $(f\ x) \overset{\rightharpoonup}{=} (K\ y\ z)$ we are defining $(f\ x)$ to return a value of constructor K, where y and z bind to the values of the arguments of K. That is to say $(f\ x)$ defines what y and z are. If we were instead to say $(f\ x) \overset{\rightharpoonup}{=} (K\ y\ y)$ we are saying that $(f\ x) \overset{\rightharpoonup}{=} (K\ y\ z)$ and that $y = z$, which adds an unconstrained equality as a condition. This problem is explained further in Section 4.5.

Figure 4.9 Well-formedness for constructor terms

$$\frac{}{wf(K)} \qquad \frac{wf(\kappa) \qquad x \notin \kappa}{wf(\kappa \cdot x)}$$

A well-formed sentence φ satisfies $wf(\varphi)$, as defined in Figure 4.10, and Figure 4.12. Note that the function *leftmost* returns the inner left-most symbol of a term and is given in Figure 4.11.

Figure 4.10 Well-formedness for unquantified **FL** sentences

$$\frac{\begin{array}{c}\exists f \in \mathcal{F} : (f = (leftmost\ \tau_1) \vee (f = (leftmost\ \tau_2))) \\ \forall(\tau \overset{\rightharpoonup}{=} \kappa) \in X : (wf(\kappa) \wedge (\forall x \subseteq_{st} \tau, x' \subseteq_{st} \kappa : X \nvdash x' \rhd x))\end{array}}{wf(\tau_1 = \tau_2 \leftarrow X)}$$

In Figure 4.10, $\exists f \in \mathcal{F} : (f = (leftmost\ \tau_1) \vee (f = (leftmost\ \tau_2)))$ enforces that at least one side of the equality being defined has a function symbol as its inner leftmost. Meaning that the equality defines the behaviour of a function. Otherwise we are defining the behaviour of a variable or a constructor, which is either meaningless or contradictory.

$X \nvdash x' \rhd x$ means we cannot show that $X \vdash x' \rhd x$. This enforces the anti-symmetry and hence acyclicity of $\rhd$ within the sentence.

Figure 4.11 *leftmost* symbol of a term

$$\begin{array}{lcl} leftmost\ s & = & s \\ leftmost\ (\tau_1 \cdot \tau_2) & = & leftmost\ \tau_1 \end{array}$$

Figure 4.12 Well-formedness for quantifying **FL** sentences

$$\frac{n \geq 0 \qquad \varphi = \forall x_1^{T_1}.\forall x_2^{T_2}...\forall x_n^{T_n}.\tau_1 = \tau_2 \leftarrow X \\ wf(\varphi) \qquad \forall(\tau \overset{\rightharpoonup}{=} \kappa) \in X : x \notin \kappa \qquad \nexists i \in [1..n] : x = x_i}{wf(\forall x^T.\varphi)}$$

In Figure 4.12 we define that a well-formed sentence can only quantify over variables that do not exist in a constructor term in one of its conditions. It also does not quantify a variable it has already quantified.

It is important to note that Function Logic that has been translated from Haskell-Core will automatically be well-formed. Well-formedness may even allow an encoding from **FL** back into **HC**, but we have not investigated this. It is fairly easy to see though that Function Logic that is not well formed will have no natural representation in **HC**.

4.8 FL Rules

In this section we define the rules one can apply in order to derive a proof or disproof in Function Logic.

Proof in **FL** is represented as implication:

$$\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi_\gamma$$

This is equivalent to the following classical formula under the type definitions Δ and environment Γ:

$$\left(\bigwedge_{i=1}^{n} [[\varphi_i]]_{FOL}^{FL} \right) \Rightarrow [[\varphi_\gamma]]_{FOL}^{FL}$$
$$where\ \Phi_\alpha = \{\varphi_1, \varphi_2, ..., \varphi_n\}$$

As in classical implication, Φ_α is referred to as the antecedent (or assumption), and φ_γ as the consequent (or goal).

In addition to proving formulas we can also create disproofs using these rules. A disproven **FL** implication is expressed as:

$$\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \varphi_\gamma$$

This represents the following classical formula:

$$\left(\bigwedge_{i=1}^{n} [[\varphi_i]]^{FL}_{FOL}\right) \Rightarrow \neg[[\varphi_\gamma]]^{FL}_{FOL}$$
$$where\ \Phi_\alpha = \{\varphi_1, \varphi_2, ..., \varphi_n\}$$

In the case of a universally quantified equality property a disproof would indicate that we have found an instance of its variables that falsifies the equality.

It it worth noting that if we have failed to find a proof in **FL** we have not necessarily found a disproof, or vice-versa. There is no excluded middle in **FL** as there is in classical logic.

The following rules represent the steps one can perform in order to prove or disprove an **FL** implication.

4.8.1 Cumulative transitivity

Just as with implication in classical logic, implication in **FL** is cumulatively transitive, both for proof and disproof. This is formally defined in Figure 4.13.

Figure 4.13 Cumulative transitivity of **FL** implication

$$(\textsc{cut1})\ \frac{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi_1 \qquad \Delta, \Gamma, (\Phi_\alpha \cup \{\varphi_1\}) \vdash_{\mathbf{FL}} \varphi_2}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi_2}$$

$$(\textsc{cut2})\ \frac{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi_1 \qquad \Delta, \Gamma, (\Phi_\alpha \cup \{\varphi_1\}) \nvdash_{\mathbf{FL}} \varphi_2}{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \varphi_2}$$

4.8.2 Quantifiers

The (∀I) (read "forall introduction") rules, defined in Figure 4.14, allows you to add a quantifier to a proven or disproven consequent, as long as the variable quantified does not exist in the antecedent.

Figure 4.14 Quantifier introduction in **FL**

$$(\forall I1)\ \frac{\Delta, (\Gamma \cup \{x \mapsto T\}), \Phi_\alpha \vdash_{\mathbf{FL}} \varphi \qquad x \notin vars(\Phi_\alpha)}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \forall x^T.\varphi}$$

$$(\forall I2)\ \frac{\Delta, (\Gamma \cup \{x \mapsto T\}), \Phi_\alpha \nvdash_{\mathbf{FL}} \varphi \qquad x \notin vars(\Phi_\alpha)}{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \forall x^T.\varphi}$$

($\forall$E) (read "forall elimination"), defined in Figure 4.15, allows you to remove a quantifier and instantiate it with any term. From this rule we can create a specific instance of a universally quantified sentence.

Figure 4.15 Quantifier elimination in **FL**

$$(\forall E)\ \frac{\Gamma \vdash \tau : T}{\Delta, (\Gamma \cup \{x \mapsto T\}), (\Phi_\alpha \cup \{\forall x^T.\varphi\}) \vdash_{\mathbf{FL}} \varphi[\tau/x]}$$

4.8.3 Adding and removing conditions

(FULFIL) allows you to remove a condition from a sentence as long as the condition is true in the rest of antecedent, as defined in Figure 4.16.

Figure 4.16 Fulfilling conditions in **FL**

$$(\textsc{fulfil})\ \frac{}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_1 = \tau_2 \leftarrow (X \cup \{\tau \overset{\rightharpoonup}{=} \kappa\}), \tau = \kappa\}) \vdash_{\mathbf{FL}} \tau_1 = \tau_2 \leftarrow X}$$

(EXPAND), defined in Figure 4.16, specifies that one can replace a constructor term (κ) with an arbitrary term (τ) by adding a condition ($\tau \overset{\rightharpoonup}{=} \kappa$), as long as κ is well-formed and does not create a cycle in the conditions, and every variable in the constructor term does not exist in the rest of antecedent, i.e. its value is not constrained in any way. See Section 4.7 for the definition of $X \nvdash x' \triangleright x$ and $wf(\kappa)$.

Figure 4.17 Expanding sentences in **FL**

$$(\textsc{expand})\ \frac{\begin{array}{cc}(\kappa \subseteq_{st} \tau_1 \lor \kappa \subseteq_{st} \tau_2) & wf(\kappa) \\ \forall x_\tau \subseteq_{st} \tau . \forall x_\kappa \subseteq_{st} \kappa : X \nvdash x_\kappa \rhd x_\tau & \forall x \subseteq_{st} \kappa : x \notin vars(\Phi_\alpha)\end{array}}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_1 = \tau_2 \leftarrow X\}) \vdash_{\mathbf{FL}} \tau_1[\tau/\kappa] = \tau_2[\tau/\kappa] \leftarrow X[\tau/\kappa] \cup \{\tau \overset{\rightharpoonup}{=} \kappa\}}$$

4.8.4 Equality

In Figure 4.18 we have the various rules surrounding equality in **FL**.

Figure 4.18 Equality rules for **FL** proof

$$(\textsc{eq-reflexive})\ \frac{}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \tau = \tau}$$

$$(\textsc{eq-symmetric})\ \frac{}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_1 = \tau_2\}) \vdash_{\mathbf{FL}} \tau_2 = \tau_1}$$

$$(\textsc{eq-transitive})\ \frac{}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_1 = \tau_2, \tau_2 = \tau_3\}) \vdash_{\mathbf{FL}} \tau_1 = \tau_3}$$

$$(\textsc{eq-left})\ \frac{}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_1 = \tau_1', (\tau_1 \cdot \tau_2) = \tau_3\}) \vdash_{\mathbf{FL}} (\tau_1' \cdot \tau_2) = \tau_3}$$

$$(\textsc{eq-right})\ \frac{}{\Delta, \Gamma, (\Phi_\alpha \cup \{\tau_2 = \tau_2', (\tau_1 \cdot \tau_2) = \tau_3\}) \vdash_{\mathbf{FL}} (\tau_1 \cdot \tau_2') = \tau_3}$$

4.8.5 Disproof

The first disproof rule defined here is for two terms in which both of their left-most symbols are constructors, but different constructors, defined in Figure 4.19. For example *Zero* is trivially unequal to *Succ* x for any value of x. The function *leftmost* is defined earlier in Figure 4.11.

Figure 4.19 Base unequality in **FL**

$$(\textsc{unequal1})\ \frac{K = (\mathit{leftmost}\ \tau_1) \qquad J = (\mathit{leftmost}\ \tau_2) \qquad K \neq J}{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \tau_1 = \tau_2}$$

The second disproof rule is for two application terms for which one side is identical, but the other side is provably unequal. This we have named "recursive unequality", and is given in Figure 4.20. *Succ* (*Succ Zero*) $\neq$ *Succ Zero*, i.e. $2 \neq 1$ is a trivial example of this. It would also give (*f Zero*) *Zero* $\neq$ (*f* (*Succ x*)) *Zero*.

Figure 4.20 Recursive unequality in **FL**

$$(\textsc{unequal2})\ \frac{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \tau_1 = \tau_2}{\begin{array}{c}\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} (\tau_1 \cdot \tau) = (\tau_2 \cdot \tau) \\ \Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} (\tau \cdot \tau_1) = (\tau \cdot \tau_2)\end{array}}$$

4.8.6 Instantiating constructors

The induction and case completion rules (Sections 4.8.8 and 4.8.7 respectively) make use of the *instantiate* function, which creates a fully instantiated constructor term from a constructor by recursing on its type and adding arguments as it goes. Along with the instantiated constructor term it returns a type environment giving the types of all the arguments of the new constructor term. This function is defined formally in Figure 4.21.

Figure 4.21 *instantiate* function for constructors

$$\mathit{instantiate} : K \to T \to (\kappa \times \Gamma)$$

$$\begin{array}{lcl}
\mathit{instantiate}\ K\ A & = & (K, \emptyset) \\
\mathit{instantiate}\ K\ (T_1 \to T_2) & = & (\kappa \cdot x, \Gamma \cup \{x \mapsto T_1\}) \\
& & \quad \mathit{where}\ (\kappa, \Gamma) = \mathit{instantiate}\ (K, T_2)
\end{array}$$

As an example, we have the instantiation of the two constructors of the natural numbers (*Zero* and *Succ*) in Figure 4.22.

Figure 4.22 Example usage of the *instantiate* function

$$\begin{array}{lcl} \textit{instantiate Zero Nat} & = & (Zero, \emptyset) \\ \textit{instantiate Succ } (Nat \to Nat) & = & ((Succ \cdot x), \{x \mapsto Nat\}) \end{array}$$

4.8.7 Case Completion

(COMPLETE-CASE1), defined in Figure 4.23, states that if an implication is proven for every possible value of a term, then it holds in general.

Figure 4.23 Rule for proof by case completion

$$(\textsc{complete-case1})\ \frac{\begin{array}{c}\Gamma \vdash \tau' : A \\ \forall K \in (\Delta A) : \Big[\Delta, (\Gamma' \cup \Gamma), (\Phi_\alpha \cup \{\tau = \kappa\}) \vdash_{\mathbf{FL}} \varphi \\ \textit{where } (\kappa, \Gamma') = (\textit{instantiate } K\ (\Gamma K))\Big]\end{array}}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi}$$

(COMPLETE-CASE2), defined in Figure 4.24, states that if an implication is disproven for every possible value of a term, then it is disproven in general.

Figure 4.24 Rule for disproof by case completion

$$(\textsc{complete-case2})\ \frac{\begin{array}{c}\Gamma \vdash \tau' : A \\ \forall K \in (\Delta A) : \Big[\Delta, (\Gamma' \cup \Gamma), (\Phi_\alpha \cup \{\tau = \kappa\}) \nvdash_{\mathbf{FL}} \varphi \\ \textit{where } (\kappa, \Gamma') = (\textit{instantiate } K\ (\Gamma K))\Big]\end{array}}{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \varphi}$$

(COUNTEREXAMPLE), defined in Figure 4.25, states that if a disproven implication holds for one paticular binding to a free variable, then it holds in general. As the variable does not exist in the antecedent, there are no rules governing its value, so it could be assigned to the constructor term

(κ) that validates the disproven implication. This constructor term κ is the counterexample we have found to the property we are trying to prove.

Figure 4.25 Finding a counterexample in **FL**

$$\textsc{(counterexample)}\ \frac{\begin{array}{c} A = (\Gamma x) \qquad x \notin \mathit{vars}(\Phi_\alpha) \\ \exists K \in (\Delta A) : \Big[\Delta, (\Gamma' \cup \Gamma), (\Phi_\alpha \cup \{\ x = \kappa\ \}) \nvdash_{\mathbf{FL}} \varphi \\ \mathit{where}\ (\kappa, \Gamma') = (\mathit{instantiate}\ K\ (\Gamma K))\Big] \end{array}}{\Delta, \Gamma, \Phi_\alpha \nvdash_{\mathbf{FL}} \varphi}$$

4.8.8 Induction

The (INDUCTION) rule, given in Figure 4.26, defines proof by structural induction on recursive datatypes in **FL**.

To inductively prove that $\tau_1 = \tau_2$ we take an arbitrary τ', a term in either τ_1 or τ_2, and prove every inductive case for it. An inductive case being a substitution of τ' for one of the constructors of its return type into $\tau_1 = \tau_2$, given as an inductive assumption the substitution of τ' for the recursive cases of that constructor into $\tau_1 = \tau_2$.

