Towards a Provably Correct Encoding from F* to SMT

Alejandro Aguirre (Université Paris Diderot, Paris 7)

Report on MPRI M2 internship supervised by Cătălin Hriţcu (Inria Paris)

September 1, 2016

Introduction

General Context

F* is an ML-like language aimed at verification (Swamy et al., 2016). It has been used for several real-world applications such as the verification of cryptographic protocols or the formalisation of a fragment of its own metatheory. F* tries to combine the richness of the logic of systems like Coq, Agda or Isabelle with the automation power of verification tools such as Why3 or Dafny. To check that a program meets its specification, F* uses a combination of manual proofs and automation via SMT solvers. However, there is no formal guarantee of the soundness of this automation process and currently has to be trusted by the users.

Research Problem

F* has a rich type system that allows the user to express preconditions and postconditions for their programs. The typechecker generates some higher-order verification conditions that are then encoded into first-order logic and passed to an SMT solver to be automatically proven. This enconding is complex, and currently has to be trusted. Having a formal proof of its soundness would be highly desirable, since this would remove the need to assume it, and would overall make F* a more reliable verification tool.

Similar translations from higher- to first-order logic are usually proven to be sound using model-theoretic reasoning, i.e., giving an algorithm that translates models of the target language into models of the source language. However, models for F* are a matter of ongoing research, and for this reason, these kind of proof techniques fail.

Our first attempt to prove soundness was unsuccessful. We were trying to mechanise a Coq proof in which we had given semantics in `Prop` to the SMT terms, and we were trying to map them back to a proof in pF* were validity was inductively defined with inference rules. This turned out to be much more impractical than we thought on the first place, since essentially we were trying to turn a semantic proof into a syntactic one. Therefore, we needed to find a better proof technique that would allow us to solve the problem.

Contribution

Our starting point for a source language was pF*, a lambda calculus for refinement types (Swamy et al., 2016). I designed a lambda lifted variant for it, $p\text{F}^\star_{LL}$ (Section 2).

I formalised the encoding from $p\mathrm{F}^{\star}_{LL}$ into $\mathrm{FOL}_{=N}$, intuitionistic FOL with equality and arithmetic for which I use a standard representation of natural deduction proofs as proof terms in eta-long normal form. The logic is presented in Section 3 and the encoding in Section 4.

I gave a proof-theoretic argument of the soundness of the encoding, i.e. I showed that if we have a proof of the encoding of a $p\mathrm{F}^{\star}$ judgement in FOL, then it can be mapped back to a proof of that judgement in $p\mathrm{F}^{\star}$ (Section 5).

Additionally we wrote an extended abstract about the results in Section 1, which was accepted to the Hammers for Type Theories workshop, and I gave a talk there presenting said results.

Finally, during the first weeks of the internship, we wrote a Coq file while attempting to do a model-theoretic proof of soundness. Even if the proof was unsuccessful, most of the definitions can be reused to mechanise the proof-theoretic approach.

Validity

We have a formal proof of the soundness of the encoding of a small subset of $\mathrm{F}^{\star}$ into intuitionistic first-order logic. The proof-theoretic techniques we chose turned out to be very effective for our setting, since they allow the problem to be solved in an elegant and modular manner, which we hope to extend to bigger subsets of $\mathrm{F}^{\star}$ using similar arguments as the ones we present.

Additionally, the fact that the proof is proof-theoretic and constructive, allows us to build a derivation in the source language from a derivation in the target language. This process, known as proof reconstruction, could let us have automatically generated $\mathrm{F}^{\star}$ proofs in the future.

Conclusion and Next Steps

Proof-theoretic techniques, as opposed to our initial unsuccessful Coq semantic proof, were ultimately the right tool to solve the problem. The choice of the eta-normal form allowed for much natural reasoning, which lead to an elegant and modular proof. We hope to be able to extend said proof to a larger setting.

There are a few natural continuations to our work, such as extending both the source and the target language or mechanising the formalisation in Coq. The main one is that our target is intuitionistic logic while the SMT solvers are classical. To achieve this we could either use some variant of the double negation translation from classical to intuitionistic logic, or add one of the classical rules we are missing. Both methods have their strengths and their weaknesses, which we will discuss in more detail later on.

Secondly, the source language can be greatly expanded. Of course the full language of $\mathrm{F}^{\star}$ has many more constructs, but there are some subsets of it that have already been studied and for which a metatheory has already been developed. In addition, some optimisations of the encoding could also be formally verified.

Proof reconstruction could also be an interesting problem to look at. Some SMT solvers, such as Z3, can return a derivation in natural deduction style . This problem has been studied in the context of Isabelle/HOL in (Böhme and Weber, 2010). If we had proof reconstruction, the SMT solver would no longer have to be trusted since we could actually check the generated proofs directly in $\mathrm{F}^{\star}$. This is a very ambitious continuation, since it would require as prerequisites being able to parse the derivation trees returned by the SMT solver and extending the proof to a much larger subset of $\mathrm{F}^{\star}$.

Finally, both the work so far and the future one could be mechanised in Coq. A good starting point would be our unsuccessful semantic proof, since most of the definitions of the source and target languages and the encoding can be reused.

1 From F^* to SMT

Verification conditions in the dependently-typed higher-order logic of F⋆ are encoded to SMT formulas and discharged automatically using Z3. This nontrivial encoding tries to strike a pragmatic balance between completeness and practical tractability. For this the encoding (1) combines a deep and a shallow embedding of F⋆ terms, (2) allows bounded unrolling of recursive and inductive definitions, and (3) performs lambda lifting for eliminating higher-order functions.

In this section we illustrate some of this by encoding the following F⋆ function computing factorial:

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

The refinement type `x:int{x>=0}` denotes the integers that are greater than or equal to 0, that is, the naturals. For the `nat` type definition, the SMT encoding generates the following very intuitive formula: a term `@x` has type `nat` when it has type `int` and when `@x` is larger than or equal to zero. Formally, F⋆ expressions are encoded using the SMT sort `Term`, the F⋆ typing relation is encoded as the `HasType` predicate.

```
(assert (forall ((@x Term))
                (! (iff (HasType @x nat)
                        (and (HasType @x int)
                             (>= (BoxInt_proj @x) 0)))
  :pattern ((HasType @x nat)))))
```

The pattern `HasType @x nat` is bound to the quantifier by the reserved symbol bang (`!`), and allows the SMT solver to instantiate the `forall` only when it encounters an SMT term that matches the left hand side of the equivalence. In the case above, whenever there is an SMT term of the form `(HasType y nat)` for some SMT term `y`, the solver will try to instantiate the quantifier producing a formula of the form

```
(iff (HasType y nat)
     (and (HasType y int)
          (>= (BoxInt_proj y) 0)))
```

Sort `Term` denotes an open data type (open to capture the ability to add new inductive types in F⋆) and `BoxInt` is a constructor taking an `Int` and returning a `Term`. In the formula above, `BoxInt_proj` projects the integer out of a term that has type `int`. F⋆ arithmetic operations are encoded in terms of the SMT solver's ones; for instance multiplication on `Term`s is defined as follows:

```
(declare-fun op_Multiply (Term Term) Term)
(assert (forall ((@x0 Term) (@x1 Term))
                (! (= (op_Multiply @x0 @x1)
                      (BoxInt (* (BoxInt_proj @x0) (BoxInt_proj @x1))))
  :pattern ((op_Multiply @x0 @x1)))))
```

Since we marked the `factorial` function as total (`Tot`), F⋆ needs to show its termination by proving that the recursive call `factorial (n - 1)` is done on a "strictly smaller" argument. For this, F⋆ asks the SMT solver to prove that `n - 1` is strictly smaller than `n`, using the following idealised SMT query:

```
(declare-fun n () Term)
(assert (not (implies (and (HasType n nat) (not (= n (BoxInt 0))))
                      (Valid (Precedes (op_Subtraction n (BoxInt 1)) n)))))
```

If the SMT solver proves this negative formula unsatisfiable then F* concludes that factorial terminates. For this, F* defines a well-founded ordering on all values called `Precedes`, which for naturals is the usual less than relation. The `Precedes` relation is deeply embedded and given meaning in the SMT solver by a `Valid` predicate. For instance, the ordering on naturals can be axiomatised as follows:

```
(assert
  (forall ((@x1 Term) (@x2 Term))
          (! (implies
                 (and (HasType @x1 int) (HasType @x2 int)
                      (> (BoxInt_proj @x1) 0) (>= (BoxInt_proj @x2) 0)
                      (< (BoxInt_proj @x2) (BoxInt_proj @x1)))
                 (Valid (Precedes @x2 @x1)))
  :pattern ((HasType @x1 int) (HasType @x2 int)
            (Valid (Precedes @x2 @x1))))))
```

While in this simple example, `Valid` of `Precedes` could be represented as just a FOL predicate, F* allows type- and formula-level computations, which can also be performed by the SMT solver. For instance, in a more complex example the `Precedes @x2 @x1` formula above could be computed by applying a type-level function to arguments.

Afterwards, F* needs to prove that `factorial` returns a `nat`. To encode this, first it defines an arbitrary natural `n` that stands for the argument to factorial. Afterwards, F* admits that for every term `x` preceding `n`, `factorial x` has type `nat`.

```
(declare-fun n () Term)
(assert (HasType n nat))

(assert (forall ((@x Term))
                (! (implies (and (HasType @x nat) (Valid (Precedes @x n)))
                            (HasType (factorial @x) nat))
          :pattern ((factorial @x)))))
```

Finally, we use the assumption above to check that `factorial` returns a `nat`. In the base case this check is trivially `1 >= 0`. For the recursive case, F* has to prove that `n*factorial(n-1) >= 0`, knowing that `n` is a `nat` different from 0. This condition is encoded as the following SMT query:

```
(assert (not
  (ite (BoxBool_proj (op_Equality int n (BoxInt 0)))
       (>= 1 0)
       (>= (BoxInt_proj
              (op_Multiply n
                (factorial (op_Subtraction n (BoxInt 1))))) 0 ))))
```

Once `factorial` has been verified to be total its definition can be used in specifications (e.g. refinements), so we want to allow the SMT solver to perform computations that unroll factorial. For this we encode the body of the factorial function as an equivalent SMT function. For efficiency, we only allow the SMT solver to perform *bounded* unrolling of recursive functions. For this, we define a new SMT function (`factorial_fuel`) that takes an additional fuel argument, which controls the number of times it can be unrolled. The fuel sort is defined as an unary natural number with `ZFuel` and `SFuel` as the constructors:

```
(declare-datatypes () ((Fuel (ZFuel) (SFuel (prec Fuel)))))

(declare-fun MaxFuel () Fuel)
(assert (= MaxFuel (SFuel (SFuel ZFuel))))
```

`MaxFuel` is a parameter that can be overridden by the user. For instance, the definition above sets `MaxFuel` to 2, which is the current default in F⋆.

