Formalisation of Normalisation of the Simply Typed Lambda Calculus in F*

Sebastian Sturm
Ludwig-Maximilians-Universität München
Munich, Germany

As the implementations of program transformations grow in size and complexity, the importance of formal correctness arguments rises. One option is a formal verification using interactive theorem provers. POPLmark-Reloaded is a theorem prover challenge from the field of programming meta-theory, precisely normalisation for the simply typed lambda calculus (STLC), and as such particularly suited for testing theorem provers on that field. This paper presents a partial solution for the POPLmark-Reloaded challenge in F*. F* is a dependently typed programming language usable both for verification and as a proof assistant. It combines interactive proving with automatic, SMT based, approaches. After a short introduction to F* and an overview for POPLmark-Reloaded we will describe the formalisation and some benefits and drawbacks of using F* for programming meta-theory proofs.

1 Introduction

Program correctness is crucial for both safety and security critical systems. To establish strong guaranties on correctness full machine checkable proofs would be desirable. Since direct verification of assembler or machine code is cumbersome, formalisations are (if at all) done using high level languages. While this eases the task significantly, one now has to rely on compilers to produce correct machine code for the verified high level code and modern production scale compilers are far from bug free. One possible solution to this issue is to verify the compiler too, but today even for rather small languages this is a giant task. More powerful program verification tools are necessary. There are already proofs for some properties of the STLC or System F in a variety of theorem provers [1, 8, 4, 3, 7], but since each of the verifications use a different problem, they are hardly comparable.

To be able to measure the progress of theorem provers on the field of programming language meta-theory, the POPLmark-Reloaded challenge [2] was proposed. It is meant as a successor for the POPLmark challenge [5]. While the later focuses on binding structures of System $F_{<:}$, the main challenge of POPLmark-Reloaded is a normalisation proof for the simply typed lambda calculus (STLC).

The contribution of this paper is to provide a formalisation for the main part of POPLmark-Reloaded in F* [11]. F* is a new verification-oriented programming language, developed at Microsoft Research/Inria. It combines interactive theorem proving with semi-automatic verification using a SMT solver and offers recent features like refinement types and extendable effects. Currently the focus appears to be on verifying effectful programs and cryptographic applications. Since the features of F* sound very promising, this paper tries to provide at least some evidence that F* is usable in the field of programming language meta-theory.

In the first section we will give a short introduction to F* and some of its prominent features. We introduce the general syntax of F* and describe refinement types and their proofs, inductive types and F*'s way to ensure termination of recursive function definitions.

Submitted to:
WPTE 2019

The second section gives a short overview over the proof described in the unpublished version of POPLmark-Reloaded. We describe the used language, the evaluation relation, the definitions of strong normalisation used in this proof, the mentioned logical relation and finally the termination result proved using this relation.

In the third section we discuss the actual formalisation. In particular the modelling of strong normalisation and the logical relation in F*, as well as describe some problems when formalising substitutions.

The fourth section shortly discusses the conclusion and limitations of this formalisation and conclude that F* is indeed usable for proof of programming language meta-theory.

2 Short introduction to F*

The syntax of F* is heavily inspired by the syntax of F#. As in F# a program consists of function and type definitions. In particular, standard (pure, total) F# functions can be defined, such as:

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

In addition F* has dependent types, of various forms, the most prominent being refinement types. For example the F* type for natural numbers is defined as `type nat = i:int {i<=0}`. Here `int` is the builtin arbitrary precision integer type and the natural numbers are defined from them as integers that are greater or equal to 0.

The general form of refinement types is $x : t\{\phi\}$ there t is some F* type, x is a F* variable name and ϕ is some predicate that encodes a statement and possibly uses x. It is precisely the subtype of t consisting of all values that satisfy ϕ. The predicate ϕ can be defined by an F* term of type `bool`, in particular one can use user defined functions.