Figure 4.26 Structural induction in Function Logic

$$\textsc{(induction)}\ \frac{\begin{array}{c} (\tau' \subseteq_{st} \tau_1 \lor \tau' \subseteq_{st} \tau_2) \qquad \Gamma \vdash \tau' : A \\ \forall K \in (\Delta A) : \\ \Big[\Delta, (\Gamma' \cup \Gamma), (\Phi_\alpha \cup \{\ (\tau_1 = \tau_2)[x/\tau'] \mid x \in \mathit{xs}\ \}) \vdash_{\mathbf{FL}} (\tau_1 = \tau_2)[\kappa/\tau'] \\ \mathit{where} \\ (\kappa, \Gamma') = (\mathit{instantiate}\ K\ (\Gamma K)) \qquad \mathit{xs} = \{x \in \mathit{dom}(\Gamma') \mid (\Gamma'\ x) = A\}\Big] \end{array}}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \tau_1 = \tau_2}$$

For each constructor of the same type (A) as the subterm (τ') we have its instantiation (κ) and the typing of all its variables (Γ'). xs is the list of all recursively typed variables of that constructor, that is all variables that have the same type as the type of the constructor (A). For each of these recursively typed variables we can add an inductive hypothesis $(\tau_1 = \tau_2)[x/\tau']$. Note that the variables in the new constructor term generated by *instantiate* can be ones already existing in the antecedent, indeed this fact is necessary for the proof given in Figure 4.29.

As an example we will describe an inductive step on the variable x in the sentence $add\ x\ Zero = x$. In Figure 4.27 we give the assignments to every variable in our (INDUCTION) inference rule, to better highlight how this rule is applied. As one can see there are two new implications that must be proven in order to complete this rule, corresponding to the two constructors for Nat ($Succ$ and $Zero$). The first branch is the "base case", $\kappa = Zero$, where we must prove $\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} add\ Zero\ Zero = Zero$. The second branch is our "inductive case", $\kappa = (Succ\ x')$, for some new x'. As x' is of the same type as x it can be rewritten for x and added as an inductive assumption ($add\ x'\ Zero = x'$) to the antecedent. This means we must prove $\Delta, (\{x' \mapsto Nat\} \cup \Gamma), (\Phi_\alpha \cup \{add\ x'\ Zero = x'\} \vdash_{\mathbf{FL}} add\ (Succ\ x')\ Zero = (Succ\ x')$ for this branch.

Figure 4.27 FL induction example

$$\frac{\begin{array}{c} (x \subseteq_{st} (add\ x\ Zero) \vee x \subseteq_{st} x) \qquad \Gamma \vdash x : Nat \\ \\ \forall K \in \{\ Zero, Succ\ \} : \\ \\ [\ K = Zero\ ] \\ \Big[\Delta, (\emptyset \cup \Gamma), (\Phi_\alpha \cup \emptyset) \vdash_{\mathbf{FL}} add\ Zero\ Zero = Zero \\ where \\ (\kappa, \Gamma') = (Zero, \emptyset) \qquad xs = \emptyset\Big] \\ \\ [\ K = Succ\ ] \\ \Big[\Delta, (\{x' \mapsto Nat\} \cup \Gamma), (\Phi_\alpha \cup \{add\ x'\ Zero = x'\} \vdash_{\mathbf{FL}} add\ (Succ\ x')\ Zero = (Succ\ x') \\ where \\ (\kappa, \Gamma') = (Succ\ x', \{x' \mapsto Nat\}) \qquad xs = \{x'\}\Big] \end{array}}{\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} add\ x\ Zero = x}$$

4.9 Example derivations in FL

In this section we give a proof, using the above rules, that shows the definition for `add` (Figure 3.9) implies that $add\ Zero\ x = x$ for any x, i.e. $\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \forall x^{Nat}.add\ Zero\ x = x$. Using the same antecedent we also give a disproof of $add\ x\ x = x$, i.e. $\Delta_{add}, \Gamma_{add}, \Phi_{add} \nvdash_{\mathbf{FL}} \forall x^{Nat}.add\ x\ x = x$. See Section 4.4 for the definitions of Δ_{add}, Γ_{add} and $\Phi_{add} \equiv \Phi_{add_1}$.

In Figure 4.28 we have trivially correct implications created using ($\forall$E) (Figure 4.15). In our main proofs we can then use cumulative transitivity (Figure 4.13) to augment our antecedent. That is to say we have that if $\Delta_{add}, \Gamma_{add}, \Phi'_{add} \vdash_{\mathbf{FL}} \varphi_\gamma$ then $\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \varphi_\gamma$, and if $\Delta_{add}, \Gamma_{add}, \Phi''_{add} \nvdash_{\mathbf{FL}} \varphi_\gamma$ then $\Delta_{add}, \Gamma_{add}, \Phi_{add} \nvdash_{\mathbf{FL}} \varphi_\gamma$.

Figure 4.28 Simple implications to augment our antecedent

$$\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \mathit{add}\ \mathit{Zero}\ \mathit{Zero} = \mathit{Zero}$$
$$\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \mathit{add}\ (\mathit{Succ}\ x')\ \mathit{Zero} = \mathit{Succ}\ (\mathit{add}\ x'\ \mathit{Zero})$$

$$\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \mathit{add}\ (\mathit{Succ}\ \mathit{Zero})\ (\mathit{Succ}\ \mathit{Zero}) = \mathit{Succ}\ (\mathit{add}\ \mathit{Zero}\ (\mathit{Succ}\ \mathit{Zero}))$$
$$\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \mathit{add}\ \mathit{Zero}\ (\mathit{Succ}\ \mathit{Zero}) = \mathit{Succ}\ \mathit{Zero}$$

$$\Phi'_{add} = \Phi_{add} \cup \{ \quad \mathit{add}\ \mathit{Zero}\ \mathit{Zero} = \mathit{Zero}, \quad \mathit{add}\ (\mathit{Succ}\ x')\ \mathit{Zero} = \mathit{Succ}\ (\mathit{add}\ x'\ \mathit{Zero})\ \}$$

$$\Phi''_{add} = \Phi_{add} \cup \{ \quad \mathit{add}\ (\mathit{Succ}\ \mathit{Zero})\ (\mathit{Succ}\ \mathit{Zero}) = \mathit{Succ}\ (\mathit{add}\ \mathit{Zero}\ (\mathit{Succ}\ \mathit{Zero})), \quad \mathit{add}\ \mathit{Zero}\ (\mathit{Succ}\ \mathit{Zero}) = \mathit{Succ}\ \mathit{Zero}\ \}$$

The derivation for our example proof is given in Figure 4.29 and the example disproof in Figure 4.30. Note that we have used (EQ) to represent multiple applications of the equality rules, combined with cumulative transitivity where necessary. Also, (REF) is shorthand for the rule (EQ-REFLEXIVE), and (UNEQ) for the application of the unequality rules.

One thing to note is that in our disproof of $\mathit{add}\ x\ x = x$ the values introduced by the application of (COUNTEREXAMPLE) give us a counterexample to our property, i.e. $x = (\mathit{Succ}\ \mathit{Zero})$.

Figure 4.29 Proof example using **FL** rules

$$
(\forall\mathrm{I}1)\ \dfrac{(\textsc{cut}1)\ \dfrac{(\textsc{induction})\ \dfrac{(\textsc{eq})\ \dfrac{(\textsc{ref})\ \dfrac{}{\Delta_{add},\Gamma_{add},(\Phi'_{add}\cup\{x=(Succ\ x')\})\vdash_{\mathbf{FL}}(Succ\ Zero)=(Succ\ Zero)}}{\Delta_{add},\Gamma_{add},(\Phi'_{add}\cup\{x=(Succ\ x')\})\vdash_{\mathbf{FL}} add\ x\ Zero=x}\qquad (\textsc{eq})\ \dfrac{(\textsc{ref})\ \dfrac{}{\Delta_{add},\Gamma_{add},(\Phi'_{add}\cup\{x=Zero\})\vdash_{\mathbf{FL}} Zero=Zero}}{\Delta_{add},\Gamma_{add},(\Phi'_{add}\cup\{x=Zero\})\vdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi'_{add}\vdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi_{add}\vdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi_{add}\vdash_{\mathbf{FL}}\forall x^{Nat}.add\ x\ Zero=x}
$$

Figure 4.30 Disproof example using **FL** rules

$$
(\forall\mathrm{I}2)\ \dfrac{(\textsc{cut}2)\ \dfrac{(\textsc{counterexample})\ \dfrac{(\textsc{counterexample})\ \dfrac{(\textsc{eq})\ \dfrac{(\textsc{uneq})\ \dfrac{}{\Delta_{add},\Gamma_{add},(\Phi''_{add}\cup\{x=(Succ\ x'),x'=Zero\})\nvdash_{\mathbf{FL}} Succ\ (Succ\ Zero)=(Succ\ Zero)}}{\Delta_{add},\Gamma_{add},(\Phi''_{add}\cup\{x=(Succ\ x'),x'=Zero\})\nvdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},(\Phi''_{add}\cup\{x=(Succ\ x')\})\nvdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi''_{add}\nvdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi_{add}\nvdash_{\mathbf{FL}} add\ x\ Zero=x}}{\Delta_{add},\Gamma_{add},\Phi_{add}\nvdash_{\mathbf{FL}}\forall x^{Nat}.add\ x\ Zero=x}
$$

4.10 Mapping HC to FL

Defined in Figure 4.10 is our encoding of **HC** expressiongs into **FL** formulas, where $[[\langle\tau, xs, X\rangle; E]]^{HC}_{FL}$ defines the encoding of **HC** expression E giving the value for τ under the conditions in X with free variables xs.

The typing of **HC** is identical to that of **FL** so there is no need to explictly encode it.

Figure 4.31 Encoding **HC** to **FL**

$$
\begin{array}{lcl}
[[P_1 \cup P_2]]^{HC}_{FL} & = & [[P_1]]^{HC}_{FL} \cup [[P_2]]^{HC}_{FL} \\
[[\{f = E\}]]^{HC}_{FL} & = & [[\langle f, \emptyset, \emptyset\rangle; E]]^{HC}_{FL} \\
 & & \\
[[\langle\tau, xs, X\rangle; s]]^{HC}_{FL} & = & \{\tau = s \leftarrow X\} \\
[[\langle\tau, xs, X\rangle; \lambda x.E]]^{HC}_{FL} & = & [[\langle(\tau \cdot x), (\{x\} \cup xs), X\rangle; E]]^{HC}_{FL} \\
[[\langle\tau, xs, X\rangle; (E_1 \cdot E_2)]]^{HC}_{FL} & = & \{\tau = (E_1 \cdot E_2) \leftarrow X\} \\
 & & \quad \text{where } term(E_1 \cdot E_2) \text{ holds} \\
[[\langle\tau, xs, X\rangle; (E_1 \cdot E_2)]]^{HC}_{FL} & = & [[\langle\tau, xs, X\rangle; let\ f_1 = E_1\ in\ let\ f_2 = E_2\ in\ (f_1 \cdot f_2)]]^{HC}_{FL} \\
 & & \quad \text{where } f_1 \text{ and } f_2 \text{ are fresh function symbols} \\
 & & \quad \text{and } term(E_1 \cdot E_2) \text{ does } \textbf{not} \text{ hold} \\
[[\langle\tau, xs, X\rangle; case\ s\ of\ \emptyset]]^{HC}_{FL} & = & \emptyset \\
[[\langle\tau, xs, X\rangle; case\ s\ of\ \{\kappa \to E\} \cup C]]^{HC}_{FL} & = & [[\langle\tau, xs, X \cup \{s \overset{\rightharpoonup}{=} \kappa\}\rangle; E]]^{HC}_{FL} \cup [[\langle\tau, xs, X\rangle; case\ s\ of\ C]]^{HC}_{FL} \\
[[\langle\tau, xs, X\rangle; case\ E\ of\ C]]^{HC}_{FL} & = & [[\langle\tau, xs, X\rangle; let\ f = E\ in\ case\ f\ of\ C]]^{HC}_{FL} \\
 & & \quad \text{where } f \text{ is a fresh function symbol} \\
[[\langle\tau, xs, X\rangle; let\ f = E_1\ in\ E_2]]^{HC}_{FL} & = & [[\langle\tau, xs, X\rangle; (let\ f = (\lambda x.E_1)\ in\ E_2[(f \cdot x)/f])]]^{HC}_{FL} \\
 & & \quad \text{where } x \in E_1 \text{ and } x \in xs \\
[[\langle\tau, xs, X\rangle; let\ f = E_1\ in\ E_2]]^{HC}_{FL} & = & [[\langle\tau, xs, X\rangle; E_2]]^{HC}_{FL}[E_1/f] \\
 & & \quad \text{where } \nexists y : y \in E_1 \land y \in xs \\
 & & \quad \text{and } term(E_1) \text{ holds} \\
[[\langle\tau, xs, X\rangle; let\ f = E_1\ in\ E_2]]^{HC}_{FL} & = & [[\langle g, \emptyset, \emptyset\rangle; E_1]]^{HC}_{FL} \cup [[\langle\tau, xs, X\rangle; E_2[g/f]]]^{HC}_{FL} \\
 & & \quad \text{where } \nexists y : y \in E_1 \land y \in xs \\
 & & \quad \text{and } term(E_1) \text{ does } \textbf{not} \text{ hold} \\
 & & \quad \text{and } g \text{ is a fresh function symbol}
\end{array}
$$

The predicate $term(E)$, defined in Figure 4.32, holds when something is a simple term expression, containing only variables (x) and application ($E_1 \cdot E_2$). Note that if $term(E)$ does hold then E is both **HC** code and an **FL** term, and hence can be used interchangeably as such.

Figure 4.32 Defining a term

$$\frac{}{term(x)} \qquad \frac{term(E_1) \qquad term(E_2)}{term(E_1 \cdot E_2)}$$

Something we have left out of our encoding was ensuring the fresh function symbols created were in the type environment, so we will assume that any function symbols we needed were there to begin with.

An issue with this representation is that Haskell, and so **HC**, has a notion of local variable scoping within an expression, whereas **FL** does not. This means that the same name can be used for different variables deeper in the expression if it is redefined with a lambda abstraction, case statement binding or a *let* expression. We can fix this by assuming this encoding will be used on **HC** expressions that have every variable is uniquely named, which is a trivial transformation for any **HC** expression.

One could also infer that the implicit cast from **HC** variables to **FL** ones will take into account which variable is which, rather than just relying on their names. Indeed from an implementation point of view this last method is easy, since variables have an underlying unique identifier independent of their name.

Chapter 5

Function Logic Tableau

In this chapter we detail Function Logic Tableau, a method in which we adapt the **FL** implication proof rules into an algorithm that our Zeno tool implements. An algorithm that takes an **FL** implication $(\Delta, \Gamma, \Phi_\alpha \vdash_{\mathbf{FL}} \varphi_\gamma)$ and attempts to prove or disprove it.

A tableau method generates a tree representing a proof according to certain rules. If a branch of the generated tree fulfils certain properties then it is said to "close", and represents a successful proof for that branch. If every branch of a tree closes then the whole tree closes and the entire proof is successful. For a set of inference rules it could be seen as a tree representing an upside-down derivation. Tableau methods are detailed more fully in Section 2.4.2.

In this chapter we first describe what a node of an **FL** tableau looks like and what it represents (Section 5.1). We then explain how one takes the implication we are trying to prove and creates the head node of the tableau from it (Section 5.2). Then we describe the two rules we can apply at any node of the tree to generate new branches, equality (Section 5.3) and induction (Section 5.4). We then detail how we can close a branch having found a proof or disproof (Section 5.5), along with an full example of an **FL** tableau proof (Figure 5.12). We finish by describing the exact search space that an **FL** tableau generation algorithm must search through (Section 5.6).

