Full Dependency and User-defined Effects in F⋆

Proposal for a talk plus a demo

Nikhil Swamy[1] Cătălin Hriţcu[2] Chantal Keller[1,3] Pierre-Yves Strub[4]
Aseem Rastogi[5] Antoine Delignat-Lavaud[2] Karthikeyan Bhargavan[2] Cédric Fournet[1,3]

[1]Microsoft Research [2]Inria [3]MSR-Inria [4]IMDEA Software Institute [5]UMD

Abstract

We present (new) F⋆, a dependently typed, higher-order, strict, effectful language, bringing formal verification to the ML language family. We illustrate the dual uses of F⋆ both as a proof assistant as well as a tool for building and verifying effectful programs. The latest version of F⋆ is programmed in F⋆, bootstraps in both OCaml and F# on most major platforms, and is open source. We encourage interested ML programmers to gradually migrate critical portions of their programs to F⋆ for verification, while easily and safely interoperating with their existing ML libraries.

F⋆ reloaded

Since its initial development in 2010 we had about five years of experience programming in F⋆ (Swamy et al. 2013a), gaining insight into parts of the language that worked well and others that didn't. As we aimed to push F⋆ further towards certifying larger pieces of software, we found our old design lacking in various ways (outlined below). This prompted us to develop new F⋆, a fresh take on the language, this time with full dependent types and user-extensible monadic effects. At ML 2015, we propose to demo this new redesigned F⋆ language, showcasing two of its complementary usage modes: F⋆ as a proof assistant, and F⋆ as a general-purpose verification-oriented dialect of ML.

Henceforth, we write F⋆ for the new F⋆ language and implementation, and write old F⋆ for F⋆ circa 2010–2014.

Value-dependent types and old F⋆

The main distinctive feature of old F⋆ was its *value-dependent refinement type system*, a middle ground between the mainstream variants of the ML type system, and the vastly more powerful dependent type theory underlying systems like Coq and Agda. The main typing construct in the language is the refinement type $x{:}t\{\phi\}$, a type inhabited only by those elements of t that also satisfy the predicate ϕ, e.g. $x{:}\mathsf{int}\{x \geq 0\}$ is the type of non-negative integers. Given a program e and purported type for it $x{:}t\{\phi\}$, the type-checker seeks to prove that e has type t and furthermore that ϕ holds for every evaluation of e, e.g., that e returns non-negative integers. Backed by an SMT solver for good automation, old F⋆'s type-checker (and several other systems that also provide value-dependent types, e.g., Backes et al. (2014); Eigner and Maffei (2013); Lourenço and Caires (2015); Rondon et al. (2008); Vazou et al. (2014)) could be used to statically check a variety of program properties.

To enable refinement predicates ϕ to be checked mostly automatically, various restrictions are usually placed on their form. While ϕ may refer to program terms (e.g., we may write $x{:}\mathsf{int}\{x > y\}$, where y is some program variable in scope), allowing constructs like $x{:}\mathsf{int}\{\mathsf{failwith}\ \texttt{"fixme"};\ \mathsf{true}\}$ is problematic: how should one interpret effects like exceptions, IO, or even non-termination within logical formulae? Sidestepping such difficulties, all the refinement type systems mentioned so far restrict ϕ to only contain *values* from the programming language (hence the name value-dependent). As such, potentially effectful code creeping into the logical fragment of the language is ruled out by syntactic fiat. Additionally, refinement formulae contain interpreted function symbols in some logic (e.g., > in the theory of integer arithmentic); and other uninterpreted function symbols.

We have used old F⋆ in several non-trivial verification efforts (Bhargavan et al. 2013; Fournet et al. 2013; Strub et al. 2012; Swamy et al. 2013b). Independently, other researchers have used their value-dependent refinement type systems with good success. The conceptual simplicity of value-dependent types is a significant selling point: for not much work, one can significantly boost the expressiveness of the type system. However, as we aim to move F⋆ forwards to the certification of larger pieces of code, programming and verifying large systems with value dependency alone is too cumbersome. We highlight three main shortcomings here.

An axiom- and annotation-heavy programming style. Consider writing a type for a sorted list of integers: one would like to write $x{:}\mathsf{list\ int}\{\mathsf{sorted}\ x\}$, for some well-defined total function sorted. In a value-dependent system, one must adopt the following style, introducing sorted as an uninterpreted function in the logic, and then providing (error-prone) axioms for it.

```
logic function sorted : list int → bool
assume Nil_S : sorted []
assume Sing_S: ∀i. sorted [i]
assume Cons_S: ∀i j tl. sorted (i::j::tl) = i ≤ j ∧ sorted (j::tl)
```

Of course, should one actually want to implement a function to test whether a list was sorted, one would need to write a program sorted_f and give it the type $l{:}\mathsf{list\ int} \to b{:}\mathsf{bool}\{b{=}\mathsf{sorted}\ l\}$, effectively writing the program twice and then proving a relation between the two, which may in turn require further annotations to go through. This style is tedious and, unfortunately, pervasive in existing developments; for instance it accounts for thousands of lines of specifications in the proof of miTLS (Bhargavan et al. 2013). Such duplication is no longer necessary in F⋆.