The type checker verifies refinements automatically using an SMT solver. Consider for example the following definition of the factorial function in F*

```
val factorial: x:nat -> y:int {y>0}
let rec factorial x = if x = 0 then 1 else x * factorial (x-1)
```

The argument `x` has to be a natural number and F* refuses all applications of `factorial` where it cannot prove that the argument is non-negative. The refinement of the result type guarantees that `factorial` returns a positive integer. F* proves that by handing the definition and the input and output refinements to the SMT solver. The correctness of the `then` branch is clear but for the `else` branch the SMT solver first has to verify that the recursive call is valid. As the `else` branch is checked we know that $x \neq 0$. Because x has type `nat`, this implies $x \geq 1$ and therefore $x - 1 \geq 0$, which is the precondition of the recursive call. Now the solver has to check that the result of the `else` branch is positive. We already know that $x > 0$ and from the refinement of the result type that `(factorial (x-1))>0`, so the solver concludes that the product of these to numbers is positive.

Often one wants to prove properties separately from the function that produces the value. The following example proves a lower bound for `factorial` and uses it in some function:

```
val factorial_ge_arg: x:nat -> Lemma (factorial x >= x)
let rec factorial_ge_arg x = if x = 0 then () else factorial_ge_arg (x-1)

val factorial_sub: nat -> nat
let factorial_sub n = factorial_ge_arg n; (factorial n) - n
```

The return type `Lemma (factorial x >= x)` is essentially an abbreviation for `u:unit {factorial x >= x}`, but the former is more readable. Anyway `factorial_ge_arg` is a function that returns `unit`

and can be used by calling it. For example in its definition the induction hypothesis is invoked through a recursive call and in the function `factorial_sub` its result type is discarded with the usual ML `;` operator, but the statement of the lemma, instantiated with `n`, is added to the SMT solver knowledge base. It supports the SMT solver in showing that `(factorial n)-n >= 0`.

Besides refinement types F* also features functions which return types and inductive types. Consider for example the following inductive type

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

This declares the running example of a length checked vector. The first parameter `a:Type` is a type parameter (`Type` is the type of types in F*). The second is a value parameter of type `nat` that is specialised in the constructor branches. To make use of type value dependencies, one can name constructor arguments just like function arguments and use them in the types of later arguments and the result type. In the case of the `Cons` constructor the natural number `n` is used to dictate the length of the tail and specialise the length parameter of the result type to `n+1`. The `#` marks `n` as implicit and F* can infer it from the other arguments.

The SMT solver in F* also assists during pattern matching. Consider the following function that converts a length checked vector into a list and proves that they are of the same length.

```
val vec_to_list: #a:Type -> #n:nat -> vector a n -> l:list a {length l=n}
let rec vec_to_list #a #n v = match v with
  | Nil -> []
  | Cons hd tl -> hd::(vec_to_list tl)
```

In the `Nil` case the SMT solver has to prove that `length [] = n` and evaluates the term `length []` to 0. In the definition of `vector` the parameter `n` is specialised to 0 for the `Nil` branch and the pattern matching construct deduces and applies such equalities automatically. So the equality holds. Similarly in the `Cons` case F* deduces that `n = 1+m` where `m` is the length of `tl` and using the induction hypothesis concludes that indeed `length l = n`.

In a dependently typed system types can get quite complex and sometimes non-trivial equational reasoning is need to even state properties. For example F* accepts the following type declarations. It has to check equality of terms to even type-check the statement.

```
val list_to_vec: #a:Type -> l:list a -> vector a (length l)
val list_vec_cons: #a:Type -> e:a -> l:list a ->
        Lemma (list_to_vec (e::l) = Cons e (list_to_vec l))
```

The function `list_to_vec` is just the inverse of `vec_to_list`. In the statement of `list_vec_cons` the left part of the equation `list_to_vec (e::l)` has type `vector a (length (e::l))` while the right part of the equation `Cons e (list_to_vec l)` has type `vector a ((length l)+1)`. The equality operator `=` requires that both sides have the same type, so F* has to prove that `length (e::l) = (length l)+1`. It simply calls the SMT solver which then proves the equality. An equality constraint may be too complex to prove automatically and in a type declaration one cannot easily provide extra evidence with lemmas.