An **FL** tableau starts with a list of sentences, the *antecedents*, and a sentence to prove or disprove, the *goal*, which both become the root node of the tableaux.

It is important to note that all the tableaux rules are the inverse of an **FL** implication inference rule. As you more down the tree you move up the derivation.

5.1 Tableau nodes

Each node in the **FL** tableau contains a list of sentences that are true at it and every child node (antecedents), and a goal sentence that is yet to be proven. Each of the antecedents are labelled so that proof can be more easily followed. An example node is given in Figure 5.1.

Figure 5.1 An **FL** tableau node

$$\begin{array}{l} (label_1)\ antecedent_1 \\ \vdots \\ (label_n)\ antecedent_n \\ \hline\hline \qquad\qquad goal \end{array}$$

As a running example in this chapter we will be proving:

$$\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \forall x^{Nat}.add\ x\ y = x \leftarrow \{y \overset{\rightharpoonup}{=} Zero\}$$

This has the starting root node in Figure 5.2. The definitions of Δ_{add}, Γ_{add} and Φ_{add} can be found in Section 4.4. Our antecedents here are the sentences found in Φ_{add}.

Figure 5.2 Starting node of our example tableau

$$\begin{array}{c} (add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \hfill \\ (add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\ \hline\hline \forall x^{Nat}.add\ x\ y = x \leftarrow \{y \overset{\rightharpoonup}{=} Zero\} \end{array}$$

Depending on the node itself, or those beneath it, a goal can be proven true or disproven.

5.2 First steps

Although we have already described what the starting node of the tableau is, there are a few starting changes we must make to it before it can be used. To begin with we must remove all of the universal quantifiers from the goal and instantiate them to be a new variable not existing in the antecedent, by the reverse of (∀I).

For our example we have instantiated x to x for simplicity (Figure 5.3).

Figure 5.3 Instantiating our goal variables

$$\begin{array}{l}
(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \\
(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\
\hline\hline
\qquad\qquad add\ x\ y = x \leftarrow \{y \overset{\rightharpoonup}{=} Zero\}
\end{array}$$

We then take all the conditions of goal sentence and add them to the list of antecedents at the root node (giving them a label of our choosing), since we can assume them to be true when proving the goal. More formally this can be done using (FULFIL) (Figure 4.16) with (CUT) (Figure 4.13). Our final and usable starting node for our example is given in Figure 5.4.

Figure 5.4 Adding our goal conditions to our antecedents

$$\begin{array}{l}
(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \\
(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\
(cond_1)\ y = Zero \\
\hline\hline
\qquad\qquad\qquad add\ x\ y = x
\end{array}$$

5.3 Applying equality

At any tableau node we can apply one of the antecedent sentences using an equality rule to generate a new goal. How this is applied depends on the nature of the antecedent sentence.

In the simplest case the antecedent has no quantifiers and no conditions, so it can just be applied as is, creating a single descendent branch (Section 5.3.1).

If we have an antecedent sentence with no quantifiers but with conditions we must create new branches in order to remove these conditions before we can apply the sentence (Section 5.3.4).

If we have a quantified antecedent sentence we must remove these quantifiers before we can apply it (Section 5.3.2).

Sometimes it is advantageous to add conditions to an antecedent sentence so that it can be applied more generally (Section 5.3.3).

We then show how many of the decisions a human would make in applying equality can be inferred by the algorithm (Section 5.3.5).

5.3.1 Conditionless unquantified sentences

The most simple application of equality would be to use an unquantified sentence with no conditions. When applying an unquantified sentence we have to match the term we are rewriting exactly with a sub-term of one side of the sentence we are applying. This is a combination of the various equality rules in Figure 4.18.

In our example we will apply our unquantified rule ($cond1$) from left to right on the variable y in the left hand side of our goal, giving us the next node in the tableau as shown in Figure 5.5. Note that at the beginning of next goal we put the name of the rule we applied.

Figure 5.5 Applying ($cond_1$) to our goal

$$\begin{array}{c}
\begin{array}{l}
(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \\
(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\
(cond_1)\ y = Zero
\end{array} \\
\hline\hline
add\ x\ y = x \\
\Big| \\
\hline\hline
[cond_1]\ add\ x\ Zero = x
\end{array}$$

If the new branch closes with a proof then this node also closes with a proof, by the equality rules (Figure 4.18) and cumulative transitivity (CUT1) (Figure 4.13). If the new branch closes with a disproof then this node also closes with a disproof, by the equality rules and (CUT2) (Figure 4.13).

5.3.2 Quantified sentences

When we have quantifiers infront of a sentence we need to bind these variables to a term before the sentence can be used. Formally this is done using ($\forall$E) (Figure 4.15). for a quantified variable x^T we choose any term of type

T, and replace all instances of that variable in the sentence with the term chosen.

In our example, from the antecedent $(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v$ we can choose v to be $Zero$, and get the new antecedent $add\ Zero\ Zero = Zero$. We will also create a new version of (add_2) using instantiation, this time picking u to be a new x', and v to again be $Zero$. Both these new antecedents are shown in Figure 5.6. Note that we have placed the name of the antecedent from which this one was generated at the beginning of the line. Note also that we are adding these antecedents to our existing node so that they can be used in the next step.

Figure 5.6 Instantiating an antecedent

$$\begin{array}{c}
\begin{array}{l}
(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \\
(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\
(cond_1)\ y = Zero
\end{array} \\
\hline\hline
add\ x\ y = x \\
| \\
\begin{array}{l}
[add_1]\ (add_{1.1})\ add\ Zero\ Zero = Zero \\
[add_2]\ (add_{2.1})\ add\ (Succ\ x')\ Zero = Succ\ (add\ x'\ Zero)
\end{array} \\
\hline\hline
[cond_1]\ add\ x\ Zero = x
\end{array}$$

5.3.3 Adding conditions to unquantified sentences

The rule (EXPAND) allows us to add a condition in order to create a sentence that can be applied more generally. We can replace a constructor term (κ) in a sentence with any other term (τ) as long as we add a condition to that effect ($\tau \overset{\rightharpoonup}{=} \kappa$). This also requires that every variable in the constructor term does not exist elsewhere in the antecedent (at this node or higher up the tree), and that adding this condition does not remove the well-formedness of the sentence. For a more detailed description please see the definition of (EXPAND) in Figure 4.17.

As an example take our previously generated antecedent:

$$(add_{1.1})\ add\ Zero\ Zero = Zero$$

We can replace our first $Zero$ constructor term with x, by adding the condition that $x \overset{\rightharpoonup}{=} Zero$. In this way we have expanded the sentence so it can be applied more generally. We can also create another useful antecedent from our $(add_{2.1})$ rule, replacing $Succ\ x'$ with x by adding the condition $x \overset{\rightharpoonup}{=} Succ\ x'$, notice that x' must not exist in the antecedent for this step to be valid. These new antecedents are depicted in Figure 5.7. Notice that we have combined the previous variable instantiation step and this step into one.

Figure 5.7 Expanding antecedents with conditions

$$\begin{array}{c}
\begin{array}{l}
(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v \\
(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v)) \\
(cond_1)\ y = Zero
\end{array} \\
\hline\hline
add\ x\ y = x \\
\Big| \\
\begin{array}{l}
[add_1]\ (add_{1.1})\ add\ x\ Zero = Zero \leftarrow x \overset{\rightharpoonup}{=} Zero \\
[add_2]\ (add_{2.1})\ add\ x\ Zero = Succ\ (add\ x'\ Zero) \leftarrow x \overset{\rightharpoonup}{=} Succ\ x'
\end{array} \\
\hline\hline
[cond_1]\ add\ x\ Zero = x
\end{array}$$

5.3.4 Removing conditions from sentences

If a sentence has conditions they must be fulfilled in order to use the sentence in an equality rule. To fulfil a condition on the value of a term τ we branch at this node, with a separate branch for every possible value of τ. That is to say a branch for every possible constructor term of the type of τ. Each of these possible branches will have this assignment to τ, i.e. $\tau = \kappa_i$ added to its antecedent. For a formal method of generating each constructor term please see the definition of *instantiate* in Figure 4.21. Notice that we can choose any term we like to branch on.

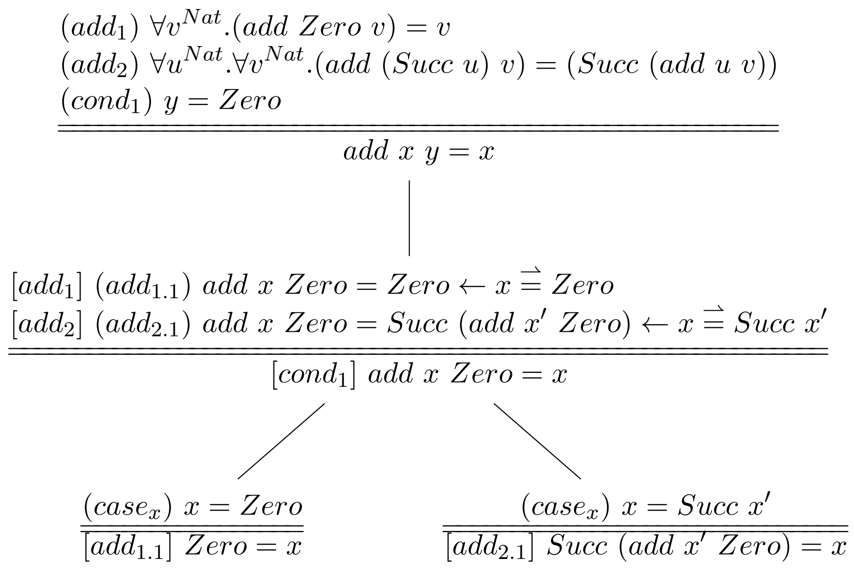
As κ is a constructor term, one of these $\tau = \kappa_i$ will fulfil $\tau \overset{\rightharpoonup}{=} \kappa$, that is to say there will be a κ_i which is equal to κ up to renaming of variables (we can just choose variable names such that they are equal), and hence in this branch the condition can be removed.

For our example we have branched on the possible values of x, giving us a branch $x = Zero$ and a branch $x = Succ\ x'$, notice that we have chosen the variable x' so it matches the condition of $(add_{2.1})$. We can now apply our $(add_{1.1})$ and $(add_{2.1})$ rules on the goal down the new branches where they are applicable. The tableau we have after this step in our example is given in Figure 5.8. Notice that we have named each of the new antecedents

($case_x$), though there is no strict naming convention and one can name new antecedents in whatever way seems most appropriate.

In this case we have branched in such a way that a rule can be applied down each branch, and though this saves time and space it is not a requirement of the branching rule. Indeed one can branch on the different values of a term without even having a condition to satisfy in mind.

Figure 5.8 Branching to apply conditional antecedents


$(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v$
$(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))$
$(cond_1)\ y = Zero$
$add\ x\ y = x$
$[add_1]\ (add_{1.1})\ add\ x\ Zero = Zero \leftarrow x \overset{\rightharpoonup}{=} Zero$
$[add_2]\ (add_{2.1})\ add\ x\ Zero = Succ\ (add\ x'\ Zero) \leftarrow x \overset{\rightharpoonup}{=} Succ\ x'$
$[cond_1]\ add\ x\ Zero = x$
$(case_x)\ x = Zero$
$[add_{1.1}]\ Zero = x$
$(case_x)\ x = Succ\ x'$
$[add_{2.1}]\ Succ\ (add\ x'\ Zero) = x$

This branching generalises (COMPLETE-CASE1), (COMPLETE-CASE2) and (COUNTEREXAMPLE), depending on the result of each branch, and the nature of τ. See Section 4.8.7 for the definitions of these rules.

- By (COMPLETE-CASE1), if every branch is proven true then this node is proven true.
- By (COMPLETE-CASE2), if every branch is proven false then this node is proven false.
- By (COUNTEREXAMPLE), if τ is a variable symbol that does not exist in the antecedent of this or any higher node, then the disproof of a single branch entails the disproof of this node.

5.3.5 Removing choice

Removing quantifiers and generating conditions require that we in some way choose what values to instantiate the quantifed variables to, or what term to replace constructor terms with. Described here is a method to generate these from any antecedent sentence and the goal sentence it is to be applied to.

We do this by creating a unifier that will rewrite the antecedent term to match the goal subterm. We then check whether this is a valid unifier and if so we use it to determine how to add conditions and instantiate variables correctly. This method is described in detail in the rest of this section.

First we choose which side of the antecedent we are applying to the goal, and which sub-term of the goal we are applying it to. As an example we will apply the left hand side of the antecedent $\forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))$, i.e. $(add\ (Succ\ u)\ v)$, to the sub-term $add\ x\ Zero$ of the goal $add\ x\ Zero = x$.

We must now unify these two chosen terms together. For this we use a recursive unification function $unify$, returning which sub-terms must be replaced with which in order that the two terms become equal. See Figure 5.3.5 for the Haskell source of our $unify$ function.

Figure 5.9 Haskell function that generates the unifier of two terms

```
unify :: Ord a => Term a -> Term a -> Set (Term a, Term a)
unify t1 t2 =
  let (t1_func : t1_args) = flattenTerm t1
      (t2_func : t2_args) = flattenTerm t2
  in if (t1_func == t2_func) && (length t1_args == length t2_args)
       then Set.unions (zipWith unify t1_args t2_args)
       else Set.singleton (t1, t2)

  where flattenTerm :: Term a -> [Term a]
        flattenTerm (App f a) = flattenTerm f ++ [a]
        flattenTerm t = [t]
```

So for our example we have $unify\ (add\ (Succ\ u)\ v)\ (add\ x\ Zero)$ which returns the set $\{(Succ\ u), x), (v, Zero)\}$.

Now that we have the unifier set we must check to make sure it is a valid unifier. If the unifier is invalid then we cannot apply this antecedent in this way to the goal. Valid unifiers fulfil two conditions:

1. They are one to many, that is no value recurs on the left side of the product set that is the unifier. So the set $\{(x, y), (x, z)\}$ is not a valid unifier since x exists twice on the left side. This is necessary as we are applying the unifier to the antecedent term in order to make it match the goal sub-term we unified it with. If the unification is not one-to-many is is not a valid mapping, and so cannot be applied as such. It should be noted that the left side of the unifier represents the sub-terms of the antecedent expression we are matching.

2. They contain no invalid unifications, a unification being one of the $(\tau \times \tau)$ pairs in our unifier. An invalid unification is one that fulfils any of these four properties:

   (a) Has a constructor term on both sides. A constructor term in this context being one that has a constructor as its left-most symbol. This is invalid as we are unifying a constructor with a constructor, which will only ever be different constructors based on how the unification algorithm works, and so are fundamentally unequal. So $(K\ y, ((J\ x)(f\ z)))$ is invalid for this reason.

   (b) Has a function term on the left hand side. A function term being one that has a function as its left-most symbol. This is invalid because we are matching a function with another arbitrary term, and so this introduces an unconstrained equality as a condition. See Section 4.5 for why this is a bad thing. So $((f\ x), y)$ is invalid for this reason.

   (c) Has a variable symbol on the left hand side that is not universally quantified in the antecedent sentence. The reasoning for this is that we cannot replace a variable that is not universally quantified.

