INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN
Lehr- und Forschungseinheit für Theoretische Informatik

Masterarbeit

Verification and Theorem proving in F*

Sebastian Sturm

Betreuer: Dr. Ulrich Schöpp
Abgabetermin: 18. Dezember 2018

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. Dezember 2018

. . . .
(Sebastian Sturm)

Abstract

F* is a dependently typed programming language usable both for verification and as a proof assistant. It combines manual proving with automatic, SMT based, approaches.

The goal of this thesis is to test F*'s verification and proving capabilities on two illustrative examples. First I implemented and verified insertion and deletion into and from red-black trees using the refinement types feature. The main contribution then was an implementation of the Poplmark-Reloaded theorem proving challenge in F*. These challenge mainly consists of proving normalisation of a version of the simply typed lambda calculus using logical relations. In that part I used mostly Curry-Horward style proofs.

Contents

1 Introduction

The technology of theorem provers and verification oriented programming languages constantly evolves. One goal of this evolution is to make theorem provers more usable and verification applicable to a wider range of practical problems. The most widely used theorem prover is Coq (Coq Development Team [2017]). It is very powerful and applicable to a wide range of verification problems, so that even a quite complete C Compiler CompCert (Kästner et al. [2018]) could be developed and verified using it. But even this only lightly optimising compiler needed several person years to be build. For more complex languages and cutting edge optimisations the verification task would nearly be unmanageable. But proofing compilers to be semantic preserving is quite important, not only because bugs introduced by the compiler are notoriously hard to find, but also because usually only the source code is verified and a buggy compiler can destroy all proven correctness properties. To resolve this dilemma, even more powerful and usable theorem provers are needed. The Poplmark challenge (Aydemir et al. [2005]) provides a framework for comparing the capabilities and the progress for theorem provers in mechanising programming language meta-theory. It focuses mainly on binding structures, such as let bindings or function abstraction. This challenge was further extended by Poplmark-Reloaded (Pientka [2018]). It contains challenges around the problem of proving normalisation of the simply typed lambda calculus (STLC) using logical relations.

Quite a few new approaches were developed in recent years, one of it is the F* verification-oriented programming language, developed at Microsoft Reseach/Inria. It is promising, since it combines interactive theorem proving with semi-automatic program verification using a SMT solver. It is a dependently typed, call-by-value language with support for verifying effectful programs, i.e. programs featuring state or exceptions. To make verified programs practically useful, one can extract it to either F# or OCaml and a limited subset can also be indirectly extracted to C. At the moment F* is used in the project Everest (Project Everest), which tries to implement a fully verified TLS stack miTLS (miTLS), including a verified cryptography library HACL* (HACL*).

The goal of this thesis was to test the basic proving capabilities of F*. This was mainly achieved by implementing the Poplmark-Reloaded challenge. As a theorem proving challenge it is especially suitable for this task. Since the use cases listed above are part of a very special field, it was not obvious from the beginning, that F* is suitable for this task. So test F* and get used to it, I first implemented and verified Red-Black trees (Okasaki [1993]) in F*. They are a commonly used data structure, which can be implemented in a reasonable amount of code, while still the correctness proves are not quite trivial. I only considered the basic proving capabilities here and therefore I did not use and discuss special features like verifying stateful programs.

In the next section I will give a short introduction to the features of F*, that are relevant for this thesis. Then in Section 3 I will discuss in some detail the verified implementation of insertion and deletion into and from red-black trees. The normalisation proof of STLC in turn is discussed in Section 4. Finally I will discuss the role of the SMT solver in F* in Section 5.

2 Short introduction to F*

The syntax of F* is inspired by the syntax of F# and OCaml with some modifications. In this presentation of F* I assume basic knowledge of a ML like language and will point out differences on the fly, while presenting the proving capabilities of F*. One mainly syntactical difference is, that F* supports separate type declarations. These type declarations are inspired by the declarations inside SML module declarations, but since F* does not support ML-style modules, they are simply prepended. E.g.

```
let f (a:int) (b:int):int = a + b
```

can also be written as

```
val f: int -> int -> int
let f a b = a + b
```

This separation is especially useful for theorems, as the type, encoding the assertion, usually gets quite complex. Since it is also common in mathematics, to split the assertions and the proof of a theorem, the separate type style is used mainly.

But a normal ML-style type system is not enough for a theorem prover. For that reason F* is designed as a dependently typed language. It essentially means, that types can depend on values. Such dependencies can be expressed through type valued functions, value parameterised general abstract data types (GADTs) and refinement types. I will discuss the last one in Section 2.2. But how can one use functions that return types. In type declarations one can name the value of a type and use it later on. E.g. `x:a -> b x` is the type of a function, that takes an argument of some type `a`. The value of this argument is now named `x`. `b` then has to be a function, that takes a value of type `a` and returns itself a type, that can depend on `x`. But type valued function would cause trouble with usual ML style type constructors, which are written postfix. E.g. `int list` could be parsed in two ways: As the `list` constructor applied to `int`, or as a function `int` applied to the argument `list`. To avoid this, type constructors in F* are handled as special type level function and thus written prefix.

But still what is the return type of `b` or in other words: What is the type of types? The type of base types is `Type0` which is a special abbreviation for `Type u#0`. Its type again is `Type u#1` and so on. With `u#n` one specifies the type universe. Definitions in F* can even be polymorphic in the type universe. For example `list` has the type `Type u#a -> Type u#a` and therefore `[1;2;3]` of type `list int` and `[int; bool; int -> bool]` of type `list Type0` are valid expressions in F*. To make F* more convenient, it has the special `Type` keyword, which tries to infer the universe, if possible.

Apart from the usual ML style type declarations, which is valid in F* too, there is also a more powerful style that supports GADTs. Consider for example the following

```
type vector (a:Type) : nat -> Type =
  | Nil :  vector a 0
  | Cons : hd:a -> #n:nat -> tl:vector a n -> vector a (n + 1)
```

This is the running example of a dependently typed list. In the declaration `vector (a:Type) : nat -> Type` the `a:Type` argument is the one, that behaves a lot like of the shelf polymorphism.

This argument is fixed in the type and does not depend on arguments of its constructors. Of course values are allowed there too. The `nat -> Type` part on the other hand indicates, that the type indeed depends on the natural number, with like in the `Cons` case, can be given though or computed from arguments. So the result type of all cases needs to name the type argument `a`, but are free to choose the natural number. Since we want it to describe the length of the vector, we declare `Nil` as `vector a 0`. `Cons` that should contain other values is declared like a function, that takes an element, a natural number and a `vector a` of that length and return a new `vector a`, that's one element longer.

You may have noticed the `#` before the `n:nat`. This marks an argument as implicit, i.e. you usually don't care about it, since F* tries to infer it, where needed. We require the natural number argument only, because we want to say, that `tl` has precisely this length. It is therefore fully determined by the `tl` argument and can easily inferred from its type.

In F* a type declaration does not only define a type. It additionally derives equality predicates, discriminators and projectors. Discriminators in F* are named like the constructors with a tailing `?` and are functions that return `true`, if the given argument is of the corresponding constructor. So `Nil? v` return `true` if and only if `v` is a vector of length 0.

A projector can be used, if one is already sure about the case of the variant and wants to get a field out of it, without a pattern match. They may use the argument names given in the type definition. The projector for the `hd` argument of our `vector a n` for example is `Cons?.hd`. The general scheme is the name of the constructor followed by `?.` and then the name of the argument. Our example projector has the type `#a:Type -> #n:nat -> v:vector a n {Cons? v} -> a`. Here again `a` and `n` are implicit arguments, required to introduce the arguments of `vector` and generally ignored/inferred. An interesting point here is the `{Cons? v}` after the `vector` type. It is called refinement and will be discussed in more detail in Section 2.2. Here it requires the given vector to be a `Cons`. This requirement is checked at verification time.

2.1 Effects

F* tries to be both, a theorem prover and a verification oriented programming language. To be sound as a theorem prover, all function used in proofs must normalise. On the other hand general purpose programming languages are useless without side effects. To support both, F* features a sophisticated effect system. The most powerful effect in F* is `ALL`. This effect allows anything, that can be written in a ML like language, including side effects. Beyond that it has special effects to write stateful computations (`STATE`) and computations with exceptions (`EXN`). It also features an effect for computations, that do not or cannot be proven terminating (`DIV`). The most restrictive effect finally is `PURE`, which marks pure, normalising computations. `PURE` is also the effect used for proof terms. But F* features extraction to F# or OCaml and one usually does not want all proofs to be included in the extraction. To this end F* has an additional effect `GHOST`, which is like `PURE`, but not extracted. If all these effects would exist in isolation, they wouldn't make much sense, since one would need to reimplement algorithms for every effect. To solve this F* arranges the effects in a lattice, with a implicit top element $\top$, like shown below.

$$\begin{tikzcd}
\mathsf{PURE} \arrow[rr] \arrow[dr] & & \mathsf{DIV} \arrow[rr] \arrow[dr] & & \mathsf{STATE} \arrow[rr] & & \mathsf{ALL} \\
& \mathsf{GHOST} & & \mathsf{EXN} \arrow[urrr] & & &
\end{tikzcd}$$

For every edge in this lattice a lifting function is provided. This way one need only one definition for pure normalising functions and can use them both in proofs and actual production code. The only limitation is, that the usage of the $\top$ element is prohibited, as it would allow to mix `GHOST` functions with extracted functions, which in turn would result in an unknown symbol error in F# or OCaml.

The direct usage of these effects is very inconvenient, since they are tightly coupled with the monadic weakest precondition framework and they can be further refined by the user. Since most of the time one does not need this extra power, F* defines wrappers with meaningful defaults. For example to specify a function as `PURE` it suffices to mark the return type with `Tot`.

```
effect Tot (a:Type) = PURE a (pure_null_wp a)
let pure_null_wp  (a:Type) (p:a -> Gtot Type0) =
                forall (any_result:a). p any_result
```

The first argument of a primitive effect is the type of the computed value and the second argument is a weakest precondition transformer, i.e a function that maps postconditions, indexed by the result of the computation, to preconditions. So `pure_null_wp a` has to be a function of type `(a -> Type0) -> Type0`. But `Tot` describes a pure computation and weakest preconditions are not meaningful in this context. Therefore any assertion that holds after the computation, has to hold as well before the computation and the validity of a assertion cannot depend on its result. `pure_null_wp` enforces this by requiring the postcondition, that holds on the result of the computation, to hold on any value of correct type as precondition.

As an example of effects consider the usual `length` function that has the type `#a:Type -> Tot (list a -> Tot nat)`. We have already seen function signatures without any effect specifications. This is valid in F*, since, if one leaves out the effect, `Tot` is used. In this paper I mostly specify the effect of the fully applied function and leave partial applications at the default. As I present only proofs and terminating algorithms, I use only `Tot` and `GTot`, which is the convenient version of `GHOST`.

So far I have talked about function with proven termination, but said nothing about how it is done. To prove normalisation F* uses a semantic criterion based on a well-founded ordering ($\prec$) `: #a:Type -> #b:Type -> a -> b -> Type`, over all terms (Swamy et al. [2016]). In addition every recursive function f has its own decreasing metric δ_f, with is itself a pure normalising function. It takes the arguments of is associated function as input. Consider a recursive function definition of the from

```
let rec f1 a1 ... an = ...
and f2 b1 ... bm = ...
```

If whenever say f_1 calls f_2, then $\delta_{f_2} b_1 \dots b_m \prec \delta_{f_1} a_1 \dots a_n$ must hold. $\prec$ is defined in many cases, as one would expect it to be, precisely

- Given i, j: `nat` it is $i \prec j$ if and only if $i < j$. Since `nat` in F* is not defined as inductive datatype, this extra rule is necessary.
- There is a special type `lex_t` described later, with forces lexicographic ordering instead of the default for inductive data types
- if $e = De_1 \dots e_n$, where e is not of type `lex_t`, D a constructor and $e_1, \dots, e_n$ arguments to the constructor, then $e_i \prec e$ for all $i \in \{1, \dots, n\}$.

The type `lex_t` is defined as

```
type lex_t =
  | LexTop  : lex_t
  | LexCons : #a:Type -> a -> lex_t -> lex_t
```

and for well typed pure terms v, v_1, v_2, v_1', v_2' the ordering is defined as

- `LexCons` v_1 $v_2 \prec$ `LexCons` v_1' v_2' if and only if $v_1 \prec v_1'$ and $v_2 \prec v_2'$
- If $v : \texttt{let}_{\texttt{t}}$ and $v \neq$ `LexTop` then $v \prec$ `LexTop`

Since `lex_t` is often used, F* provides syntactic sugar for it. `%[e1;e2;e2]` is the same as `LexCons e1 (LexCons e2 (LexCons e3 LexTop))`. The default decreasing metric is the `lex_t` list of all function arguments. If the default is not sufficient, one can override it with a `decreases` clause. For example

```
val ackermann_swap: n:nat -> m:nat -> Tot nat (decreases %[m;n])
let rec ackermann_swap n m =
   if m=0 then n + 1
   else if n = 0 then ackermann_swap 1 (m - 1)
   else ackermann_swap (ackermann_swap (n - 1) m)
                       (m - 1)
```

defines the ackermann function with arguments flipped. The `(decreases %[m;n])` then explicitly states that it decreases in m and n in lexicographic order. E.g. in the first recursive call `\%[m-1;1]`$\prec$`\%[m;n]` because of the first rule for `lex_t`.

2.2 Refinement Types and SMT in F*

A refinement of a type statement $x : t$ is of the from $x : t\{\phi\}$ where ϕ is a term of type `Type0`, possibly using x. It is the type of all values x of type t with satisfy the predicate ϕ. This construct then is a subtype of t and thus $x : t\{\phi\}$ can freely be used, where a t is expected. One can view an expression of type $x : t\{\phi\}$ as one of type t with the additional knowledge, that it satisfies ϕ. A commonly used refinement type in F* is `nat`, the type of arbitrary precision natural numbers. It is defined as `type nat = i:int {i >= 0}`. `int` is in F* a primitive type for arbitrary precision signed integers. This definition has the advantage, that F* only needs one version of the basic arithmetic operators, because `nat` is a subtype and therefore the operators work with `nat`s too. The only extra requirement are lemmata, that certain operations again yield a `nat`.

Refinement types are especially useful, when code should be extracted. F* then simply removes the refinement and extracts the unrefined type. Therefore one has a lot control over the runtime representation of types and the extracted code, while one can still enrich them with assertions and proofs. `int` and therefore `nat` for example are represented in OCaml as the integer type of Zarith, a highly optimised arbitrary precision integer library. In contrast Coq, with its inductively defined data type for natural numbers, needs special logic in the extractor, to gain decent performance.