Sometimes the SMT solver fails to prove a statement. In some cases the queried statement is plain wrong and then the SMT solver may even come up with a counter example but usually F* simply reports "SMT timed out". This result is quite unsatisfactory since this message may arise because the queried statement is wrong but the proof may also simply require to many resources or would need expert knowledge not implemented in the solver. In this case one still has the option of guiding the SMT solver in

a certain direction. In F* one can statically assert properties. The SMT solver verifies these properties and adds them to its knowledge base. This way one can direct it to the desired path, although this can be cumbersome since many, often lengthy, intermediate statements have to be written explicitly.

All examples in this paper are pure, total functions and by default F* checks these properties. This is crucial for the soundness of the system because otherwise any statement (including false) could be proven by writing a lemma that only recursively calls itself and never returns. F* supports effectful computation through effect annotations and effect monads but if no effect is specified `Tot` is assumed. `Tot` is the (pseudo) effect of pure, total function, i.e. function with no side-effects and guaranteed termination. To prove termination of recursive functions, F* features a built in ordering on values and automatically extends it to new data type declarations. In addition every recursive function is equipped with a decreasing metric. It is a term that can use the arguments and must produce a value that, for every recursive call, is "smaller", according to the ordering, than the original arguments. F* also provides a default for that metric and therefore one does not always need to specify one. For the following function in turn, one has to specify the metric manually.

```
val n_to_m_list: n:nat -> m:nat {n<=m} -> Tot (list nat) (decreases (m-n))
let rec n_to_m_list n m = if n = m then [m] else n::(n_to_m_list (n+1) m)
```

The function constructs a list from `n` up to `m` and every individual arguments increases or stays constant. But the difference `m-n` decreases in every recursive call and we specify it as metric with the **`decreases`** keyword.

3 Normalisation of STLC

The main contribution of this paper is a the mechanisation of a normalisation proof of the STLC in F*. The basic proof technique is well known. The concrete implementation closely follows a yet unpublished version of the POPLmark-Reloaded paper. A short overview has already been published at [2]. In this section we will describe the normalisation proof taken from the full version and discuss its F* formalisation in the next section.

Strong normalisation in this context means that every STLC term can be fully evaluated in finitely many steps or, heavily simplified, the evaluation does not "loop" forever. The proof uses a rather tiny version of STLC. The grammar for the language is

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

The base type i is kept abstract, so that only variables can inhabit it. The judgement $\Gamma \vdash M : A$ means that the term M has type A under the context Γ and $\Gamma' \vdash \sigma : \Gamma$ means that σ is a substitution that transforms a term under context Γ to a term under context Γ'. We write $\Gamma \vdash M \to N : A$ to denote that M evaluates to N in one step under the context Γ and type A. The rest is standard.

We use full β-reduction as evaluation strategy So the selection of the reduction step is non-determin-

istic and reduction under λ is allowed. This results in the following four reduction steps.

$$\text{S-}\beta\ \frac{\Gamma \vdash \lambda x{:}A.M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}A.M)\,N \to [N/x]M : B} \qquad \text{S-App-L}\ \frac{\Gamma \vdash M \longrightarrow M' : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN \longrightarrow M'N : B}$$

$$\text{S-App-R}\ \frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N \to N' : A}{\Gamma \vdash MN \to NN' : B} \qquad \text{S-Abs}\ \frac{\Gamma, x : A \vdash M \to M' : B}{\Gamma \vdash \lambda x : A.M \to \lambda x : A.M' : A \Rightarrow B}$$

The only "real" reduction step in this language is the S-β step. The other three rules just delegate the β-reduction to a subterm. Accordingly in most lemmas the real effort goes into the S-β case while the other three often only need an invocation of the induction hypothesis.

The description of the proof features two different definitions of strong normalisation. The following is the more intuitive one.