   (d) Has a not well-formed constructor term on the left hand side. This is because to deal with a constructor term on the left hand side we introduce a new condition to the antecedent sentence by the (EXPAND) rule (Figure 4.17), which requires this property. All of the variables in the constructor term must also be universally quantified as we will be renaming them to fresh variables in the expanding process.

Now we know our unifier is valid we have to use it to convert the antecedent sentence into one that can be applied to our goal sub-term. How we apply each unification depends on its left hand side:

- If it is a variable we simply apply this unification (x, τ) as a renaming $[\tau/x]$ on the antecedent sentence.

- If it is a constructor term we are applying the (EXPAND) rule so we first rename all of the variables in the term to be fresh, then apply the renaming $[\tau/\kappa]$ to the antecedent sentence, and then finally add the unification as a condition $(\tau \overset{\rightharpoonup}{=} \kappa)$.
- By the definition of a valid unifier these are the only two options the left hand side can be.

See Figure 5.3.5 for a formal definition of how each unification is applied.

Figure 5.10 Applying a unification to an **FL** sentence

$$\begin{aligned}
&applyUni &&: \varphi \rightarrow (\tau \times \tau) \rightarrow \varphi \\
\\
&applyUni\ \varphi\ (x, \tau) &&= \varphi[\tau/x] \\
&applyUni\ (\tau_1 = \tau_2 \leftarrow X)\ (\kappa, \tau) &&= (\tau_1[\tau/\kappa'] = \tau_2[\tau/\kappa'] \leftarrow X[\tau/\kappa'] \cup \{\tau \overset{\rightharpoonup}{=} \kappa'\}) \\
& && \qquad where\ \kappa' = freshen\ \kappa
\end{aligned}$$

Now we should have an unquantified antecedent sentence where one side is equal to the sub-term of the goal sentence we were aiming to apply it too. In our example we will have converted our antecedent sentence from $\forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))$ into $(add\ x\ Zero) = (Succ\ (add\ x'\ Zero)) \leftarrow x \overset{\rightharpoonup}{=} Succ\ x'$, where we had $Succ\ x' = freshen\ (Succ\ u)$ generating fresh variable x'. This can easily be applied to our goal $add\ x\ Zero = x$, giving us $(Succ\ (add\ x'\ Zero)) = x$ when $x = Succ\ x'$.

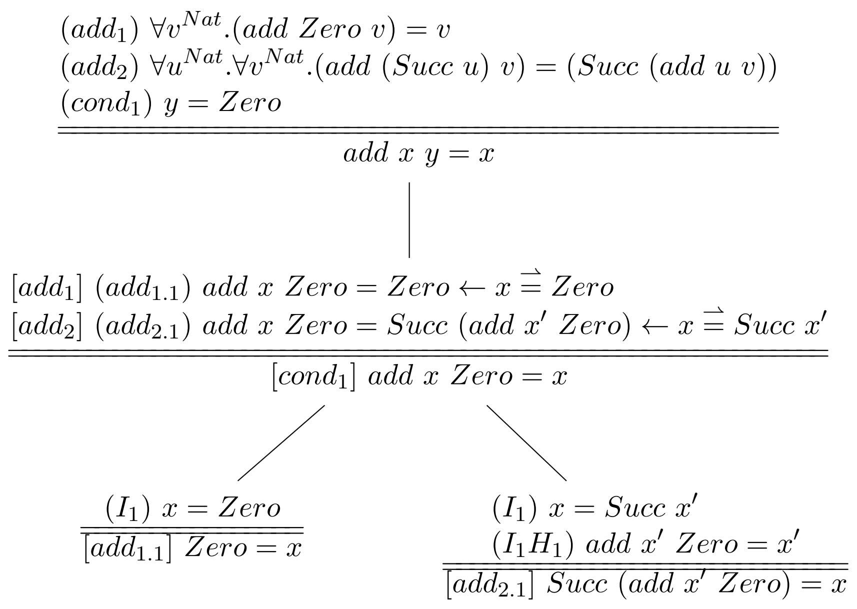
5.4 Applying induction

At any node we can either apply an equality, or we can apply induction. To apply induction we first pick a sub-term of our goal sentence (τ) as our inductive term. We then branch on every possible constructor value for τ such that every recursively typed variable in the constructor value (variables with the same type as τ) are added as an inductive hypothesis to that branch. See the definition of the (INDUCTION) rule in Figure 4.26 for a more formal defintion, in addition to the definition for *instantiate* for generating constructor terms in Figure 4.21.

If every branch of an inductive step closes with a proof, then this node closes with a proof. Note that induction can only be used to generate a positive proof, its usage is not defined for the creation of a disproof.

In our running example we will ignore the case-completion branching step we took in Figure 5.8 and instead apply induction on the term x of our goal $add\ x\ Zero = x$. The two constructor terms x could be are still $Zero$ and $Succ\ x'$ for some new x'. In the $Zero$ branch we simply add the antecedent $x = Zero$ and continue the proof as with case-completion. In the $Succ\ x'$ branch though we have x' as a recursive variable, since it is of type Nat, so we can add $(add\ x\ Zero = x)[x'/x] \equiv (add\ x'\ Zero = x')$ as an inductive hypothesis down this branch in addition to $x = Succ\ x'$. This inductive step is shown in Figure 5.11.

Figure 5.11 Applying an induction step


$(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v$
$(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))$
$(cond_1)\ y = Zero$
$add\ x\ y = x$
$[add_1]\ (add_{1.1})\ add\ x\ Zero = Zero \leftarrow x \overset{\rightharpoonup}{=} Zero$
$[add_2]\ (add_{2.1})\ add\ x\ Zero = Succ\ (add\ x'\ Zero) \leftarrow x \overset{\rightharpoonup}{=} Succ\ x'$
$[cond_1]\ add\ x\ Zero = x$
$(I_1)\ x = Zero$
$[add_{1.1}]\ Zero = x$
$(I_1)\ x = Succ\ x'$
$(I_1H_1)\ add\ x'\ Zero = x'$
$[add_{2.1}]\ Succ\ (add\ x'\ Zero) = x$

Notice we have labelled the inductive assignments I_1 to show that they are the antecedent of the first inductive step we have performed. The label I_1H_1 denotes the first inductive hypothesis of the first inductive step. Remember that labelling need only be unique within each branch, so multiple branches could each contain a rule I_1 or I_1H_1.

5.4.1 Inferring auxiliary lemmas through induction

When we perform a successful induction step we are proving a sentence to be true for any input given. Performing induction on a node of the tree that is not the root node means we are proving a sentence that was not our original goal, so we have proven a lemma auxiliary to our proof.

An important part of our induction method is the ability to pick an arbitrary subterm, rather than just a variable, allowing non-trivial lemmas to be generated on the way to a proof. Furthermore, the ability to have multiple induction steps performed down the tree enables you to prove properties of arbitrary complexity in the same tableau, rather than relying on sub-proofs of these lemmas. To illustrate this further we would draw your attention to the various proofs created using Function Tableau, either by us or Zeno, in Chapter 7.

5.5 Closing branches

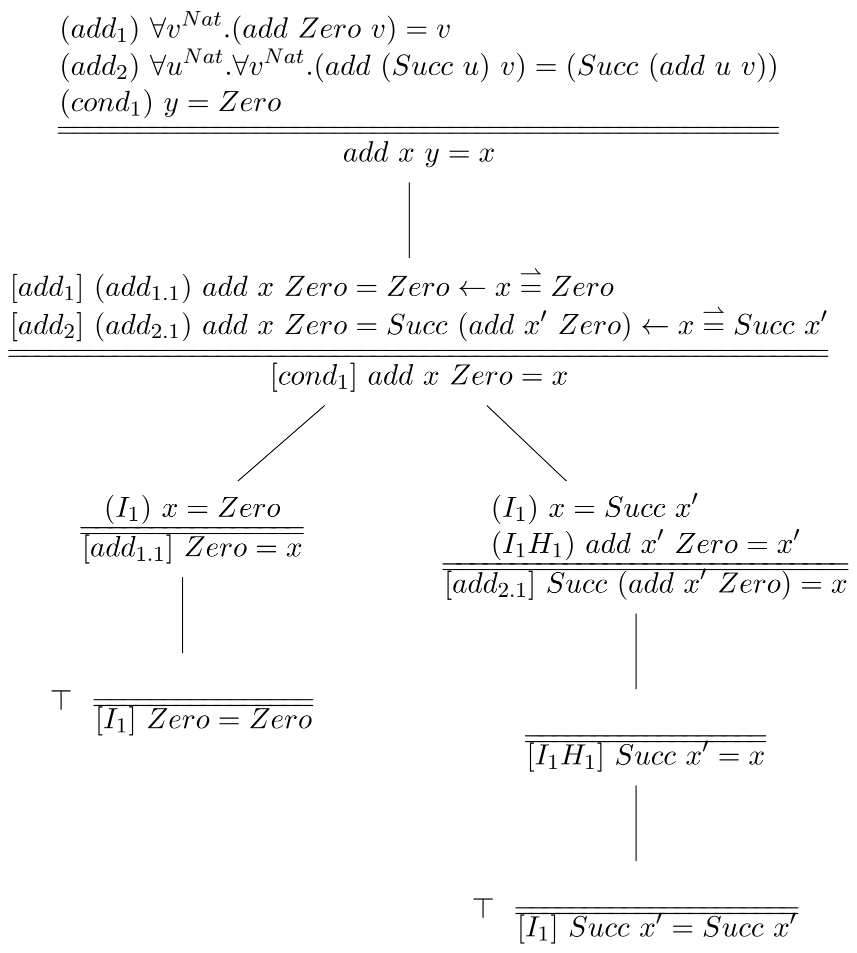
Now that we have described the application of various rules to alter the goal sentence, we must have a way of deciding when a goal has been proven or disproven.

5.5.1 Proving a branch

The rule we apply for proof is just (EQ-REFLEXIVE) (Figure 4.18) which corresponds to the reflexivity of equality, and means we can close a branch with a proof if the left and right hand sides of a goal are equal. So goals like $x = x$, $(Succ\ x) = (Succ\ x)$ and $(f\ (g\ x)\ y) = (f\ (g\ x)\ y)$ mean we have proven this branch. To show a proven branch we place a $\top$ to the left of the node.

To demonstrate our proof rules we will complete our running example of a proof for $\Delta_{add}, \Gamma_{add}, \Phi_{add} \vdash_{\mathbf{FL}} \forall x^{Nat}.add\ x\ y = x \leftarrow \{y \overset{\rightharpoonup}{=} Zero\}$ in Figure 5.12. Note that this tableau corresponds almost exactly with the derivation we gave earlier in Figure 4.29.

Figure 5.12 Full proof of the identity of zero under addition



$(add_1)\ \forall v^{Nat}.(add\ Zero\ v) = v$
$(add_2)\ \forall u^{Nat}.\forall v^{Nat}.(add\ (Succ\ u)\ v) = (Succ\ (add\ u\ v))$
$(cond_1)\ y = Zero$
$add\ x\ y = x$
$[add_1]\ (add_{1.1})\ add\ x\ Zero = Zero \leftarrow x \overset{\rightharpoonup}{=} Zero$
$[add_2]\ (add_{2.1})\ add\ x\ Zero = Succ\ (add\ x'\ Zero) \leftarrow x \overset{\rightharpoonup}{=} Succ\ x'$
$[cond_1]\ add\ x\ Zero = x$
$(I_1)\ x = Zero$
$[add_{1.1}]\ Zero = x$
$\top$
$[I_1]\ Zero = Zero$
$(I_1)\ x = Succ\ x'$
$(I_1H_1)\ add\ x'\ Zero = x'$
$[add_{2.1}]\ Succ\ (add\ x'\ Zero) = x$
$[I_1H_1]\ Succ\ x' = x$
$\top$
$[I_1]\ Succ\ x' = Succ\ x'$

5.5.2 Disproving a branch

The two rules we apply for disproof are (UNEQUAL1) (Figure 4.19) and (UNEQUAL2) (Figure 4.20). We will refer you to the definitions of these rules for how they work, but they allow us to disprove goals such as $Succ\ (Succ\ Zero) = Succ\ Zero$, and $(f\ Zero)\ Zero = (f\ (Succ\ x))\ Zero$. To show a disproven branch we place a $\bot$ to the left of the node.

5.6 Search space

The **FL** tableau method requires that one make a number of choices at each node as to what rule to apply, and how to apply it. Unfortunately we have not found any intelligent way of determining which choice to make at each node, so the implementation of our tool can only check every possibility at every node.

Obviously if we have already found a proof/disproof there is no need to check the rest of the search space, but this still leaves us with an algorithm with a worst-case run-time that is exponential in the number of steps the proof/disproof requires.

The choices to be made at each node break down as follows:

- Choose to apply an equality rule with an antecedent. This means we must also choose which antecedent to apply and which sub-term of the goal to apply it to.
- Choose to apply induction. This means we must also choose which sub-term of the goal to apply induction on.

All of these choices are from a finite set but they still represent a lot of different possibilities at each node. Induction in particular can add new antecedents through induction hypotheses and so creates more ways to apply equality in any branches beneath it.

Chapter 6

Zeno

In this chapter we describe the overall structure and usage of the Zeno tool. First we give an broad overview of Zeno (Section 6.1). We then show how Function Logic can be represented in a file format (.flf) for Zeno to load and for Haskell to be translated into (Section 6.2). Next we describe how Zeno translates Haskell code into **FL** along with how we represent lemmas to be proven within our Haskell (Section 6.3). We then give the command line options for running the Zeno tool oneself (Section 6.4). Finally we explain the format in which Zeno outputs the proofs that it generates, along with some example proofs found by Zeno (Section 6.5).

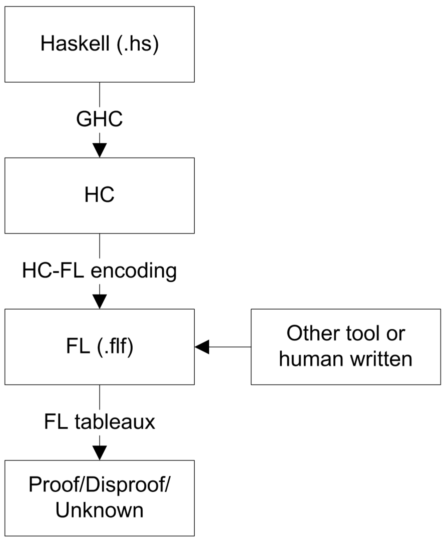
6.1 Overview

Zeno consists of two main parts, the mapping of a Haskell/**HC** program to a Function Logic formula, and the attempted proof or disproof of properties with respect to an **FL** formula using our **FL** tableau method.

We start in most cases with a Haskell file (.hs) which Zeno can parse using the GHC API in order to get its GHC-core representation. As GHC-core is a more complex version of our Haskell-Core (Section 3.2) we remove the functions that are not valid **HC** and then use an implementation of our **HC** to **FL** encoding (given in Figure 4.10) to generate **FL** code. This **FL** code is represented in the **FL** file format, which Zeno can then parse and attempt to solve using our **FL** tableau method (Chapter 5). Note that we could have generated our **FL** file in another fashion, perhaps with a mapping from another functional language, or by hand-writing it. In Figure 6.1 we have a flow chart of this process.