But how do one write a expression which has type $x : t\{\phi\}$? The short answer is: One writes a expression of type t and prove that it indeed satisfies ϕ. Normally proving ϕ means, giving an expression of type ϕ. This is not the case with refinement types. The refinement of a type proven by the SMT solver. In the current implementation of F* the Z3 SMT solver (de Moura and Bjørner [2008]). One can support it by bringing other refinements in scope, from with ϕ follows, but one usually prove ϕ not directly. Consider for example the following type declaration

```
val factorial: x:int{x>=0} -> Tot (y:int {y>0})
```

If I want to implement `factorial` I can assume, that the argument `x` is indeed ≥ 0, but the SMT solver must be able to somehow derive, that the result is > 0. So the refinement `x>=0` is a precondition and the refinement of the result `y>0` is a postcondition. Giving pre- and postcondition as refinement of arguments and result is not the only way of specifying pre- and postconditions, as described later, but I do it this way quite often. For this admittedly quite easy function it is not necessary to give extra lemmata. Here I give the whole implementation and simultaneously the proof, that the result is indeed > 0.

```
let rec factorial x = if x = 0 then 1
                  else x * factorial (x-1)
```

How can the SMT solver prove the postcondition? First it can do a case analysis. In the `then` case it is quite obvious, that the postcondition holds. The interesting part is the `else` case. Here we know, that $x \neq 0$ or `x <> 0` in F* syntax. For the type of `factorial` we also know, that $x \geq 0$, so combined we know, that $x > 0$. Now we have to prove, that $x - 1 \geq 0$, because the call to `factorial` requires it. This follows directly from $x > 0$. But then we know from the postcondition of `factorial`, that `factorial (x-1)` is > 0. Finally multiplying two numbers > 0 yields a number > 0 and we are done.

I pretended so far that the refinement ϕ in $x : t\{\phi\}$ must be a type. This is not completely true. For instance `>=` and `>` in the example above are the boolean comparison operators. If ϕ is a boolean expression instead of a type, the F* automatically inserts a call to `b2t` with converts a boolean to it's corresponding type. `b2t` is not the only construct specially introduced for the use in refinement types. There are also special type versions for and `/\` and or `\/` as well as type constructors for implication `==>`, (logical) equivalence `<==>`, equality `==`, `forall` and `exists`. All these type constructors are not the usual Curry Howard ones, but special for use with SMT and refinement types. More Curry Howard style proofs are supported too, as described in the next section.

The task of the SMT solver is not limited to proofing refinements. It can also prove patterns to be unfeasible and therefore one often can leave whose patterns arms out completely. Also congruence goals are fed into the SMT solver. Most of the time this is advantageous, but, as described in more detail in Section 5.2, it can also be burden, since SMT goals are not proven directly. As already mentioned they are proven by giving helpful lemmata and letting the SMT solver sort out the rest. This sometimes makes it complicated to prove equality in F*. In addition to that, it sometimes complicates the search for errors. If the SMT solver cannot prove, that the erroneous assigned types are not equal up to congruence, it mostly simply gives back "SMT timed out".

As already mentioned further lemmata are given to the SMT solver, by bringing additional refinements into scope. One way to do so, is tying the proof together with the value in question, like in the `factorial` example. This is called the intrinsic style. But can I prove things also separately, possibly in another module? As usual, then we are not interested in the value returned, we simply use the return type `unit` and refine it with the lemma in question. Since there is no point in executing a pure normalising function with return type unit, this is the point for applying our "ghost" effect `GTot`. In the end a lemma finally looks like this

```
val factorial_ge_arg: x:int {x>=0} -> (u:unit {factorial x >= x})
```

This is then the extrinsic style. But how do I invoke such a lemma. To that end F* has the ML `;` operator and like with ML side effecting functions, lemmata can be chained with `;` in F*. But specifying assertion as refinements of the `unit` type is cumbersome and obstructs the view on the actual purpose. Since the real goal is the prove a assertion, not to produce a value, a more handy and powerful way of specifying lemmata was introduced. The keyword for it is `Lemma`. When we use the most simple from our example becomes

```
val factorial_ge_arg: x:int {x>=0} -> Lemma (factorial x >= x)
```

The pattern for this version is `Lemma (ϕ)`, where ϕ is the statement of the lemma. The most general and powerful version is `Lemma (requires pre) (ensures post) (decreases d) [SMTPat pat]`, where `pre` is a precondition to the lemma (can be used instead of refining the arguments), `post` is the postcondition, i.e. the statement of the lemma, `d` is a decreasing metric described in the previous section and finally `pat` is a SMT pattern. Whenever such a pattern appears the lemma is automatically given to the SMT solver, so that the user does not need to state it. Here is our running example again, in the general form, together with a proof. With `assert` one can add additional goals to the SMT solver, that need to hold. It is used here instead of a lemma, to show how invocations of such constructs work.

```
val factorial_ge_arg: x:int -> Lemma (requires x>=0)
  (ensures factorial x >= x)
  (decreases x)
  [SMTPat (factorial x)]
let rec factorial_ge_arg x = if x = 0 then ()
                  else ( assert (x <> 0);
                         factorial_ge_arg (x-1)
                       )
```

But there are theories, that are not efficiently SMT-automatable. In order to use such theories in otherwise SMT automated proofs, F* lately introduced tactics. Similar to Coq tactics, they are small proofs scripts, that automate parts of the proof. Since this feature is added recently, nearly no documentation exist yet and therefore I didn't use this feature at all.

2.3 Constructive Proofs in F*

The little proofs we have seen so far are mostly done through refinement types, but also feature the classic Curry-Howard function arrow as forall. Since F* is a full blown dependently typed language, one can write proofs in it, even without the use of refinement types. As F* has no tactic support for this proof style, one needs to directly write the proof term with the usual Curry-Howard translation of constructs. This style is more explicit than the proofs via refinement types, but on the other hand it is hard to read. This style was mostly used, in the proof of normalisation of STLC and a full example of it can be found in Section 4.3. The rationale there was, that the direct style follows the paper proofs more closely and there is also no point in extraction pure proofs. The F* standard library also has an extra module `FStar.Constructive` to support this style, but since the standard tuples, functions and dependent tuples have better library support, they were preferred. Only the types for constructive disjuction `cor` and equality `ceq` were used.

To give you a taste of constructive proofs in F*, I included a basic example

```
type le (x:int): int -> Type =
  | LeBase: le x x
  | LeStep: #y:int -> inner:le x y -> le x (y+1)

val le_trans: #x:int -> #y:int -> #z:int
  -> le1:le x y -> le2:le y z -> Tot (le x z)
  (decreases le2)
let rec le_trans #n #m #l le1 le2 = match le2 with
  | LeBase -> le1
  | LeStep i -> LeStep (le_trans le1 i)
```

The type `le` represents less-or-equal for integers. In the base case the numbers are equal. The step says, that I can increase the right hand side by one, while retaining less-or-equalness. The function `le_trans` then is the proof of transitivity by induction. In the base case `x` and `y` are equal and therefore `le1` is already a proof for `le x y`. Otherwise I use the induction hypothesis, by calling `le_trans` recursively, and get a value of type `le x (z-1)`. Finally applying `LeStep` yields the right type and completes the proof.

3 Case Study: Red-Black-Trees

Red-black trees are a commonly used and well understood data structure. They were chosen to test the verification capabilities of F*, because these algorithms have good functional implementations in the style of Okasaki (Okasaki [1993]) and they are not trivial, while one can still implement them in a few hundred lines of code. In addition the correctness proofs for the red-black tree algorithm feature some commonly used patterns like invariants. The algorithms verified here conform to the description in Cormen et al. [2013], which presents the algorithms with imperative pseudo code. Nevertheless the descriptions and the proofs can be easily adapted to functional code.

As usual this thesis only presents the important and illustrative parts of the verification. Also to make my points more clearly, I sometimes present code, that uses functions or lemmata, that are discussed later. I think by following the "control flow" the algorithm and verification is easier to understand.

3.1 Basic Setup

First of all we have to define our tree

```
type colour =
  | Red: colour
  | Black: colour

type tree =
  | Leaf: tree
  | Node: c:colour -> l:tree -> v:int -> r:tree -> tree
```

The definition is specialised to integers, because it requires a bit less boilerplate than a polymorphic definition and the point was to test F*'s verification capabilities and not to provide a practically useful red-black tree implementation. The `tree` is a normal binary tree, with a colour field in the nodes. This is needed for the balancing. Leafs do not need a colour, since they are always black in red-black trees. The first interesting property of this tree is sortedness.

```
val min_elem: t:tree{Node? t} -> GTot int
let rec min_elem t = match t with
  | Node _ Leaf v _ -> v
  | Node _ l _ _ -> min_elem l

val sorted: tree -> GTot bool
let rec sorted t = match t with
  | Leaf -> true
  | Node _ l v r -> sorted l && sorted r
        && (if Node? l then v > max_elem l else true)
        && (if Node? r then v < min_elem r else true)
```

Here `min_elem` is the leftmost element of a tree. Similarly `max_elem` is defined as the rightmost element of a tree. The order used here is <, so if the tree is sorted, then the `min_elem`/`max_elem` of a tree are indeed the minimal/maximal element of the tree. I would like to require sortedness in the type of `min_elem`/`max_elem`, but, since they are used in the definition of sorted, this is not possible. As only sorted trees are considered here and the definition is not extracted, because of the `GHOST` effect, this is not an issue.

In the definition of `sorted` the requirement of the sorted subtrees is somewhat clear, so lets have a look at the last two lines. F* does not provide a boolean operator for logical implication, so the `if` simulates one. This is necessary, since `min_elem`/`max_elem` is not meaningful on leaves. For instance the last line says, that if the right subtree is not a leaf, then the value of our current root node `v` has to be smaller than the minimal element of the right subtree. `r` is sorted, as we required it before, so `min_elem` is indeed the minimal element. But then all elements of `r` are greater than `v`. With similar reasoning one sees, that also all elements of `l` are smaller than `v` and therefore the whole current tree is sorted. This approach, taken from (RBTree.fst), was not the first one I considered. At first I tried a version, where I required all elements of the left tree to be smaller than `v` and all elements of the right tree to be greater. To make sure, that the new definition is correct, I retained the more naive one and proved that they are equivalent.

```
val gmember: tree -> int -> GTot bool
let rec gmember t x = match t with
  | Leaf -> false
  | Node _ l v r -> v = x || gmember l x || gmember r x

val sorted_vals : tree -> GTot Type0
let rec sorted_vals = function
  | Leaf -> True
  | Node _ l v r -> sorted_vals l /\ sorted_vals r
        /\ (forall x. gmember l x ==> x < v)
        /\ (forall x. gmember r x ==> v < x)
```

`gmember` stands for `GHOST` member, because it is the membership query for unsorted trees and quite a waste of processor time for sorted trees. But it is very handy in verification, because it is very stable to changes in the tree structure. `sorted_vals` yields a `Type0` instead of a bool, because `forall` is only available as a type constructor. Apart from that, only the two last lines are different. Here I state for example, that all elements, that are member of the left subtree, are greater, then `v`. `sorted_vals` is useful especially, then using sortedness, because of the `forall`. If one knows, that a value is member of a subtree, the SMT solver immediately derives some inequalities. But, as pointed out in more detail in Section 5.2, it is often difficult to prove a `forall`. Therefore with `sorted_vals` one is often at the SMT solver's mercy. With `sorted` on the other hand, the assertion to prove is more concrete.

Sortedness is only the most basic red-black tree property. Two more properties are necessary to ensure, that a tree is balanced. First of all, all children of a red node have to be black. This is formulated in the `red_black_childs` property. The second important predicate is `black_height_correct`. It states, that all paths form the root to a leaf containing the same number of black nodes. An equivalent easier to verify version is used here.

```
val black_height: tree -> GTot nat
let rec black_height t = match t with
  | Leaf -> 0
  | Node Red l _ r -> max (black_height l) (black_height r)
  | Node Black l _ r -> 1 + max (black_height l) (black_height r)


val black_height_correct: tree -> GTot bool
let rec black_height_correct t = match t with
  | Leaf -> true
  | Node _ l _ r -> black_height_correct l && black_height_correct r
      && black_height l = black_height r
```

The `black_height` function calculates the maximal number of black nodes on a path from the root to a leaf. When first implementing it, I had the choice between `max` and `min`. I chose `max` by chance, but `min` would have worked as well. `black_height_correct` then says, that for every node, its left and right subtree must have the same `black_height`. This separation is important, since in the deletion algorithm unbalanced trees can arise as intermediate steps and one needs a meaningful value for `black_height`, to proof the whole algorithm correct.

```
val is_srbtree: t:tree -> GTot bool
let is_srbtree t = sorted t && red_black_childs t
                     && black_height_correct t
```

This definition collects the three requirement discussed so far. By convention, one additionally require that the root of a red-black tree is black. There is another predicate `is_rbtree`, that adds this requirement. `is_srbtree` in some sense describes a subtree of a valid red-black tree. For the sake of convenience I use `is_rbtree` also for subtrees of a red-black tree, which need to have a black root.

3.2 Insertion

As it appeared to be the simpler part, I started with insertion. First one needs to come up with the goal of the specification. So lets have a look at the type of `insert`.

```
val insert : x:int -> t:tree {is_rbtree t} ->
  Tot (t':tree {is_rbtree t'
    /\ (forall z. gmember t z ==> gmember t' z)
    /\ (forall z. gmember t' z ==> z == x \/ gmember t z)
    /\ gmember t' x})
let insert x t = let t' = insert0 Red x t in
   match t' with
    | Node Red a v b -> Node Black a v b
    | _ -> t'
```

As the red-black tree properties are meant as invariants, the `is_rbtree` requirements on both the input and output tree are quite obvious. The second line of the output refinement additionally require, that all elements of the input tree appear in the resulting tree. Similarly I assert, that all elements of the result, but the inserted `x`, are elements of the original tree. Finally the inserted element must be a member of the output tree.

The final `insert` is mainly a wrapper around the real worker function `insert0`. I used a separate worker function, because one needs to restore the black root property at the end and this way the specification of `insert` is clear and not cluttered with internal helper predicates.