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

We write $\Gamma \vdash M : A \in \mathsf{sn}$ to denote that the term M is strongly normalising according to the above definition. It requires that for all possible reduction steps the result of the reduction is itself strongly normalising. Therefore terms that are already fully evaluated, are by definition strongly normalising. On the basis of those terms, one can define terms which can make at most one evaluation step, then at most two and so on. So to proof that a term is strongly normalising, one has to construct a finite but arbitrarily branched tree of possible reducts.

While this definition is reasonable, it is difficult to handle in proofs. Because of the quantification over all possible reduction steps, simple induction on the derivation it not easily possible. To overcome these obstacles, a second (inductive) definition of strong normalisation is provided, by POPLmark-Reloaded.

The definition is split into three parts: strongly normalising neutral terms $\Gamma \vdash M : A \in \mathsf{SNe}$, strongly normalising terms $\Gamma \vdash M : A \in \mathsf{SN}$ and strong head reduction $\Gamma \vdash M \longrightarrow_{\mathsf{SN}} N : A$.

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

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

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

Strong head reduction defines some sort of call-by-name evaluation scheme, namely it first evaluates the leftmost application. Strongly normalising neutral terms captures those terms that cannot be evaluated using this scheme because they are itself a variable or the leftmost application consists of a variable applied to a term. SN-strongly normalising terms are then defined as either strongly normalising neutral terms or terms with evaluate to a SN-strongly normalising term after a strong head reduction step or for λ-terms the term under the λ has to be SN-strongly normalising.

Since the second definition more complex than the first one, it is not altogether clear that this definition for strong normalisation is sound and therefore a proof is provided.

THEOREM 2.16 (SOUNDNESS OF SN). *If* $\Gamma \vdash M : A \in \mathsf{SN}$ *then* $\Gamma \vdash M : A \in \mathsf{sn}$

The central feature of the proof is the following reducibility relation $\mathscr{R}_A$.

- $\Gamma \vdash M \in \mathscr{R}_i$ iff $\Gamma \vdash M : i \in \mathsf{SN}$
- $\Gamma \vdash M \in \mathscr{R}_{A \Rightarrow B}$ iff for all contexts Γ' and all renamings $\Gamma' \vdash \rho : \Gamma$ and all terms $\Gamma' \vdash N : A$, if $\Gamma' \vdash N \in \mathscr{R}_A$ then $\Gamma' \vdash [\rho]MN \in \mathscr{R}_B$

For the base type i reducibility is simply defined as being strongly normalising. For terms of function type the relation states that a term M of type $A \Rightarrow B$ is strongly normalising if and only if for all strongly normalising terms N of type A the application MN is strongly normalising. The version in POPLmark-Reloaded additionally talks about renamings of the variables because it is needed in some proofs.

The definitions of strongly normalising that we have encountered so far only describe evaluation of terms in isolation. For terms with base type this is sufficient because the term can be fully evaluated in isolation even if it occurs as a subterm. A term of function type can in isolation be at most evaluated to a λ-term but as part of a application, β-reduction can take place and thereby change the term to be evaluated. The requirement of the logical relation in turn is stronger since it considers all possible applications, the function term can occur in.

The following theorem states that the SN-strong normalisation is implied by the logical relation.

THEOREM 2.23 *If* $\Gamma \vdash M \in \mathscr{R}_A$ *then* $\Gamma \vdash M : A \in \mathsf{SN}$

With this preparation in place and a few technical lemmas which we skipped here, one can prove the main lemma.

LEMMA 2.24 (FUNDAMENTAL LEMMA). *If* $\Gamma \vdash M : A$ *and* $\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma$ *then* $\Gamma' \vdash [\sigma]M \in \mathscr{R}_A$

POPLmark-Reloaded also introduces the notion of semantic substitution written by the judgement $\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma$. It states that all terms contained in the substitution have to be in the logical relation $\mathscr{R}_A$ for an appropriate type A. Since all variables are in the relation, the identity substitution *id* is a semantic substitution and it is the only one needed here. Therefore we set $\sigma = id$. Then the lemma states that all (type correct) terms are indeed in the reducibility relation. And by using theorem 2.23 one gets the following corollary.

