Towards a Provably Correct Encoding from F⋆ to SMT
(Rough Diamond)

Alejandro Aguirre[1] Cătălin Hriţcu[2] Chantal Keller[3] Nikhil Swamy[4]

[1]IMDEA Software Institute [2]Inria Paris [3]LRI, Univ. Paris-Sud [4]Microsoft Research

Abstract F⋆ is a dialect of ML aimed at program verification that puts together the expressive power of a proof assistant based on dependent types with the automation of a deductive verification tool backed by an SMT solver. F⋆ features an expressive type system combining dependent types, refinement types, monadic effects, and a weakest-precondition calculus. The F⋆ type-checker generates verification conditions that are dependently typed and higher-order, encodes these verification conditions into first-order logic, and discharges them with the help of an SMT solver. This paper formalizes this encoding into first-order logic for a small fragment of F⋆ and proves its soundness using a proof-theoretic argument.

1 Introduction

F⋆ [1, 19] is a verification system for ML programs that allows a novel combination of manual and automatic proofs. This enables the scalable verification of functional correctness and security for more realistic applications, such as a reference implementation of the TLS protocol framework. The F⋆ type-system puts together dependent types, refinement types, monadic effects, and a weakest-precondition calculus. Verification conditions in the dependently-typed higher-order logic of F⋆ are encoded to SMT formulas and discharged automatically using Z3. However, currently there is no formal guarantee of the soundness of this complex encoding that now has to be trusted by the users.

Encodings from higher-order to first-order logic are usually proven sound using model-theoretic reasoning, i.e., by translating models of the target language into models of the source language. However, models for dependent type theory are complex and in the case of F⋆ a matter of ongoing research.

Contribution This extended abstract describes our formalization of the encoding of a fragment of F⋆ including refinement types into intuitionistic FOL with equality and arithmetic. We prove the soundness of this encoding using a proof-theoretic argument, by mapping FOL proof terms in normal form to derivations in the F⋆ fragment we consider.

Outline §2 introduces the encoding of F^* to SMT on a simple example. §3 gives a summary of our formal results. §4 presents related work and §5 discusses future steps. A more comprehensive technical report includes definitions and paper proofs: `http://prosecco.gforge.inria.fr/personal/hritcu/students/alejandro/report.pdf`

2 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)))))
```

Sort `Term` denotes an open data type (open to capture the ability to add new inductive types in F⋆) and `BoxInt` constructs a `Term` from an `Int`, with the corresponding destructor `BoxInt_proj`. F⋆ arithmetic operations are lifted to `Terms` from the SMT solver's ones.

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)))))
```

F⋆ can conclude that factorial terminates if the SMT solver can prove this negative formula unsatisfiable. 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.

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))))
```

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))))
```

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))))))))))
```

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

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

Finally, a 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))))
```

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. Suppose we have the following definition:

```
(lambda x. (lambda y. x + y) x) e
```

A lambda-lifted equivalent program consists of a list of closed functions definitions:

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

3 Summary of Formalization

We have formalised the SMT encoding for *p*F* (read "pico F*"), a small lambda-lifted fragment of F* with refinement types. Its syntax can be seen on Figure 1. The meta-variable t represents a type, that can be *nat*, the type of natural number; 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 mapping the free variables to their type, and a list of function definitions.

$$
\begin{aligned}
t &::= \mathit{nat} \mid x{:}t_1 \to t_2 \mid x : t\{\phi\} \\
\varphi &::= e_1{<}e_2 \mid e_1{=}e_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x{:}t.\phi \mid \bot \\
e &::= x \mid n \mid f \mid e_1\ e_2 \mid S\ e \mid \mathit{pred}\ e\ e_O\ e_S
\end{aligned}
$$