I only include a prettified version of the extracted OCaml code for `insert0` here, because the implementation is quite lengthy. This is due to the fact, that I used intrinsic style here, because the properties highly depend on each other, so it was not possible to split it in multiple lemmata and I saw no point in separating it into a implementation and one big proof.

```
let rec insert0  (pc:colour) (x:int) (t:tree) : tree =
        match t with
        | Leaf  -> Node (Red, Leaf, x, Leaf)
        | Node (c,a,v,b) ->
            if x < v then
              let a' = insert0 c x a  in
              let t0 = Node (c, a', v, b)  in
              insert_fixup pc t0
            else if v < x then
              let b' = insert0 c x b  in
              let t0 = Node (c, a, v, b')  in
              insert_fixup pc t0
            else t
```

The basic structure of `insert0` is very close to a normal insertion into a sorted binary tree. A leaf is simply replaced by the new element. It is important to remember, that elements are only added at leaf positions. The inserted node is always red. This ensures, that the `black_height_correct` invariant is still valid. Only the `red_black_childs` invariant may be violated.

On a node we test whether the inserted element is smaller, greater or equal to the value in the current root and depending on that we recursively call `insert0` on the left or right subtree or stop and return respectively. After the recursive call we have to reconstruct the node with the new subtree. Since the `red_black_childs` invariant may be broken, we call `insert_fixup`. Later I will discuss in some detail, what fixing up means and discuss one of the possible cases.

Now that we know how `insert0` works, let's discuss its type

```
val insert0 : pc: colour -> x:int
  -> t:tree{is_srbtree t /\ child_colour pc (get_rcol t)}
  -> Tot (t':tree {post_balance pc t'
       /\ (forall z. gmember t' z <==> z==x \/ gmember t z)
       /\ black_height t == black_height t'
       /\ (Node? t ==> max_elem t' == max x (max_elem t)
                    /\ min_elem t' == min x (min_elem t))
       /\ (Leaf? t ==> max_elem t' == x /\ min_elem t' == x)})
       (decreases t)
```

As in `insert` one requires the original tree, to satisfy the red-black tree invariant. Because `insert0` operates also on subtrees, only `is_srbtree` can be demanded. `get_rcol` extracts the colour of the root node or returns `Black` for a leaf. `pc` stands for parent colour. It is only used during the verification and usually filled with the colour of the parent of the current tree. When we call `insert0` on the whole tree, like in `insert`, we use `Red` as the parent colour, since the root has to be black and should be black after the fixup. `child_colour` checks, that at least one of its argument is `Black` and therefore `t` can be a child of a node with colour `pc`.

In the postcondition one can find common patterns with `insert` too. All three requirements regarding membership are joined in the one predicate `(forall z.  gmember t' z <==> z==x gmember t z)`. Unfortunately the resulting tree does not always satisfy `is_srbtree`. However, as we will see later, the `post_balance` predicate asserts something quite similar. In the code of `insert0` we call it recursively on a subtree and afterwards reconstruct the node. To prove, that the `black_height_correct` invariant is satisfied for the reconstructed node, one needs to know that the new tree has the same `black_height` as the original tree. This is expressed in the next line of the postcondition and holds true, since we only insert red nodes and the fixup does not change the `black_height`. The rest of the assertions of the postcondition are mainly needed for sortedness. Our `sorted` is defined over the maximal and minimal elements of the subtrees. So if I reconstruct a node after a recursive call to `insert0` and I like to prove it sorted, I need to know, how `insert0` may change the `max_elem` and `min_elem`. Since the only new element is `x`, they are either the old maximal/minimal element or `x`.

As the name suggests, `post_balance` is one half of a pair of predicates. `post_balance` describes a tree after the call to `insert_fixup` and `pre_balance` before respectively.

```
val pre_balance : pc:colour -> t:tree -> GTot bool
let pre_balance pc t = match pc with
  | Black -> is_srbtree t || icase1 t || icase2 t || icase3 t
      || ired_red t
  | Red -> is_srbtree t || icase1 t || icase2 t || icase3 t

val post_balance : pc:colour -> t:tree -> GTot bool
let post_balance pc t = match pc with
  | Black -> is_srbtree t || ired_red t
  | Red -> is_srbtree t
```

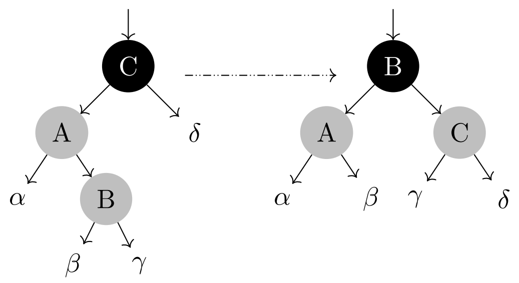
These definitions use many predicates that are not introduced yet. I present the code this way, because I think the big picture is important to understand the details of the predicates. First let's have a look at `post_balance`. `pc` is the colour of the (possibly imaginary) parent node. If it is red, then our tree has satisfy all red-black tree invariants, expect the one for the root node. Otherwise it alternatively can satisfy `ired_red`.

```
val ired_red : tree -> GTot bool
let ired_red t = match t with
  | Node Red (Node Red a _ b) _ c
  | Node Red a _ (Node Red b _ c) ->
    let ba = black_height a in
    is_rbtree a && is_rbtree b && is_rbtree c
    && ba = black_height b && ba = black_height c
    && sorted t
  | _ -> false
```

`ired_red` states, that the root and one of its children is red and it is otherwise a valid red-black subtree. I use `is_rbtree t` here as an abbreviation for `is_srbtree t && get_rcol t == Black`. To reflect, that only trees with two red nodes in a row can occur during the algorithm, `ired_red` is only a valid option for trees with black parent node. Trees satisfying `ired_red` violate the `red_black_childs` invariant, but one does not fix them up immediately and therefore they have to be mentioned in `post_balance`.

If a tree satisfying `ired_red` is returned to `insert`, then `red_black_childs` is restored by repainting the root black. The `black_height` changes, but this is valid for the whole tree. Note that I included the complete code for `insert`. The fact that repainting the root yields a tree satisfying `is_rbtree` is proven alone by the SMT solver.

On the other hand if `insert0` was called recursively, then during the "node reconstruction" one may build a tree with two red nodes starting in a child of the root. There `pre_balance` enters the stage. Since trees satisfying `is_srbtree` or `ired_red` can be yielded directly from insertion, all this cases are valid in `pre_balance` too. But if one reconstructs a node from a tree satisfying `ired_red` and a valid red-black subtree, one gets a tree with a defect in a child and a grandchild. These defects can be fixed and Cormen et al. [2013] distinguishes three kinds of such trees. They are described by the predicates `icase1`, `icase2` and `icase3`. For each of this cases there is its own fixup code. Since the proofs and fixup routines for all three cases are quite similar, I will discuss only one of them, namely `icase2`. The following picture shows a pattern for trees in `icase2` on the left side and their corresponding fixed version on the right side.



As usual circles represent nodes and red is replaced by grey. In the predicate I included the symmetric case too.

```
val icase2 : tree -> GTot bool
let icase2 t = match t with
  | Node Black (Node Red a _ (Node Red b _ c)) _ d
  | Node Black a _ (Node Red (Node Red b _ c) _ d) ->
    let ba = black_height a in
    is_rbtree a && is_rbtree b && is_rbtree c && is_rbtree d
    && ba = black_height b && ba = black_height c
    && ba = black_height d && sorted t
  | _ -> false
```

The first pattern corresponds to the left hand side of the picture. This shows the tree before the fixup. In this case α, β, γ and δ have to be leaves or have to have a black root. Additionally all four subtrees have to be of the same `black_height` and as usual we only work with sorted trees. Writing the actual fixup code is then a simple transcription from the picture. Later on I proved the necessary properties separately. First I proved, that the fixup preserves membership. In F* code it is `forall x. gmember t x <==> gmember (insert_fixup_c23 t) x)`. `insert_fixup_c23` is the fixup function for `icase2` and `icase3`. I combined them, since these cases are quite similar. I will not show the actual proof here, since there is no real one. All nodes and subtrees appear in both, the original and the fixed version. The proof is then so easy that the SMT solver was able to figure it out completely itself.

The next property was sortedness. There the SMT solver actually required some support to see it. Cormen et al. [2013] fixes `icase2` not directly, but converts it to `icase3`. Working this way, the proofs for `icase2` would depend on the proofs for `icase3`. Therefore I inlined the step and fixed it directly. But then the tree structure changes to much. So I proved, that my direct fixup is equivalent to the two step fixup and this was enough for the SMT solver.

Finally I proved that the fixed trees have the same `black_height` then the defect trees and that the fixed trees satisfy the `post_balance` predicate, by proving that they even satisfy `is_srbtree`. The handling of the other cases was analogue, except what sortedness was no matter. All the fixup functions and proofs were then combined to the big `insert_fixup` function called in the code of `insert0`.

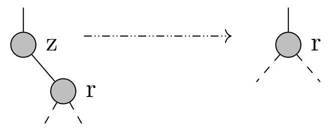
3.3 Deletion

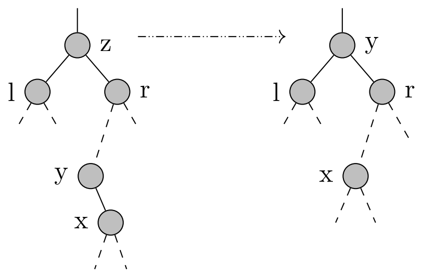
As with insertion I will start with the specification of `delete`.

```
val delete: x:int -> t:tree {is_rbtree t}
  -> Tot (t':tree { is_rbtree t'
  /\ (forall y. gmember t y ==> x == y \/ gmember t' y)
  /\ (forall y. gmember t' y ==> gmember t y)
  /\ not (gmember t' x) })
```

The specification of `delete` is somehow a reversed version of `insert`. Again we require a valid red-black tree as argument and also return one. Now all elements, but `x` appear in the output tree too and we add no element. The last line finally states, that `x` is not a member of the resulting tree. Much like `insert` `delete` is only a wrapper. It calls `delete0` and repaints the root if necessary. The code is almost identical to the code of `insert` and therefore I will not include it here.

When deleting an element, one first has to find it in the tree. This happens through the normal search in a sorted binary tree. Unfortunately the element is not guaranteed to be a leaf element, so we have to distinguish several cases. The function, that performs the actual deletion is called `delete1`.





```
type add_black_pos =
  | PosNone
  | PosSelf
  | PosLeft
  | PosRight

val delete1: t:tree {Node? t /\ is_srbtree t}
                    -> Tot (tp:(tree * add_black_pos)
  {delete1_t t (fst tp) (snd tp)})
let delete1 t = match t with
  | Node Red Leaf _ r -> r,PosNone
  | Node Black Leaf _ r -> fixup_delete (r,PosSelf)
  | Node Red l _ Leaf -> l,PosNone
  | Node Black l _ Leaf -> fixup_delete (l,PosSelf)
  | Node c l _ r -> delete1_help0 t
```

I will come back to add_black_pos and delete1_t later. The argument t is the subtree, which root should be deleted. Therefore it has to be a node and as a subtree of a red-black tree it also satisfies is_srbtree. Now consider the first image and the first two patterns of delete1. The left child of our node to be deleted is a leaf, so we can simply replace the node by its right child. This retains sortedness. If the deleted node is red, then its parent is black and the red_black_childs invariant stays valid. If the deleted node is black, then its parent and the root of r could be red. But in this case fixup_delete will repaint r black. I will discuss fixup_delete later.

The invariant threatened here is black_height_correct. If we remove a black node, we reduce the black_height of the tree by one. If it is a subtree, then the black_height does not match the black_height of its sibling any more. At this point we cannot always fix this issue and therefore we mark the tree as having a too small black_height. One can think of that mark as a additional black. The type add_black_pos describes there, relative to the current root of the tree in question, the mark sits. Since I inlined the fixup into the deletion, I could restrict add_black_pos to four alternatives, which makes verification a lot easier. PosNone says, that no mark exists and therefore the tree requires no fixing. PosSelf says, that the current root carries the mark and PosLeft and PosRight, that the left or right child has the additional black respectively. As we will see during the discussion of fixup_delete, certain, but not all, trees can be fixed immediately and hence fixup_delete is called already in delete1. You may have noticed, that fixup_delete takes a pair of a tree and an add_black_pos as argument and also returns such a pair, so in theory fixup_delete could be implemented as the identity function. Of course it is not and we will see during the discussion of delete1_t, that it precisely describes the possible results. In particular fixup_delete and delete1 always return either PosNone or PosSelf. The third and fourth pattern are symmetric to the first and second.

Instead of add_black_pos one could have also used a datatype for a full path through the tree and separate the fixup from the deletion. On the one hand this would make the code more readable and modular, on the other hand the specification would become more complex, since I would have needed a predicate that states, that the tree has correct black_height, up to the subtree with the mark.

The most interesting pattern of delete1 is the last one. There I like to delete a node, whose children are both no leaves. Since large rebuilding operations are far too expensive, we need some element to replace the deleted element, while still retaining sortedness. Possible candidates are the maximal element of the left tree and the minimal element of the right tree, because they are the next smaller and next larger elements in the tree and therefore would both fit in between the two subtrees. Cormen et al. [2013] chooses the minimal element of the right tree. The second picture illustrates this operation. del_minimum and delete1_help0 are both a bit lengthy, because I used intrinsic proofs. Therefore I only show prettified versions of the extracted OCaml code.

```
let rec del_minimum (t:tree) : (int,tree,add_black_pos) = match t with
    | Node (Red,Leaf,v,r) -> (v, r, PosNone)
    | Node (Black,Leaf,v,r) -> fixup_delete (r, PosSelf)
    | Node (c,l,v,r) -> (match del_minimum l with
         | (vy,x,p) ->
              let tx = Node (c, x, v, r)  in
              (match p with
               | PosSelf  ->
                     let (tx',p') = fixup_delete (tx, PosLeft) in
                     (vy, tx', p')
               | PosNone  -> (vy, tx, p)
              ))

let delete1_help0 (t:tree) :(tree,add_black_pos) = match t with
    | RBTree.Node (c,l,v,r) ->
        (match del_minimum r with
         | (vy,r',p) ->
             let tx = Node (c, l, vy, r')  in
             (match p with
              | PosNone  -> (tx, PosNone)
              | PosSelf  -> fixup_delete (tx, PosRight)
              ))
```

The function `del_minimum` deletes the minimal element of a sorted binary tree and returns it together with the new tree. Again the minimal element is the leftmost element in the tree, i.e. the left child of the minimal element is a leaf. So it is no surprise, that the first two patterns of `del_minimum` are almost identical to those of `delete1`. And like in `delete1` a fixup is needed, if the deleted node is black, for the very same reason. The only difference is, that in `del_minimum` we need to return the deleted value. In the third pattern we simply call `del_minimum` recursively on the left subtree and reconstruct the node with the new left subtree. `fixup_delete` does not always fix a defect completely, but sometimes needs to propagate the defect up. The `add_black_pos` returned by `fixup_delete` is either `PosNone` or `PosSelf` and the same applies to the `add_black_pos`, that `del_minimum` returns. As already mentioned `PosNone` means no defect at all and therefore we can simply return the reconstructed tree together with the minimal element `vy`. If on the other hand the recursive call returns `PosSelf` the left subtree of our reconstructed tree carries the mark and therefore we need to call `fixup_delete` with `PosLeft`. The proof that the node reconstruction and the transition form `PosSelf` to `PosLeft` is sound requires some special handling, but I would like to put this discussion of, until I come to `cases1`.