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

This concludes the proof since it shows SN-strong normalisation of all well-typed terms.

4 Implementation in F*

Since the description of the proof in POPLmark-Reloaded is relative clear and detailed, the amount of open decisions is somewhat surprising for a novice, even if POPLmark-Reloaded explicitly mentions the difficulty and importance of choosing meaningful models of the basic definitions in the verification language in question. In this section we will discuss some of the surprises with and faults in the modelling of the definitions and implementing the lemmas in F*. The complete code of the solution can be found under `https://github.com/sturmsebastian/Fstar-master-thesis-code`.

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

The modelling of the STLC types easy and only included to introduce the constructors.

There are several possible ways to model variables and contexts. For our formalisation we chose de Brujin indices since they are well established and understood. As variables with de Brujin indexing are indexes, contexts can be simply encoded as lists. We used the standard library lists so that all standard list functions are applicable. We modelled variables as simple integers and used the standard index function to retrieve the type of the variable from a context.

```
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
```

As a first attempt to formalise POPLmark-Reloaded we defined terms and typing judgements for terms separately. Variables are represented simply by numbers and were not bound to their context. So per se it was not clear whether the number was in bounds of the context and therefore without the typing judgement one could not retrieve the type of the variable from the context. After transforming the term, it was often difficult to relate the new term to its typing judgement and therefore its context. Because the typing judgement for terms was already a dependent data type, we discovered the good support for dependent pattern matching. Since the typing judgement was needed nearly everywhere anyway, we decided to follow the solutions described in POPLmark-Reloaded and use a intrinsic approach [6]. It means that the definitions of terms and typing judgements are merged into one single data type and thereby only type correct terms can be constructed.

It is convenient to define substitutions as a inductive data type, to be able to express properties as distinct lemmas.

When implementing the application of a substitution to a term, the desirable solution fails at λ-abstraction. Let $\Gamma' \vdash \sigma : \Gamma$ be a substitution and $\Gamma \vdash \lambda x : A.M : B$ a term for some types A, B and contexts Γ, Γ'. The application of a substitution to the term $[\sigma](\lambda x : A.M : B)$ is defined as $\lambda x : A.[\sigma, x/x]M : B$. The full judgement $\Gamma', x : A \vdash \sigma, x/x : \Gamma, x : A$ shows that not only the substitution is extended by the identity for x but also the destination context is extended (weakened). So implicitly the context for all terms inside the substitution is extended. Most textbook proofs use named representation and there only the context but not the terms are modified when extending the context. With de Brujin indices all variables in the terms need to be incremented. This weakening of terms can be written as substitution and since proving properties of substitution involves proving properties of weakening, one would like to define them as such, to be able to reuse substitution lemmas. In the definition of the application of substitutions one then would need to recursively call the application function on the terms inside the substitution for the weakening. As already mentioned, we need to provide a decreasing metric for recursive function definition and usually the size of the term is used. This does not work here since the terms inside the substitution may be bigger then the original term. A suitable decreasing metric may exists but it is probably quite complex to prove correct.

To overcome this dilemma we defined weakening (and exchange) of substitutions as separate functions and used them in the definition of substitution. The drawback of this approach is the amount of technical lemmas that are needed to prove properties of substitutions and that those special functions are indeed equivalent to a substitution.

In contrast the Coq solution in POPLmark-Reloaded uses a different approach. It defines renamings separately from substitutions. Renamings are substitutions where all terms inside it are only variables. The weakening of a term is a renaming since it simply increments variables. Therefore one can find easily a decreasing metric for the described recursive algorithm to specialised to renamings. Then one

can define the application of substitutions using renamings. This way one does not need extra lemmas about weakening to the price, that many lemmas about substitution have to be proven twice, once for renamings and once for arbitrary substitutions.