Figure 1. Syntax of *p*F*

*p*F* has 6 mutually inductive typing judgments: (1) Formula validity ($\Gamma \vDash \varphi$); (2) Expression typing ($\Gamma \vdash e : t$); (3) Subtyping ($\Gamma \vdash t_1 <: t_2$); (4) Well-formedness of an environment ($\Gamma \vdash \mathit{wf}$); (5) Well-formedness of a formula ($\Gamma \vdash \varphi\ \mathit{wf}$); (6) Well–formedness of a type ($\Gamma \vdash \tau\ \mathit{wf}$). The target of our encoding is an intuitionistic sorted first-order logic with equality and arithmetic. The logical encoding E maps *p*F* formulas to first-order formulas and *p*F* expressions to first-order terms. The main theorems we proved (on paper) is soundness for the formula validity and expression typing judgment:

Theorem 1 *Let Γ be an pF* environment and φ an pF* 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 $\Gamma \vDash \varphi$.*

Theorem 2 *Let Γ be a pF* environment, e an pF* 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 $\Gamma \vdash e : t$.*

The proof relies on the property of the target logic that every proof has an equivalent η-long normal form. Proofs of atomic predicates in this normal form consist of a series of eliminations performed on an axiom in the context, which can be translated back to a similar derivation in the logic of *p*F*. More details can be found on the technical report.

4 Related Work

Translations from HOL to FOL have been implemented in the past [2, 5, 7, 16]. In particular, Sledgehammer [3, 18] allows proving Isabelle/HOL proof goals by translation to FOL provers and SMT solvers. Similar hammer tools have been build for HOL Light and HOL4 [14] as well as for Mizar [15]. 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, with a Coq hammer proposed by Czajka and Kaliszyk in recent independent work [11].

There have also been some efforts in proving the soundness of hammer encodings. A recent proof targets the soundness of an encoding between Pure Type Systems and minimal FOL [10]. While our use of η-long normal forms was inspired by this proof, our intuitionistic target logic is more expressive that minimal logic. Encoding from multisorted to unsorted FOL has also been proved sound with a monotonicity argument in [6]. This work is extended in [4] to encode polymorphic FOL into monomorphic FOL.

Why3 [13] encodes higher-order functions using lambda lifting [9] and provides a Coq tactic called `why3` that can revert this translation.

Lean [12] is an ongoing effort to implement SMT-style automation directly for type theory. Encoding F* into Lean is the subject of ongoing experiments.

5 Next Steps

There are several natural continuations to our work. The main current limitation is that the target of our formalization is an intuitionistic logic while SMT solvers are classical. To remove this limitation we could either use some variant of the double negation translation from classical to intuitionistic logic [8], or move to classical logic with its proof terms and normal forms, such as the $\lambda\mu$ calculus [17].

Secondly, it would be interesting to extend the source language to include, for instance, dependent types, recursive functions, and other features of F* [1, 19]. In addition, it would also be interesting to verify some optimizations of the F* SMT encoding. For example, F* uses SMT patterns to guide the instantiaton of quantifiers in a way that ensures that instantiation is only attempted with a term of the appropriate type. This allows type guards to be potentially removed without losing soundness. Proving this sound would make good use of our proof theoretic argument, where quantifier patterns are easy to express, as opposed to model theory.

Proof reconstruction could also be an interesting problem to look at. Some SMT solvers can return a derivation in natural deduction style. Since our proof gives an algorithm for turning FOL derivations into F* ones, we could try to generate F* terms that prove the verification conditions. Then, the SMT solver would no longer have to be trusted since we could actually recheck the generated proofs directly in F*.

Finally, we hope that both our current proof and the future extensions can be mechanised in a theorem prover like Coq or F*. In fact, we already have a Coq definition of pF* and of its encoding to FOL, which could be used as a base for mechanizing the soundness proof.

Bibliography

[1] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

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

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

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

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

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

[7] C. E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. *JAR*, 51(1):57–77, 2013.

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

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

[10] L. Czajka. A shallow embedding of pure type systems into first-order logic. Online Report, 2016.

[11] L. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. Submitted to JAR, 2017.

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

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

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

[15] C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *JAR*, 55(3):245–256, 2015.

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

[17] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. *LPAR*. 1992.

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

[19] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.