The actual deletion is then finally perfomed by `delete1_help0`. It first calls `del_minimum` on the right subtree, i.e. extracts and deletes the minimal element of the right subtree. The node is then reconstructed from the unchanged left subtree `l`, the extracted element `vy` and the new right subtree `r'`. The resulting tree is still sorted, because `sorted` says, that `vy` is greater than deleted value `v` with itself is greater than the maximal element of `l`. Therefore `vy` is greater than the maximal element of `l`. Similarly `vy` was the minimal element of the old right subtree `r` and is therefore smaller than all elements of `r'`, in particular smaller than the minimal element of `r'`. Like in `del_minimum` `fixup_delete` may need to propagate the defect up and therefore we may need to call `fixup_delete` and need to return a `add_black_pos`.

As promised, I will now finally discuss `delete1_t`, the specification of `delete1`.

```
val black_tree_pos_height: (tree * add_black_pos) -> GTot nat
let black_tree_pos_height (t, p) = match p with
  | PosLeft
  | PosRight
  | PosNone -> black_height t
  | PosSelf -> 1+black_height t

type delete1_t (t:tree) (t':tree) (p:add_black_pos) =
    Node? t /\ post_fixup (t',p)
    /\ black_height t == black_tree_pos_height (t',p)
    /\ (forall x. gmember t x <==> x == (Node?.v t) \/ (gmember t' x))
    /\ not (gmember t' (Node?.v t))
    /\ (get_rcol t == Black ==> get_rcol t' == Black)
```

The function `black_tree_pos_height` is a wrapper for `black_height`, that takes the additional black into account. One can see this in the `PosSelf` pattern, where I added one to the actual `black_height`. To understand the patterns `PosLeft` and `PosRight` one has to remind the definition of `black_height`. There I took the maximum of the `black_heights` of the subtree. In this cases one subtree has smaller `black_height` and therefore `max` selects the other unmarked subtree. So `black_height` returns the right value.

In `delete1_t` the argument `t` is the tree before the deletion and `t'` the resulting tree. `p` is the position of the mark of `t'`. Since we want to delete the root of `t`, we require, that it is a node. `post_fixup` is a predicate, that describes the result of `fixup_delete` much like `post_balance` describe the result of `insert_fixup`. I will come back to it later. Then we state that the resulting tree has the same `black_height`, than the original tree, if we take our additional black into account. Like in the `insert` algorithm, we need this during node reconstruction, to proof, that the resulting tree is again balanced or at least can be balanced by `fixup_delete`. The next two lines are the usual membership assertions that we already saw in the specification of `delete`. The last line finally states, that if `t` has a black root, then the resulting tree will have a black root too. This is necessary to keep the `red_black_childs` invariant valid. The parent of a black node may be red and if the resulting tree would have a red root, the `red_black_childs` invariant would be broken during node reconstruction.

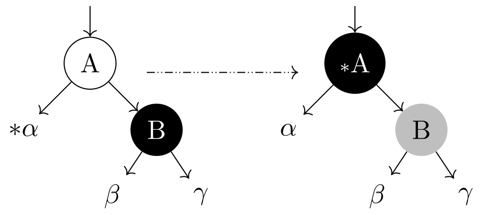
Like in the verification of insertion I defined a pair of predicates for `fixup_delete` too. `pre_fixup` describes a tree, that can be or is already fixed and `post_fixup` specifies the result of the fixup.

```
val post_fixup : sp:(tree * add_black_pos)  -> GTot bool
let post_fixup (t,p) = match p with
    | PosNone -> is_srbtree t
    | PosSelf -> is_rbtree t
    | _ -> false

val pre_fixup : (tree * add_black_pos) -> GTot bool
let pre_fixup (t,p) = match p with
  | PosNone -> is_srbtree t
  | PosSelf -> is_srbtree t
  | PosLeft -> dcase1l t || dcase2l t
      || dcase3l t || dcase4l t
  | PosRight -> dcase1r t || dcase2r t
      || dcase3r t || dcase4r t
```

As already mentioned only `PosNone` and `PosSelf` are marked positions for a fixed tree. Therefore the match only lists requirements for these to cases. `PosNone` indicates, that all defects are resolved and no additional black is required and as such the tree should satisfy `is_srbtree`. In the `PosSelf` case, the tree has to be a valid red-black tree too, but an additional black sits in the root. Now if the root is red, then we can simply repair the `black_height` by repainting it black. As I said in the discussion of `insert`, repainting the root black does not directly break an invariant, but increases the `black_height` by one, if the root is red. Since we can fix this so easily, we want `fixup_delete` to do so and disallow it by requiring `is_rbtree` instead of `is_srbtree`. Recall that `is_rbtree` implies `is_srbtree`, but additionally requires, that the root is black.

A tree with a black root and the `PosSelf` mark cannot be fixed directly, but for trees with a `PosLeft` or `PosRight` mark Cormen et al. [2013] lists four fixup cases. In the case of insertion I handled symmetric cases together, but since deletion is more complicated, I split them up into a left and a right version. As this thesis is not about red-black trees, I will limit the discussion to one illustrative case namely `dcase2l`.



The tree on the left side shows `dcase2l` and the tree on the right side the corresponding fixed version. The $*$ indicates the position of the mark. Like before black stands for black and grey for red. The white node A can be black or red. The node A in the fixed tree carries the mark, if A was black before the fixup. The subtrees α, β and γ must all be leaves or have a black root. All three must have the same `black_height` and must satisfy `is_srbtree`. I choose this case, because it illustrates the propagation of the mark. After the fixup α has the same `black_height` than its sibling, but a new additional black now may sit in the root of our tree.

```
val dcase2l : tree -> GTot bool
let dcase2l t = match t with
  | Node co ta _ (Node Black tb _ tc) ->
      let l = black_height ta in
       is_rbtree ta && is_rbtree tb
       && is_rbtree tc && sorted t
       && l = black_height tb && l = black_height tc
  | _ -> false
```

The code summarises these requirements. α, β and γ are named `ta`, `tb` or `tc` respectively. `is_rbtree` is used again for a red-black subtree with black root. The fixup code also closely relates to the picture.

```
val fixup_dcase2l : t:tree{dcase2l t}
    -> Tot (tp':(tree * add_black_pos){
       red_black_childs (fst tp')
    /\ (forall x. gmember t x <==> gmember (fst tp') x)
    /\ get_rcol (fst tp') == Black
    /\ (get_rcol t == Black ==> snd tp' == PosSelf)
    /\ (get_rcol t == Red ==> snd tp' == PosNone)})
let fixup_dcase2l = function
  | Node co ta v1 (Node Black tc v2 te) ->
    let p = if co = Red then PosNone else PosSelf in
    (Node Black ta v1 (Node Red tc v2 te), p)

val fixup_dcase2l_rbtree: t:tree{dcase2l t}
  -> Lemma (post_fixup (fixup_dcase2l t)
  /\ black_height t == black_tree_pos_height (fixup_dcase2l t))
let fixup_dcase2l_rbtree t = match t with
```

```
  | Node Red ta v1 (Node Black tc v2 te) -> ()
  | Node Black ta v1 (Node Black tc v2 te) -> ()
```

The resulting code returns always a tree with a black node and sets the mark dependently of the colour of A. The code presented here is directly taken from the verification. It shows the power of the SMT solver. All assertions are easy to see, but not all are altogether trivial. As the only red node we know of is the child of a black node and has children with black roots, the `red_black_childs` invariant is obviously fulfilled. Since we only change the colour of nodes, it is trivial to see that membership does not change. The rest are useful observations from the code, that are completely trivial to prove.

In `fixup_dcase2l_rbtree` I collected assertions with a small twist. For `post_fixup` one essential has to prove, that the resulting tree satisfies `is_srbtree`. It requires sortedness and the invariants `red_black_childs` and `black_height_correct`. I accidentally asked the SMT solver to reprove `red_black_childs`. The `dcase2l` predicate already states, that the tree is sorted and we just repainted some nodes, so there is actually nothing to prove. `black_height_correct` holds, because α,β and γ are all of the same `black_height` and the red node B just binds β and γ together, but does not increase the `black_height`. So the siblings β and γ are balanced and α has the same `black_height` then the tree with root B.

For the second assertion recall, that α, β and γ are all of the same `black_height`. I will call it h for now. Since `black_height` uses `max`, the `black_height` of the tree with root B is the one that matters before the fixup. Now we have to make a case distinction. If the root of the original tree was red, then its `black_height` is $1 + h$, since B is black. The `black_height` of the fixed version is $1 + h$ too. There B does not contribute to it, but A is black and therefore it stays the same. Since `fixup_dcase2l` return `PosNone` in our case, `black_tree_pos_height` is the same as `black_height`.

If the root of the original tree was black on the other hand, then the `black_height` is $2 + h$, since both A and B are counted. The `black_height` of the fix tree is the same as in the other case, but a mark sits in its root, so `black_tree_pos_height` increments the `black_height`. So in this case the equality holds too.

All the case specific fixup functions were later combined to the `fixup_delete` function. The associated lemmata like `fixup_dcase2l_rbtree` are combined to a big `fixup_delete_rbtree` theorem. As we have seen in while the discussion of `dcase2l`, `fixup_delete` does not always resolve the defect completely and propagates the defect up. If we reach the root of the whole tree, we can simply throw away the mark, since reducing the `black_height` is no issue. On the other hand, if the marked tree has a parent and a sibling, then need to reconstruct the parent node and call `fixup_delete` again, like I did in `del_minimum`. But before the call `fixup_delete`, we have to prove, that the tree is in one of our four cases. As this is often needed and not completely trivial to see, I introduced helper predicates and lemmata. Again I splited left and right versions and since they are analogous I will only discuss the left version.

```
val casesl: t:tree -> GTot bool
let casesl t = match t with
  | Node Red l _ r -> 1+black_height l = black_height r
    && is_rbtree l && is_rbtree r && sorted t
  | Node Black l _ r -> 1+black_height l = black_height r
    && is_rbtree l && is_srbtree r && sorted t
  | _ -> false

val is_casesl: t:tree
  -> Lemma (requires casesl t)
  (ensures dcase1l t \/ dcase2l t
        \/ dcase3l t \/ dcase4l t)
let is_casesl t = match t with
  | Node Black ta _ (Node Red tc _ te) -> assert (dcase1l t); ()
  | Node co ta _ (Node Black _ _ (Node Red e _ f)) ->
        assert (dcase4l t); ()
  | Node co ta _ (Node Black (Node Red c _ d) _ te) ->
        assert (dcase3l t); ()
  | Node co ta _ (Node Black tc _ te) -> assert (dcase2l t); ()
```

`casesl` mainly states, that the `black_height` of the left subtree is by one smaller than the right subtree. As usual the whole tree must be sorted and the subtrees have to be valid red-black subtrees. Again `is_rbtree` is used to additionally require black roots, to retain the `red_black_childs` invariant. Whenever I used `casesl` the subtree `l` was the result of `fixup_delete` together with `PosSelf`. As such `(l,PosSelf)` satisfies `post_fixup` and therefore `l` has to satisfy `is_rbtree`. In addition the fixup cases described in Cormen et al. [2013] all assume that the marked nodes are black.

Proving that a reconstructed node satisfies `casesl` is easy and `is_casesl` then provides the precondition for `fixup_delete`. The proof of `is_casesl` is again mainly done by the SMT solver. I only needed to point in the right direction.

4 Case Study: Normalisation of STLC

Now we come to the main topic of this thesis, the solution for Poplmark-Reloaded theorem prover challenge (Pientka [2018]) in F*. As a challenge Poplmark-Reloaded is especially appropriate to test the power and expressiveness of F* and the solution presented here shows that F* can indeed be used as general purpose theorem prover and is not restricted to cryptographic applications. Since the goal of this thesis is to analyse F*'s proving capabilities and not to solve Poplmark-Reloaded, I will only give an overview over the challenge and its F* implementation in this section. For a full discussion of Poplmark-Reloaded I refer the reader to Pientka [2018] and the full implementation can be found at `https://github.com/sturmsebastian/Fstar-master-thesis-code` and on the enclosed CD.

As the title suggests, Poplmark-Reloaded and this section is about a normalisation proof of a STLC. Normalisation in this context means, that the evaluation of a term of our language terminates or heavily simplified the evaluation does not "loop" forever. At first this sounds trivial, since STLC has no construct for recursion or any other looping construct, but it turns out, that the proof is not altogether trivial. If one has fixed the language, one first has to define, what evaluation actually means. Poplmark-Reloaded does this via a step relation, which summarises all possible evaluation steps. The language used here is quite small and the central evaluation step is β-reduction. It can be applied, if the term in question is an application of a λ term $\lambda x : A \,.\, e_1$ on some other term e_2. The β reduction then results in the term e_1, where all occurrences of the variable x are substituted by the term e_2, provided that the application is type correct. So substitution is a essential part of the proof.

Once the step relation is in place, one has to formally define normalisation. Poplmark-Reloaded includes even two definitions of strong normalisation, one via a accessibility relation and an inductive one. Then the later definition is shown to imply the first. Finally the central logical relation can be set up and used to prove the main result, namely that all type correct terms indeed strongly normalise.