The following equation defines the fuel-instrumented function using a recurrence relation. Notice that the recursive call is made with one less fuel unit, ensuring that the unrolling can only happen a bounded number of times (determined by `MaxFuel`). When the fuel runs out `factorial_fuel` is treated as an uninterpreted function.

```
(assert
  (forall ((@f Fuel) (x Term))
    (! (implies (HasType x nat)
                (= (factorial_fuel (SFuel @f) x)
                   (ite (= (op_Equality int x (BoxInt 0)) (BoxBool true))
                        (BoxInt 1)
                        (op_Multiply x
                               (factorial_fuel @f
                                    (op_Subtraction x (BoxInt 1)))))))
   :pattern ((factorial_fuel (SFuel @f) x)))))
```

Another equation relates the original and the fuel-instrumented factorial SMT functions:

```
(assert
   (forall ((@x Term))
           (! (= (factorial @x)
                 (factorial_fuel MaxFuel @x))
   :pattern ((factorial @x)))))
```

An SMT function called `ApplyTT` is defined for representing function applications. In the case of factorial, we introduce a fresh token `factorial@tok` for which we define `ApplyTT` as follows:

```
(assert (forall ((@x Term))
                (! (= (ApplyTT factorial@tok @x) (factorial @x))
                   :pattern ((ApplyTT factorial@tok @x)))))
```

The F⋆ encoding also uses lambda lifting for eliminating first-class functions. Lambda lifting is a process in which lambda constructs are removed by giving them global names and passing them their free variables as extra arguments. It is a commonplace technique in similar translations. Rather than giving a formal definition, we will see a small example. Suppose we have the following definition:

```
lambda x = (lambda y = x + y) x
```

On a first stage, lambdas are given names, and each of them gets passed the free variables it uses. As a result, we have:

```
let f x =
  let g y x = x + y in
  g x x in
...
```

$$
\begin{aligned}
t &::= nat \mid x{:}t_1 \to t_2 \mid x : t\{\phi\} \\
\varphi &::= e_1{<}e_2 \mid e_1{=}e_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x{:}t.\phi \mid \bot \\
n &::= O \mid S\ n \\
e &::= x \mid n \mid f \mid e_1\ e_2 \mid S\ e \mid pred\ e\ e_O\ e_S \\
bs &::= (x : t) \quad \mid (x : t), bs \\
ds &::= \quad \mid f\ bs\ :\ t = e \\
\Gamma &::= bs; ds
\end{aligned}
$$

Figure 1: Syntax of $p\mathrm{F}^\star_{LL}$

After that, `g` is closed and can be lifted out to the top level:

```
let g y x = x + y in
let f x = g x x in
...
```

The result is a program than no longer has lambda abstractions, but rather consists in a list of closed functions definitions (also called *supercombinators* in the literature) and a main body:

```
let f1 x11 ... x1k = b1 in
...
let fn xn1 ... x1j = bn in
e
```

2 Source Language: $p\mathbf{F}^\star_{LL}$

We present the fragment of F* that we encode to first-order logic in a provably correct way. For the calculus we assume that lambda-lifting has already been performed. We call this calculus $p\mathrm{F}^\star_{LL}$, but for brevity we will usually call it just pF* (read pico F*).[1]

On actual F*, lambda lifting is done at the same time as the encoding, but separating the two phases allows us to ease reasoning, and leads to more modular definitions and proofs.

2.1 Syntax

The syntax for $p\mathrm{F}^\star_{LL}$ can be seen in Figure Figure 1. The meta-variable t represents a type, that can be *nat*, the type of natural numbers, a function type from t_1 to t_2, or a refinement type $x : t\{\varphi\}$, which is the type of all the elements e of t for which $\varphi[e/x]$ is a valid formula. A formula φ is either an equality or an inequality between two expressions, a conjunction or implication of two formulae, or a universal quantification of a formula. An expression is either a natural number, a variable, an application of one expression to another, or the successor of an expression. We use *bs* to represent a list of bindings and *ds* to represent a list of function definitions (can be read as a `let f (x1:t1) ... (xn:tn) : t = b in ...`). Finally, an environment Γ is a list of bindings of the free variables to their type, and a list of function definitions.

[1] An extension of this calculus with weakest preconditions instead of refinements was already presented by Swamy et al. (2016)

2.2 Static Semantics

$p\text{F}^\star$ has 6 mutually inductive typing judgements:

- Formula validity: $\Gamma \vDash \varphi$
- Expression typing: $\Gamma \vdash e : t$
- Subtyping : $\Gamma \vdash t_1 <: t_2$
- Well-formedness of an environment: $\Gamma \vdash \mathit{wf}$
- Well-formedness of a formula: $\Gamma \vdash \varphi\ \mathit{wf}$
- Well-formedness of a type: $\Gamma \vdash \tau\ \mathit{wf}$

The complete rules for these judgements can be found in the Appendix; in Figure 2 we present the most interesting rules. To type a term with a refinement type, we need to prove that the term has the base type and that the refinement formula is valid when instantiated for that term (T-Ref). The (V-Ref") rule is dual, and allows us to use the information contained in a refinement type to prove validity of other formulas. The universal quantifier is typed, so we need to prove the typing of the term that is used to instantiate it (Forall-Elim). Explosion can be used either to derive any validity judgement (V-Expl) or any subtyping judgement (Sub-Expl). Notice that via subtyping, the following rule is admissible:

$$\frac{\Gamma \vDash \bot \qquad \mathit{fv}(e) \subseteq \mathit{dom}(\Gamma) \qquad \Gamma \vdash t\ \mathit{wf}}{\Gamma \vdash e : t}\ \textsf{(T-Expl)}$$

that is, under a proof of $\bot$ we can type a term with any well-formed type, as long as we can type all of its free variables. This can be done only if they are bound in Γ to some type.

Finally, we will see some rules that relate the various judgements. The (T-Sub) rule relates the typing and the subtyping judgements, by allowing to retype a term with any supertype. And the (V-SubstEq) lets us substitute an expression e_1 inside a formula by another one for which we have proven the equality $e_1 = e_2$. From (V-ReflEq) and (V-SubstEq) one can get the other properties of equality, such as symmetry and transitivity.

3 Target Logic: $FOL_{=N}$

3.1 Syntax and Semantics

The target of our encoding is a sorted first-order logic with equality and arithmetic. In this work we use an intuitionistic version of it, and at the end discuss how our results can be extended to a classical setting. Terms and formulae have the usual syntax, that we recall in the following definitions:

Definition 1 (Term)**.** *Terms are defined inductively:*

- *A variable x of sort s is a term of sort s.*
- *An application $f(t_1, \ldots, t_n)$ where f is a function symbol of arity $s_1 \times \ldots \times s_n \to s$, and $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ respectively, is a term of sort s.*

Definition 2 (Formula)**.** *Formulae are defined inductively:*

$$\frac{\Gamma \vdash e : x : t\{\varphi\}}{\Gamma \vDash \varphi[e/x]}\ \textsf{(V-Ref'')} \qquad \frac{\Gamma, x : t \vDash \varphi}{\Gamma \vDash \forall(x : t).\varphi}\ \textsf{(Forall-Intro)}$$

$$\frac{\Gamma \vDash \forall(x : t).\varphi \qquad \Gamma \vdash e : t}{\Gamma \vDash \varphi[e/x]}\ \textsf{(Forall-Elim)} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t \qquad \Gamma \vdash e \rightsquigarrow e'}{\Gamma \vDash e = e'}\ \textsf{(V-RedE)}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vDash \varphi[e/x]}{\Gamma \vdash e : x : t\{\varphi\}}\ \textsf{(T-Ref)} \qquad \frac{\Gamma \vDash \bot \qquad \Gamma \vdash \varphi\ wf}{\Gamma \vDash \varphi}\ \textsf{(V-Expl)}$$

$$\frac{\Gamma \vDash \bot \qquad \Gamma \vdash t_1\ wf \qquad \Gamma \vdash t_2\ wf}{\Gamma \vdash t_1 <: t_2}\ \textsf{(Sub-Expl)} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash t <: t'}{\Gamma \vdash e : t'}\ \textsf{(T-Sub)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vDash e = e}\ \textsf{(V-ReflEq)} \qquad \frac{\Gamma \vDash e_1 = e_2 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vDash \varphi[e_1/x]}{\Gamma \vDash \varphi[e_2/x]}\ \textsf{(V-SubstEq)}$$

Figure 2: Some rules for $p\mathrm{F}^\star$

- *An application $P(t_1, \ldots, t_n)$ where P is a predicate symbol of arity $s_1 \times \ldots \times s_n$, and $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ respectively, is a formula.*
- *False ($\bot$) is a formula.*
- *If ψ_1 and ψ_2 are formulae, so are $\psi_1 \wedge \psi_2$, $\psi_1 \to \psi_2$*
- *If ψ is a formula, and x is a variable of sort s, $\forall x^s.\psi$ is a formula.*

Now we will present the logical signature in which we will be working.

Definition 3 (Signature)**.** *Our signature is going to contain the following symbols:*

- *Two sorts, Type for types, and Term for terms.*
- *A function symbol @ of arity Term $\times$ Term $\to$ Term to represent application.*
- *A constant 0 of sort Term, and a constant Nat of sort Type*
- *A function symbol succ of arity Term $\to$ Term to represent the succesor.*
- *A function symbol pred of arity Term $\times$ Term $\times$ Term $\to$ Term to represent the branch operator over the naturals.*
- *A predicate HasType of arity Term $\times$ Type to represent typing guards and judgements.*
- *Predicate symbols eq and lt of arity Term $\times$ Term to represent equality and less-than relations respectively.*
- *In addition, we will have a constant of sort Term for every $pF^\star$ variable we encode, and a constant of sort Type for every $pF^\star$ type that we encode.*

Validity of formulae in this logic is defined via natural deduction rules. We will have proof terms representing a derivation in natural deduction, in the usual Curry-Howard style. Simply-typed lambda calculus has to be extended with some type dependency in order to be able to have proof terms for both

$$\eta ::= \lambda x{:}s.\ \eta \mid \lambda X{:}\alpha.\ \eta \mid \delta \mid \langle \eta_1, \eta_2 \rangle \mid fe_{R(e)}\delta$$
$$\delta ::= X \mid \delta\ \eta \mid \delta\ t \mid pr_1\ \delta \mid pr_2\ \delta$$

Figure 3: Grammar for η-long normal form FOL proof terms

$$\frac{\Delta \vdash \eta_1 :_\eta \psi_1 \qquad \Delta \vdash \eta_2 :_\eta \psi_2}{\Delta \vdash \langle \eta_1, \eta_2 \rangle :_\eta \psi_1 \wedge \psi_2}\ (\eta\mathsf{I}\wedge) \qquad \frac{\Delta \vdash \eta :_\eta \psi}{\Delta \vdash \lambda x.\eta :_\eta \forall x.\psi}\ (\eta\mathsf{I}\forall) \qquad \frac{\Delta, X :_\delta \psi_1 \vdash \eta :_\eta \psi}{\Delta \vdash \lambda X : \psi_1.\eta :_\eta \psi_1 \to \psi_2}\ (\eta\mathsf{I}\to)$$

$$\frac{\Delta \vdash \delta :_\delta R(\vec{t})}{\Delta \vdash \delta :_\eta R(\vec{t})}\ (\eta\delta) \qquad \frac{\Delta \vdash \delta :_\delta \bot}{\Delta \vdash fe_{R(\vec{t})} :_\eta R(\vec{t})}\ (\eta\mathsf{E}\bot)$$

$$\frac{\Delta \vdash X : \psi \in \Delta}{\Delta \vdash X :_\delta \psi}\ (\delta\mathsf{X}) \qquad \frac{\Delta \vdash \delta :_\delta \forall x.\psi}{\Delta \vdash \delta\ t :_\delta \psi[t/x]}\ (\delta\mathsf{E}\forall)$$

$$\frac{\Delta \vdash \delta :_\delta \psi_1 \to \psi_2 \qquad \Delta \vdash \eta :_\eta \psi_1}{\Delta \vdash \delta\ \eta :_\delta \psi_2}\ (\delta\mathsf{E}\to) \qquad \frac{\Delta \vdash \delta :_\delta \psi_1 \wedge \psi_2}{\Delta \vdash pr_i\ \delta :_\delta \psi_i}\ (\delta\mathsf{E}\wedge)$$

Figure 4: Natural deduction rules for η-long normal form FOL proof terms

universal quantification (abstraction over terms) and implication (abstraction over proofs), but we leave the details out.