All of our tool is implemented in Haskell, and we used the Happy[1] parser generator to create a parser for **FL** files. We believe it is noteworthy that the **FL** tableau solving engine of our tool amounts to only 354 lines of regularly formatted Haskell code.

Figure 6.1 Flow chart of the Zeno tool



6.2 Function Logic files (.flf)

In order that problems from other languages could be solved by Zeno we created an external file format with which to express **FL** formulas and their associated types and typing. It is into this file format that Haskell programs are compiled and it is these files that Zeno loads and solves lemmas from.

The EBNF grammar for the **FL** file structure is given in Figure 6.2.

[1]http://www.haskell.org/happy/

Figure 6.2 EBNF grammar for **FL** files

$$
\begin{array}{lcl}
\mathit{FL} & ::= & (\mathit{Stmt}\ \text{‘}\texttt{.}\text{’})^{*} \\
 & & \\
\mathit{Stmt} & ::= & \text{‘}\texttt{variables}\text{’}\ \mathit{TypedNames} \\
 & | & \text{‘}\texttt{functions}\text{’}\ \mathit{TypedNames} \\
 & | & \text{‘}\texttt{type}\text{’}\ \mathit{Name}\ \text{‘}\texttt{=}\text{’}\ \mathit{TypeCons} \\
 & | & \text{‘}\texttt{axiom}\text{’}\ \mathit{Name}\ \mathit{Sentence} \\
 & | & \text{‘}\texttt{lemma}\text{’}\ \mathit{Name}\ \mathit{Sentence} \\
 & & \\
\mathit{Type} & ::= & \mathit{Name} \\
 & | & \text{‘}\texttt{(}\text{’}\ \mathit{Type}\ \text{‘}\texttt{->}\text{’}\ \mathit{Type}\ \text{‘}\texttt{)}\text{’} \\
 & & \\
\mathit{Term} & ::= & \mathit{Name} \\
 & | & \text{‘}\texttt{(}\text{’}\ \mathit{Term}\ \mathit{Term}\ \text{‘}\texttt{)}\text{’} \\
 & & \\
\mathit{TypedNames} & ::= & \mathit{TypedName}\ (\text{‘}\texttt{,}\text{’}\ \mathit{TypedName})^{*} \\
\mathit{TypedName} & ::= & \mathit{Name}\ \text{‘}\texttt{:}\text{’}\ \mathit{Type} \\
 & & \\
\mathit{TypeCons} & ::= & \mathit{Name}^{+}\ (\text{‘}\texttt{|}\text{’}\ \mathit{Name}^{+})^{*} \\
 & & \\
\mathit{Sentence} & ::= & \mathit{Equality}\ (\text{‘}\texttt{:-}\text{’}\ \mathit{Equality}^{+})^{?} \\
\mathit{Equality} & ::= & \mathit{Term}\ \text{‘}\texttt{=}\text{’}\ \mathit{Term}
\end{array}
$$

FL files don't only have to be parsable by this grammar to be valid, they must also correspond to well-formed **FL**, as defined in Section 4.7, and which is checked by Zeno when it loads a file.

In Figure 6.3 we have given an example of a well-formed **FL** file containing the datatype of the natural numbers (`type Nat`) and the addition function on the naturals (`add`). It also has three lemmas, two true and one false. The first (`addZero`) expresses that `Zero` is the identity when given as the second argument to `add`. The second lemma (`addSym`) expresses the symmetry of the addition function. The final lemma (`addSelf`) expresses that adding something to itself does not change its value, which is provably false. This **FL** file is the direct encoding of the Haskell file given in Figure 6.4.

Figure 6.3 Function Logic file example

```
type Nat = Succ Nat | Zero.

functions
  add : (Nat -> (Nat -> Nat)).

variables
  x : Nat,
  x' : Nat,
  y : Nat.

axiom add1
  ((add x) y) = y :- x = Zero.
axiom add2
  ((add x) y) = (Succ ((add x') y)) :- x = (Succ x').

lemma addZero
  ((add x) Zero) = x.

lemma addSym
  ((add x) y) = ((add y) x).

lemma addSelf
  ((add x) x) = x.
```

There were two important design decisions made in creating the **FL** file format. The first of which was typing variable names globally as per the `variables` keyword, rather than typing them at the head of each **FL** axiom and lemma, as is done in the formal definition of **FL** . The reason for this was to remove clutter and repetition from the definition of axioms/lemmas as well as providing some naming consistency across the **FL** file. This means that if you see `x` defined as a natural number then you know it will always be a natural number wherever it is used.

The second design decision was to give a name to every axiom and lemma in the file. These names exist so that when a proof or disproof is generated by Zeno it can be more easily followed, as every equality step can be labelled with the name of the axiom or lemma which was applied. When Haskell code is encoded into **FL** these names are automatically generated from the name of the function, as in Figure 6.3.

6.3 Translating Haskell files to Function Logic files

The translation of a Haskell file (.hs) to a Function Logic file (.flf) first requires that we parse the Haskell code, for which we have employed the open-source Glasgow Haskell Compiler[2] API. This API outputs a format called GHC-core, which is a greatly simplified representation of the functions in a Haskell program, along with any datatypes defined. As GHC-core is a super-set of our own formal language Haskell-Core (**HC**), defined in Section 3.2, we can easily remove any functions from our GHC-core representation that are not valid **HC**.

Now that we have the **HC** code representing our Haskell file we can apply the **HC** to **FL** encoding given in Figure 4.10. All functions that are not lemmas (see Section 6.3.1) are simply translated into axioms in our **FL** file and named using the name of the function, with a different number after each name if the function has multiple axioms representing it.

6.3.1 Representing lemmas in Haskell code

Rather than have Haskell files translated into **FL** and then require the programmer to add their lemmas to the **FL** file manually we created a method whereby lemmas could be expressed in the Haskell file iteself for automatic translation by the Zeno tool.

This is useful in two respects. The first is that it means a programmer does not have to understand the **FL** syntax in order to use the Zeno tool, they need only understand Haskell. Secondly, without this feature every time we change our Haskell code and have to retranslate it to **FL** we would lose the description of all these properties and have to re-enter them into the new **FL** file.

Lemmas are added by defining a top-level function which returns a value of the `Lemma` datatype. The `Lemma` datatype is provided in the `Zeno` library module and has a single constructor `Equals`, which takes two parameters of the same type and expresses an equality between these two parameters. Any variable taken as an input to the function returning the `Lemma` type is then a universally quantified variable in the resulting **FL** code. For convenience we have added the infix operator `===` which corresponds to the `Equals` constructor for the `Lemma` datatype.

When encoding **HC** into **FL** Zeno will search through for any functions that return type `Lemma` and translate them into lemmas rather than axioms, using the name of the function as the name of the lemma.

[2]http://haskell.org/ghc

In Figure 6.4 we have the Haskell code defining the datatype for the natural numbers (`Nat`) and their addition function (`add`), along with a few lemmas about addition. This translates to the **FL** file given previously in Figure 6.3. As you can see the name of the function that returns the `Lemma` type is used as the name of the produced **FL** lemma.

The `Zeno` library module contains nothing but the definition for `Lemma` and `===` and is shown in Figure 6.5.

Figure 6.4 Symmetry of addition in Haskell

```
module Add where
import Zeno

data Nat
  = Succ Nat
  | Zero

add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = Succ (add x y)


addZero :: Nat -> Lemma Nat
addZero x = (add x Zero) === x

addSym :: Nat -> Nat -> Lemma Nat
addSym x y = (add x y) === (add y x)

addSelf :: Nat -> Lemma Nat
addSelf x = (add x x) === x
```

Figure 6.5 Zeno library module

```
module Zeno where

data Lemma a = Equals a a
(===) = Equals
```

6.4 Using Zeno

Zeno is a command line tool, with a few options that are explained below:

-f filename This option allows us to supply Zeno with a **FL** file (.flf) to load and prove properties about.

-hs filename This option allows us to supply Zeno with a Haskell file (.hs) to translate into **FL** which it then automaticaly loads as if it were given with a **-f** option.

-g lemma We can use this option to choose which lemma in the **FL** file Zeno should attempt to prove (our *goal* lemma).

-fl When this flag is set it means Zeno should output the **FL** it has loaded before printing any proof information. If we use this flag without a goal given the output will just be the **FL** file. We can use this to convert Haskell code to **FL** code, for example `zeno -hs test.hs -fl > test.flf` will output the encoding of the Haskell file `test.hs` as **FL** to the file `test.flf`.

-i number This option allows us to specify the number of inductive steps Zeno is allowed to perform in its proof search. If we do not give a value it defaults to zero.

-e number This option allows us to specify the number of equality steps Zeno is allowed to perform in its proof search. If we do not give a value Zeno will start at 1, then iteratively increment the value if a proof/disproof is not found. Without this option specified a Zeno proof search may not terminate.

-l lemma With this option we can supply Zeno with an auxiliary lemma to assume to be true and use in its proof search. This parameter can be given multiple times, corresponding to multiple lemmas.

Examples of invoking Zeno from the command line are given in the next section (Section 6.5).

6.5 Proof output

The proofs/disproofs that Zeno outputs are textual representations of the corresponding **FL** tableau that Zeno has generated. For the definition of an **FL** tableau please see Chapter 5. Listed below are individual facts about the proof output and are best read in context with a Zeno proof, like the one in Figure 6.6.

- Each line represents the goal to be proven at that step. The label at the beginning of each line is the axiom/lemma that was applied in order to get to this line from the previous one. The only exception to this is the first line, which is always labelled with `[Goal]`, as it represents what we are trying to prove overall.
- Multiple branching in the tableau, by the application of case-completion or induction rules, is represented by indentation and boxes around each branch.
- Inductive steps are labelled as `[I`n`]`, where `[I1]` is the first inductive step, `[I2]` the second, and so on. Application of an inductive hypothesis is labelled as `[I`n` Hyp]`, where `[I1 Hyp]` is an inductive hypothesis of the first inductive step.
- If a line is disproven it will have a `not` before it.
- Along with the proof, the output contains a list of lemmas that were proven along the way, where each lemma corresponds to a successful inductive step in the tableau. In this list of lemmas, the variable upon which induction was performed is replaced with the symbol `!x` to highlight how induction took place.
- It will also show the lemmas that were used in the proof/disproof, and the number of equality and induction steps that Zeno had to make and the number of seconds that the process took.
- New variables introduced by Zeno in the proof process are prefixed by a `?` symbol to differentiate them from those defined by the user.

A simple Zeno proof, that `add x Zero = Zero` for all `x : Nat`, is given in Figure 6.6. It is an exact representation of the formal **FL** tableau proof given in Figure 5.12. This was generated with the command `zeno -hs Add .hs -g addZero -i 1`. In this, and the next two examples, the file `Add.hs` corresponds to that given in Figure 6.4.

A more complex Zeno proof, in this case the proof of the symmetry of addition expressed by the lemma `addSym` in Figure 6.3, is given in Figure 6.7. This was generated with the command `zeno -hs Add.hs -g addSym -i 2`.

A disproof found by Zeno, that `add x x = x`, is given in Figure 6.8. This was generated with the command `zeno -hs Add.hs -g addSelf -i 0`.

Figure 6.6 Zeno's proof of addition with zero

```
[Goal] ((add x) Zero) = x

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] ((add (Succ ?yd)) Zero) = (Succ ?yd)
  | [add2] (Succ ((add ?yd) Zero)) = (Succ ?yd)
  | [I1 Hyp] (Succ ?yd) = (Succ ?yd)
  | True
  /////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] ((add Zero) Zero) = Zero
  | [add1] Zero = Zero
  | True
  ///////////////////////////////

Lemmas proven:
  ((add !x) Zero) = !x

Lemmas used: none
Equality steps: 3
Induction steps: 1
Computation time: 0.000 sec
```

Figure 6.7 Zeno's proof of the symmetry of addition

```
[Goal] ((add x) y) = ((add y) x)

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] ((add (Succ ?yd)) y) = ((add y) (Succ ?yd))
  | [add2] (Succ ((add ?yd) y)) = ((add y) (Succ ?yd))
  | [I1 Hyp] (Succ ((add y) ?yd)) = ((add y) (Succ ?yd))

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [I2] (Succ ((add (Succ ?fe)) ?yd)) = ((add (Succ ?fe)) (Succ ?yd))
    | [add2] (Succ ((add (Succ ?fe)) ?yd)) = (Succ ((add ?fe) (Succ ?yd)))
    | [add2] (Succ (Succ ((add ?fe) ?yd))) = (Succ ((add ?fe) (Succ ?yd)))
    | [I2 Hyp] (Succ ((add ?fe) (Succ ?yd))) = (Succ ((add ?fe) (Succ ?yd)))
    | True
    ////////////////////////////////////////////////////////////////////////

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [I2] (Succ ((add Zero) ?yd)) = ((add Zero) (Succ ?yd))
    | [add1] (Succ ((add Zero) ?yd)) = (Succ ?yd)
    | [add1] (Succ ?yd) = (Succ ?yd)
    | True
    ////////////////////////////////////////////////////////

  //////////////////////////////////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] ((add Zero) y) = ((add y) Zero)
  | [add1] y = ((add y) Zero)
  | [add1] y = ((add y) Zero)

    \\\\\\\\\\\\\\\\\\\\
    | [add1] Zero = Zero
    | True
    ////////////////////

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [add1] (Succ ?le) = ((add (Succ ?le)) Zero)

      \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
      | [I2] (Succ (Succ ?me)) = ((add (Succ (Succ ?me))) Zero)
      | [add2] (Succ (Succ ?me)) = (Succ ((add (Succ ?me)) Zero))
      | [I2 Hyp] (Succ (Succ ?me)) = (Succ (Succ ?me))
      | True
      ///////////////////////////////////////////////////////////

      \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
      | [I2] (Succ Zero) = ((add (Succ Zero)) Zero)
      | [add2] (Succ Zero) = (Succ ((add Zero) Zero))
      | [add1] (Succ Zero) = (Succ Zero)
      | True
      ///////////////////////////////////////////////

    /////////////////////////////////////////////////////////////

  ///////////////////////////////////////////////////////////////

Lemmas proven:
  ((add !x) y) = ((add y) !x)
  ((add !x) (Succ ?yd)) = (Succ ((add !x) ?yd))
  ((add (Succ !x)) Zero) = (Succ !x)

Lemmas used: none
Equality steps: 7
Induction steps: 2
Computation time: 5.725 sec
```

Figure 6.8 Zeno's disproof of self-addition

```
not [Goal] ((add x) x) = x
| not [add2] (Succ ((add ?ce) (Succ ?ce))) = (Succ ?ce)
| not [add1] (Succ (Succ Zero)) = (Succ Zero)
| True

Lemmas used: none
Equality steps: 2
Induction steps: 0
Computation time: 0.016 sec
```

Chapter 7

Evaluation

In this chapter we will evaluate the usefulness and overall success of our tool. First we enumerate a few interesting proofs and disproofs that it was able to find (Section 7.1). We then demonstrate problems that our tool is theoretically able to solve but the complexity of which is too high to demonstrate a fully automated proof (Section 7.2). Then we will give some problems it was unable to solve and examine the reason for this shortfall (Section 7.3).

In all of these sections we will often give the equivalent proofs found using the proof verifier Isabelle/HOL[36]. Note that we have not used Haskabelle[37] to create this Isabelle code directly from the Haskell source, but have manually transcoded it. Note also that we have often used (`#`) to represent the list `Cons` operator, rather than the (`:`) used by Haskell, as Isabelle/HOL does not parse (`:`) correctly.