4.1 Basic Setup

Before we come to the actual challenges, lets have a look at the grammar for terms and types:

$$\begin{array}{lll}
\text{Terms} & M, N & ::= \; x \mid \lambda x : A \,.\, M \mid M\, N \\
\text{Types} & A, B & ::= \; A \Rightarrow B \mid i \\
\text{Contexts} & \Gamma & ::= \; \cdot \mid \Gamma, x : A \\
\text{Substitutions} & \sigma & ::= \; \cdot \mid \sigma, M/x
\end{array}$$

The base type i is kept abstract, so that only variables can inhabit it. The rest is standard and the bare minimum needed for STLC.

```
type ty =
  | TBase : ty
  | TArr : ty -> ty -> ty

type ctx = list ty
```

The F* type `ty` is used for types. I used an abbreviation, since `type` is a reserved keyword. For variable bindings I used pure de Brujin indexing. In de Brujin indexing variables are simply natural numbers, there the number n refers the the n-th enclosing lambda, if there are at least n enclosing lambdas. Otherwise if there are only $m < n$ enclosing lambdas, then the variable has the type as index $n - m$. So contexts are simply lists of types, since variables have no name. Please note, that contexts in the paper are in principle lists of variable type pairs, with are extended on the right. Since standard functional list grow on the left, the order of appearance is reversed. In contrast to the Coq solution in the Poplmark-Reloaded paper, I encoded variables simply as natural numbers, instead of defining a separate data type. This way I was able to use F*'s standard library functions and lemmata for list concatenation and indexing, but still needed some refinements.

```
type texp (g:ctx): ty -> Type =
  | TyVar: v:nat {v < length g} ->
           texp g (index g v)
  | TyLam: tlam:ty ->
           #t:ty ->
           e':texp (tlam::g) t ->
           texp g (TArr tlam t)
  | TyApp: #t1:ty ->
           #t2:ty ->
           e1:texp g (TArr t1 t2) ->
           e2:texp g t1 ->
           texp g t2
```

Expressions are represented by `texp` in my formalisation. The name stands for typed expression, because I inlined the typing statement into the expression. The effect of this intrinsic typing is, that all expressions already carry their type and the context under with the typing judgement holds. In the variable case, the type is fully determined by the context and the natural number or the de Brujin index itself. The refinement to `v` is necessary, since the standard library function `index` directly yields the element on the specified index and therefore is only applicable to natural numbers that are smaller than the length of the list. As we can see in the `TyLam` case, context extensions are simply encoded as list constructor applications.

The typing judgement for `TyLam` requires, that the inner term `e'` is of the result type, under the premise, that the context is extended with the abstracted type `tlam`. These are exactly the premises, that the type of `e'` expresses. Similarly the `TyApp` constructor takes two terms and the corresponding typing judgement has two premises. The first expression has to be of a function type, with the result type matching the type of the whole expression, the second expression on the other hand has to be of type `t1`, so that the function term can be applied to it. This way `TyApp` ensures, that only type correct terms can be built.

For example take the term $\lambda x.f\ x$ with the typing judgement $f : i \to i \vdash \lambda x.f\ x$. This would be encoded as `TyLam TBase (TyApp (TyVar 1) (TyVar 0))` of type `texp [TArr TBase TBase] TBase`. Why is this a correct type? The outermost constructor is `TyLam` and it requires, that `TyApp (TyVar 1) (TyVar 0)` has type `TBase` under the context `[TBase; TArr TBase TBase]`. To show this we need to consider `TyVar 1` and `TyVar 0`. The type of `TyVar 1` is listed in our context at index 1 and this is `TArr TBase TBase`. It is a function type and the result type matches the required type `TBase`. Now we need to show, that `TyVar 0` has type `TBase`, so what the function can be applied, but the type at index 0 in our context is exactly `TBase` and so our example term has indeed the asserted type.

```
type sub (g':ctx): g:ctx -> Type =
  | STNil: sub g' []
  | STCons: #t:ty ->
            #g:ctx ->
            e:texp g' t ->
            hst:sub g' g ->
            sub g' (t::g)
```

Substitutions map variables from one context to terms in another. In the definition the variables of context `g` are mapped to terms in context `g'`. In the `STNil` case `g` is empty and therefore there are no variables to map. `STCons` simultaneously extends a substitution by a expression and the context `g` by its corresponding type. In the paper proofs `s:sub g' g` is written as $\Gamma' \vdash \sigma : \Gamma$, where Γ and Γ' are replaced by `g` and `g'` and σ by `s`. If the substitution is a renaming ρ, then the notion $\Gamma' \leq_\rho \Gamma$ is used instead. I will come to the definition of renamings, when I discuss the `subst` function.

The usage of intrinsic typings was motivated by Benton et al. [2012] and the Coq solution in Pientka [2018]. It is beneficial for various reasons, first as Benton et al. [2012] points out, one needs the typing for terms and substitution anyway in nearly any proof and inlining the typing saves a lot of work. This is especially an issue for F*, where simultaneous pattern matching on values, that depend on each other, is not yet fully elaborated. When one tries to do so, F* often fails with “Incompatible version for unification variable”. Additionally with extrinsic typing, one needs extra logic for applying substitutions to terms or other substitutions, since the subsection must contain a term for every variable, that arises. So one has to track the maximal variable that arises in terms. With intrinsic typing on the other hand, a substitution is only applicable, if the mentioned contexts match, but this already ensures, that applying the substitution is valid.

The usual downside of this approach is, that pattern matching values with intrinsic typing is significantly more complex. Benton et al. [2012] uses special Coq tactics to work with non-trivial dependencies. Fortunately this is not an issue in F*, as it generates equality constraints automatically, without citing a special tactic. E.g. consider a term `e` of type `texp g t` for some `t:ty` and `g:ctx`. Then if one matches `e` with the pattern `TyLam tlam e'`, then F* derives that `t` has the form `TArr tlam t'` for some `t':ty`.

4.2 Substitutions

Another important question was, how to implement the application of a substitution to a term. I desired to first implement the weakening of term directly.

```
val insertAt: l:list 'a -> e:'a -> n:nat {n <= length l} -> Tot (list 'a)
let rec insertAt l e n = if n = 0 then e::l
  else match l with
    | hd::tl -> hd::(insertAt tl e (n-1))

val insert_index: l:list 'a -> e:'a -> n:nat {n <= length l} ->
  i:nat {i < length (insertAt l e n)} ->
  Lemma ((i < n ==> index (insertAt l e n) i == index l i)
    /\ (i = n ==> index (insertAt l e n) i == e)
    /\ (i > n ==> index (insertAt l e n) i == index l (i-1)))

val texp_weaken_at: #g:ctx -> #t:ty -> n:nat {n <= length g}
  -> t':ty -> e:texp g t -> Tot (texp (insertAt g t' n) t)
  (decreases e)
let rec texp_weaken_at #g #t n t' e  = match e with
  | TyVar x -> if x >= n then (
        insert_index g t' n (x+1); TyVar (x+1)
      ) else ( insert_index g t' n x; TyVar x )
  | TyApp e1 e2 -> TyApp (texp_weaken_at n t' e1)
                   (texp_weaken_at n t' e2)
  | TyLam tlam e' -> TyLam tlam (texp_weaken_at (n+1) t' e')

val texp_weaken: #g:ctx -> #t:ty -> t':ty -> e:texp g t ->
  Tot (texp (t'::g) t)
let texp_weaken #g #t t' e = texp_weaken_at 0 t' e
```

Usually in papers weakening only extends the context on the end, but for the proofs, one needs a more general version. `insertAt g t' n` inserts `t'` before the position `n` in the list `g`. `texp_weaken_at` can therefore weaken the context of a term at any position. The most interesting case in the implementation is the `TyVar` pattern. It contains a natural number, that i.a. is an index into the context. To preserve the type, all numbers, that are greater then the weakening position must be incremented, since by inserting a type before that position, the indicies of all types at and after that position increment.

The lemma `insert_index` describes exactly this correlation between indicies, before and after a call to `insertAt`. One has to distinguish three cases. The indicies of elements before the inserted element do not change. The second line of the assertions says, that `insertAt` inserts indeed at the right position and the indicies of elements after the inserted one increases by one. In the `TyApp` case the function is simply called recursively on the two subterms. The context of the subterm contained in the `TyLam` constructor is already extended by the type `tlam`. Therefore to insert the new type at the same position, we have to increment the argument `n` by one. In F* terms, since the result has to have type `texp (insertAt g t' n) t`, where `t` is equal to `TArr tlam t'` for some `t'`, we have to prove, that the recursive call yields something of type `texp (tlam::(insertAt g t' n)) t'`. But, since `insertAt` is defined by simple recursion, the SMT solver can derive, that `insertAt (tlam::g) t' (n+1) == tlam::(insertAt g t' n)` and the recursive call yields a result of type `texp (insertAt (tlam::g) t' (n+1)) t'`. The version of weakening `texp_weaken`, that corresponds to the lemma in the paper, is then simply a specialisation of `texp_weaken_at`, with a fixed position `0`.

Weakening of a substitution, implemented in the function `sub_weaken`, is then simply defined through weakening every contained term.

```
val sub_weaken: #g':ctx -> #g:ctx -> t:ty -> s:sub g' g ->
        Tot (sub (t::g') g)
let rec sub_weaken #g' #g t s = match s with
  | STNil -> STNil
  | STCons e s' -> STCons (texp_weaken t e) (sub_weaken t s')

val sub_elam: #g':ctx -> #g:ctx -> t:ty -> s:sub g' g ->
  Tot (sub (t::g') (t::g))
let rec sub_elam #g' #g t s = STCons (TyVar 0) (sub_weaken t s)
```

The function `sub_weaken` only weakens the target context, but for the application of substitutions to terms, I need a function, that extends both contexts with the same type. Since it was initially intended for use in the lambda case of `subst` I called this function `sub_elam`. It weakens the target context with `sub_weaken` and then extends the substitution by the identity for variable `0`.

```
val sindex: #g':ctx -> #g:ctx -> s:sub g' g -> n:nat {n < length g} ->
    Tot (texp g' (index g n))
let rec sindex #g' #g s n = match s with
  | STCons e s' -> if n = 0 then e else sindex s' (n-1)

val subst: #g':ctx -> #g:ctx -> #t:ty -> s:sub g' g -> e:texp g t ->
  Tot (texp g' t) (decreases e)
let rec subst #g' #g #t s e = match e with
  | TyApp e1 e2 -> TyApp (subst s e1) (subst s e2)
  | TyLam tlam e' -> TyLam tlam (subst (sub_elam tlam s) e')
  | TyVar x -> sindex s x
```

Application of a substitution to a term is written $[\sigma]N$ in the paper, where σ is the substitution and N is the term. Again in the `TyApp` case we simply recurse on the two subterms. In the `TyLam` case the subterm `e'` has type `texp (tlam::g) t'` for some `t'` and we need a new term of type `texp (tlam::g') t'`. There `sub_elam` enters the stage. It produces a substitution from `s`, by extending all term with `tlam` and extending the result itself, so that the new variable `0` of type `tlam` is not changed. The actual work happens in the `TyVar` case. There `subst` extracts the term at the right index form `s`. Since `s` is not an ordinary list, I need a distinct implementation for indexing. Despite the dependencies on the contexts, the implementation of `sindex` is quite standard.

One special type of substitutions is particularly interesting for normalisation proofs of STLC, namely substitution, that arises from the β rule. In the paper proofs $[N/x]$ denotes a substitution, that replaces the variable x by the term N and is the identity substitution on all other variables.

```
val sub_beta: #g:ctx -> #t:ty -> e:texp g t -> Tot (sub g (t::g))
let sub_beta #g #t e = STCons e (id_sub g)
```

When applying the β rule, one substitutes always the variable bound by the outermost lambda of a term. In case of de Brujin this is variable 0. At the same time one eliminates the outermost lambda. By doing so the binding level decreases in the whole term and therefore one has to decrease all variables in the term. `id_sub g` is the identity substitution for the context `g`, i.e. the term at position `n` is the term `TyVar n`. If one extends this identity substitution, in our case by the term `e`, then the term `TyVar n` is at index `n+1` and therefore `sub_beta` decreases all variables.

One advantage of this definition of `subst` is that renamings can be defined as a special case of substitutions.

```
val renaming: #g':ctx -> #g:ctx -> s:sub g' g -> Tot Type0
let rec renaming #g' #g s = match s with
  | STNil -> True
  | STCons e s' -> TyVar? e /\ renaming s'
```

Renamings are substitutions that map variables to variables, not to general terms. The predicate `renaming` expresses exactly this, by requiring every contained term to be a variable. By using the predicate as a refinement, all lemmata for substitutions are equally applicable to renamings.

The implementation I used for `subst` is not the only one one can think of. One would like to implement `texp_weaken` as a substitution too, but then `sub_weaken` and `subst` would have to be defined as mutual recursive, where `subst` is called recursively very often and this would be very difficult to prove terminating. Benton et al. [2012] then uses the fact, that weakening is a renaming and defines renamings separately from substitutions. Applying a renaming to a term is a lot easier than applying a general substitution and sub_weaken can then be defined with renamings. The advantage of the definition in Benton et al. [2012] is, that one needs only a few helper lemmata, that describe the relationship between substitutions and renamings. The downside is, that all subsequent lemmata must be proven twice, for renamings and for substitutions.

My definition in contrast defines texp_weaken and sub_weaken separately, so that renamings can be defined as a special case of substitutions and therefore all properties of substitutions automatically apply to renamings too. But this comes at a cost too. To show, that the separately defined texp_weaken can be written in terms of substitutions, one needs quite a few technical helper lemmata. Most of them are equalities and in F* equalities are solved by the SMT solver, but with SMT one has less control over the proof. So defining weakening as a separate function was also a way to work around the limitations of the SMT supported proving approach used in F*. I will discuss the up and downsides of F* in more detail in Section 5.2, but since it fits very well, let us examine one of the helper lemmata used in my verification.

```
val insert_exchange: l:list 'a -> e1:'a -> n1:nat {n1 <= length l}
  -> e2:'a -> n2:nat {n2 <= n1} ->
  Lemma (insertAt (insertAt l e1 n1) e2 n2
    == insertAt (insertAt l e2 n2) e1 (n1+1))
  [SMTPat (insertAt (insertAt l e1 n1) e2 n2)]
let rec insert_exchange l e1 n1 e2 n2 = if n2 = 0 then ()
  else match l with hd::tl -> insert_exchange tl e1 (n1-1) e2 (n2-1)

val texp_weaken_exchange: #g:ctx -> #t:ty -> t1:ty -> t2:ty
  -> e:texp g t -> n:nat {n <= length g} -> m:nat {m <= n} ->
  Lemma (ensures texp_weaken_at (n+1) t1 (texp_weaken_at m t2 e) ==
    texp_weaken_at m t2 (texp_weaken_at n t1 e))
  (decreases e)
let rec texp_weaken_exchange #g #t t1 t2 e n m = match e with
  | TyApp e1 e2 -> texp_weaken_exchange t1 t2 e1 n m;
    texp_weaken_exchange t1 t2 e2 n m
  | TyLam tlam e' ->
    assert (texp_weaken_at (n+1) t1 (texp_weaken_at m t2 e) ==
      TyLam tlam (texp_weaken_at (n+2) t1 (texp_weaken_at (m+1) t2 e')));
    assert (texp_weaken_at m t2 (texp_weaken_at n t1 e) == TyLam tlam
      (texp_weaken_at (m+1) t2 (texp_weaken_at (n+1) t1 e')));
    texp_weaken_exchange t1 t2 e' (n+1) (m+1)
  | TyVar v -> texp_weaken_exchange_var g t1 t2 v n m
```

The lemma texp_weaken_exchange describes, what happens, when one exchanges two weakenings. To ensure, that both sides of the stated equation are terms over the same context and therefore have the same type, the helper lemma insert_exchange is needed. But before I go into detail, I would like to give a high level overview. As weakening mostly affects the context, let us ignore the actual term and its type for a while and focus on the context. The following diagram illustrates its modifications.

$$\begin{tikzcd}
\Gamma, \Delta, E \arrow[r, "\downarrow y:T_2"] \arrow[d, "\downarrow x:T_1"'] & \Gamma, \Delta, y : T_2, E \arrow[d, "\downarrow x:T_1"] \\
\Gamma, x : T_1, \Delta, E \arrow[r, "\downarrow y:T_2"] & \Gamma, x : T_1, \Delta, y : T_2, E
\end{tikzcd}$$

We start with a context represented by the variable `g` on the top left corner. For convenience I already split it up in three parts Γ, Δ and E, each of which may be empty. The left side of the equation first weakens the context with T_2, so we follow the arrow to the right and get a new context with T_2 inserted between Δ and E. Then we weaken with T_1 and therefore have to follow the downside arrow. We reach the double weakened context in the bottom right corner. But we can reach the very same context, by first following the arrow down and then the arrow to the right. This path is the representation of the right hand side of the equation.