No fallback when the SMT solver fails. Automated proving via an SMT solver is key to the success of F⋆—without it, even small developments would be too tedious. Still, relying on the SMT solver as the only way to complete a proof can be frustrating. Particularly when trying to prove complex properties involving induction, quantifiers, or non-linear arithmetic, SMT solvers can be unpredictable, or even hopeless. In such cases, we need finer control—in the limit, being able to supply a manually-constructed proof term, and to receive assistance from the tool to build such a term. Constructive proofs built semi-interactively are now feasible in F⋆: one of our larger developments to date is a formalization of the metatheory of System F_ω in F⋆, with the formalization of a subset of F⋆ itself underway.

Limited support for reasoning about effects. Refinement types are great for stating invariants, e.g., ref $(x{:}\mathsf{int}\{x \geq 0\})$ is a convenient way of enforcing that a reference cell is always non-negative. However, refinement types are usually inapplicable when trying to prove

non-monotonic properties about mutable state, e.g., proving that a reference is incremented. In the absence of this, despite having support for effects, verification efforts in old F* often resorted to writing code in a purely functional style that often rendered the code inefficient.

Full dependency with user-defined effects

F* now provides (1) a core dependently typed logic of normalizing terms, expressive enough to do proofs by well-founded induction; (2) embedded within an effectful language, with the capability of writing precise functional correctness specification; (3) with as much automated proving as possible from an SMT solver; (4) packaged into a usable surface language, with good type inference; (5) easy interoperability with existing ML dialects (in particular, F# and OCaml), and (6) deployability on multiple platforms.

The central organizing principle of our design is a new type-and-effect system, which separates effectful code from a core logic of pure functions (rather than relying on the syntactic value restriction of previous value-dependent refinement type systems). To ensure the core language of pure functions is normalizing, we do semi-automatic semantic termination proofs based on well-founded relations.

Beyond pure code, F* supports reasoning about divergence, state, and exceptions via a weakest pre-condition calculus parameterized by the specific effect. Through user-defined effects, F* can be easily configured to also reason about other effect-like features. One new example involves the use of effects to encapsulate computationally irrelevant code in a Ghost monad; another involves reasoning about relational properties using a monad for pairs of computations.

F* is programmed in about 20,000 lines of F* itself, highlighting its practicality as a general-purpose programming language. In addition to the compiler, we have, to date, programmed and verified an additional 20,000 lines of code—ranging from simple canonical algorithms (e.g., quicksort) to a proof of type-soundness for System Fω. Also underway is an effort to build a high-performance implementation of TLS/SSL with certified security—we expect this as well as our mechanized formalization of F* in F* to be complete by the time of the ML workshop.

The F* compiler bootstraps into OCaml and F#, yielding binaries for all major platforms. We have an online tutorial and interface to our type-checker, all available from `https://fstar-lang.org`. An interactive mode for F* integrated in Atom editor is also available `https://github.com/FStarLang/fstar-interactive`.

A talk and a demo

We request a slot at the ML workshop to deliver first a 10-minute talk, providing a brief overview of the design of F*, followed by a 15-minute demo of F* in action.

Illustrating the use of F* *as a programming language*, we will show how to start with an OCaml program, port one module in it to F*, verify it, compile it back to OCaml link and run it with the rest of the original program. This example could be based on the verification of a canonical algorithm, such as quicksort, shown below.

```
val qsort: f:('a → 'a → Tot bool){total_order f}
        → l:list 'a
        → Tot (m:list 'a{sorted f m ∧ (∀ i. count i l = count i m)})
              (decreases (length l))
let rec qsort f = function
  | [] → []
  | pivot::tl → let hi, lo = partition (f pivot) tl in
                qsort f lo @ pivot::qsort f hi
```

The type states that if f is a total order on 'a and list:'a, then qsort f l is a total computation (signified by the effect Tot) returning a m:list 'a, a permutation of l sorted according to f. To prove that qsort terminates, we provide a measure decreases (length l) which suffices to convince F* that the recursion in qsort is well-founded.

To prove that qsort has this type, we rely on a library of lemmas proven about functions like List.partition and List.@. F* then generates verification conditions, and feeds them to an SMT solver (together with previous proven lemmas) and the SMT solver completes the proof.

In other cases, when the solver is unable to complete the proof, one can program proofs for the properties in question, explicating just those parts of the proof that are hard for an SMT solver. In the limit, one can provide the entire proof manually, defaulting to a style reminiscent of programming in other non-SMT-enabled dependently typed languages such as Coq or Agda.

Time-permitting, illustrating F* *as a proof assistant*, we plan to sketch a proof of syntactic type-safety for the simply typed lambda calculus, based on the development at `https://github.com/FStarLang/FStar/blob/master/examples/metatheory/stlc_strong_db_parsubst.fst`.

References

M. Backes, C. Hritcu, and M. Maffei. Union, intersection, and refinement types and reasoning about type disjointness for secure protocol implementations. *JCS*, 22(2):301–353, 2014.

K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. *S&P (Oakland)*, 2013.

F. Eigner and M. Maffei. Differential privacy by typing in security protocols. *CSF*, 2013.

C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully Abstract Compilation to JavaScript. *POPL*, 2013.

L. Lourenço and L. Caires. Dependent information flow types. *POPL*, 2015.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. *PLDI*, 2008.

P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. *POPL*, 2012.

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *JFP*, 23(4):402–451, 2013a.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying Higher-order Programs with the Dijkstra Monad. *PLDI*, 2013b.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for Haskell. *ICFP*, 2014.