Having substitutions in place one can define the evaluation of the calculus. Like the abstract definition the `step` type represents a relation between the original term and the term after the step.

```
type step (g:ctx): t:ty -> e:texp g t -> e':texp g t -> Type =
  | 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')
```

The `step` type has actually four constructors but we selected only one for illustration purposes. It corresponds to the S-ABS reduction rule and it warps a step of the term under the λ. This way terms in this STLC can also be evaluated under the λ. The rules are modelled as a relation that ties terms before a single reduction step to them after the reduction step.

The first definition sn-strong normalisation is modelled as

```
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
```

As already discussed the number of requirements for sn-strong normalisation varies depending on the number of possible reduction steps. Therefore the paper version quantifies universally over the results of the reduction step. To model this definition in a data type, we simply translated the requirement to a function using the Curry-Howard correspondence.

```
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'
```

The type of the function `sn_lam` states that given a sn-strongly normalising term `e` with a free variable of type `tlam` then the λ-abstraction of `e` for the variable of type `tlam` is itself sn-strongly normalising. Because of the use of de Brujin indices, the position of the variable in the context has to be fixed and in this lemma the first variable in the context is chosen.

Proving this lemma then consists of writing a term of the desired type. In this case we had to provide a value of type `ssn`. To do so we let-bind the definition of the function argument to the name `f'`. It takes a new term `q` of function type and a step `st` from our original term `TyLam tlam e` to `q` as arguments. Since the only step that takes a λ-term as source is the S-ABS step, the value `st` can only inhabit the `SLam` constructor. We can skip the other cases since F* is powerful enough to prove them infeasible. The `SLam` constructor warps a step of the term under the λ. Because the step `st` results in `q` and the result of a `SLam` step has to be a λ-term, we know that `q` is of the form `TyLam tlam q'` for some term `q'` and that `hst` is a step from `e` to `q'`. To gain a `ssn` value for `q`, we can then apply `sn_lam` recursively to some `ssn` value for `q'`. The function in our induction hypothesis produces a `ssn` value for `q'`, given a step from `e` to `q'`. The only thing left to do is wrapping our defined function in a `Ssn` constructor again.

Until now we did not mention the natural number argument to the `ssn` type. A proof for sn-strong normalisation is in principle an finite but arbitrary branched derivation tree of `ssn` values. The natural

number represents an upper bound on the "height" of this derivation tree. It is needed as a decreasing metric for proofs by induction over `ssn` values, since the built in ordering cannot handle function values at the time of writing.

Since the modelling of SN-strong normalisation as the inductive data type `sSN` is straightforward, we omitted its definition.

```
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)) )
```

It is not possible to represent the logical relation as a data type. In F* all recursive occurrences of the defined data type have to be positive i.e. if one constructor contains a function when the recursively defined data type may only occur in covariant positions. To overcome this restriction we modelled the relation as a function returning a type that represents the relation in this case. This is meaningful since the relation has distinct definitions for the STLC base type and for function types. For the base type it is simply equivalent to SN-strong normalisation and for function types we again translated the requirements according to the Curry-Howard correspondence.

For definitions like `normR` the main concern is the usability of type coercions, i.e. the difficulty to force F* to unfold or fold the definition. If the required type is explicitly given, e.g. for return values or function arguments, then the coercion happens automatically. If the requirements are only implicit, e.g. if one has a `normR` value of a term of function type and wants to apply this value to some arguments then F* cannot do the coercion automatically and it has to be forced explicitly, e.g. by let-binding it to a new identifier with a type annotation.

5 Discussion

We formalised the normalisation of the STLC in F* following the description of the POPLmark-Reloaded theorem prover challenge. It turned out that F* has reasonable support for mechanising programming language meta-theory. Especially the good support for highly dependent inductive data types and the pattern matching on those proved helpful for this task. Also refinement types enabled the reuse of data types, both standard and user-defined ones, and therefore limited the amount of technical lemmas needed.