My supervisor, Professor Sophia Drossopoulou, runs half of a course entitled "Reasoning about Programs" for first year students. Her half details the proof of inductive properties of functional programs and a few of the properties proven here are those set as exercises for students.

The Zeno proofs listed here were produced using both 3GHz cores of a Intel® Core 2™ Duo E6850[1].

[1]http://ark.intel.com/Product.aspx?id=30785

7.1 Solved Problems

7.1.1 Idempotence of list reversal

A function is idempotent if applying it twice to a value returns the original value, that is to say $f^{A \to A}$ is idempotent if $\forall x^A : f\ (f\ x) = x$. The idempotence of list reversal is therefore that the reverse of the reverse of a list is equal to the original list. This is intuitively true[2], but finding a formal proof of this is not simple.

This was a very important problem for me as it is the tutorial example given for proving Higher Order Logic formulas in the Isabelle/HOL manual[36]. The proof in Isabelle also requires three lemmas to be input by the user, so is obviously non-trivial.

In Figure 7.1 we have the Haskell source that represents this problem, where the function `app` is list appending and the function `rev` is list reversal.

Figure 7.1 Idempotence of list reversal in Haskell

```
data Nat = Zero | Succ Nat
data List = Empty | Cons Nat List

app :: List -> List -> List
app Empty ys = ys
app (Cons x xs) ys = Cons x (xs ‘app‘ ys)W

rev :: List -> List
rev Empty = Empty
rev (Cons x xs) = (rev xs) ‘app‘ (Cons x Empty)

revIdm :: List -> Lemma
revIdm xs = (rev (rev xs)) === xs
```

Proof in Zeno

In Figure 7.2 we have the encoding of this Haskell code into a **FL** file.

[2]Except for infinite lists, but these are not discussed in our method (see Section 3.3)

Figure 7.2 Idempotence of list reversal in **FL**

```
functions
  rev : (List -> List),
  app : (List -> (List -> List)).

variables
  xs : List,
  ys : List,
  x : Nat.

type Nat = Succ Nat | Zero.

type List = Cons Nat List | Empty.

axiom rev1
  (rev Empty) = Empty.
axiom rev2
  (rev ((Cons x) xs)) = ((app (rev xs)) ((Cons x) Empty)).

axiom app1
  ((app Empty) ys) = ys.
axiom app2
  ((app ((Cons x) xs)) ys) = ((Cons x) ((app xs) ys)).

lemma revIdm
  (rev (rev xs)) = xs.
```

Proving `revIdm` in Zeno requires two inductive steps and nine equality steps giving the proof shown in Figure 7.3. Note that I have reformatted it slightly so that it fits more nicely onto the page.

Figure 7.3 Zeno's proof of the idempotence of reverse

```
[Goal] (rev (rev xs)) = xs

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev ((Cons ?yd) ?zd))) = ((Cons ?yd) ?zd)
  | [rev2] (rev ((app (rev ?zd)) ((Cons ?yd) Empty))) = ((Cons ?yd) ?zd)
  | [I1 Hyp] (rev ((app (rev ?zd)) ((Cons ?yd) Empty))) = ((Cons ?yd) (rev (rev ?zd)))

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [I2] (rev ((app ((Cons ?ee) ?fe)) ((Cons ?yd) Empty))) = ((Cons ?yd) (rev ((Cons ?ee) ?fe)))
    | [app2] (rev ((Cons ?ee) ((app ?fe) ((Cons ?yd) Empty)))) = ((Cons ?yd) (rev ((Cons ?ee) ?fe)))
    | [rev2] ((app (rev ((app ?fe) ((Cons ?yd) Empty)))) ((Cons ?ee) Empty)) =
                 ((Cons ?yd) (rev ((Cons ?ee) ?fe)))
    | [rev2] ((app (rev ((app ?fe) ((Cons ?yd) Empty)))) ((Cons ?ee) Empty)) =
                 ((Cons ?yd) ((app (rev ?fe)) ((Cons ?ee) Empty)))
    | [I2 Hyp] ((app ((Cons ?yd) (rev ?fe))) ((Cons ?ee) Empty)) =
                   ((Cons ?yd) ((app (rev ?fe)) ((Cons ?ee) Empty)))
    | [app2] ((Cons ?yd) ((app (rev ?fe)) ((Cons ?ee) Empty))) =
                 ((Cons ?yd) ((app (rev ?fe)) ((Cons ?ee) Empty)))
    | True
    //////////////////////////////////////////////////////////////////////////////////////////////

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [I2] (rev ((app Empty) ((Cons ?yd) Empty))) = ((Cons ?yd) (rev Empty))
    | [app1] (rev ((Cons ?yd) Empty)) = ((Cons ?yd) (rev Empty))
    | [rev2] ((app (rev Empty)) ((Cons ?yd) Empty)) = ((Cons ?yd) (rev Empty))
    | [rev1] ((app Empty) ((Cons ?yd) Empty)) = ((Cons ?yd) (rev Empty))
    | [app1] ((Cons ?yd) Empty) = ((Cons ?yd) (rev Empty))
    | [rev1] ((Cons ?yd) Empty) = ((Cons ?yd) Empty)
    | True
    ////////////////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev Empty)) = Empty
  | [rev1] (rev Empty) = Empty
  | [rev1] Empty = Empty
  | True
  ///////////////////////////////

Lemmas proven:
  (rev (rev !x)) = !x
  (rev ((app !x) ((Cons ?yd) Empty))) = ((Cons ?yd) (rev !x))

Lemmas used: none
Equality steps: 9
Induction steps: 2
Computation time: 78.565 sec
```

Notice that Zeno has discovered one auxiliary lemma in its proof, corresponding to the inner induction step `[I2]`. Remember that `!x` is the symbol for the variable that induction was performed on. The auxiliary lemma Zeno has found is that the reverse of a list (`!x`) with a single element appended to the end (`?yd`) is the same as the reverse of that list with the element concatenated at the start `((Cons ?yd)(rev !x))`.

Proof in Isabelle/HOL using lemmas

Given in Figure 7.4 is the Isabelle/HOL definitions of our `app` and `rev` functions, almost exactly as given in the Isabelle/HOL tutorial[36]. Note that the `#` operator is an infix version of the `Cons` and `++` is an infix version of list append. These were kept to make the proof in Isabelle/HOL more readable than the one in Zeno, which does not have the ability to display infix operators. One important change from the tutorial version is the removal of polymorphism from the list type to make the proof more closely reflect what is attempted in Zeno. Note that we do not need to define `nat` as it is built into Isabelle.

Figure 7.4 Reverse/Append definitions in Isabelle/HOL

```
datatype list
  = Empty                             ("[]")
  | Cons nat list                     (infixr "#" 65)

primrec app :: "list => list => list"  (infixr "++" 65)
where
  "app [] ys = ys" |
  "app (x # xs) ys = x # (app xs ys)"

primrec rev :: "list => list"
where
  "rev [] = []" |
  "rev (x # xs) = (rev xs) ++ (x # [])"
```

The proof of the idempotence of reverse as given in the Isabelle/HOL manual is shown in Figure 7.5. It cannot complete the proof without the lemma `revApp`, which itself cannot be proven without the lemmas `appEmpty` and `appAssoc`.

Figure 7.5 Proof of the idempotence of list reversal in Isabelle/HOL

```
lemma appEmpty [simp]: "xs ++ [] = xs"
  apply (induct_tac xs)
  apply (auto)
done

lemma appAssoc [simp]: "(xs ++ ys) ++ zs = xs ++ (ys ++ zs)"
  apply (induct_tac xs)
  apply (auto)
done

lemma revApp [simp]: "rev (xs ++ ys) = (rev ys) ++ (rev xs)"
  apply (induct_tac xs)
  apply (auto)
done

lemma revIdm [simp]: "rev (rev xs) = xs"
  apply (induct_tac xs)
  apply (auto)
done
```

The Isabelle/HOL tutorial quotes the necessary lemma to be `revApp`, which is `rev (xs ++ ys)= (rev ys)++ (rev xs)`. Zeno on the other hand found the required to lemma to be `rev((x # [])++ xs)= x # (rev xs)`. One could consider Zeno's lemma to be a specific case of the one required by Isabelle. That is to say the case where `ys = (x # Empty)`.

Indeed if we replace the `revApp` with the lemma found by Zeno we find that Isabelle/HOL can still prove the lemma. In addition, this new version `revApp2` requires no auxiliary lemmas, so the proof becomes much shorter, requiring only what is shown in Figure 7.6.

Figure 7.6 Simpler proof of `revIdm` in Isabelle/HOL

```
lemma revApp2 [simp]: "rev((x # []) ++ xs) = x # (rev xs)"
  apply (induct_tac xs)
  apply (auto)
done

lemma revIdm [simp]: "rev (rev xs) = xs"
  apply (induct_tac xs)
  apply (auto)
done
```

So Zeno has helped us find a shorter proof which can be checked to be correct in Isabelle.

Proof in Zeno using lemmas

Of course we can also augment the proof search in Zeno by giving it some user provided lemmas. If we supply Zeno the lemma it found in the fully automated proof, that is `(rev ((app xs)((Cons x)Empty)))= ((Cons x) (rev xs))`, it can discover a new proof in a trivial amount of time and with only four equality steps and one inductive step. Zeno's output for this proof is given in Figure 7.7, where we have called the given lemma `revApp2`.

Figure 7.7 `revApp2` augmented list reversal idempotence proof in Zeno

```
[Goal] (rev (rev xs)) = xs

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev ((Cons ?yd) ?zd))) = ((Cons ?yd) ?zd)
  | [rev2] (rev ((app (rev ?zd)) ((Cons ?yd) Empty))) = ((Cons ?yd) ?zd)
  | [revApp2] ((Cons ?yd) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [I1 Hyp] ((Cons ?yd) ?zd) = ((Cons ?yd) ?zd)
  | True
  ////////////////////////////////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev Empty)) = Empty
  | [rev1] (rev Empty) = Empty
  | [rev1] Empty = Empty
  | True
  ////////////////////////////////////////

Lemmas proven:
  (rev (rev !x)) = !x

Lemmas used: revApp2
Equality steps: 4
Induction steps: 1
Computation time: 0.031 sec
```

We could also augment the proof search with the lemma that the Isabelle/HOL tutorial gives us, which is `(rev ((app xs)ys))= ((app (rev ys))(rev xs))`, but this only saves 25 seconds of time, one equality step and one induction step, making Zeno much slower at lemma assisted proofs than Isabelle. Zeno's output for this is given in Figure 7.8

Figure 7.8 revApp augmented list reversal idempotence proof in Zeno

```
[Goal] (rev (rev xs)) = xs

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev ((Cons ?yd) ?zd))) = ((Cons ?yd) ?zd)
  | [rev2] (rev ((app (rev ?zd)) ((Cons ?yd) Empty))) = ((Cons ?yd) ?zd)
  | [revApp] ((app (rev ((Cons ?yd) Empty))) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [rev2] ((app ((app (rev Empty)) ((Cons ?yd) Empty))) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [rev1] ((app ((app Empty) ((Cons ?yd) Empty))) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [app1] ((app ((Cons ?yd) Empty)) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [app2] ((Cons ?yd) ((app Empty) (rev (rev ?zd)))) = ((Cons ?yd) ?zd)
  | [app1] ((Cons ?yd) (rev (rev ?zd))) = ((Cons ?yd) ?zd)
  | [I1 Hyp] ((Cons ?yd) ?zd) = ((Cons ?yd) ?zd)
  | True
  //////////////////////////////////////////////////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (rev (rev Empty)) = Empty
  | [rev1] (rev Empty) = Empty
  | [rev1] Empty = Empty
  | True
  ////////////////////////////////

Lemmas proven:
  (rev (rev !x)) = !x

Lemmas used: revApp
Equality steps: 8
Induction steps: 1
Computation time: 53.011 sec
```

7.1.2 Length of lists

The problem solved in the section is whether the length of two lists appended to each other is the length of each list added together. We show this proof as it is the first question on the "Structural Induction" tutorial of the "Reasoning about Programs" course my supervisor gives.

The Haskell code for this proof is given in Figure 7.9, the **FL** code in Figure 7.10 and Zeno's proof of the lemma `lengthApp` is given in Figure 7.11.

Figure 7.9 List application length Haskell code

```haskell
module Length where
import Zeno

data Nat = Zero | Succ Nat
data List = Empty | Cons Nat List

app :: List -> List -> List
app Empty ys = ys
app (Cons x xs) ys = Cons x (xs ‘app‘ ys)

add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = Succ (add x y)

length :: List -> Nat
length Empty = Zero
length (Cons x xs) = Succ (length xs)

lengthApp :: List -> List -> Lemma Nat
lengthApp xs ys =
       length (xs ‘app‘ ys)
                ===
  ((length xs) ‘add‘ (length ys))
```

Figure 7.10 List application length **FL** code

```
type Nat = Succ Nat | Zero.
type List = Cons Nat List | Empty.

functions
  app : (List -> (List -> List)),
  add : (Nat -> (Nat -> Nat)),
  length : (List -> Nat).

variables
  x : Nat,
  y : Nat,
  xs : List,
  ys : List.

axiom app1
  ((app Empty) ys) = ys.
axiom app2
  ((app ((Cons x) xs)) ys) = ((Cons x) ((app xs) ys)).

axiom add1
  ((add Zero) y) = y.
axiom add2
  ((add (Succ x)) y) = (Succ ((add x) y)).

axiom length1
  (length Empty) = Zero.
axiom length2
  (length ((Cons x) xs)) = (Succ (length xs)).

lemma lengthApp
  (length ((app xs) ys)) = ((add (length xs)) (length ys)).
```

Figure 7.11 List application length proof in Zeno

```
[Goal] (length ((app !x) ys)) = ((add (length !x)) (length ys))

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (length ((app ((Cons ?yd) ?zd)) ys)) = ((add (length ((Cons ?yd) ?zd))) (length ys))
  | [app2] (length ((Cons ?yd) ((app ?zd) ys))) = ((add (length ((Cons ?yd) ?zd))) (length ys))
  | [length2] (Succ (length ((app ?zd) ys))) = ((add (length ((Cons ?yd) ?zd))) (length ys))
  | [length2] (Succ (length ((app ?zd) ys))) = ((add (Succ (length ?zd))) (length ys))
  | [add2] (Succ (length ((app ?zd) ys))) = (Succ ((add (length ?zd)) (length ys)))
  | [I1 Hyp] (Succ (length ((app ?zd) ys))) = (Succ (length ((app ?zd) ys)))
  | True
  ////////////////////////////////////////////////////////////////////////////////////////////////

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  | [I1] (length ((app Empty) ys)) = ((add (length Empty)) (length ys))
  | [app1] (length ys) = ((add (length Empty)) (length ys))
  | [length1] (length ys) = ((add (length Empty)) (length ys))

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [length1] Zero = ((add (length Empty)) (length Empty))
    | [length1] Zero = ((add (length Empty)) Zero)
    | [length1] Zero = ((add Zero) Zero)
    | [add1] Zero = Zero
    | True
    ////////////////////////////////////////////////////////////////

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    | [length1] (length ((Cons ?me) ?ne)) = ((add (length Empty)) (length ((Cons ?me) ?ne)))
    | [length1] (length ((Cons ?me) ?ne)) = ((add Zero) (length ((Cons ?me) ?ne)))
    | [add1] (length ((Cons ?me) ?ne)) = (length ((Cons ?me) ?ne))
    | True
    ///////////////////////////////////////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////////////////////////////////////////

Lemmas proven:
  (length ((app !x) ys)) = ((add (length !x)) (length ys))

Lemmas used: none
Equality steps: 6
Inductive steps: 1
Computation time: 15.793 sec
```

Having solved this with just one inductive step in Zeno we hypothesised that the inductive tactic (`induct_tac`) and the automated prover tactic (`auto`) in Isabelle would be able to do the proof, which was indeed the case. In Figure 7.12 we have the successful proof. Note that we knew which variable to perform induction upon only because we already had the proof from Zeno, without Zeno this would have to be inferred by the user[3].