The diagram shows three concatenated contexts, while the F* lemma takes about inserting into one context. In essence both versions are equivalent, but the version with the concatenated contexts hide one important fact. If I insert something into a list the indicies of all elements after the inserted element increment by one. So if I first insert T_2 between Δ and E, then the index of the gap between Γ and Δ increments by one. Recall that contexts use the reverse list representation. In contrast if I first insert T_1, then the index of the gap between Δ and E stays the same. This is why I had to insert `t1` at index `n+1` on the left hand side of the equation.

This index shift is not special to context, but property of lists. The lemma `insert_exchange` shows this by a simple recursion. As already mentioned, this lemma is vital already for the type correctness of the assertion of `texp_weaken_exchange`. Its proof then describes the effect of the weakenings on the term itself.

If one uses a named variable representation, like most paper proofs do, a weakening only affects the context and does not change the term itself, but in this formalisation de Brujin indexing is used. Since variables are indexes into their context and insertion into a context changes the index of some of its types, the weakening function has to adjust the variables too.

The `TyApp` and `TyLam` cases can be proven by simply recursively calling `texp_weaken_exchange` on their contained subterms. In the `TyLam` case, one has to guide the SMT solver with some extra assertions. Both simply unfold the definition of `texp_weaken_at` twice and thereby force a specific representation of the subterm contained in the `TyLam`. The SMT solver can then match these subterms with both sides of the equality resulting from the recursive call and thus complete the proof for this case.

The `TyVar` case in turn is so challenging for the SMT solver, that I needed a separate helper lemma `texp_weaken_exchange_var`.

```
val texp_weaken_exchange_var: g:ctx -> t1:ty -> t2:ty
  -> v:var {v < length g} -> n:nat {n <= length g} -> m:nat {m <= n} ->
  Lemma (ensures texp_weaken_at (n+1) t1 (texp_weaken_at m t2 (TyVar #g v))
    == texp_weaken_at m t2 (texp_weaken_at n t1 (TyVar #g v)))
let texp_weaken_exchange_var g t1 t2 v n m =
  if v >= n then (
    insert_index g t1 n (v+1);
    assert (texp_weaken_at n t1 (TyVar #g v)
      == TyVar #(insertAt g t1 n) (v+1));
    assert (v+1+1 == v+2);
    insert_index (insertAt g t1 n) t2 m (v+2);
    assert (texp_weaken_at m t2 (TyVar #(insertAt g t1 n) (v+1))
      == TyVar (v+2));

    insert_index g t2 m (v+1);
    assert (texp_weaken_at m t2 (TyVar #g v)
      == TyVar #(insertAt g t2 m) (v+1));
    insert_index (insertAt g t2 m) t1 (n+1) (v+2);
    assert (texp_weaken_at (n+1) t1 (TyVar #(insertAt g t2 m) (v+1))
      == TyVar (v+2))
  ) else if v >= m then (
    assert (v < n);
    insert_index g t1 n v;
    assert (texp_weaken_at n t1 (TyVar #g v) == TyVar #(insertAt g t1 n) v);
    insert_index (insertAt g t1 n) t2 m (v+1);
    assert (texp_weaken_at m t2 (TyVar #(insertAt g t1 n) v) ==
      TyVar (v+1));

    insert_index g t2 m v;
    assert (texp_weaken_at m t2 (TyVar #g v)
      == TyVar #(insertAt g t2 m) (v+1));
    insert_index (insertAt g t2 m) t1 (n+1) (v+1);
    assert (texp_weaken_at (n+1) t1 (TyVar #(insertAt g t2 m) (v+1))
      == TyVar #(insertAt (insertAt g t2 m) t1 (n+1)) (v+1))
  ) else (
    assert (v < n); assert (v < m);
    insert_index g t1 n v;
    assert (texp_weaken_at n t1 (TyVar #g v) == TyVar #(insertAt g t1 n) v);
    insert_index (insertAt g t1 n) t2 m v;
    assert (texp_weaken_at m t2 (TyVar #(insertAt g t1 n) v) ==
      TyVar #(insertAt (insertAt g t1 n) t2 m) v);

    insert_index g t2 m v;
    assert (texp_weaken_at m t2 (TyVar #g v) == TyVar #(insertAt g t2 m) v);
    insert_index (insertAt g t2 m) t1 (n+1) v;
    assert (texp_weaken_at (n+1) t1 (TyVar #(insertAt g t2 m) v)
      == TyVar #(insertAt (insertAt g t2 m) t1 (n+1)) v)
  )
```

The assertion of the helper lemma `texp_weaken_exchange_var` is essentially the same as that of `texp_weaken_exchange`. The only difference is that the first specifically require a variable, there the later works for general terms, with is no surprise, since `texp_weaken_exchange_var` is the helper lemma introduced exactly for variables.

The proof of the lemma distinguishes three cases. In the first case the variable is greater or equal then both insert positions. The second case handles variables smaller then the high insert position `n` and greater or equal to the lower insert position `m` and in the final case the remaining variables that are smaller than both insert positions. I will discuss only the proof for the first case, because the other two are mostly analogue.

The only helper lemma used here is `insert_index`. It describes correlates the indicies of the elements of a list with the indicies after an insertion. Notably it says that the index of all element after the insert position increments by one.

The proof of each case again consists of two parts, separated by an empty line. The first part unfolds the definition of `texp_weaken_at` in the right side of the asserted equation and the second in the left part, so that both sides meet in a common term. The function `texp_weaken_at` applied to a `TyVar` again results in a `TyVar`, but the contained natural number may change depending on its value. In the first case the variable `v` is greater or equal then both insert positions, so each unfolding increments the variable `v`. The context used for the assertions has to be explicitly given, since it can not inferred in this situation. I needed to assert `v+1+1 == v+2`, because otherwise the SMT solver would not necessarily exploit this equation and then would fail to apply the assertion of the first `insert_index` invocation. After the both sides of the equation are fully evaluated, one reaches the desired common term.

4.3 Proving Strong Normalisation

The definition of our language is still not complete yet. Some sort of operational semantics is missing.

```
type step (g:ctx): t:ty -> e:texp g t -> e':texp g t -> Type =
  | SAppL: #t1:ty -> #t2:ty -> #e1:texp g (TArr t1 t2) ->
          #e1':texp g (TArr t1 t2) -> e2:texp g t1 ->
          hst:step g (TArr t1 t2) e1 e1' ->
           step g t2 (TyApp e1 e2) (TyApp e1' e2)
  | SAppR: #t1:ty -> #t2:ty -> e1:texp g (TArr t1 t2) ->
          #e2:texp g t1 -> #e2':texp g t1 ->
          hst:step g t1 e2 e2' ->
            step g t2 (TyApp e1 e2) (TyApp e1 e2')
  | SLam: tlam:ty -> #t:ty -> #e:texp (tlam::g) t ->
         #e':texp (tlam::g) t -> hst:step (tlam::g) t e e' ->
           step g (TArr tlam t) (TyLam tlam e) (TyLam tlam e')
  | SBeta: tlam:ty -> #t:ty -> e1:texp (tlam::g) t ->
          e2:texp g tlam ->
            step g t (TyApp (TyLam tlam e1) e2)
              (subst (sub_beta e2) e1)
```

`step` defines a non-deterministic small step reduction relation. In the paper `step g t e1 e2` is noted as $\Gamma \vdash M \longrightarrow N : A$, where Γ corresponds to `g`, A to the type `t` and M and N are represented by the terms `e1` and `e2`. Due to the limited amount of language constructs, the number of possible reductions is also quite small.

The definition of `step` is quite cluttered with implicit arguments. In the following descriptions I will completely ignore them, since they are completely determined by the explicit arguments.

First of all, one can reduce a term by making a step in the left or right side of an application. The corresponding constructors are `SAppL` and `SAppR` respectively. `SAppL` takes a term `e2` of type `t1` and a reduction `hst` of function type `TArr t1 t2` and builds a step of type `t2`. The corresponding paper version is

$$\frac{\Gamma \vdash M \longrightarrow M' : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N \longrightarrow M'\,N : B}$$

I replaced M and N with `e1` and `e2` and A and B with `t1` and `t2`, but apart from that the constructor is a one-to-one translation. `SAppR` is the analogue constructor for a step on the right side of a `TyApp`. Note that already at this point we experience non-determinism. If both sides of a `TyApp` can be further evaluated, both steps are equally applicable. As usual in λ-calculi and unlike in most practical programming languages one can evaluate terms under a lambda here. The `SLam` constructor models this by wrapping the reduction of the inner expression. The `tlam` argument is not marked as implicit, so that its correspondence to the `TyLam` constructor is more obvious. The most difficult part is β reduction, since substitution is needed here. The `SBeta` constructor takes the parts of a `TyLam` term, `tlam` and `e1` as arguments, rather than the `TyLam` term itself, because the definition was easier this way, since one needs `e1` directly. The term `e2` is the argument in the application.

On top of the single step relation Poplmark-Reloaded defines a multi-step relation too.

```
type multi_step (g:ctx) (t:ty) (e1:texp g t): e2:texp g t -> Type =
  | MRefl: multi_step g t e1 e1
  | MSingle: #e2:texp g t -> #e3:texp g t ->
             hst:step g t e1 e2 -> mst:multi_step g t e2 e3 ->
             multi_step g t e1 e3
```

The notation $\Gamma \vdash M \longrightarrow^* N : A$ is used for `multi_step g t e1 e2` with the same translation then for `step` ($\Gamma \to$ `g`, $A \to$ `t`, $N \to$ `e1`, $M \to$ `e2`). The arguments of the `multi_step` type are in principle the same as those of `step`, but since no specialisation is required for most of them, I could fix mark them as fixed by the type. At a high level one can think of `multi_step` as a chain of steps. The `MRefl` constructor is some sort of empty chain, because no reduction happens at all and the output term is the same as the input one. The `MSingle` constructor plays the role of a dependently typed cons. It takes a step `hst` from the input term `e1` to some intermediate term `e2` and a `multi_step` `mst` from `e2` to the output term `e3` and constructs a `multi_step` from `e1` to `e3`.

Some properties that are easily to prove, but required for later use, are included for `multi_step`. Since none of them give deeper insights and since the proofs are not especially interesting, I refer the reader to Pientka [2018] and the source code of the verification.

Poplmark-Reloaded features two different definitions of strong normalisation. The first one is defined via an accessibility relation.

$$\frac{\Gamma \vdash M : A \qquad \forall N. \Gamma \vdash M \longrightarrow N : A \Rightarrow \Gamma \vdash N : A \in \mathsf{sn}}{\Gamma \vdash M : A \in \mathsf{sn}}$$

The definition says, that a expression M strongly normalises, if all expressions N, to which M can step, strongly normalise. In this version no extra typing statement is required, since I use intrinsic typing, so lets have a look at the second precondition. The interesting point there is the quantification over the term N. Since at any time only finitely many reduction steps are possible, one can also think of this precondition as collection of precondition, one for each possible reduction step. But then modelling this definition is F*, we have to cope with the $\forall$ anyway. Under the Curry-Howard correspondence a $\forall$ becomes a function and I modelled the precondition exactly this way.

```
noeq type ssn: nat -> g:ctx -> t:ty -> texp g t -> Type =
  | Ssn: #g:ctx -> #t:ty ->
         #e:texp g t ->
         n:nat ->
         f:(#e':texp g t ->
           hst:step g t e e' ->
           Tot (ssn n g t e')
           ) ->
          ssn (n+1) g t e
```

Again `g` stands for Γ, `t` for A and `e` and `e'` for M and N. The function `f` mainly is a direct translation of the precondition. It takes the term `e'` and a step from `e` to `e'` as arguments and returns a proof, that `e'` strongly normalises. When we have such a function, we know, that `e` itself also strongly normalises.

But does this definition really reflect strong normalisation? To answer that question first consider a situation, where we have no value of a `ssn` type, nor a function that yields one. Under that circumstances we can only construct values of `ssn` type for term, that can not be further evaluated. For such terms no step exists and therefore a function `f` can be provided through a pattern match on the step. To construct a `ssn` value for a term, that has reduction steps, we need `ssn` values for all term our term can step to. This way we somehow build up a finite tree, with fully evaluated term in its leafs. A more abstract version of this discussion also shows, that a function, that is called recursively on a result of some inner function `f` terminates. Simply stated: The recursion works on a finite tree and therefore it terminates. For exactly that reason the Coq prover generates a special induction principle for this case. F* uses metrics and a build in partial order, instead of generated induction principles. Swamy et al. [2016] lists the rule

For any function $f:\ (x:t \to \mathsf{Tot}\ t')$ and $v:t,\ f(v) \prec f$

for the partial order $\prec$, but this rule is apparently not yet implemented. Therefore I left it out during the discussion of the F* features. This rule would be necessary to prove assertions by induction of `sn`, with are translated into recursive functions on some value of type `ssn`. Say for example, we want to prove some property `prop` by induction on `ssn`, without the natural number. It would look like this

```
val prop: #g:ctx -> #t:ty -> #e:texp g t -> hssn:ssn g t e -> Tot (...)
    (decreses hssn)
let rec prop #g #t #e hssn = match hssn with
    | Ssn f -> let hssn' = f (...) in
        prop hssn'
```

First we would unpack the inner function `f`, then we would call it with some argument and recursively call `prop` on the result. To accept this definition, F* has to show, that the recursion terminates, i.e. has to show that `hssn'` is less than `hssn` according to our relation $\prec$. But the third rule for $\prec$ in Section 2.1 says, that `f` $\prec$ `hssn`, because it is contained in `hssn`. Then our extra rule states, that `hssn`$'$ $\prec$ `f`, because `hssn'` is the result of `f` applied to some arguments, and then by transitivity of $\prec$ is follows that `hssn`$'$ $\prec$ `hssn` and therefore our recursive dummy function `prop` indeed terminates. To work around the issue, that this rule is not implemented, I introduced the additional natural number argument in `ssn`. It represents an upper bound on the high of the derivation tree. As such the natural number in the result type of `f` is smaller by one than the result type of the `Ssn` constructor. This number is then used as decreasing metric in inductive proofs over a `ssn` typed value.