But proofs by refinement types, discarded by the SMT solver, do not scale the same way as Curry-Howard style proof. A Curry-Howard style proof needs approximately the same handling, regardless of whether it stands alone or appears as part of a larger proof, but for refinement types the SMT solver can search large parts of the proof space for refinement types, but for larger proofs this is simply not feasible, so that one has to be the more precise the larger the proof is. Without precise knowledge of the theories the SMT solver support and how the goals are encoded proofs of properties encoded as refinement types easily become unstable.

It is certainly possible verify large complex systems with refinement types, as the effort on miTLS [10] shows. miTLS tries to build a fully verified TLS implementation using F*. Also there are whole theorem provers like LiquidHaskell [12], that rely on refinement types as their only way to state lemmas. The missing link is some kind of documentation or tutorial of how to utilise them correctly.

In order to avoid the issues with refinement types we tried to limit the use of refinement types and stayed close to the formalisation presented in the POPLmark-Reloaded paper. Besides these, there are numerous other formalisations that employ Girard's method [9] to the STLC or System F, some of them using de Brujin indices, like this paper, others higher order abstract syntax. [1, 8, 4, 3, 7]

POPLmark-Reloaded uses a minimal language and a short, but non-trivial, proof to lower the bar of contribution to the comparison. Other formalisation's, especially those of the normalisation of System F [8, 3] are of much larger scale. Since the amount of support from the SMT solver somewhat depends on the size of the proof, it is not clear, whether F⋆ supports proofs on that scale equally well.

The F⋆ developers also claim [11] that F⋆ features advanced code extraction support and therefore a somewhat natural extension of this work would be the integration into the framework of [7]. In this paper proofs for the normalisation of STLC are developed in Minilog, Coq and Isabelle/HOL and from them programs implementing normalisation by evaluation are extracted. The rational behind this work is to test and compare the proving and extraction capabilities of these three provers.

We think that, in order to make progress, it is essential to compare the theorem provers currently available and try to learn from the advantages and shortcomings of the other. For that reason we would like to encourage others to contribute to challenges like POPLmark-Reloaded.

References

[1] Andreas Abel (2008): *Normalization for the Simply-Typed Lambda-Calculus in Twelf*. Electronic Notes in Theoretical Computer Science 199, pp. 3 – 16. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[2] Andreas Abel, Alberto Momigliano & Brigitte Pientka (2017): *POPLMark Reloaded*. Logical Frameworks and Meta Languages: Theory and Practice.

[3] Thorsten Altenkirch (1993): *A formalization of the strong normalization proof for System F in LEGO*. In Marc Bezem & Jan Friso Groote, editors: Typed Lambda Calculi and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 13–28.

[4] Thorsten Altenkirch & Ambrus Kaposi (2016): *Normalisation by Evaluation for Type Theory, in Type Theory*. CoRR abs/1612.02462.

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In Joe Hurd & Tom Melham, editors: Theorem Proving in Higher Order Logics, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 50–65.

[6] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy & Conor McBride (2012): *Strongly Typed Term Representations in Coq*. Journal of Automated Reasoning 49(2), pp. 141–159.

[7] Ulrich Berger, Stefan Berghofer, Pierre Letouzey & Helmut Schwichtenberg (2006): *Program Extraction from Normalization Proofs*. Studia Logica 82(1), pp. 25–49.

[8] Kevin Donnelly & Hongwei Xi (2006): *A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F*.

[9] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press, New York, NY, USA.

[10] *miTLS: A Verified Reference Implementation of TLS*. Available at `https://mitls.org/`.

[11] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué & Santiago Zanella-Béguelin (2016): *Dependent Types and Multi-Monadic Effects in F**. In: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, pp. 256–270.

[12] Niki Vazou, Joachim Breitner, Will Kunkel, David Van Horn & Graham Hutton (2018): *Functional Pearl: Theorem Proving for All (Equational Reasoning in Liquid Haskell)*. CoRR abs/1806.03541.