[3]This might require the user to enumerate the entire search space of two...

Figure 7.12 List application length proof in Isabelle

```
theory Length
imports Datatype
begin

datatype list
  = Empty                              ("[]")
  | Cons nat list                      (infixr "#" 64)

primrec app :: "list => list => list"  (infixr "++" 65)
where
  "app [] ys = ys" |
  "app (x # xs) ys = x # (app xs ys)"

primrec add :: "nat => nat => nat"
where
  "add 0 y = y" |
  "add (Suc x) y = Suc (add x y)"

primrec length :: "list => nat"
where
  "length [] = 0" |
  "length (x # xs) = Suc (length xs)"

lemma lengthApp [simp]:
    "length (xs ++ ys) = add (length xs) (length ys)"
  apply (induct_tac xs)
  apply (auto)
done
```

7.2 Partially Solved Problems

Described here are problems which we are able to solve myself with an **FL** tableau, but which require too many steps for Zeno to be able to find them in a reasonable amount of time, due to the exponential nature of its search space.

7.2.1 Reverse of appended lists

In Isabelle's proof of the idempotence of list reversal (see Section 7.1.1), it makes use of the intermediary lemma `revApp`, presented below in Isabelle's syntax:

```
lemma revApp [simp]: "rev (xs ++ ys) = (rev ys) ++ (rev xs)"
```

The Isabelle/HOL proof of this lemma requires two intermediary lemmas, appEmpty and appAssoc, and is detailed in Figure 7.13. The definitions of the various functions are as given in Figure 7.4.

Figure 7.13 revApp proof in Isabelle

```
lemma appEmpty [simp]: "xs ++ [] = xs"
  apply (induct_tac xs)
  apply (auto)
done

lemma appAssoc [simp]: "(xs ++ ys) ++ zs = xs ++ (ys ++ zs)"
  apply (induct_tac xs)
  apply (auto)
done

lemma revApp [simp]: "rev (xs ++ ys) = (rev ys) ++ (rev xs)"
  apply (induct_tac xs)
  apply (auto)
done
```

Unfortunately the number of steps and axioms in this proof mean that it cannot be found by Zeno in a reasonable amount of time. We have however found an **FL** tableau of this proof that Zeno would have eventually found, which we have detailed in Figure 7.14 using a similar style to the one in which Zeno outputs its proofs. Note that we have used the infix symbols # for Cons and ++ for app, as well as simplified bracketing, to aid reading the proof. The definitions for all the rules used can be found in Figure 7.2.

Figure 7.14 Simplified **FL** tableau of the proof of `revApp`

```
[Goal] rev (xs ++ ys) = rev ys ++ rev xs

  [I1] rev ((x # xs') ++ ys) = rev ys ++ rev (x # xs')
  [rev2] rev (xs' ++ ys) ++ (x # []) = rev ys ++ rev (x # xs')
  [I1 Hyp] (rev ys ++ rev xs') ++ (x # []) = rev ys ++ rev (x # xs')

    [I2] ((z # zs) ++ rev xs') ++ (x # []) = (z # zs) ++ rev (x # xs')
    [app2] (z # (zs ++ rev xs')) ++ (x # []) = (z # zs) ++ rev (x # xs')
    [app2] z # ((zs ++ rev xs') ++ (x # [])) = (z # zs) ++ rev (x # xs')
    [I2 Hyp] z # (zs ++ rev (x # xs')) = (z # zs) ++ rev (x # xs')
    [app2] z # (zs ++ rev (x # xs')) = z # (zs ++ rev (x # xs'))
    True

    [I2] ([] ++ rev xs') ++ (x # []) = [] ++ rev (x # xs')
    [app1] rev xs' ++ (x # []) = [] ++ rev (x # xs')
    [app1] rev xs' ++ (x # []) = rev (x # xs')
    [rev2] rev (x # xs') = rev (x # xs')
    True

  [I1] rev ([] ++ ys) = rev ys ++ rev []
  [rev1] rev ys = rev ys ++ rev []
  [rev1] rev ys = rev ys ++ []

    [I2] (z # zs) = (z # zs) ++ []
    [app2] (z # zs) = z # (zs ++ [])
    [I2 Hyp] (z # zs) = (z # zs)
    True

    [I2] [] = [] ++ []
    [app1] [] = []
    True
```

The auxiliary lemmas generated in this proof are the ones corresponding to our two `[I2]` steps:

```
(!x ++ rev xs') ++ (x # []) = !x ++ rev (x # xs')
!x = !x ++ []
```

We can see that the second lemma is the exact `appEmpty` lemma required by Isabelle for the proof in Figure 7.13. The first lemma is on close inspection a very specific case of the `appAssoc` lemma. If we apply the definition of `rev` to the `rev (x # xs')` term we get the term below, which is more obviously a specific case of `appAssoc`.

```
(!x ++ rev xs') ++ (x # []) = !x ++ (rev xs' ++ (x # []))
```

We can in fact replace `appAssoc` with the more specific case found by the **FL** tableau and Isabelle can still complete the proof, which is shown in Figure 7.15.

Figure 7.15 `revApp` proof in Isabelle/HOL v2

```
lemma appEmpty [simp]: "xs ++ [] = xs"
  apply (induct_tac xs)
  apply (auto)
done

lemma appAssoc2 [simp]:
    "(xs ++ rev ys) ++ (x # []) = xs ++ rev (x # ys)"
  apply (induct_tac xs)
  apply (auto)
done

lemma revApp [simp]: "rev (xs ++ ys) = (rev ys) ++ (rev xs)"
  apply (induct_tac xs)
  apply (auto)
done
```

7.2.2 Flattening binary trees

The problem we have solved here is whether the number of nodes in a binary tree is equal to the length of the list of the flattened tree. The Haskell definition of this is given in Figure 7.16.

This problem, as specified in the given Haskell code but with a polymorphic `BTree` type, is the only problem in the first coursework for the "Reasoning about Programs" course my project supervisor sets. In the coursework we are given the lemma about the length of appended lists found in Section 7.1.2 as an assumption, this lemma is called `lengthApp` in the Haskell code.

In Figure 7.17 we have the proof of this property using Isabelle/HOL. Notice that we have added another lemma (`addTail`) on top of `lengthApp`, which is required to complete the proof. It worth noting however that if we replace our definition with Isabelle's built in definition of addition then this lemma is no longer required.

As we can see from the success of this proof Isabelle is as good as a first-year at inductive reasoning. Unfortunately however, Zeno is not. It has yet to generate a proof, even after spending much longer than the average first-year does on their coursework. However in Figure 7.19 we give a proof for the coursework as an **FL** tableau (in something resembling Zeno's output style) without any lemmas, just the **FL** axioms for the function definitions in Figure 7.18. Note that we have used the infix symbols `#` for `Cons`, `++` for `app` and `+` for `add`, as well as simplified bracketing, all to aid in reading the proof.

Figure 7.19 FL tableau proof of the binary tree flattening problem

```
[Goal] numBEs ts = length (flatten ts)

  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  [I1] numBEs (BTnode lhs x rhs) = length (flatten (BTnode lhs x rhs))
  [numBEs2] Succ ((numBEs lhs) + (numBEs rhs)) = ...
  [I1 Hyp] Succ ((length (flatten lhs)) + (numBEs rhs)) = ...
  [I1 Hyp] Succ ((length (flatten lhs)) + (length (flatten rhs))) = ...
  [flatten2] ... = length ((flatten lhs) ++ ((x # []) ++ (flatten rhs)))
  [app2] ... = length ((flatten lhs) ++ (x # ([] ++ (flatten rhs))))
  [app1] ... = length ((flatten lhs) ++ (x # (flatten rhs))

    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    [I2] Succ ((length (y # ys)) + (length (flatten rhs))) =
           length ((y # ys) ++ (x # (flatten rhs)))
    [length2] Succ ((Succ (length ys)) + (length (flatten rhs))) = ...
    [add2] Succ (Succ (length ys + (length (flatten rhs)))) = ...
    [I2 Hyp] Succ (length (ys ++ (x # (flatten rhs)))) =
                 length ((y # ys) ++ (x # (flatten rhs)))
    [app2] ... = length (y # (ys ++ (x # (flatten rhs))))
    [length2] Succ (length (ys ++ (x # (flatten rhs)))) =
                Succ (length (ys ++ (x # (flatten rhs))))
    True
    /////////////////////////////////////////////////////////////////////


    \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    [I2] Succ ((length []) + (length (flatten rhs))) =
           length ([] ++ (x # (flatten rhs)))
    [length1] Succ (Zero + (length (flatten rhs))) = ...
    [add1] Succ (length (flatten rhs)) =
             length ([] ++ (x # (flatten rhs)))
    [app1] ... = length (x # (flatten rhs))
    [length2] Succ (length (flatten rhs)) =
                Succ (length (flatten rhs))
    True
    /////////////////////////////////////////////////////////

  [I1] numBEs BTempty = length (flatten BTempty)
  [numBEs1] Zero = length (flatten BTempty)
  [flatten1] Zero = length []
  [length1] Zero = Zero
  True
  ////////////////////////////////////////////////////////////////////////
```

The second induction step here corresponds to following lemma:

```
Succ (length !x + (length (flatten rhs))) =
    length (!x ++ (x # (flatten rhs)))
```

Which if we replace `flatten rhs` with an arbitrary term `ys`, and rename `x!` to `xs`, we get:

```
Succ (length xs + length ys) = length (xs ++ (x # ys))
```

This is an interesting lemma in itself, and similar to the `lengthAdd` lemma required by Isabelle. This further demonstrates our method's ability to generate the lemmas a human would otherwise have to infer in the proof process.

7.3 Unsolved Problems

In this Section we detail a few problems that our **FL** tableau method is unable to solve, even though a simple proof does exist, and present the reason for these shortcomings.

7.3.1 A null list is empty

The Haskell function `null` returns `True` if given the empty list, and `False` otherwise. However in its current incarnation **FL** tableau, and indeed any of the **FL** rules underlying it, are unable to find a proof that `null xs = True` implies that `xs` is the empty list. This problem is represented in **FL** with the lemma `nullEmpty` in Figure 7.20. The proof in Isabelle/HOL is very simple and is shown in Figure 7.21, where `case_tac` is the tactic of case analysis, and `==>` is logical implication.

The reason this proof cannot be found is that when Zeno branches on the value of `xs` in the lemma it proves the case where `xs = y # ys` to be false, as it shows `xs` not to be the empty list. It cannot infer as Isabelle does that the condition `null xs = True` means that `xs = y # ys` is a contradiction, and so this branch does not have to be considered. See Section 8.5 for our proposal to fix this issue.

7.3.2 Non-standard recursion

Some functions recurse in such a way that neither Zeno nor Isabelle/HOL can perform standard structural induction on them. Principally, they require a different well-founded ordering than simply the ordering of definite subterms ($\sqsubset_{st}$) used by Zeno (see Figure 4.7).

Take for example an alternative definition of the addition function over the natural numbers (see Figure 3.9 for our original definition):

```
add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = add x (Succ y)
```

This is a perfectly correct definition of `add`. It is in fact tail-recursive and so one could consider it a preferable definition to our previous one from a programmer's standpoint. However it is impossible for our **FL** rules, or Isabelle/HOL's induction tactic, to create inductive proofs using this definition. The Isabelle/HOL proof of the `addZero` lemma in Figure 7.22 fails, as does the equivalent **FL** proof attempt.

The reason for this is that we cannot simply use the well-founded ordering of definite sub-terms $\sqsubset_{st}$ (Figure 4.7), otherwise known as structural induction, on the first argument. We must use a new ordering that takes both arguments of `add` into account. To demonstrate we will define a new and trivially well-founded ordering $\sqsubset_{st}^{\times}$ as:

$$(x', y') \sqsubset_{st}^{\times} (x, y) \Longleftrightarrow (x' \sqsubset_{st} x \vee (x' = x \wedge y' \sqsubset_{st} y))$$

This ordering takes into account both arguments, and through this we can prove that our new definition of `add` is equivalent to our original one:

$$\forall x^{Nat}.\forall y^{Nat} : (Succ\ x) + y = Succ\ (x + y)$$

Note that we have used the infix notation $(+)$ for the following proofs, and that this represents our new definition of `add`.

The base case of $(x, y) = (0, 0)$ is trivial so we have ignored it, but the inductive case of $(x, y) = (Succ\ x', y)$ using our new well-founded ordering is given in Figure 7.23.

Once we have that both definitions are equivalent we can use all the proofs of the old definition for the new definition.