Instead, in our presentation we focus on η-long normal proof terms, which are produced by the context-free grammar in Figure 3. Deduction rules and their corresponding proof terms can be seen in Figure 4. We will leave well-sortedness judgements implicit, but they are needed in the introduction and elimination of quantifiers.

To give a more intuitive idea, the η-terms end in an introduction, while the δ-terms end in on an elimination.

The same set of derivation rules without the η and δ sub-indices correspond to the usual intuitionistic FOL, and produce proof terms which are not necessarily in normal form. We will denote such generalised proof terms with a capital letter ranging in N,P,Q, and the derivations without a sub-index for the colon, i.e. $\Delta \vdash P : \psi$.

3.2 Properties

The main property is that η-long normal forms are complete with respect to intuitionistic FOL.

Lemma 1. *For every environment Δ and formula ψ, if there exists a proof-term P such that $\Delta \vdash P : \psi$, then there exists an η proof-term η such that $\Delta \vdash \eta :_\eta \psi$*

Proof. By η-expansion and β-reduction. □

In other words, there exists always a proof ending in an introduction. Notice that this theorem would not work well with disjunction or existentials, since it would be equivalent to the witness property, which intuitionistic logic has, but it is lost in the presence of axioms.

Before stating the main property about η proof terms, we need two more definitions. The *head* of a δ proof term is the axiom that it is instantiating. One can see from the grammar that the δ production rule must necessarily end with an X, which is a proof variable corresponding to an axiom.

Definition 4. *The head of a δ-proof-term is inductively defined as follows:*

$$\begin{aligned} \mathit{head}(X) &:= X \\ \mathit{head}(\delta_1\delta_2) &:= \mathit{head}(\delta_1) \\ \mathit{head}(pr_i\delta) &:= \mathit{head}(\delta) \end{aligned}$$

On the other hand, the *targets* of an axiom is the set of atoms that can be proven after all possible eliminations and instantiations.

Definition 5. *The set of targets of a FOL-formula is inductively defined as follows:*

$$\begin{aligned} \mathit{targs}(R(\vec{t})) &:= R \\ \mathit{targs}(\bot) &:= \bot \\ \mathit{targs}(\psi_1 \wedge \psi_2) &:= \mathit{targs}(\psi_1) \cup \mathit{targs}(\psi_2) \\ \mathit{targs}(\psi_1 \to \psi_2) &:= \mathit{targs}(\psi_2) \\ \mathit{targs}(\forall x.\psi) &:= \mathit{targs}(\psi) \end{aligned}$$

The main proposition about proofs of atoms is the following:

Proposition 1. *If $\Delta \vdash \eta :_\eta R(\vec{t})$ then exactly one of the following is true:*

1. *There is some $X : \psi$ in Δ such that $\mathit{head}(\eta) = X$ and $R \in \mathit{targs}(\psi)$*
2. *$\eta = fe_{R(\vec{t})}(\eta')$ for some η' such that $\eta' : \bot$*

A similar proposition for $\bot$ can be stated as follows:

Proposition 2. *If $\Delta \vdash \eta :_\eta \bot$, then there is some $X : \psi$ in Δ such that $\mathit{head}(\eta) = X$ and $\bot \in \mathit{targs}(\psi)$*

These results allow us assume that all of our proof terms were obtained by a Prolog-style proof search:

- If we have a proof of $\psi_1 \wedge \psi_2$, we have a proof of both sides.
- If we have a proof of $\psi_1 \to \psi_2$, we have a proof of ψ_2 with ψ_2 as a premise.
- If we have a proof of $\forall x.\psi$, we have one of ψ.
- If we have a proof of $R(\bar{t})$, either there is an axiom with R as a target and we have a proof of all its premises, or there is a proof of $\bot$.
- If we have a proof of $\bot$, there is an axiom with $\bot$ as a target and we have a proof of all its premises.

3.3 Axioms for equality and arithmetic

In every derivation $\Delta \vdash N : \psi$ we use for our encoding, we assume that we implicitly have some axioms in Δ for equality and arithmetic. The full list of axioms is in the Appendix.

- For equality we have reflexivity, symmetry and transitivity, as well as congruence rules for some of the symbols of our signature.
- For arithmetic we have the first four Peano axioms.

- For inequality, we have a weak system, but it allows some basic reasoning about transitivity.

A short remark on the axioms for *pred*. In F* *pred* is polymorphic, but we do not have polymorphism in FOL. To simulate this, we suppose that in FOL we will have *pred* axioms of the form

$$\forall xyz.\mathit{HasType}(x,\mathit{nat}) \to \mathit{HasType}(y,\nu(t)) \to \mathit{HasType}(z,\nu(\mathit{nat}\to t)) \to \mathit{eq}(\mathit{pred}(0,y,z),y)$$
$$\forall xyz.\mathit{HasType}(x,\mathit{nat}) \to \mathit{HasType}(y,\nu(t)) \to \mathit{HasType}(z,\nu(\mathit{nat}\to t)) \to \mathit{eq}(\mathit{pred}(\mathit{succ}(x),y,z),z@x)$$

for every pred_t that we encounter during the encoding.

Our intuitionistic setting further restricted to just the $\wedge,\to,\forall$ connectives prevents us from adding some of the usual arithmetic axioms such as the inversion axiom:

$$\forall x.\mathit{HasType}(x,\mathit{Nat}) \to (\mathit{eq}(x,0) \vee \exists y.\mathit{eq}(x,\mathit{succ}(y))$$

In the future work section we discuss in more depth about adding classicality.

Moreover, we are currently missing the induction principle for naturals. It is worth recalling that induction is not a first-order axiom, so the alternative would have been be to add it as an axiom scheme, but that would have complicated our proofs.

4 The encoding

The full encoding is going to take a pF* judgement, either $\Gamma \vDash \varphi$ or $\Gamma \vdash e : t$, and it is going to produce a FOL judgement $\Delta \vdash \psi$ that needs to be proven. We will use the capital E to represent the encoding function.

The encoding of an expression is almost the identity, but to encode application we introduce a binary symbol @.

Definition 6 (Encoding of expressions)**.**

$$\begin{aligned} E(x) &:= x \\ E(e_1\ e_2) &:= @(E(e_1),E(e_2)) \\ E(0) &:= 0 \\ E(S\ e) &:= \mathit{succ}(E(e)) \\ E(\mathit{pred}(e,e_0,e_s)) &:= \mathit{pred}(E(e),E(e_0),E(e_s)) \end{aligned}$$

There are three main reasons we do not encode functions to function symbols and use the FOL application:

- Functions can be applied partially, e.g. $\mathit{map}\ (+1)\ l$.
- Functions can appear abstracted by a variable, e.g. $\forall f.\ \mathit{twice}@f@x = f@f@x$. Writing $\forall f.\ \mathit{twice}(f,x) = f(f(x))$ would not be correct since in that context f is just a variable. Notice that this is a restriction of first-order logic, since in higher-order, we could quantify over functions.
- Functions can be passed as arguments, as seen in the previous examples.

Now we will see the encoding of types. When encoding a type t, we introduce a fresh FOL variable $\nu(t)$, where ν is just an injective function that gives a name to a type. Additionally, we will recursively generate some equations describing the behaviour of t and the types that syntactically form part of t, if it is an arrow or a refinement. The set theoretic notation is just to ensure that we do not generate the same axiom twice (for example in the case of $t' \to t'$).

Definition 7 (Equations for types)**.**

$$\begin{aligned} E(nat) &:= \emptyset \\ E(t_1 \to t_2) &:= \\ &\quad \{\forall f. \mathit{HasType}(f, \nu(t_1 \to t_2)) \to (\forall x. \mathit{HasType}(x, \nu(t_1)) \to \mathit{HasType}(f@x, \nu(t_2))\} \\ &\quad \cup E(t_1) \cup E(t_2) \\ E(x : t\{\varphi\}) &:= \{\forall x. \mathit{HasType}(x, \nu(x : t\{\varphi\})) \leftrightarrow (\mathit{HasType}(x, \nu(t)) \wedge E(\varphi))\} \cup E(t) \end{aligned}$$

These equations try to give some meanings to the types:

- If a term f has type $t_1 \to t_2$ then for all x of type t_1 the application $f@x$ has type t_2.
- A term d has a refinement type $x : t\{\varphi\}$ if and only if it has type t and $\phi[d/x]$ is valid. Notice that the substitution that we have at the F* level, for example in the (V-Ref) rule, is going to be automatically performed when the quantifier is instantiated, since the variable x in φ is going to be bound by the quantifier and then replaced with the term for which the for all is instantiated.

In a similar manner as the types, encoding a function is going to generate a FOL constant of sort *Term* to represent the function, which we can assume to be the same as in pF* and an equation to represent the behaviour of the function under application. We also encode a type guard for the function itself:

Definition 8 (Encoding of function definitions)**.**

$$\begin{aligned} E(f(x_1 : t_1) \dots (x_n : t_n) : t = e) := \\ \forall x_1 \dots x_n.\, \mathit{HasType}(x_1, \nu(t_1)) \to \dots \to \mathit{HasType}(x_n, \nu(t_n)) \to \mathit{eq}(f@x_1@\dots@x_n, E(e)) \\ \cup E(f : t_1 \to \dots \to t_n \to t) \end{aligned}$$

The encoding of a function definition simulates the β-reduction in the target logic, by using equality. The function is type guarded, which means that it can only be applied if the arguments have the correct type.

Now, encoding an environment Γ is just a matter of encoding its elements one by one:

Definition 9 (Encoding of environments)**.**

$$\begin{aligned} E(\emptyset) &:= \emptyset \\ E(x : t; \Gamma) &:= E(x : t) \cup E(\Gamma) \\ E(f(x_1 : t_1) \dots (x_n : t_n) : t = e; \Gamma) &:= E(f(x_1 : t_1) \dots (x_n : t_n) : t = e) \cup E(\Gamma) \end{aligned}$$

The encoding of formulae is almost the identity, except in the case of the universal quantifier, in which we also encode a typing guard.

