From F* to SMT (Extended Abstract)

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

[1]Inria Paris [2]Université Paris Diderot (Paris 7) [3]LRI, Université Paris-Sud
[4]Microsoft Research, Redmond

1 Introducing F*

F* (Swamy et al., 2016) 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. F* puts together full dependent types, monadic effects, refinement types, 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. In addition, F* provides some of the power and expressiveness of a proof assistant like Coq, which enable users to prove arbitrarily complex properties manually. In this talk we will explain the main ideas behind F*'s SMT encoding (§2) and present ongoing work on formalizing it for a core subset of F* (§3). We conclude with related work (§4).

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 combines (1) 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.

The following F* function computing a factorial illustrates some of this:

```
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)))))
```

Submitted to:
HaTT 2016

The pattern HasType @x nat is bound to the quantifier by the reserved symbol bang (!), and allows the SMT solver to instantiate the forall whenever 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 as a formula of the form

```
(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 always 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 Terms 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 idealized 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 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 axiomatized 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 not interpreted.

```
(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:

$$\begin{aligned}
t &::= \mathit{nat} \mid x{:}t_1 \to t_2 \mid x : t\{\phi\} \\
\phi &::= e_1{<}e_2 \mid e_1{=}e_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x{:}t.\phi \\
n &::= O \mid S\ n \\
v &::= n \mid \lambda x{:}t.e \mid \mathit{let\ rec}\,(f^d{:}t)\ x = e \\
e,d &::= x \mid v \mid e_1\ e_2 \mid S\ e \mid \mathit{pred}\ e\ e_O\ e_S
\end{aligned}$$

Figure 1: Syntax of the fragment

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

The F⋆ encoding also uses lambda lifting for eliminating first-class functions. 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)))))
```

3 Ongoing formalization

We are currently formalizing in Coq the SMT encoding for *p*F⋆ (read "pico F⋆"), a fragment of F⋆ that includes dependent function types, refinement types, fixpoints with semantic termination check, a well-founded ordering on naturals and subtyping. Its syntax can be seen on Figure 1.

An expression e in this language can be a variable x, a value v, an expression e_1 applied to another e_2 (written $e_1\ e_2$), a successor of an expression e (written $S\ e$), or the destructor $\mathit{pred}\ e\ e_0\ e_S$, that reduces to e_0 or $e_S\ n$ depending on whether the first argument reduces to O or to $S\ n$. A value v is either a natural number, which is constructed from O and successor S, a lambda abstraction or a recursive definition $\mathit{let\ rec}\,(f^d{:}t)\ x = e$, where d is a ranking function ensuring its termination. A type t is nat, a dependent function type $x{:}t_1 \to t_2$, or a type t refined by a logical formula ϕ (written $x : t\{\phi\}$).

In this fragment we have typing judgments of the form: (1) $\Gamma \vdash e : t$, where Γ is a typing environment relating variables to types, e is an expression, and t is a type; (2) formula well-formedness $\Gamma \vdash \phi$; and (3) a logical validity judgment of the form $\Gamma \models \phi$.

For the formalization we have embedded into Coq the syntax of *p*F⋆ and FOL with arithmetic, as well as the Tarski semantics $[\![\cdot]\!]$ of FOL with arithmetic that maps SMT formulas to `Prop`. The logical encoding $[\cdot]$_ maps *p*F⋆ formulas to SMT formulas and *p*F⋆ expressions to SMT terms.

The main property we are in the process of proving is the following:

If $\Gamma \vdash \phi$ and $[\![[\Gamma]]\!] \to [\![[\phi]_\Gamma]\!]$ then $\Gamma \models \phi$.

A particularly interesting case for ϕ is the one of equality ($e_1{=}e_2$), for which we need to prove:

If $\Gamma \vdash e_1 : t$ and $\Gamma \vdash e_2 : t$ and $[\![[\Gamma]]\!] \to [\![[e_1]_\Gamma = [e_2]_\Gamma]\!]$ then $\Gamma \models e_1 = e_2$.

4 Related work

Translations from HOL to FOL have been implemented and formalized in the past (Meng and Paulson, 2008; Benzmüller et al., 2015; Brown, 2013; Blanchette et al., 2016). In particular, Sledgehammer (Paulson and Blanchette, 2010; Blanchette et al., 2013) 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, since we target reproducible SMT behavior and the soundness of the encoding, while hammers do not aim for reproducibility or soundness for the FOL encoding 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).

Counterexample finders such as Nitpick and Nunchaku rely on different kinds of translations from HOL to FOL (Blanchette and Nipkow, 2010; Cruanes et al., 2016). Extending counterexample finders to dependent types is also a topic of ongoing research (Cruanes and Blanchette, 2016; Blanchette, 2015).

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).

Why3 (Filliâtre and Paskevich, 2013) encodes higher-order functions using lambda lifting (Clochard et al., 2014). Moreover, it provides a Coq tactic called `why3` that can revert the translation from Why3 to Coq. Lean (de Moura and Selsam, 2016; de Moura et al., 2015; Lewis and de Moura, 2016) is an ongoing effort to implement better automation for type theory.

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. Toward Nitpick and Sledgehammer for Coq. Talk at Coq Workshop, 2015.

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

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

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

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 type theory. To appear at HaTT, 2016.

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

L. Czajka and C. Kaliszyk. Components of a hammer for Coq: Goal translation and proof reconstruction. To appear at HaTT, 2016.

L. de Moura and D. Selsam. Congruence closure in intensional type theory. To appear at IJCAR, 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*. Mar. 2013.

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.

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

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

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

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. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. Jan. 2016.