Figure 7.16 Haskell code for the binary tree flattening problem

```
module BTree where
import Zeno

data Nat
  = Zero
  | Succ Nat

data List
  = Empty
  | Cons Nat List

data BTree
  = BTempty
  | BTnode BTree Nat BTree

app :: List -> List -> List
app Empty ys = ys
app (Cons x xs) ys = Cons x (xs `app` ys)

add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = Succ (add x y)

length :: List -> Nat
length Empty = Zero
length (Cons x xs) = Succ (length xs)

flatten :: BTree -> List
flatten BTempty = Empty
flatten (BTnode lhs x rhs) =
  (flatten lhs) `app` ((Cons x Empty) `app` (flatten rhs))

numBEs :: BTree -> Nat
numBEs BTempty = Zero
numBEs (BTnode lhs x rhs) =
  Succ ((numBEs lhs) `add` (numBEs rhs))

lengthApp :: List -> List -> Lemma Nat
lengthApp xs ys =
  (length (xs `app` ys)) === ((length xs) `add` (length ys))

coursework :: BTree -> Lemma Nat
coursework ts =
  (numBEs ts) === (length (flatten ts))
```

Figure 7.17 Isabelle/HOL proof of the binary tree flattening problem

```
theory BTree
imports Datatype
begin

datatype list
  = Empty                              ("[]")
  | Cons nat list                      (infixr "#" 64)

datatype btree
  = BTempty
  | BTnode btree nat btree

primrec app :: "list => list => list"  (infixr "++" 65)
where
  "app [] ys = ys" |
  "app (x # xs) ys = x # (app xs ys)"

primrec add :: "nat => nat => nat"
where
  "add 0 y = y" |
  "add (Suc x) y = Suc (add x y)"

primrec length :: "list => nat"
where
  "length [] = 0" |
  "length (x # xs) = Suc (length xs)"

primrec flatten :: "btree => list"
where
  "flatten BTempty = []" |
  "flatten (BTnode lhs x rhs) =
    (flatten lhs) ++ ((x # []) ++ (flatten rhs))"

primrec numBEs :: "btree => nat"
where
  "numBEs BTempty = 0" |
  "numBEs (BTnode lhs x rhs)
    = Suc (add (numBEs lhs) (numBEs rhs))"

lemma lenApp [simp] :
    "length (xs ++ ys) = add (length xs) (length ys)"
  apply (induct_tac xs)
  apply (auto)
done

lemma addTail [simp] : "add x (Suc y) = add (Suc x) y"
  apply (induct_tac x)
  apply (auto)
done

lemma flatLen [simp] : "numBEs ts = length (flatten ts)"
  apply (induct_tac ts)
  apply (auto)
done
```

Figure 7.18 FL code for the binary tree flattening problem

```
type Nat = Succ Nat | Zero.
type List = Cons Nat List | Empty.
type BTree = BTNode BTree Nat BTree | BTEmpty.

functions
  length : (List -> Nat),
  app : (List -> (List -> List)),
  add : (Nat -> (Nat -> Nat)),
  flatten : (BTree -> List),
  numBEs : (BTree -> Nat).

variables
  x : Nat,
  y : Nat,
  xs : List,
  ys : List,
  lhs : BTree,
  rhs : BTree,
  ts : BTree.

axiom length1
  (length Empty) = Zero.
axiom length2
  (length ((Cons x) xs)) = (Succ (length xs)).

axiom app1
  ((app Empty) ys) = ys.
axiom app2
  ((app ((Cons x) xs)) ys) = ((Cons x) ((app xs) ys)).

axiom add1
  ((add Zero) y) = y.
axiom add2
  ((add (Succ x)) y) = (Succ ((add x) y)).

axiom flatten1
  (flatten BTEmpty) = Empty.
axiom flatten2
  (flatten (((BTNode lhs) x) rhs)) =
    ((app (flatten lhs)) ((app ((Cons x) Empty)) (flatten rhs))).

axiom numBEs1
  (numBEs BTEmpty) = Zero.
axiom numBEs2
  (numBEs (((BTNode lhs) x) rhs)) =
    (Succ ((add (numBEs lhs)) (numBEs rhs))).

lemma coursework
  (numBEs ts) = (length (flatten ts)).
```

Figure 7.20 FL code for whether a null list is empty

```
type Nat = Succ Nat | Zero.
type Bool = True | False.
type List = Cons Nat List | Empty.

functions
  null : (List -> Bool).

variables
  x : Nat,
  xs : List.

axiom null1
  (null Empty) = True.
axiom null2
  (null ((Cons x) xs)) = False.

lemma nullEmpty
  xs = Empty :- (null xs) = True.
```

Figure 7.21 Isabelle/HOL proof that a null list is empty

```
theory Null
imports Datatype
begin

datatype list
  = Empty                             ("[]")
  | Cons nat list                     (infixr "#" 64)

primrec null :: "list => bool"
where
  "null [] = True" |
  "null (x # xs) = False"

lemma nullEmpty [simp]:
    "null xs ==> xs = []"
  apply (case_tac xs)
  apply (auto)
done
```

Figure 7.22 Failed Isabelle/HOL proof using tail-recursive add

```
theory Add
imports Datatype
begin

primrec add :: "nat => nat => nat"
where
  "add 0 y = y" |
  "add (Suc x) y = add x (Suc y)"

lemma addZero [simp]: "add x 0 = x"
  apply (induct_tac x)
  apply (auto)
...
```

Figure 7.23 Inductive case of addition definition equivalence

$$\forall (x'', y'') \sqsubset^{\times}_{st} (\mathit{Succ}\ x', y) :$$

$(\mathit{Succ}\ x'') + y'' = \mathit{Succ}\ (x'' + y'')$	inductive hypothesis	(7.1)
$(x', \mathit{Succ}\ y) \sqsubset^{\times}_{st} (\mathit{Succ}\ x', y)$	definition of $\sqsubset^{\times}_{st}$	(7.2)
$(\mathit{Succ}\ x') + (\mathit{Succ}\ y) = \mathit{Succ}\ (x' + (\mathit{Succ}\ y))$	∀E on (7.1) with (7.2)	(7.3)
$(\mathit{Succ}\ (\mathit{Succ}\ x')) + y = \mathit{Succ}\ ((\mathit{Succ}\ x') + y)$	definition of (+) on (7.3)	(7.4)

This is our inductive case so we are done. □

Chapter 8

Extensions

In this chapter we will detail a few ways in which we would have applied more work to our project to develop it further.

8.1 Soundness of FL

While we have yet to find a case of our tool producing an incorrect result (proving a formula that is false, or disproving one that is true) this does not preclude such a possibility. Before any further work is done to extend our system it is very important that we first show our existing system to be sound, and that every step it takes is correct with respect to its encoding into first-order logic.

We would need to formally prove all our **FL** rules from Chapter 4 are sound, and in particular that our (INDUCTION) rule (Section 4.8.8) is sound with respect to the principle of well-founded induction on which it is based. Once we have this we will have that our **FL** tableau method is sound, and hence that Zeno itself is sound.

A good method of doing this would be to encode all of our rules into Isabelle and create a verifiable proof of each.

8.2 Covering all of Haskell

Currently Zeno will ignore any functions typed polymorphically or involving primitive types, but these are used used in almost all real Haskell programs so should be addressable in future versions of our tool.

Polymorphically typed variables are interesting as they cannot case analysed or have induction performed on them. However we do not believe they would be very difficult to add into **FL**, requiring only a few extensions to the type system and a guard against using them in the induction and case-completion rules.

Primitively typed variables we believe would be very difficult to add in a useful way to our project, as they have no simple representation as a recursive datatype, and operations on them are not efficient as recursive functions. Taking integer addition as an example, if we calculate $100 + x$ this may require up to one hundred applications of a recursive definition of addition over recursively defined integers.

A trivial extension would be to deal with them as variables of a type we cannot case-analyse or perform induction on, but then they are just ignored rather than addressed.

Another method would be to send all problems involving primitive types to an SMT solver (see Section 2.4.1). This was the approach taken by Dana Xu in her method of functional program verification[39] (see Section 2.6). The problem with this method is that it does not allow for Zeno's proofs to be integrated with proofs over primitive types, it just separates the two so they can be solved individually.

8.3 Integration with an existing proof tool

As a standalone tool Zeno serves as a proof of concept for our **FL** tableau method of functional program verification. To be useful though we believe it should be integrated into an existing theorem proving tool as an augmentation to its existing method of proof.

One interesting integration would be as a proof tactic for Isabelle/HOL. This could, for example, shorten the tutorial example[36] of proving the idempotence of list reversal from the one in Figure 7.5, to the one in Figure 8.1.

Figure 8.1 Zeno as a tactic in Isabelle/HOL

```
theorem revIdm [simp]: "rev (rev xs) = xs"
  apply (zeno)
done
```

Another place we could integrate our **FL** tableau method would be an SMT solver, allowing it to handle the style of proofs that Zeno tackles. The added advantage of this method is that it may allow the intermixing of SMT and **FL** proofs, augmenting the power of the **FL** tableau and allowing it to handle non-recursively typed values, such as integers.

8.4 Lemma generalisation

Zeno's method of applying multiple inductive steps allows it to generate intermediary lemmas that could then possibly be reused in other proofs. One problem with the lemmas that it outputs is they often represent a specific version of a more general lemma, where the more general lemma would be much more useful in future proofs. A good extension would therefore be a method whereby the more specific intermediary lemmas could be generalised.

As an example take the intermediary lemma found in the **FL** tableau proof for the reverse of appended lists in Figure 7.3, written below in Haskell syntax:

```
(xs ++ rev as) ++ [a] = xs ++ rev (a : as)
```

To generalise this we can first apply the definition of `rev` to `rev (a : as)`, giving:

```
(xs ++ rev as) ++ [a] = xs ++ (rev as ++ [a])
```

We can now generalise `rev ys` to be any variable of `List` type (which we name `ys`), and generalise `[a]` to also be any variable `List` type (which we name `zs`), giving us:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

The new generalised lemma above is the associativity of list append, and would be a much more useful lemma to have than the one we started with.

8.5 Tackling contradictory tableau branches

Currently the inability to recognise a contradiction in a branch means that our **FL** tableau method cannot find certain proofs, as demonstrated with the lack of a proof that a null list is empty in Section 7.3.1. To reiterate, our method is unable to prove that `null xs = True` implies that `xs = Empty` (**FL** definition in Figure 7.20). It fails because when it branches on the possible values for `xs` it finds the case when `xs = (y : ys)` to disprove the property, even though we know that `xs = (y : ys)` is a contradiction when `null xs = True`.

An extension to our **FL** rules that accounts for this would be something like the following:

$$(\textsc{contradiction})\ \frac{\Delta, \Gamma, \Phi_1 \nvdash_{\mathbf{FL}} \Phi_2}{\begin{array}{c}\Delta, \Gamma, \Phi_1 \cup \Phi_2 \vdash_{\mathbf{FL}} \Phi_3 \\ \Delta, \Gamma, \Phi_1 \cup \Phi_2 \nvdash_{\mathbf{FL}} \Phi_3\end{array}}$$

This encapsulates the idea that from a contradiction we can prove anything, and means in an **FL** tableau a contradictory branch can be ignored. Before we add an antecedent to a branch we can check whether it can be disproven by the current antecedents, and therefore that adding it would introduce a contradiction.

8.6 Beautifying the proof output

The proofs that Zeno outputs are plain-text and hard to follow. One simple improvement would be to remove all unnecessary bracketing, as it currently contains every possible bracket.

Another feature we could add would be to output proofs in some markup language that can be easily displayed. HTML would be a good choice as we could then display the proof in a web-browser. We could have separate proof branches be collapsable for easier reading, possibly with highlighting/-colouring for different symbols or for different variable types.

A possible option would be to output proofs as a theorem in Isabelle/HOL. This serves the extra purpose of making sure the proof we have produced is sound in the absence of an overall soundness proof for our tool.

8.7 Reducing complexity

The run-time of a proof in **FL** tableau is, in worst case, exponential in the length of the proof required. Unfortunately our current implementation in Zeno seems to lead to exponential run-time in the average case, and many proofs that can be done using an **FL** tableau on paper are unable to be found in a reasonable time. Zeno is in great need of more methods to trim its search space, perhaps in such a way that we lose some proofs, but so that most proofs can be found quickly.

As a side note if we implemented **FL** tableau as a non-deterministic algorithm it would have run-time polynomial to the length of the proof. Obviously then a useful extension to this project would be a method that allows non-deterministic polynomial time algorithms to be calculated deterministically in polynomial time.

Chapter 9

Conclusions

In Function Logic we believe we have found a nice representation of functional programs, and a useful subset of logical properties we can express about them. While it has many restrictions compared to first-order logic, particularly the lack of negation and existential quantifiers, we believe a great majority of program properties can be expressed in this way.

Function Logic Tableauu, and so Zeno, has what we consider to be one major advantage and one major flaw:

- The advantage of our **FL** tableauu method is its ability to infer auxiliary lemmas by performing nested induction on arbitrary terms. It can therefore complete complex proofs without any input from the programmer. This is particularly useful when one is using many self-defined functions, for which there will be no existing background lemmas defined.
- The disadvantage of **FL** tableau is that in worst case its complexity is exponential in the length of the proof to be found. Most practical uses of functional programming have very complex functions with many auxiliary function calls. Proofs of properties over these functions would be very deep, and so completely infeasible for Zeno. It is this issue that places **FL** tableau more in the realm of automated theorem proving than program verification.

It is our opinion that there is a a lot of scope for developing the work detailed here, and that this method could one day allow for the fully automated proof of complex theorems, if only when we have the computational power to make it feasible.

Bibliography

[1] Yices: An smt solver. `http://yices.csl.sri.com`. Technical documentation and tutorials for the Yices SMT Solver.

[2] Prover9 manual. `http://www.cs.unm.edu/~mccune/mace4/manual/Dec-2007`, December 2007.

[3] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, New York, NY, USA, 2005. ACM.

[4] Peter Dybjer Ana Bove. Dependent types at work. Lecture Notes for the LerNet Summer School, February 2008.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs.

[6] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[7] Constantinos Bartzis. Decision procedures for presburger arithmetic. Presentation.

[8] Matthias Blume and David McAllester. A sound (and complete) model of contracts. *SIGPLAN Not.*, 39(9):189–200, 2004.

[9] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in smt solvers. In *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5, New York, NY, USA, 2008. ACM.

[10] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, 2000.

[11] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories.

[12] Leonardo de Moura and Nikolaj Bjorne. Z3: An efficient smt solver. *Lecture Notes in Computer Science*, pages 337–340, 2008.

[13] David L. Detlefs, K. Rustan M. Leino, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. 1998.

[14] Bruno Dutertre and Leonardo de Moura. The yices smt solver.

[15] Levent Erkök and John Matthews. Using yices as an automated solver in isabelle/hol. In *In Automated Formal Methods '08*, pages 3–13. ACM Press, 2008.

[16] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM.

[17] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM.

[18] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39(2):176–210, 1934.

[19] Jean Goubault-Larrecq. A simple deduction system for first-order logic with equality, free constructors and induction. 1999.

[20] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *In FLOPS '06: Functional and Logic Programming: 8th International Symposium*, pages 208–225. Springer-Verlag, 2006.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[22] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant - A Tutorial*, October 2008.

[23] Michael Huth. A constraint-based validity solver for intuitionistic propositional logic.

[24] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boute, Warren Burton, Joseph Fase, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Report on the Programming Language Haskell 98*, February 1999.

[25] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 27–38, New York, NY, USA, 2008. ACM.

[26] John Longley and Randy Pollack. Reasoning about cbv functional programs in isabelle/hol. In *TPHOLs*, pages 201–216, 2004.

[27] Claude Marché. *The Krakatoa Verification Tool for JAVA programs*, dec 2009.

[28] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[29] Ulf Norell. *Dependently Typed Programming in Agda.* Chalmers University, Gothenburg.

[30] Steve Reeves. Semantic tableaux as a framework for automated theorem-proving. In *on Advances in artificial intelligence*, pages 125–139, New York, NY, USA, 1987. John Wiley & Sons, Inc.

[31] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[32] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA, 2008. ACM.

[33] Robert E. Shostak. On the sup-inf method for proving presburger formulas. *Journal of the Association for Computing Machinery*, 24, October 1977.

[34] Cesare Tinelli Silvio Ranise. *The SMT-LIB Standard: Version 1.2*, August 2006.

[35] Geoff Sutcliffe, Christoph Benzmüller, Chad E. Brown, and Frank Theiss. Progress in the development of automated theorem proving for higher-order logic. In *CADE-22: Proceedings of the 22nd International Conference on Automated Deduction*, pages 116–130, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] Markus Wenzel Tobias Nipkow, Lawrence C. Paulson. *Isabelle HOL - A Proof Assistant for Higher-Order Logic.* University of Cambridge, Universitt Mnchen, April 2009.

[37] Florian Haftmann Tobias Rittweiler. *Haskabelle - converting Haskell source files to Isabelle/HOL theories*, September 2009.

[38] Dana N. Xu. Extended static checking for haskell. Presented at the Haskell Workshop 2006, September 2006.

[39] Dana N. Xu. *Static Contract Checking for Haskell.* PhD thesis, Churchill College, University of Cambridge, August 2008.