Definition 10 (Encoding of formulae)**.**

$$\begin{aligned} E(e_1 = e_2) &:= \mathit{eq}(E(e_1), E(e_2)) \\ E(e_1 < e_2) &:= \mathit{lt}(E(e_1), E(e_2)) \\ E(\varphi_1 \to \varphi_2) &:= E(\varphi_1) \to E(\varphi_2) \\ E(\varphi_1 \wedge \varphi_2) &:= E(\varphi_1) \wedge E(\varphi_2) \\ E(\forall (x : t).\ \varphi) &:= \forall x.\ E(x : t) \to E(\varphi) \end{aligned}$$

Finally, with all the definitions in place, we can define the encoding of a $p\mathrm{F}^\star$ judgement, which is a FOL judgement that needs to be proven. The encoding of a validity judgement is just the encoding of both sides of the $\vDash$ by separate. In the encoding of a typing judgement, we also need to add to the left the encoding of the type; while on the right we will have a *HasType* predicate:

Definition 11 (Encoding of judgements)**.**

$$\begin{aligned} E(\Gamma \vDash \varphi) &:= E(\Gamma) \vdash E(\varphi) \\ E(\Gamma \vdash e : t) &:= E(\Gamma) \cup E(t) \vdash \mathit{HasType}(E(e), \nu(t)). \end{aligned}$$

5 Correctness Proof

We will take a proof-theoretic approach, that is, we will show that any proof in FOL can be translated back into a proof in $p\mathrm{F}^\star$. This is interesting from a practical point of view, since it opens the possibility of performing proof reconstruction, and users would no longer have to trust the SMT solver.

In this section we will present a high-level view of the proof. We will state the theorems and some of the helping lemmas that we use, and we will give an idea about why they are needed and how they are proved. More details can be found in the Appendix.

The two main theorems correspond to the soundness of the validity and the typing judgements in $p\mathrm{F}^\star$. First we see the one for validity:

Theorem 1 (Soundness for the validity judgement)**.** *Let Γ be an F* environment and φ an F* formula such that $\Gamma \vdash$ wf and $\Gamma \vdash \varphi$ wf. If there exists a proof term P so that $E(\Gamma) \vdash P : E(\varphi)$, then $\Gamma \vDash \varphi$.*

Whenever F* needs to prove a validity judgement, it will encode the environment and the formula to be proved, and it will call the SMT solver. In our case, we encode the environment Γ into $E(\Gamma)$ and the formula φ to $E(\varphi)$. We also require that Γ and ϕ are well-formed, which we use to recover some typing conditions.

In the proof we will restrict ourselves to η-proof-terms, which we know to always exist. This lets us assume that the last step was an introduction of the top connective and that the premises of that

introduction are also proved by an η-proof-term. Then, we can do a simple induction up to the base cases, which we have to prove separately. In fact, most of the proof effort went into proving these base cases.

The base cases for a formula are equality, inequality and absurdity. We have the following lemmas for these:

Lemma 2. *Let Γ be a pF* environment, e_1 and e_2 be pF* expressions such that $\Gamma \vdash wf$ and $\Gamma \vdash e_1 = e_2$ wf. If $E(\Gamma) \vdash E(e_1 = e_2)$ then $\Gamma \vdash e_1 = e_2$*

Lemma 3. *Let Γ be a pF* environment, e_1 and e_2 be pF* expressions such that $\Gamma \vdash wf$ and $\Gamma \vdash e_1 < e_2$ wf. If $E(\Gamma) \vdash E(e_1 < e_2)$ then $\Gamma \vdash e_1 < e_2$*

Lemma 4. *Let Γ be a well-formed pF* environment ($\Gamma \vdash wf$). If $E(\Gamma) \vdash \bot$ then $\Gamma \vdash \bot$*

In order to prove those lemmas we use Proposition 1. For instance, the encoding of $E(e_1 = e_2)$ is $eq(E(e_1), E(e_2)$, and any η-proof term consists of a series of projections and applications on an axiom X that has eq among its targets. This allows us to perform an induction on the size of the η-proof terms. We know the shape of the axioms we can have on the left-hand size of a sequent:

- Some of the axioms of equality, inequality or arithmetic.
- The axiom for reducing applications of a function.
- The axiom for reducing applications of *pred*

We prove inductively that for all of these cases we can map back the proof to pF*.

A useful device for mapping back proofs from FOL to pF* is the fact that the encoding is a bijection at the expression level, that is, we can define a decoding D such that the pair E,D form a bijection between the pF* expressions and the FOL terms of sort *Term*.

We present some examples of how FOL proofs are mapped to pF*. Consider for instance the case of application of a function. In this case, the axiom X is:

$$\forall x_1 \dots x_n.\ E(x_1 : t_1) \to \dots \to E(x_n : t_n) \to eq(f @ x_1 @ \dots @ x_n, E(b))$$

and an η-proof term would be $X\ d_1 \dots d_n\ \eta_1 \dots \eta_n$, where each η_i is a proof term of $E(d_i, t_i)$.

By the bijection, there exists some F* expressions $D(d_1) \dots D(d_n)$ that are encoded to $d_1 \dots d_n$, and we need to prove that

$$\Gamma \vDash f\ D(d_1) \dots D(d_n) = b[D(d_1)/x_1] \dots [D(d_n)/x_n].$$

By well-formedness, both sides have the same type, and by the (V-RedEq) rule we have the result:

$$\dfrac{\dfrac{wf\ hyp}{\Gamma \vdash b[D(c_1)/x_1] \dots [D(c_n)/x_n] : t} \qquad \dfrac{\dfrac{wf\ hyp}{\Gamma \vdash fD(a_1) \dots D(a_1) : t} \quad \begin{array}{c}\Gamma \vdash D(a_1) : t_1 \\ \dots \\ \Gamma \vdash D(a_n) : t_n\end{array} \quad f(D(a_1) : t_1) \dots (D(a_n) : t_n) : t' = b \in \Gamma}{\Gamma \vdash fD(a_1) \dots D(a_n) \rightsquigarrow b[D(a_1)/x_1] \dots [D(a_1)/x_n]}\ \textsf{(R-Beta)}}{\Gamma \vDash fD(a_1) \dots D(a_1) = b[D(a_1)/x_1] \dots [D(a_n)/x_n]}\ \textsf{(V-RedEq)}$$

Another case worthy of note is the case of transitivity of the equality. Our induction hypothesis, is quite strong, and in this case it is not trivial to do the inductive step. Formally, we have the following axiom:

$$\forall xyz.\ eq(x,y) \to eq(y,z) \to eq(x,z)$$

If we are proving $eq(E(e_1), E(e_2)$, the proof term is going to look like $X\ E(e_1)\ y\ E(e_2)\ \eta_1\ \eta_2$, where η_1 and η_2 are respectively proofs of $eq(E(e_1), y)$ and $eq(y, E(e_2))$. But in order to use the induction hypothesis we need to prove that: (1) y is the image of a translation $E(e_3)$ and (2) $e_1 = e_3$ is well-formed, otherwise the proof does not go through.

One important insight is that the encoding is "well-behaved" at the term level, meaning that if we have a FOL proof of an equality or inequality where we only know one of the two terms, it can be mapped back to an equality or inequality in F* that is well-typed. This is due to needing to prove type guards in order to instantiate axioms about equality or inequality, on which we will comment on later. But for now we will state the lemma:

Lemma 5. *For all Γ, e, t such that $\Gamma \vdash wf$ and $\Gamma \vdash e : t$, if there is a FOL term d and a proof term Q such that $E(\Gamma) \vdash Q : eq(E(e), d)$ or $E(\Gamma) \vdash Q' : eq(d, E(e))$ then $\Gamma \vdash e = D(d)$ wf (and $\Gamma \vdash D(d) = e$ wf).*

A similar lemma can also be proven for inequality. With these two, the tricky cases such as transitivity, are covered, since at least one of the sides will always be typed.

Finally, we will also mention the case of type refinements. When we have a refinement type $x{:}t\{\varphi\}$, our encoding generates an axiom:

$$\forall x.\mathit{HasType}(x, \nu(x : t\{\varphi\})) \to (\mathit{HasType}(x, \nu(t)) \land E(\varphi))$$

If we have a proof of $\mathit{HasType}(d, \nu(x : t\{\varphi\})$ for some d, we can use it to instantiate the axiom and get a proof of $E(\varphi)[d/x]$. However, that proof is not an η proof, but rather a δ proof (unless $E(\varphi)[d/x]$ is atomic). So we cannot directly the induction hypothesis to get a proof of $\varphi[D(d)/x]$, but we do get a proof of $d : (x{:}t\{\varphi\})$, so by the (V-Ref") rule, we can instantiate $\varphi[D(d)/x]$, and proceed from there. Another workaround would have been to define an appropriate metric on η and δ proof terms that would let us do the induction on both.

The theorem of invertibility of the typing judgement is even more interesting.

Theorem 2. *For all Γ, t, d if $\Gamma \vdash wf$, $\Gamma \vdash t$ wf, $E(\Gamma), E(t) \vdash \mathit{HasType}(d, \nu(t))$ then $\Gamma \vdash D(d) : t$.*

This almost immediately gives us soundness for types, since the encoding is reversible at the term level. Additionally, it justifies the use of the type guards, since every term that satisfies them in FOL is going to have the right type at the F* level.

Theorem 3. *Let Γ be a pF* environment, e an F* expression and t a pF* type such that $\Gamma \vdash wf$, $\Gamma \vdash t$ wf and $fv(e) \subseteq dom(\Gamma)$. If $E(\Gamma), E(t) \vdash E(e : t)$, then $\Gamma \vdash e : t$.*

Finally, we will briefly comment on how we deal with explosion (ηE$\bot$). We have to take into account that every judgement of an atomic predicate $E(\Gamma) \vdash R(\bar{t})$ can also be proved from $E(\Gamma) \vdash \bot$, rather than by a series of eliminations from the axioms. To deal with those cases, we need to reconstruct those proofs in F* by using the explosion principle. Fortunately, we have few predicates in our signature, so the case analysis is simple enough:

- In the case of *eq*, if we prove $E(\Gamma) \vdash eq(d_1, d_2)$ by explosion, then $E(\Gamma) \vdash \bot$ and $\Gamma \vDash \bot$, so by the (V-Expl) rule we can prove $\Gamma \vdash D(d_1, d_2)$ (assuming well-formedness).

- The case of *lt* is the same.
- In the case of *HasType*, if we prove $E(\Gamma) \vdash \mathit{HasType}(d_1, \nu(t))$ by explosion, then $\Gamma \vDash \mathit{bot}$, but we still need to give e some type t', and then prove by (Sub-Expl) that $t' <: t$. Since all the constants in d are encoded F* variables then all variables of $D(d)$ can be typed, and by applying (Sub-Expl) enough times we can give $D(d)$ a well-formed type.

6 Related Work

Translations from HOL to FOL have been implemented in the past (Meng and Paulson, 2008; Benzmüller et al., 2015; Brown, 2013; Blanchette et al., 2016b). In particular, Sledgehammer (Paulson and Blanchette, 2010; Blanchette et al., 2013a) allows proving Isabelle/HOL proof goals by translation to FOL provers and SMT solvers. Similar tools have been build for HOL Light and HOL4 (Kaliszyk and Urban, 2014) as well as for Mizar (Kaliszyk and Urban, 2015). However, our use case is different in F*, since we target efficient, scalable, and reproducible SMT solver behavior and the soundness of the encoding, while hammers often do not aim for reproducibility and attempt to reconstruct a proof term after the fact to recover soundness. Extending hammers to dependent types is a topic of ongoing research (Czajka and Kaliszyk, 2016; Czajka, 2016).