To give you a taste of what proofs in F* without refinement types look like I include the proof of a simple property of strong normalisation here.

LEMMA 2.10 (2): *If* $\Gamma, x:A \vdash M : B \in \mathsf{sn}$ *then* $\Gamma \vdash \lambda x:A.M : A \Rightarrow B \in \mathsf{sn}$

The lemma asserts that adding a lambda abstraction does not destroy strong normalisation. The proof from the appendix of the paper is quite formal.

Induction on $\Gamma, x:A \vdash M : B \in \mathsf{sn}$

$$\text{Assume } \Gamma \vdash \lambda x:A.M \longrightarrow Q : A \Rightarrow B \tag{4.1}$$
$$\Gamma, x:A \vdash M \longrightarrow M' : B \text{ and } Q = \lambda x:A.M' \qquad \text{by reduction rule for } \lambda \tag{4.2}$$

$$\Gamma, x:A \vdash M' : B \in \mathsf{sn} \qquad \text{by assumption } \Gamma, x:A \vdash M : B \in \mathsf{sn} \tag{4.3}$$
$$\Gamma \vdash \lambda x:A.M' : A \Rightarrow B \in \mathsf{sn} \qquad \text{by IH} \tag{4.4}$$
$$\Gamma \vdash Q : A \Rightarrow B \in \mathsf{sn} \qquad \text{since } Q = \lambda x:A.M' \tag{4.5}$$
$$\Gamma \vdash \lambda x:A.M : A \Rightarrow B \in \mathsf{sn} \qquad \text{since } \Gamma \vdash \lambda x:A.M \longrightarrow Q : A \Rightarrow B \text{ was arbitrary} \tag{4.6}$$

The corresponding F* proof looks like this

```
val sn_lam: #n:nat -> #g:ctx -> tlam:ty -> #t:ty -> #e:texp (tlam::g) t
  -> hssn:ssn n (tlam::g) t e ->
  Tot (ssn n g (TArr tlam t) (TyLam tlam e))
  (decreases n)
let rec sn_lam #n #g tlam #t #e hssn = match hssn with
  | Ssn m f -> let f' (#q:texp g (TArr tlam t))
    (st:step g (TArr tlam t) (TyLam tlam e) q)
    : Tot (ssn m g (TArr tlam t) q) =
      match st with
      | SLam tlam hst ->
         sn_lam tlam (f hst)
      in Ssn m f'
```

To my surprise, when writing directly the proof lambda terms, one can closely follow the paper proof, since the paper proofs in the Poplmark-Reloaded appendix are quite elaborated. Unfortunately the F* proof is near to unreadable, but to support my point I will illustrate the connections of `sn_lam` to its paper version. Apart from `tlam` the explicit arguments of `sn_lam` are a quite exact translation of the statement in the paper. The variable `tlam` is not marked as implicit, since F* sometimes fails to derive it. As explained above the `n` argument is an upper bound on the normalising steps and serve more as a workaround for the limitations of F*'s termination checker. `g` corresponds to Γ and `tlam::g` to $\Gamma, x : A$. Since de Brujin encoding is used here, no name is assigned to the variable. In paper proofs the position of a variable in a context is irrelevant, as it can be reordered freely. De Brujin encoding is more restrictive in that point, but for most proofs one does not need to care about it. The name `tlam` was used here, to make clear, which type we abstract over. A interesting difference is the structure induction is held on. Whereas the paper proof uses sn for induction, this is not possible in F*, as explained earlier. Instead the natural number `n`, introduced for that purpose, is used.

Proving something on the structure of sn means recursively calling our function on a substructure of it. Due to the special requirement `ssn`, it contains these substructures not directly, but a function that returns a new `ssn`. So it is quite sure that we will need the inner function `f` and hence I immediately pattern matched on our `hssn` argument. I also used the pattern variable `m` to match the inner natural number. We will need it later on. Now our goal is to construct a value of type `ssn n g (TArr tlam t) (TyLam tlam e)`. The challenge here is the new inner function. So I started to define it as `f'` in a let. The type annotations are provided explicitly, to give the F* type checker names to refer to in error messages. The argument `q` is here the term we step to and named Q in the paper. The step assumed in line (4.1) of the paper proof is represent by the variable `st`. Note that I used the natural number `m` from the pattern match in the result type of `f'`. In the type definition of `ssn` I assert, that if I manage to provide a `ssn l g t q` for all `q`, one can step to from `e`, and some natural number `l`, then I get a `ssn (l+1) g t e`. In the type declaration of `sn_lam` I claim to produce a value of type `ssn n g (TArr tlam t) (TyLam tlam e)`, so `f'` has to produce a value of type `ssn (n-1) g (TArr tlam t) q`, there `(n-1)` has to be a natural number. The issue here is, that `n` can be any natural number, for which I can produce a value for the `hssn` argument. Instead of proving, that `n > 0`, I use the value `m`. One (and the SMT solver) can easily prove, that `m == n-1`, but `m` is by assumption a natural number, since the `Ssn` constructor requires it.

I constructed `sn_lam` after the paper proof, so lets have a look at it. Line (4.2) states, that we can get a step from M to M' where $Q = \lambda x : A \,.\, M'$, by the reduction rule for λ. The reduction rule for λ is in our case the `SLam step` constructor and it contains indeed a step of the correct type. This inner step is caught by the variable `hst`. In the pattern match F* automatically derives the equality $Q = \lambda x : A.M'$. Since only one step constructor can take a `TyLam` value as input term, F* knows, that the other cases are unfeasible, and therefore they need not to be considered.

Line (4.3) then uses the assumption $\Gamma, x : A.M : B \in \mathsf{sn}$. Its precondition states, that every term, to with M can reduce to, itself strongly normalises. But we have a step from M to M'. The assumption is in our proof the inner function `f`, so using the assumption means invoking `f` on our step variable `hst`. This gives us a value that corresponds to $\Gamma, x : A\,.\,M' : B \in \mathsf{sn}$.

This value is immediately used by line (4.4). While the paper proof cites the induction hypothesis, we recursively call our `sn_lam` function. At this point we have to make sure, that the arguments of the recursive call are "smaller" than the original arguments according to our decreasing metric. With `(decreases n)` I specified our upper bound on the tree high, what was introduced exactly for that purpose, as the value that decreases. The original argument `hssn` includes `n` in its type. The inner function `f` in turn yields a value of type `ssn m (tlam::g) t e'`, where `e'` is the value corresponding to M', what was left implicit here. From the previous discussion we know, that `m == n-1`, so indeed `m` $\prec$ `n` and the recursive call is valid. This already completes the definition of the new inner function `f'`.

Line (4.5) only reminds the reader, that the term, we produced a `ssn` value for, is actually our term Q. (4.6) then concludes the proof by referring back to the definition of `sn`. We have shown the precondition and therefore the paper proof is done. To complete our F* proof too, we need to wrap the function `f'`, that corresponds to the precondition, in the `Ssn` constructor, together with the natural number `m`.

In addition to `sn` Poplmark-Reloaded introduces a notion of `sn`-strong head reduction $\longrightarrow_{\mathsf{sn}}$.

$$\frac{\Gamma \vdash N : A \in \mathsf{sn} \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M)\ N \longrightarrow_{\mathsf{sn}} [N/x]M : B} \qquad \frac{\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N \longrightarrow_{\mathsf{sn}} M'\ N : B}$$

This predicate describes some sort of call-by-name reduction. The first case describes a β reduction step. The precondition $\Gamma \vdash N : A \in \mathsf{sn}$ includes a typing statement for N and the typing statements for N and M are necessary to ensure, that only well typed terms can be used in `sn` strong head reduction. I will later discuss, why N has to be in addition `sn` strongly normalising. In the second case describes a `SAppL` step, i.e. a reduction on the left side of an application. The reduction of the function part must be itself `sn` strongly normalising. Again the typing judgement for N is only included to ensure, that $M\ N$ is indeed well typed.

The most important properties in Poplmark-Reloaded regarding `sn` strong head reduction are closure and backward closure. Closure of $\longrightarrow_{\mathsf{sn}}$ means that, if $\Gamma \vdash M : A \in \mathsf{sn}$ and $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$, then $\Gamma \vdash M' : A \in \mathsf{sn}$ and backward closure states the same with the roles of M and M' flipped. To prove backward closure for the β reduction, it is necessary that N is in `sn` as previously mentioned. If the variable x does not occure in the term M, then $[N/x]M$ is the same as M. But one cannot derive `sn` strong normalisation of N from that of M. So to proof backward closure, one needs the requirement $\Gamma \vdash N : A \in \mathsf{sn}$.

Using this notion $\longrightarrow_{\mathsf{sn}}$ Poplmark-Reloaded formulates an interesting lemma.

> Lemma 2.13 (Confluence of `sn`). *If* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} N : A$ *and* $\Gamma \vdash M \longrightarrow N' : A$ *then either* $N = N'$ *of there* $\exists Q$ *s.t.* $\Gamma \vdash N' \longrightarrow_{\mathsf{sn}} Q : A$ *and* $\Gamma \vdash N \longrightarrow^{*} Q : A$.

Confluence is in most non-deterministic versions of STLC a non-trivial property, but I mostly include it here because of its assertion. Three aspects are of special interest here. First the statement is a disjuction, second the first option states an equality and third the second option is an existential assertion. My F* formalisation looks like this.

```
val sn_confulence: #g:ctx -> #t:ty -> #e1:texp g t -> #e2:texp g t
  -> #e2':texp g t -> hshr:ssn_head_red g t e1 e2
  -> hst:step g t e1 e2' -> Tot (cor (ceq e2 e2')
    (q:texp g t & ssn_head_red g t e2' q * multi_step g t e2 q))
  (decreases hshr)
```

Under the Curry Howard correspondence a disjuction is translated to a sum type. The F* module `FStar.Constructive` provides data types for most of the standard constructs of constructive logic, but since the F* standard library supports them better, tuples and functions where used so for conjunction, implication and forall quantification. For disjuctions in turn no such well-supported standard construct is provided and therefore I used the type `cor` form `FStar.Constructive`. Its constructors are `IntroL` and `IntroR` and they take a value of the left of right type respectively, much like the OCaml `option` type or Haskells `Either`.

The second point here is the equality as the first alternative. Previously equalities were stated using the `Lemma` keyword and SMT proofs. Since the equality is contained inside a constructive disjuction, the standard `==` type was no option here. To our rescue the `FStar.Constructive` module defines the type `ceq` with the single constructor `Refl`. The equality stated by the use of `Refl` is nevertheless proven by the SMT solver, but `ceq` provides a constructive value usable e.g. in `cor`.

The last new construct used here is the existential quantification. I felt free again to replace the type provided by `FStar.Constructive` with a standard F* construct, namely the dependent pair. The type constructor for it is `&` and the value constructor `Mkdtuple2`. Note, that in the assertion of `sn_confulence` the standard tuple type constructor takes precedence over the dependent pair type constructor. Therefore I could use the term `q` as output term in both the produced strong head reduction and the `multi_step`.

I already mentioned, that sn is not the only definition of strong normalisation in the Poplmark-Reloaded. It also features a inductive definition of it. Strongly normalising neutral terms: $\Gamma \vdash M : A \in \mathsf{SNe}$:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A \in \mathsf{SNe}} \qquad \frac{\Gamma \vdash R : A \Rightarrow B \in \mathsf{SNe} \qquad \Gamma \vdash M : A \in \mathsf{SN}}{\Gamma \vdash R\ M : B \in \mathsf{SNe}}$$

Strongly normalising terms: $\Gamma \vdash M : A \in \mathsf{SN}$:

$$\frac{\Gamma, x : A \vdash M : B \in \mathsf{SN}}{\Gamma \vdash \lambda x : A.M : A \Rightarrow B \in \mathsf{SN}} \qquad \frac{\Gamma \vdash R : A \in \mathsf{SNe}}{\Gamma \vdash R : A \in \mathsf{SN}} \qquad \frac{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} M' : A \qquad \Gamma \vdash M' : A \in \mathsf{SN}}{\Gamma \vdash M : A \in \mathsf{SN}}$$

Strong head reduction: $\Gamma \vdash M \longrightarrow_{\mathsf{SN}} N : A$:

$$\frac{\Gamma \vdash N : A \in \mathsf{SN} \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M)\ N \longrightarrow_{\mathsf{SN}} [N/x]M : B} \qquad \frac{\Gamma \vdash \longrightarrow_{\mathsf{SN}} M' : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N \longrightarrow_{\mathsf{SN}} M'\ N}$$

Since the F* formalisation is again a more or less direct translation of the definition I only include the shorter paper version here. It is composed of three mutual recursively defined types. First SNe the neutral terms. They are neutral in the sense, that if it is an application, the function cannot be evaluated to a lambda and therefore it is impossible to ever apply a β reduction on the whole term. Then it defines strong normalising terms SN itself. A λ term is SN strongly normalising, if the inner term is SN strongly normalising. SNe terms are in SN too and finally if one can reduce the term via SN strong head reduction to a SN strongly normalising term, then the initial term is SN strongly normalising too. The definition of SN strong head reduction is completely analogue to the definition of sn strong head reduction and therefore I refer the reader to its discussion.

When discussing the strong normalisation via a accessibility relation sn, I also gave some reasons for the soundness of this definition. For the inductive definition on the other hand it is not quite clear why term in SN should normalise. Therefore Poplmark-Reloaded proves, that all terms in SN are in sn too.

THEOREM 2.16 (SOUNDNESS OF SN).

(1) *If* $\Gamma \vdash M : A \in \mathsf{SN}$ *then* $\Gamma \vdash M : A \in \mathsf{sn}$

(2) *If* $\Gamma \vdash M : A \in \mathsf{SNe}$ *then* $\Gamma \vdash M : A \in \mathsf{sn}$

(3) *If* $\Gamma \vdash M \longrightarrow_{\mathsf{SN}} M' : A$ *then* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$

Since the definition of SN is mutual recursive, the proof of this theorem is done by mutual structural induction on the given derivations, using some additional lemmata. Again I skip the F* proof, because it is both simple and lengthy.