There have been some efforts in proving the soundness of hammer encodings. A proof of soundness of an encoding between Pure Type Systems and minimal FOL can be found in (Czajka, 2016). Their proof uses η-long normal forms as well. However, our target logic is more expressive, having conjunction and, above all, explosion. Encoding from multisorted to unsorted FOL has also been proved sound with a monotonicity argument in (Blanchette and Popescu, 2013). This work is extended in (Blanchette et al., 2013b) to encode polymorphic FOL into monomorphic FOL.

Counterexample finders such as Nitpick (Blanchette and Nipkow, 2010) and Nunchaku (Cruanes et al., 2016) rely on different kinds of translations from HOL to FOL, which are sound for counterexamples not sound for proofs. Extending counterexample finders to dependent types is also a topic of ongoing research (Cruanes and Blanchette, 2016).

F*'s bounded unrolling of recursive and inductive definitions was influenced by previous work in Dafny on "computing with an SMT solver" (Amin et al., 2014). Such fuel bounds are also used in counterexample finders like Nitpick (Blanchette and Nipkow, 2010) and Nunchaku (Cruanes et al., 2016).

Why3 (Filliâtre and Paskevich, 2013) encodes higher-order functions using lambda lifting (Clochard et al., 2014) and provides a Coq tactic called `why3` that can revert the translation from Why3 to Coq. It also encodes polymorphic FOL into many-sorted FOL (Bobot and Paskevich, 2011).

Lean (Selsam and de Moura, 2016; de Moura et al., 2015; Lewis and de Moura, 2016) is an ongoing effort to implement better automation for type theory. Encoding F* into Lean is the subject of ongoing experiments.

7 Next Steps

There are many ways to continue this work; we outline the most important ones below.

7.1 Targetting classical FOL

A crucial extension for the future will be making the target FOL be classical, since SMT solvers work in classical logic. One way to do it would be to move to a richer calculus that allows to express classicality, such as the $\lambda\mu$ calculus (Parigot, 1992; Saurin, 2010).

Another way we could try to approach this is using a double negation translation. It is a well known result that a formula has a classical proof if and only if its negative translation has an intuitionistic proof (Glivenko, 1929; Brown and Rizkallah, 2014). This would allow us to still translate only intuitionistic proofs for doubly negated formulas and then use double negation elimination in F* to obtain a proof of the original formula. This method would also allow us to encode $\vee$ and $\exists$ in base to the other connectives.

7.2 Extending the axiomatisation of arithmetic

Our axiomatisation of arithmetic in FOL is considerably weak. At the very least, we would need to add an induction axiom scheme, and figure out how to adapt the proof. The standard way of adding this axiom scheme is to produce an axiom for every formula ψ with free variables $\vec{y}$:

$$\forall \vec{y}.\, \psi[0, \vec{y}] \to (\forall x. \mathit{HasType}(x, \mathit{Nat}) \to \psi[x, \vec{y}] \to \psi[\mathit{succ}(x), \vec{y}]) \to (\forall x. \mathit{HasType}(x, \mathit{Nat}) \to \psi[x, \vec{y}])$$

One of the issues with this is that ψ can be a formula that is not the image of a translation and therefore, cannot be mapped back to F*. So we would probably need some restrictions on the form of ψ to be able to preserve soundness, and then we would have to justify why these restrictions do not affect the proving power of the target logic.

7.3 Growing the source language

A first possible extension to pF* would be adding dependent arrow types, i.e. $x{:}t_1 \to t_2$. Extending our current formalisation to dependent arrow types should be an achievable but somewhat tedious task. The main reason is that types would have free variables that we would have to deal with. For instance, an arrow type $x{:}t_1 \to y{:}t_2\{\varphi\}$ where $x \in \mathit{fv}(\varphi)$. When encoding the type on the right we would need to have some information about what x is. The way this is done in actual F* is by allowing applications between a type and a term. Roughly, the encoding would be:

$$\forall f. \mathit{HasType}(f, \nu(x{:}t_1 \to y{:}t_2\{\varphi\})) \to \forall x. \mathit{HasType}(x, t_1) \to \mathit{HasType}(f@x, \nu(y{:}t_2\{\varphi\})@x)$$
$$\forall yx. \mathit{HasType}(y, \nu(y{:}t_2\{\varphi\})@x) \to \mathit{HasType}(y, \nu(t_2)) \wedge E(\varphi)$$

When dealing with more complex function types, one would have to keep track of which variables need to be passed to the type on the right-hand side and in which order, as well as what variables are free in the type and need to be expected as arguments. Essentially, this resembles some kind of lambda-lifting at the type level.

7.4 Justifying aggressive optimisations such as removing type guards using SMT patterns

Additionally, there are some optimisations that we are trying to implement in the actual F* encoding that would also require a formal proof. The main one is the removal of the type guards, which improves the performance of the SMT solver. This consists in encoding a function $f(x_1 : t_1) : t_2 = e$ just as:

$$\forall x. \mathit{eq}(f@x, e)$$

We believe that, since the quantifier is annotated with a pattern $\mathit{HasType}(x_1, \nu(t_1))$, it should only get instantiated with a term of type t_1. Proving this sound will make good use of out proof theoretic argument, where quantifier patterns are easy to express (as opposed to model theory).

Previous work on hammers was focused on removing type guards using monotonicity (Claessen et al., 2011; Blanchette et al., 2013b; Blanchette and Krauss, 2011; Blanchette, 2012; Blanchette and Popescu, 2013; Blanchette et al., 2016a) when moving from sorter to unsorted FOL. Our problem is only superficially similar, since inductive types in F* are not monotonous and thus we cannot apply this optimisation. Using SMT triggers for removing useless type guards is instead more similar to one of the Boogie translations (Leino and Rümmer, 2010, §3.1), which uses SMT triggers to prevent ill-typed instantiations in conjunction with type arguments. As far as we know, no formal justification was ever given for this Boogie translation.

7.5 Proof Reconstruction

There has already been some work on recovering Isabelle proofs from Z3 natural deduction derivations (Böhme and Weber, 2010). A very ambitious continuation of our work would be to try to perform proof reconstruction in F*. This is a rather challenging task, since F* still has to typecheck the generated proof term and if it needs to prove the validity of some formula, it will call the SMT solver again, and thus never bottom out. So not only we would have to figure out how to parse and translate the Z3 derivations, but also some language redesign would be needed. The reward, however, would be greatly valuable, since it would remove the need to trust the SMT solver any longer.

7.6 Mechanising formalisation in Coq or F*

Finally, we hope that both our current formalisation and the future extensions can be mechanised in Coq or F*. In fact, we already have an extensive Coq formalisation of pF* and of its encoding to FOL, which we wrote during the first weeks of the internship, when we was trying to do a model-theoretical proof, and most of the definitions there could be reused.

Larger F* subsets have been previously formalised in F* (Swamy et al., 2016) for the purpose of proving subject reduction, which implies the correctness of the weakest precondition calculus. This work, however, simply assumes the correctness of the logical encoding.

8 Conclusion and summary

We have shown how to formalise and prove the soundness of the encoding of a small fragment of F* into a first-order intuitionistic logic. The main simplification with respect to reality is that the logic of SMT solvers is classical, since they are based on a SAT solver.

Choosing a proof-theoretic approach has turned out to be the right decision. Once an appropriate normal form was chosen, the formalisation of the encoding and the proof of soundness were much more natural. Our method has, however, some shortcomings. It is not completely clear how to extend it to classical logic, and this remains as future work. But the main advantage is that it shows a direct connection between proofs in FOL and proofs in pF* that could be exploited in the future to perform proof reconstruction.

Other potential next steps are extending both the source and the target languages in order to cover as big a subset of F* as possible, and mechanising the proof in Coq to have a greater guarantee of correctness.

Appendix

$p\mathbf{F}^\star$ rules

Reduction relation

The right arrow $\rightsquigarrow$ denotes the reduction relation. We use $\rightsquigarrow^*$ to denote its transitive and reflexive closure.

$$\frac{f(x_1 : t_1) \ldots (x_n : t_n) : t = e \ \in \ \Gamma \qquad \Gamma \vdash e_1 : t_1 \qquad \ldots \qquad \Gamma \vdash e_n : t_n}{\Gamma \vdash f e_1 \ldots e_n \rightsquigarrow e[e_1/x_1] \ldots [e_n/x_n]} \ \textsf{(R-Beta)}$$

$$\Gamma \vdash \mathit{pred}(0, e_0, e_S) \rightsquigarrow e_0 \ \textsf{(R-Pred0)} \quad \Gamma \vdash \mathit{pred}(Se, e_0, e_S) \rightsquigarrow e_s e \ \textsf{(R-Pred0)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow^* e_2 \qquad \Gamma \vdash e_3 \rightsquigarrow^* e_4}{\Gamma \vdash e_1 e_2 \rightsquigarrow^* e_3 e_4} \ \textsf{(R-App)} \quad \frac{\Gamma \vdash e_1 \rightsquigarrow^* e_2}{\Gamma \vdash Se_1 \rightsquigarrow^* Se_2} \ \textsf{(R-Succ)}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e'}{\Gamma \vdash \mathit{pred}(e, e_0, e_S) \rightsquigarrow \mathit{pred}(e', e_0, e_s)} \ \textsf{(R-Pred)}$$

Validity judgement

$$\frac{\Gamma \vdash e \ : \ x : t\{\varphi\}}{\Gamma \vDash \varphi[e/x]} \ \textsf{(V-Ref'')}$$

$$\frac{\Gamma \vdash v_1 : t_1 \qquad \Gamma \vdash v_2 : t_2 \qquad v_1 \ll v_2}{\Gamma \vDash v_1 < v_2} \ \textsf{(V-PrecedesI)} \quad \text{where } v_1 \ll v_2 = \begin{cases} n_1 < n_2 \text{ if } v_1 = n_1, v_2 = n_2 \\ \bot \text{ otherwise} \end{cases}$$

$$\frac{\Gamma \vDash e_1 < e_2 \qquad \Gamma \vDash e_2 < e_3}{\Gamma \vDash e_1 < e_3} \ \textsf{(V-PredecesTrans)} \quad \frac{\Gamma \vDash \varphi_1 \qquad \Gamma \vDash \varphi_2}{\Gamma \vDash \varphi_1 \wedge \varphi_2} \ \textsf{(V-AndI)} \quad \frac{\Gamma \vDash \varphi_1 \wedge \varphi_2}{\Gamma \vDash \varphi_i} \ \textsf{(V-AndE}_i\textsf{)}$$

$$\frac{\Gamma, \_ : \mathit{nat}\{\varphi_1\} \vDash \varphi_2}{\Gamma \vDash \varphi_1 \to \varphi_2} \ \textsf{(V-ImpI)} \quad \frac{\Gamma \vDash \varphi_1 \to \varphi_2 \qquad \Gamma \vDash \varphi_1}{\Gamma \vDash \varphi_2} \ \textsf{(V-ImpE)}$$

$$\frac{\Gamma, x : t \vDash \varphi}{\Gamma \vDash \forall (x : t).\varphi} \ \textsf{(V-ForallI)} \quad \frac{\Gamma \vDash \bot \qquad \Gamma \vdash \varphi \ \mathit{wf}}{\Gamma \vDash \varphi} \ \textsf{(V-FalseI)}$$

$$\frac{\Gamma \vDash \forall (x : t).\varphi \qquad \Gamma \vdash e : t}{\Gamma \vDash \varphi[e/x]} \ \textsf{(V-ForallE)} \quad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t \qquad \Gamma \vdash e \rightsquigarrow e'}{\Gamma \vDash e = e'} \ \textsf{(V-RedEq)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vDash e = e} \ \textsf{(V-ReflEq)} \quad \frac{\Gamma \vDash e_1 = e_2 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vDash \varphi[e_1/x]}{\Gamma \vDash \varphi[e_2/x]} \ \textsf{(V-SubstEq)}$$

$$\frac{\Gamma \vDash Se_1 = Se_2}{\Gamma \vDash e_1 = e_2} \ \textsf{(V-SuccInj)} \quad \frac{\Gamma \vDash e_1 < e_2}{\Gamma \vDash Se_1 < Se_2} \ \textsf{(V-SuccMon)} \quad \frac{\Gamma \vdash e : \mathit{nat} \qquad \Gamma \vDash e < 0}{\Gamma \vDash \bot} \ \textsf{(V-FalseLt)}$$

$$\frac{\Gamma, \_ : \mathit{nat}\{\varphi_1\} \vDash \varphi_2 \qquad \Gamma, \_ : \mathit{nat}\{\varphi_1 \to \bot\} \vDash \varphi_2}{\Gamma \vDash \varphi_2} \ \textsf{(V-ExMiddle)}$$

Typing judgement

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ \textsf{(T-Var)}\quad \frac{f(x_1 : t_1) \ldots (x_n : t_n) : t = e\ \in\ \Gamma}{\Gamma \vdash f : t_1 \to \ldots \to t_n \to t}\ \textsf{(T-Fun)}$$

$$\frac{\Gamma \vdash e : t_1 \to t_2 \qquad \Gamma \vdash e' : t_1}{\Gamma \vdash e\ e' : t_2}\ \textsf{(T-App)}\quad \Gamma \vdash 0 : nat\ \textsf{(T-Zero)}$$

$$\frac{\Gamma \vdash e : nat}{\Gamma \vdash Se : nat}\ \textsf{(T-Succ)}\quad \frac{\Gamma \vdash e : t \qquad \Gamma \vDash \varphi[e/x]}{\Gamma \vdash e\ :\ x : t\{\varphi\}}\ \textsf{(T-Ref)}\quad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash t <: t'}{\Gamma \vdash e : t'}\ \textsf{(T-Sub)}$$

$$\frac{\Gamma \vdash e : nat \qquad \Gamma \vdash e_0 : nat \qquad \Gamma \vdash e_S : nat \to nat}{\Gamma \vdash pred(e, e_0, e_S) : nat}\ \textsf{(T-Pred)}$$

Subtyping

$$\Gamma \vdash nat <: nat\ \textsf{(Sub-Nat)}\quad \frac{\Gamma \vdash t_3 <: t_1 \qquad t_2 <: t_4}{\Gamma \vdash t_1 \to t_2 <: t_3 \to t_4}\ \textsf{(Sub-Fun)}$$

$$\frac{\Gamma \vdash t <: t'}{\Gamma \vdash x : t\{\varphi\} <: t'}\ \textsf{(Sub-Ref-Left)}\quad \frac{\Gamma \vdash t <: t' \qquad \Gamma, x : t \vdash \varphi}{\Gamma \vdash t <: x : t'\{\varphi\}}\ \textsf{(Sub-Ref-Right)}$$

$$\frac{\Gamma \vDash \bot \qquad \Gamma \vdash t_1\ wf \qquad \Gamma \vdash t_2\ wf}{\Gamma \vdash t_1 <: t_2}\ \textsf{(Sub-Expl)}$$

Formula well-formedness

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash t\ wf}{\Gamma \vdash e_1 = e_2\ wf}\ \textsf{(WF-eq)}\quad \frac{\Gamma \vdash e_1 : nat \qquad \Gamma \vdash e_2 : nat}{\Gamma \vdash e_1 < e_2\ wf}\ \textsf{(WF-lt)}$$

$$\frac{\Gamma \vdash \varphi_1\ wf \qquad \Gamma \vdash \varphi_2\ wf}{\Gamma \vdash \varphi_1 \wedge \varphi_2\ wf}\ \textsf{(WF-and)}\quad \frac{\Gamma \vdash \varphi_1\ wf \qquad \Gamma \vdash \varphi_2\ wf}{\Gamma \vdash \varphi_1 \to \varphi_2\ wf}\ \textsf{(WF-imp)}$$

$$\frac{\Gamma, x : t \vdash \varphi\ wf \qquad \Gamma \vdash t\ wf}{\Gamma \vdash \forall (x : t).\varphi}\ \textsf{(WF-forall)}$$

Type well-formedness

$$\Gamma \vdash nat\ wf\ \textsf{(WF-Nat)}\quad \frac{\Gamma \vdash t_1\ wf \qquad \Gamma \vdash t_2\ wf}{\Gamma \vdash t_1 \to t_2\ wf}\ \textsf{(WF-arr)}\quad \frac{\Gamma \vdash t\ wf \qquad \Gamma, x : t \vdash \varphi\ wf}{\Gamma \vdash x : t\{\varphi\}}\ \textsf{(WF-ref)}$$

Environment well-formedness

$$\frac{\Gamma \vdash t\ wf \qquad x \notin \Gamma}{\Gamma, x : t \vdash wf}\ \textsf{(WF-EVar)}$$

$$\frac{\Gamma \vdash t_1\ wf \ldots \Gamma \vdash t_n\ wf \qquad \Gamma \vdash t\ wf \qquad \Gamma, e_1 : t_1 \ldots e_n : t_n \vdash e\ wf \qquad f \notin \Gamma}{\Gamma, f(x_1 : t_1) \ldots (x_n : t_n) : t = e \vdash wf}\ \textsf{(WF-EFun)}$$

Axioms for the first-order theories

Axioms for equality

Equality is reflexive, symmetric and transitive. We also add some axioms for congruence.

$$\forall x.eq(x, x)$$
$$\forall xy.eq(x, y) \to eq(y, x)$$
$$\forall xyz.eq(x, y) \to eq(y, z) \to eq(x, z)$$
$$\forall fgxy.eq(f, g) \to eq(x, y) \to eq(f@x, g@y)$$
$$\forall xyt.eq(x, y) \to HasType(x, t) \to HasType(y, t)$$
$$\forall x'y'xy.eq(x, x') \to eq(y, y') \to lt(x', y') \to lt(x, y)$$

Axioms for arithmetic

$$HasType(0, Nat)$$
$$\forall x.HasType(x, Nat) \to HasType(succ(x), Nat)$$
$$\forall x.HasType(x, Nat) \to \neg eq(succ(x), 0)$$
$$\forall xy.HasType(x, Nat) \to HasType(y, Nat) \to eq(succ(x), succ(y)) \to eq(x, y)$$

Axioms for inequality

$$\forall x.HasType(x, Nat) \to lt(x, 0) \to \bot$$
$$\forall x.HasType(x, Nat) \to lt(x, succ(x))$$
$$\forall xyz.lt(x, y) \to lt(y, z) \to lt(x, z)$$

Axioms for predecessor

For every function type of the form $nat \to t$

$$\forall xyz.HasType(x, nat) \to HasType(y, \nu(t)) \to HasType(z, \nu(nat \to t)) \to eq(pred(0, y, z), y)$$
$$\forall xyz.HasType(x, nat) \to HasType(y, \nu(t)) \to HasType(z, \nu(nat \to t)) \to eq(pred(succ(x), y, z), z@x)$$
$$\forall xx'yz.eq(x, x') \to eq(pred(x, y, z), pred(x', y, x))$$

Proofs of the results

Definition 12 (Decoding)**.** *Given a FOL term of sort Term, we define the following function inductively:*

- $D(x) := x$
- $D(a_1@a_2) := D(a_1)D(a_2)$
- $D(0) := 0$
- $D(succ(a)) := SD(a)$
- $D(pred(a_1, a_2, a_3) := pred(D(a_1), D(a_2), D(a_3))$

Lemma 6. *The maps E and D form a bijection from $pF^\star$ expressions to FOL terms of sort Term.*

Lemma 7 (Substitution under the encoding)**.** *Consider an environment Γ, $F^\star$ terms e_1 and e_2 such that $\Gamma \vdash wf$. Then, for all variable x free in e_1, $E(e_1[e_2/x]) = E(e_1)[E(e_2)/x]$.*

Proof. By induction on e_1:

- If $e_1 = x$, $E(x[e_2/x]) = E(e_2) = E(x)[E(e_2)/x]$
- If $e_1 = e_{11}e_{12}$. $E((e_{11}e_{12})[e_2/x]) = @(E(e_{11}[e_2/x]), E(e_{12}[e_2/x]))$ and by the induction hypothesis that is equal to $@(E(e_{11})[E(e_2)/x], E(e_{12})[E(e_2)/x]) = @(E(e_{11}), E(e_{12}))[E(e_2)/x] = E(e_{11}e_{12})[E(e_2)/x]$
- If $e_1 = Se_{11}$, or $e_1 = pred(e_{11}, e_{10}, e_{1S})$, the reasoning is similar.

□

From here on, ALL the lemmas and theorems are mutually inductive. However, well-foundedness is maintained. To see this, it suffices to observe that the proof terms always get smaller.

The first lemma gives us some notion of "well-behaving" of the encoding, namely, that an encoded term can only be proven equal to another encoded term.

Lemma 8. *For all Γ, e, t such that $\Gamma \vdash wf$, $\Gamma \vdash e : t$, if there is a FOL term d such that $E(\Gamma) \vdash Q : eq(E(e), d)$ or $E(\Gamma) \vdash Q' : eq(d, E(e))$ then $\Gamma \vdash e = D(d)$ wf (or $\Gamma \vdash D(d) = e$ wf).*

Proof. By induction on the η-proof term η. If the proof is by explosion, then $\Gamma \vDash \bot$, and we can give both e and $D(d)$ the same type, for instance *nat*. Otherwise, we do case analysis on $X = head(\eta)$.

- $X : \forall x.HasType(x, \nu(x : u\{\varphi\})) \to (HasType(x, \nu(u)) \wedge E(\varphi))$, when there is a refinement type $x : u\{\varphi\}$ appearing in Γ and $eq \in targets(E(\varphi))$. Then, the proof term η contains $pr_2(Xd'\eta_1)$, which is a proof term of $E(\varphi)[d'/x]$. In particular, $\eta_1 : HasType(d', \nu(x : u\{\varphi\}))$ and by Theorem 5, we will have that $\Gamma \vdash D(d') : x : u\{\varphi\}$, and therefore, $\Gamma \vDash \varphi[D(d')/x]$. Also, since $\Gamma \vdash wf$, $\Gamma \vDash \varphi[D(d')/x]$ *wf*. Notice that $eq(D(d), e)$ is a subformula of $\varphi[D(d')/x]$, and that is proven by a series of eliminations. Then, we just have to prove that every elimination preserves well-formedness. The only interesting case is the forall elimination. Since forall clauses $\forall(x : u).\varphi'$ are translated with a guard $\forall x.HasType(x, \nu(v)) \to E(\varphi')$, to eliminate them we need to give a term d'' and a proof of $HasType(d'', \nu(v))$. By Theorem 5, $\Gamma \vdash D(d'') : u$, if $\Gamma \vdash \forall(x : u).\varphi'$ *wf*, then $\Gamma \vdash \varphi'[D(d'')/x]$ *wf*.
- $X : \forall x.eq(x, x)$. Then $D(d) = e$.