The last challenge in Poplmark-Reloaded is the definition of the following predicate.

- $\Gamma \vdash M \in \mathcal{R}_i$ iff $\Gamma \vdash M : i \in \mathsf{SN}$
- $\Gamma \vdash M \in \mathcal{R}_{A \Rightarrow B}$ iff for all $\Gamma' \leq_\rho \Gamma$ such that $\Gamma' \vdash N : A$, if $\Gamma' \vdash N \in \mathcal{R}_A$ then $\Gamma' \vdash [\rho]M\, N \in \mathcal{R}_B$

To show, that a term M is in $\mathcal{R}_{A \Rightarrow B}$, one has to consider all extensions of Γ in which one may use M. The challenge here is, that this definition is not strictly positive, so that one cannot simply define a data type for it, like I did for all the other relations so far.

```
val normR: g:ctx -> t:ty -> e:texp g t -> Tot Type0
  (decreases t)
let rec normR g t e = match t with
  | TBase -> sSN g TBase e
  | TArr ta tb -> (g':ctx ->
      r:sub g' g {renaming r} ->
      e0:texp g' ta ->
      norm0:normR g' ta e0 ->
      Tot (normR g' tb (TyApp (subst r e) e0))
     )
```

I solved that issue by defining `normR` as a type level function instead of a data type. If our type is the abstract base type, then `normR g t e` is the same as `sSN g t e`, but if the type of our term is a function type, then `normR` yields a function, that corresponds to the requirements. The context `g'` corresponds to the extended context Γ', which is given implicitly in the paper definition. The renaming ρ is represented by the refined substitution `r`. Note that the typing judgement $\Gamma' \leq_\rho \Gamma$ is already contained in the type of `r`. `e0` and `norm0` stands for the new term N and this requirement, that it is in $\mathcal{R}_A$. The return type also a simple translation of the conclusion $\Gamma' \vdash [\rho]M\ N \in \mathcal{R}_B$.

The logical predicate $\mathcal{R}_A$ has some interesting properties, but I will discuss only the most important one.

THEOREM 2.23

(1) *CR1: If* $\Gamma \vdash M \in \mathcal{R}_A$ *then* $\Gamma \vdash M : A \in \mathsf{SN}$

(2) *CR2: If* $\Gamma \vdash M : A \in \mathsf{SNe}$ *then* $\Gamma \vdash M \in \mathcal{R}_A$

In the original paper this theorem consists of three parts and the proof is by mutual recursion. Since the proof is again quite long, I refer the reader to the full source code. The only thing worth mentioning is the handling of the type level function `normR` in F*. If the expected type of a F* term is fully determined by the environment, then F* is able to fold and unfold `normR` automatically, e.g. it accepted a `normR g TBase e`, where a `sSN g TBase e` was expected as return type, but when I applied a `normR g (TArr ta tb) e` to a context, a renaming, etc. of the corresponding types, F* failed, because it was unable to do the coercion. In this case I had to use explicit type declarations.

Poplmark-Reloaded also extends the definition of $\mathcal{R}_A$ to substitutions.

$$\frac{}{\Gamma' \vdash \cdot \in \mathcal{R}.} \qquad\qquad \frac{\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma \qquad \Gamma' \vdash M \in \mathcal{R}_A}{\Gamma' \vdash \sigma, M/x \in \mathcal{R}_{\Gamma, x:A}}$$

The definition basically requires, that every term in the substitution is contained in $\mathcal{R}_A$, where A is the type of the term. It is only a helper definition and I verified these requirements only for one substitution, namely the identity substitution. But this is easy, since it contains only variables. Variables are in SNe by the definition and theorem 2.23 (2) then shows, that all terms in the identity substitution are in $\mathcal{R}$ of their corresponding type. But then $\Gamma \vdash id \in \mathcal{R}_\Gamma$.

We now come to the final result of Poplmark-Reloaded and the corollary that concludes the proof of strong normalisation.

Lemma 2.24 (Fundamental lemma). *If* $\Gamma \vdash M : A$ *and* $\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma$ *then* $\Gamma' \vdash [\sigma]M \in \mathcal{R}_A$

Corollary 2.25. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M : A \in \mathsf{SN}$

The fundamental lemma shows that all type correct terms with any possible context extension is in $\mathcal{R}_A$. We do not actually need it in this generality. As I mentioned earlier, the identity substitution is in $\mathcal{R}_\Gamma$, so $\Gamma \vdash M \in \mathcal{R}_A$ for every term $\Gamma \vdash M : A$. Theorem 2.23 (1) then states, that $\Gamma \vdash M \in \mathsf{SN}$ and this finally is the statement we desired. All type correct terms strongly normalise.

5 SMT in F* Proofs

5.1 General role of the SMT solver

At several points I already mentioned the role of the SMT solver in F*, but never discussed it in detail. The goal of this chapter is to catch up on this topic.

First let's discuss, when the SMT solver is used in F*. Most importantly refinement types make heavily use of the SMT solver. When writing a function, first the refinements of the arguments are taken as the initial preconditions. Later on for each function called, the refinements of the called functions arguments are set as postconditions and proven by the SMT solver. The refinement of the result type is then added to the set of preconditions and finally the refinement of the return type has to be proven with the set of collected preconditions.

But even if one does not use refinement types, F* often calls the SMT solver. In F* as a dependently typed language it is not trivial to check the equality of types, since types depend on and are calculated from values. But while type checking, one often has to prove to types equal, e.g. that the type of a value supplied to a function match its specified argument type. The problem of statically deciding equality of values can be arbitrarily complex, but in many practical cases applying simple lemmata like the basic arithmetic laws suffice. So in F* the SMT solver tries to prove equality constraints, whether they arise from the problem of type equality or not. To enhance this automatic proving, F* also collects equality constraints e.g. while pattern matching. But pattern matching is not only a source of additional information. The SMT solver also checks the completeness of a pattern match and by doing so can prove certain cases infeasible.

The following lemma illustrates the role of the SMT solver while pattern matching very well. It is a utility lemma needed in Section 4 to proof the equality of two substitutions.

```
val sub_eq_index: #g':ctx -> #g:ctx -> s1:sub g' g -> s2:sub g' g
  -> (n:nat {n < length g} -> Tot (ceq (sindex s1 n) (sindex s2 n))) ->
  Lemma (s1 == s2)
let rec sub_eq_index #g' #g s1 s2 f = match s1 with
  | STNil -> ()
  | STCons e1 s1' -> (match s2 with
          | STCons e2 s2' -> let _ = f 0 in
            sub_eq_index s1' s2' (fun n -> f (n+1))
          )
```

The statement of `sub_eq_index` is that the substitutions `s1` and `s2` are equal if all contained terms are equal. Instead of specifying this requirement as a refinement, the requirement was translated according to the Curry-Howard correspondence to the function `(n:nat {n < length g} -> Tot (ceq (sindex s1 n) (sindex s2 n)))`. It selects the contained terms by their index `n` through the substitution indexing function `sindex`. The type `ceq` is declared in the module `FStar.Constructive` and is a constructive substitute for the normal F* equality. Its single constructor `Refl` is only applicable if the second argument of `ceq` is equal to the first.

In the proof the SMT solver was utilised at several points. In the `STNil` case the `()` indicates, that we give the rest of the proof over to the SMT solver. It then concludes from the definition of sub, that g has to be equal to [], because the line | STNil: sub g' [] of the definition requires it to be. But since s1 and s2 have the same type and STNil is the only constructor, there the second context must be empty, one knows, that s2 is equal to STNil too. This concludes the proof for that case, because both s1 and s2 are equal to STNil and therefore equal.

In the other case s1 has the form STCons e1 s1' for some term e1 of type texp g' (index g 0) and the tail of the substitution s1' of type sub g' g0 for some type t and some context g0. We then immediately pattern match on s2. From the first pattern match we know, that g is equal to (index g 0)::g0 and since s1 and s2 are of the same type and STCons is the only constructor for sub, that can take a non-empty list as second context, we know, that s2 is a STCons too. So we only have to provide a proof for that case. By invoking our assumption on the index 0, we give the SMT solver the knowledge, that the terms e1 and e2 are equal. Since there is no use for the actual ceq value, I used the wildcard _ here. Then I invoked the induction hypothesis, by calling sub_eq_index recursively on the tails of the substitutions. To see that the supplied lambda has the correct type, one has to prove that sindex s1 (n+1) is equal to sindex s1 n and the same for s2, but again the SMT solver was able to derive this.

5.2 Problems with the SMT solver in F*

Since the new F* tactics feature was not used in this thesis, as described in Section 2.2, the following discussion applies only to F* without tactics. For a meaningful discussion of tactics, more documentation about its use would be necessary.

The most direct use of the SMT solver in F* is via refinement types. As already mentioned in Section 2.2 one does not prove a refinement directly, instead one supports the SMT solver in proving it by giving helpful lemmata. In contrast in Curry Howard style proofs one directly manipulates the assertions represented by data types and stored in variables. So without the use of the new tactics feature, one has less control over proofs of assertions stated as refinements. Especially if one has to prove an equality by step wise rewriting parts of a term using transitivity, this indirect style is obstructive, because there is no easy way to point out the area to be rewritten. This issue arose in particular when working with substitutions in Section 4.2, since there were many such equalities needed. An example of such an equality is texp_weaken_exchange. There often handing all necessary equalities to the SMT solver was not sufficient, since it did not always find the right subterm to substitute. In this situation the assert keyword was helpful. It takes an assertion in form of an expression of type Type0 and returns unit, like a F* Lemma. The SMT solver checks, whether the given assertion is correct and if so it adds the statement to the preconditions. By stating intermediate steps via this keyword, it is possible to guide the SMT solver in a certain direction.

But this is not the only obstacle. To ensure adequate response times the SMT solver is started with time and space bounds. If an assertion for any reason cannot be proven with the given time bounds, F* returns the error “SMT timed out”, without any further information. If on the other hand the space bounds are too restrictive, F* reports that a seemingly arbitrary assertion failed. Most of the time it is one of the auto generated assertions that arise from F*s handling of effect monads like the pure_null_wp condition, that was introduced in Section 2.1. Both of them can happen due to a lack of evidence, i.e. the provided lemmata are not strong enough in their assertions or because the lemma is too big, so that too much resources needed during the search for a proof. To find the source of the problem, I used the admit keyword. It takes a () and returns whatever type is needed. By using admit () instead of the return value, I was able to find the failing assertion by step-wise checking each of them via the assert keyword, until the problem was found.

6 Discussion

F* has a lot of new promising features, but since the problems presented here are quite common in verification, only the most basic ones are tested. This includes of course dependent types in F*, but also the integrated SMT solver. As I pointed out in Section 3 the SMT solver not only takes the burden of proving nearly trivial helper assertions of the user, but also can discard much more difficult obligations such as the proof, that the fixup routines retain sortedness. This in turn applies only, if the number and difficulty of pre- and postconditions is limited and if the required theories are well supported by the SMT solver. If these conditions does not hold, then it is laborious to force the SMT solver in the right direction, as described in Section 5.2. But these results aren't new. Martínez et al. [2018] comes to a similar conclusion and presents tactics as a possible solution. As already mentioned, I did not use them due to the lack of documentation, but they appear to be a very promising approach and once they are more stable and better documented they may solve these issues.

Loading of work to the SMT solver is very welcome to a lazy programmer, like me, but it often tempted me to first try, whether the SMT solver can prove an assertion, before I actually thought of a possible proof. Although this is no technical issue, it is still important to note, since such proofs easily get unreadable and therefore unmaintainable.

Though refinement types are the proposed way to write proofs, Curry-Howard style proofs like the ones in Section 4 are also well supported. Especially the inference of implicits arguments make it clearer and shorter, because only significant arguments have to be specified. Additionally the derivation on equally constraints while pattern matching lighten the handling of dependently defined data types. Only then proving equalities, one has to rely on the SMT solver, with all its strengths and weaknesses.

One of the research goals in type theory is to make theorem provers more powerful and convenient, so that verification becomes standard in much more areas of programming. Since many easy verification goals can be discarded by the SMT solver in F* and refinement types provide good control over the extracted code and its performance, F* seems to be a step towards this goal. Of course the approach has also its drawbacks. Proofs, where one has to guide the SMT solver, are quite verbose for instance. But to render a final judgement over F* one would have to take its new specialised features like effects and tactics into account. The problems considered here were not suitable for an examination of those, but some further work may correct this.

Bibliography

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 515–529, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009878. URL `http://doi.acm.org/10.1145/3009837.3009878`.

A. W. Appel. Efficient verified red-black trees. Sept. 2011.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 50–65, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31820-0.

N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 49(2):141–159, Aug. 2012. ISSN 1573-0670. doi: 10.1007/s10817-011-9219-0. URL `https://doi.org/10.1007/s10817-011-9219-0`.

B. Beurdouche. Verified cryptography for firefox 57, 2017. URL `https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/`.

T. Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. URL `http://coq.inria.fr`.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. De Gruyter Oldenbourg, 4th edition, 2013. ISBN 978-3-486-74861-1.

L. de Moura and N. Bjørner. Z3: an efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340. R. Ramakrishnan, C and Rehof, Jakob, Apr. 2008.

HACL*. URL `https://github.com/project-everest/hacl-star`.

S. Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11, July 2001.

D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, Toulouse, France, Jan. 2018. 3AF, SEE, SIE. URL `https://hal.inria.fr/hal-01643290`.

G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. arXiv:1803.06547, July 2018. URL `https://arxiv.org/abs/1803.06547`.

miTLS. miTLS: A verified reference implementation of TLS. URL `https://mitls.org/`.

C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 1, Jan. 1993.

B. Pientka. Poplmark reloaded: Mechanizing logical relations proofs (invited talk). In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 1–1, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5586-5. doi: 10.1145/3167102. URL `http://doi.acm.org/10.1145/3167102`.

B. C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2004. ISBN 0262162288.

Project Everest. URL `https://project-everest.github.io/`.

RBTree.fst. URL `https://github.com/FStarLang/FStar/blob/743819b909b804f1234975d78809e18fd9ea0b99/examples/data_structures/RBTree.fst`.

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. URL `http://prosecco.gforge.inria.fr/personal/karthik/pubs/secure_distributed_programming_fstar_jfp15.pdf`.

N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, Jan. 2016. ISBN 978-1-4503-3549-2. URL `https://www.fstar-lang.org/papers/mumon/`.

The F* Team. Verified programming in F* - A tutorial. URL `https://fstar-lang.org/tutorial/`.