- $X : \forall x_1 \dots x_n . E(x_1 : t_1) \to \dots \to E(x_n : t_n) \to eq(f@x_n@\dots@x_n, E(b))$. Then, there is a function $f(x_1 : t_1) \dots (x_n : t_n) : t$ in Γ. There are two cases:
  - $E(e)$ is on the left. Then, there are some FOL terms $a_1 \dots a_n$ such that the proof term is of the form $X a_1 \dots a_n M_1 \dots M_n$, $E(e) = f@a_1@\dots@a_n$, and $d = E(b)[a_1/x_1] \dots [a_n/x_n]$. Then, $D(d) = b[D(a_1)/x_1] \dots [D(a_n)/x_n]$, and $e = f D(a_1) \dots D(a_n)$. By Theorem 2, we have that $\Gamma \vdash D(a_1) : t_1, \dots, \Gamma \vdash D(a_n) : t_n$. Then, by the app rule, $\Gamma \vdash e : t$, and by well-formedness, also $\Gamma \vdash b[D(a_1)/x_1] \dots [D(a_n)/x_n]$
  - E(e) is on the right. Then, there are some FOL terms $a_1 \dots a_n$ such that the proof term is of the form $X a_1 \dots a_n M_1 \dots M_n$, $E(e) = E(b)[a_1/x_1] \dots [a_n/x_n]$, $d = f@a_1@\dots@a_n$. Then, $D(d) = f D(a_1) \dots D(a_n)$, and $e = b[D(a_1)/x_1] \dots [D(a_n)/x_n]$. The reasoning is the same as in the other case.
- $X : \forall xy . HasType(x, Nat) \to HasType(y, Nat) \to eq(succ(x), succ(y)) \to eq(x, y)$. There are two symmetric cases, so w.l.o.g. let's suppose we have a proof of $eq(E(e), d)$. Then, the proof term looks like $X E(e) d \eta_1 \eta_2 \eta_3$. By soundness of the typing encoding we have that $\Gamma \vdash e : nat$ and $\Gamma \vdash D(d) : nat$.
- The rest of the cases should be straightforward. Notice that in all of them we have either *HasType* or an *eq* as one of the premises, and therefore we can just use induction or Theorem 5.

□

Proposition 3. *For all* Γ, e_1, e_2, *if* $\Gamma \vdash e_1 = e_2$ *wf and* $E(\Gamma) \vdash Q : eq(E(e_1), E(e_2))$, *then* $\Gamma \vDash e_1 = e_2$.

Proof. We do it by induction on the proof term. If the proof is by explosion, then $\Gamma \vdash \bot$, and so $\Gamma \vdash e_1 = e_2$. Otherwise, case analysis on X:

- $X : \forall x . eq(x, x)$. Then, $E(e_1)$ and $E(e_2)$ are the same FOL term, and by injectivity of the encoding e_1 and e_2 are the same $p\mathrm{F}^\star$ term. By well-formedness and the (V-ReflEq) rule, we have the result.
- $X : \forall x_1 \dots x_n . E(x_1 : t_1) \to \dots \to E(x_n : t_n) \to eq(f@x_1@\dots@x_n, E(b))$. Then, there exist some FOL terms $a_1 \dots a_n$ such that $E(e_1) = f@a_1@\dots@a_n$, so e_1 is $f D(a_1) \dots D(a_n)$ On the other hand, $E(e_2) = E(b)[a_1/x_1] \dots [a_n/x_n]$ and therefore we have $e_2 = b[D(a_1)/x_1] \dots [D(a_n)/x_n]$. To prove equality:

$$\dfrac{\dfrac{wf\ hyp}{\Gamma \vdash f D(a_1) \dots D(a_1) : t} \qquad \dfrac{\begin{array}{c}\Gamma \vdash D(a_1) : t_1 \\ \dots \\ \Gamma \vdash D(a_n) : t_n\end{array} \quad f(D(a_1) : t_1) \dots (D(a_n) : t_n) : t' = b \ \in \ \Gamma}{\Gamma \vdash f D(a_1) \dots D(a_n) \rightsquigarrow b[D(a_1)/x_1] \dots [D(a_1)/x_n]}}{\Gamma \vDash f D(a_1) \dots D(a_1) = b[D(a_1)/x_1] \dots [D(a_n)/x_n]}$$

  where we get the typing conditions from the well-typedness hypothesis.
- $X : \forall xy . eq(x, y) \to eq(y, x)$. Then the proof term is of the form $X E(e_2) E(e_1) M$, where M is a proof of $eq(E(e_2), E(e_1))$, and by I.H. $\Gamma \vDash e_2 = e_1$. By well-formedness and application of the subst rule:

$$\frac{\dfrac{I.H}{\Gamma \vDash e_2 = e_1} \qquad \dfrac{wf + refl}{\Gamma \vDash e_1 = e_1}}{\Gamma \vDash e_1 = e_2}\ \textsf{(V-SubstE)}$$

- $X : \forall xyz.eq(x,y) \to eq(y,z) \to eq(x,z)$. The proof term is of the form $XE(e_1)dE(e_2)\eta_1\eta_2$, where η_1 is a proof of $eq(E(e_1), d)$. By Lemma 8, $\Gamma \vdash e_1 = D(d)$ *wf*. By the I.H., $\Gamma \vDash e_1 = D(d)$ and $\Gamma \vDash D(d) = e_2$. By the (V-SubstE) rule, we get $\Gamma \vDash e_1 = e_2$.

- $X : \forall fgxy.eq(f,g) \to eq(x,y) \to eq(f@x, g@y)$. Then, the proof term is of the shape $XE(e_{11})E(e_{21})E(d_{12})E(d_{22})$ where $e_1 = e_{11}e_{12}$ and $e_2 = e_{21}e_{22}$. By the I.H., we have proofs of $\Gamma \vDash e_{11} = e_{21}$ and $\Gamma \vDash e_{12} = e_{22}$. By a chain of substitutions, $\Gamma \vDash e_{11}e_{12} = e_{11}e_{12}$, $\Gamma \vDash e_{11}e_{12} = e_{21}e_{12}$, and $\Gamma \vDash e_{11}e_{12} = e_{11}e_{22}$.

- $X : \forall xy.HasType(x, Nat) \to HasType(y, Nat) \to eq(succ(x), succ(y)) \to eq(x,y)$. Then the proof term is of the form $XE(e_1)E(e_2)\eta_1\eta_2\eta_3$, where $\eta_3 : eq(succ(E(e_1)), succ(E(e_2)))$, and by the I.H. $\Gamma \vdash Se_1 = Se_2$.

- $X : \forall yz.HasType(y, \nu(t)) \to HasType(z, \nu(nat \to t)) \to eq(pred(0,y,z), y)$. Then, the proof term is of the form $XE(e_2)d$ and $e_1 = pred(0, e_2, D(d))$. By the (R-Pred0) rule, we get that $e_1 \rightsquigarrow e_2$, and since $\Gamma \vdash e_1 = e_2$ *wf*, by (V-RedEq) we have that $\Gamma \vDash e_1 = e_2$.

- $X : \forall xyz.HasType(x, Nat) \to HasType(y, \nu(t)) \to HasType(z, \nu(nat \to t)) \to eq(pred(succ(x), y, z), z@x)$. Then, the proof term is of the form $Xd_1d_2d_3\eta_1$, $E(e_1) = pred(succ(d_1), d_2, d_3)$ and $E(e_2) = d_3@d_1$. The reasoning is the same as in the previous case, but with the (R-PredS) rule.

- $X : \forall xx'yz.eq(x,x') \to eq(pred(x,y,z), pred(x',y,z)$. Then, $e_1 = pred(e_{11}, e_{10}, e_{1S})$, $e_2 = pred(e_{21}, e_{20}, e_{2S})$, and by I.H., $\Gamma \vDash e_{11} = e_{21}$, so by substitution $\Gamma \vDash pred(e_{11}, e_{10}, e_{1S}) = pred(e_{21}, e_{20}, e_{2S})$.

- $\forall x.HasType(x, \nu(t)) \to (HasType(x, \nu(t_1)) \wedge E(\varphi))$ for some refinement type $t = x : t_1\{\varphi\}$ such that $eq \in targs(E(\varphi))$. In particular we have a proof $pr_2(Xd\eta_1)$ of $E(\varphi)$, and there exists a proof term $P[pr_2(Xd\eta_1)]$ of $eq(E(e_1), E(e_2))$. By Theorem 5, $\Gamma \vdash D(d) : x : t\{\varphi\}$, so by the (V-Ref") rule, we have $\Gamma \vDash \varphi[D(d)/x]$. Then, the same proof can be simulated in $p\mathrm{F}^\star$.

  - If $\varphi[D(d)/x] = eq(E(e_1), E(e_2))$, we already have a proof.
  - If $\varphi[D(d)/x] = \varphi_1 \to \varphi_2$, it will get encoded to $E(\varphi_1) \to E(\varphi_2)$. Since the proof is by elimination, necessarily we will have a proof term η_2 of $E(\varphi_1)$, which by Theorem 4 gives us a proof of $\Gamma \vDash \varphi_1$, and $pr_2(Xd\eta_1)\eta_2$ is a proof term of $E(\varphi_2)$. In $p\mathrm{F}^\star$ this corresponds to the implication elimination.
  - If $\varphi[D(d)/x] = \varphi_1 \wedge \varphi_2$, it will get encoded to $E(\varphi_1) \wedge E(\varphi_2)$. Since the proof is by elimination, necessarily we will have a projection on one side. In $p\mathrm{F}^\star$ this corresponds to the conjunction elimination on the same side.
  - If $\varphi[D(d)/x] = \forall(x : t').\varphi_1$, it will get encoded to $\forall x.HasType(x, \nu(t)) \to E(\varphi_1))$. Since the proof is by elimination, necessarily we will have a term d' and proof term η_2 of $HasType(d', \nu(t)$, which by Theorem 5 gives us a proof of $\Gamma \vdash D(d) : t$ In $p\mathrm{F}^\star$ this corresponds to the for-all elimination.

□

We now prove state similar lemmas for the less-than judgement

Lemma 9. *For all Γ, e, such that $\Gamma \vdash wf$, $\Gamma \vdash e : nat$, if there is a FOL term d such that $E(\Gamma) \vdash Q : lt(E(e), d)$ (or $E(\Gamma) \vdash Q' : lt(d, E(e))$) then $\Gamma \vdash e < D(d)$ wf (or $\Gamma \vdash D(d) < e$ wf).*

Proof. Similar to Lemma 8 □

Proposition 4. *For all Γ, e_1, e_2, if $\Gamma \vdash e_1 < e_2$ wf and $E(\Gamma) \vdash Q : lt(E(e_1), E(e_1))$, then $\Gamma \vDash e_1 < e_2$.*

Proof. Similar to Proposition 3 □

Now the main lemma about proofs of $\bot$:

Lemma 10. *For all Γ, $\Gamma \vdash wf$, if $E(\Gamma) \vdash Q : \bot$, then $\Gamma \vDash \bot$*

Proof. Case analysis on X:

- $X : \forall x.HasType(x, Nat) \to eq(0, succ(x)) \to \bot$. Then, the proof term is of the form $Xd\eta_1\eta_2$, where $\eta_1 : HasType(d, Nat)$. By Theorem 5, there is an e with $\Gamma \vdash e$, and $E(e) = d$. Then, $\Gamma \vDash 0 = Se$. To prove $\Gamma \vDash 0 < 0$, we just prove that $\Gamma \vDash 0 < Se$ with (V-PrecedesI) and that $\Gamma \vDash 0 = 0$ by (V-ReflEq), and then apply the substitution rule. Then, from $\Gamma \vDash 0 < 0$ we can apply (V-FalseLt) to get $\Gamma \vDash \bot$.
- $X : \forall x.HasType(x, Nat) \to lt(x, 0) \to \bot$. Then, the proof term is of the form $Xd\eta_1$, where $\eta_1 : lt(d, 0)$. By Lemma 9, $\Gamma \vDash D(d) < 0$, and by Theorem 5 $\Gamma \vdash D(d) : nat$ so by (V-FalseLt) $\Gamma \vDash \bot$.
- If we instantiate a refinement, the reasoning is similar as in the other lemmas for the same case.

□

We now proceed to prove the main theorems.

Theorem 4 (Soundness for entailment)**.** *For all Γ, φ, if $\Gamma \vdash \varphi$ wf and $E(\Gamma) \vdash Q : E(\varphi)$, then $\Gamma \vDash \varphi$.*

Proof. By induction on the η-proof term.
We consider possible forms of φ.

- If $\varphi = (e_1 = e_2)$, $\varphi = (e_1 < e_2)$ or $\varphi = \bot$, the preceding lemmas give us the result.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $E(\varphi) = E(\varphi_1) \wedge E(\varphi_2)$, the proof term is $\langle \eta_1, \eta_2 \rangle$, $E(\Gamma) \vdash \eta_1 : E(\varphi_1)$ and $E(\Gamma) \vdash \eta_2 : E(\varphi_2)$. By the I.H., $\Gamma \vDash \varphi_1$ and $\Gamma \vDash \varphi_2$, and by the conjunction intro rule, $\Gamma \vDash \varphi_1 \wedge \varphi_2$.
- If $\varphi = \varphi_1 \to \varphi_2$, then $E(\varphi) = E(\varphi_1) \to E(\varphi_2)$, the proof term is $\lambda X : E(\varphi_1).\eta$, and $E(\Gamma), X : E(\varphi_1) \vdash E(\varphi_2)$. Let n be a fresh variable not in φ_1. Then, by weakening, $E(\Gamma), X : E(\varphi_1), HasType(n, nat) \vdash E(\varphi_2)$, and by I.H. $\Gamma, n : nat\{\varphi_1\} \vDash \varphi_2$, and by the implication intro rule, $\Gamma \vDash \varphi_1 \to \varphi_2$.
- If $\varphi = \forall(x : t).\varphi_1$, then $E(\varphi) = \forall x.E(x : t) \to E(\varphi_1)$, the proof term is $\lambda x.\lambda \eta : E(x : t).\eta$, and $E(\Gamma), \eta : E(x : t) \vdash E(\varphi_1)$. By I.H. $\Gamma, x : t \vDash \varphi_1$, and by the for-all intro rule, $\Gamma \vDash \forall(x : t).\varphi_1$.

□

Now the lemma of invertibility of the typing judgement:

Theorem 5 (Invertibility for typing)**.** *For all Γ, t, d if $\Gamma \vdash wf$, $\Gamma \vdash t$ wf, $E(\Gamma) \cup E(t) \vdash HasType(d, \nu(t))$ then $\Gamma \vdash D(d) : t$.*

Proof. If there is a proof by explosion, then $\Gamma \vDash \bot$, and then we can give any type to $D(d)$, in particular t. Otherwise, there is an η-proof term $\eta :_\eta \mathit{HasType}(d, \nu(t))$ with $\mathit{head}(\eta) = X$ and $\mathit{HasType} \in \mathit{targets}(X)$. The proof is on the size of η. We consider possible cases for X:

- X is in the typing context (i.e., a translation for some binding $x : t$) . Then, the proof is by axiom and $D(x)$ is x.
- $X : \forall f.\mathit{HasType}(f, \nu(t_1 \to t_2)) \to (\forall x.\mathit{HasType}(x, \nu(t_1)) \to \mathit{HasType}(f@x, \nu(t_2)))$. Then, the proof term is of the form $X d_1 \eta_1 d_2 \eta_2$, for some d_1,d_2, $\eta_1 : \mathit{HasType}(d_1, \nu(t_1 \to t_2))$, and $\eta_2 : \mathit{HasType}(d_2, \nu(t_1))$, and then $d = d_1@d_2$ and $t = t_2$. By the I.H. $\Gamma \vdash D(d_1) : t_1 \to t_2$, and $\Gamma \vdash D(d_2) : t_1$, so by the (T-App) rule $\Gamma \vdash D(d_1)D(d_2) : t_2$.
- $X : \forall x.\mathit{HasType}(x, \nu(x : t\{\varphi\})) \to (\mathit{HasType}(x, \nu(t)) \wedge E(\varphi))$. Then the proof term is of the form $pr_1(X d \eta)$, for some d, where $\eta : \mathit{HasType}(d, \nu(x : t\{\varphi\}))$. By I.H., $\Gamma \vdash D(d) : x : t\{\varphi\}$, and by the subtyping rule, we get $\Gamma \vdash D(d) : t$.
- $X : \forall x.(\mathit{HasType}(x, \nu(u)) \wedge E(\varphi)) \to (\mathit{HasType}(x, \nu(x : u\{\varphi\})))$. Then, the proof term is of the form $X d \eta$, where $\eta : (\mathit{HasType}(d, \nu(u)) \wedge E(\varphi)[d/x])$. Then, clearly $pr_1(\eta) : \mathit{HasType}(d, \nu(u))$ and $pr_2(\eta) : E(\varphi)[d/x]$. By I.H., $\Gamma \vdash D(d) : u$. Also, since $\Gamma \vdash x : u\{\varphi\}$ *wf*, so $\Gamma, x : u \vdash \varphi$ *wf*, and $\Gamma \vdash \varphi[D(d)/x]$ *wf*. We then apply Theorem 4 with $pr_2(\eta)$, and we get $\Gamma \vdash \varphi[D(d)/x]$. By the (T-Ref) rule, $\Gamma \vdash D(d) : x : u\{\varphi\}$.
- $X : \mathit{HasType}(0, \mathit{Nat})$. Then, $d = 0$, $D(d) = 0$ and trivially $\Gamma \vdash 0 : \mathit{nat}$.
- $X : \forall xyt.\mathit{eq}(x, y) \to HT(x, t) \to HT(y, t)$. Then the proof term is of the form $X d' d \nu(t) \eta_1 \eta_2 . \eta_2 : \mathit{HasType}(d', \nu(t))$ and by I.H. $\Gamma \vdash D(d') : t$. Also by Lemma 8, $\Gamma \vdash D(d') = D(d)$ *wf* and by well-formedness, $\Gamma \vdash e : t$

□

A remark on the induction hypothesis. We have a premise of the form $E(\Gamma) \cup E(t) \vdash \mathit{HasType}(d, \nu(t))$. That is, the type on the right of the $\vdash$ and on its left must be the same. However, when we apply the induction hypothesis, we have still t on the left but a different type t' on the right. Since types are encoded recursively, in some of the cases $E(t) \subseteq E(t')$, so if $E(\Gamma) \cup E(t) \vdash \mathit{HasType}(d, \nu(t'))$, then also $E(\Gamma), E(t') \vdash \mathit{HasType}(d, \nu(t'))$. The only case where this does not happen is fourth one, where we have $E(\Gamma) \cup E(\nu(x : u\{\varphi\})) \vdash \mathit{HasType}(d, \nu(u))$. But the fact that we have that axiom means that the type $x : u\{\varphi\}$ already appeared in Γ, and therefore, $E(\Gamma) \cup E(x : u\{\varphi\}) = E(\Gamma) \cup E(u) = E(\Gamma)$, and we can apply the induction hypothesis.

After the previous result, the next follows easily:

Theorem 6 (Soundness for typing)**.** *For all* Γ*,* e*,* t*, if* $\mathit{fv}(e) \subseteq \mathit{dom}(\Gamma)$ *and* $E(\Gamma) \cup E(t) \vdash Q : E(e : t)$ *then* $\Gamma \vdash e : t$*.*

Proof. Just apply the previous theorem with $d = E(e)$.

□

References

N. Amin, K. R. M. Leino, and T. Rompf. Computing with an SMT solver. *TAP*, 2014.

C. Benzmüller, N. Sultana, L. C. Paulson, and F. Theiss. The higher-order prover Leo-II. *J. Autom. Reasoning*, 55(4):389–404, 2015.

J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2012.

J. C. Blanchette and C. Kaliszyk, editors. *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016*, volume 210 of *EPTCS*, 2016.

J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. *J. Autom. Reasoning*, 47(4):369–398, 2011.

J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *ITP*. 2010.

J. C. Blanchette and A. Popescu. Mechanizing the metatheory of Sledgehammer. *FroCoS*. 2013.

J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *JAR*, 51(1): 109–128, 2013a.

J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *TACAS*. 2013b.

J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. To appear in Logical Methods in Computer Science., 2016a.

J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016b.

F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, 2011.

S. Böhme and T. Weber. Fast lcf-style proof reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. 2010.

C. E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. *J. Autom. Reasoning*, 51(1):57–77, 2013.

C. E. Brown and C. Rizkallah. Glivenko and Kuroda for simple type theory. *J. Symb. Log.*, 79(2):485–495, 2014.

K. Claessen, A. Lilliestr̈om, and N. Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*. 2011.

M. Clochard, J. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. *VSTTE*. 2014.

S. Cruanes and J. C. Blanchette. Extending nunchaku to dependent type theory. In Blanchette and Kaliszyk (2016).

S. Cruanes, J. Blanchette, and A. Reynolds. Nunchaku: Flexible model finding for higher-order logic. Talk in Montpellier, 2016.

L. Czajka. A shallow embedding of pure type systems into first-order logic (preliminary version). Online Report, 2016.

L. Czajka and C. Kaliszyk. Goal translation for a hammer for coq (extended abstract). In Blanchette and Kaliszyk (2016).

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. *CADE*, 2015.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. *ESOP*. 2013.

V. Glivenko. Sur quelques points de la logique de m. brouwer. *Bulletins de la classe des sciences*, 15: 183–188, 1929.

C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 2010.

R. Lewis and L. de Moura. Automation and computation in the Lean theorem prover. Talk at HaTT, 2016.

J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *JAR*, 40(1):35–60, 2008.

M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*. 1992.

L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL*. 2010.

A. Saurin. Typing streams in the $\Lambda\mu$-calculus. *ACM Trans. Comput. Log.*, 11(4), 2010.

D. Selsam and L. de Moura. Congruence closure in intensional type theory. In N. Olivetti and A. Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*. 2016.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Z. Beguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.