Semantic Purity and Effects Reunited in F*

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 language for writing general-purpose programs with effects, specifying them within its functional core, and verifying them semi-automatically, by a combination of type inference, SMT solving, and manual proofs.

A central difficulty is to ensure the consistency of a core language of proofs within a larger language in which programs may exhibit effects such as state, exceptions, non-termination, IO, concurrency, etc. Prior attempts at solving this problem generally resort to a range of ad hoc methods. Instead, the main novelty of F* is to safely embrace and extend the familiar style of type-and-effect systems with fully dependent types.

F* is founded on a λ-calculus with a range of primitive effects and a dependent type system parameterized by a user-defined lattice of Dijkstra monads. For each monad, the user provides a predicate-transformer that captures its effective semantics. At the bottom of the lattice is a distinguished Pure monad, such that computations typeable as Pure are normalizing—our termination argument is based on well-founded relations and is fully semantic.

We illustrate our design on a series of challenging programming examples. We outline its metatheory on a core calculus, for which we prove soundness and termination of the Pure fragment. We also discuss selected aspects of its fresh typechecker. The F* system is open source; it fully supports our new design; it generates F# and OCaml code; and it bootstraps to several platforms.

1. Introduction

F* needed an overhaul. Since its initial development in 2010 (and presentation at ICFP 2011) we have had about five years of programming experience with it to discern the parts of the language that work well and those that are painful.

The main distinctive feature of 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. The main typing construct in the language is the refinement type $\mathsf{x{:}t}\{\phi\}$, a type inhabited only by those elements of t that also satisfy the predicate ϕ, e.g. $\mathsf{x{:}int}\{\mathsf{x} \geq 0\}$ is the type of non-negative integers. Backed by an SMT solver for good automation, F*'s type-checker can be used to statically check a variety of program properties.

We have used value-dependent refinement types to carry out several non-trivial verifications. Some highlights using F* and its predecessor F7 include: the verification of the security of many cryptographic protocols, including, most substantially, miTLS (Bhargavan et al. 2013), a verified implementation of TLS-1.2 (Dierks and Rescorla 2008); several memory invariants of an embedded semantics for JavaScript (Swamy et al. 2013b); a compiler from (a subset of) F* to JavaScript, proved fully abstract (Fournet et al. 2013); and even, using a technique called self-certification, a proof of the correctness of the core verifier of F* itself (Strub et al. 2012). Independently, other researchers have also explored value-dependent refinement types, with good success (Backes et al. 2014; Eigner and Maffei 2013; Lourenço and Caires 2015; Rondon et al. 2008; Vazou et al. 2014).

This positive experience suggests that value dependent types are a sweet spot in the design space of integrating dependent types in a full-fledged programming language. However, as we aim to move F* forwards to the certification of larger pieces of code, we find value dependency to be lacking—as detailed below. (Looking forward, we henceforth write F* for the new F* language and F* v1.0 implementation, and write old F* for F* circa 2010–2014.)

Value-dependent types: why and what for? Given a program e and purported type for it $\mathsf{x{:}t}\{\phi\}$, 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.

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 $\mathsf{x{:}int}\{\mathsf{x} > \mathsf{y}\}$, where y is some program variable in scope), allowing constructs like `x:int{failwith "fixme"; 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 (1) values from the programming language; (2) interpreted function symbols in some logic (e.g., $>$ in the theory of integer arithmentic); and (3) other uninterpreted function symbols. As such, potentially effectful code creeping into the logical fragment of the language is ruled out by syntactic fiat.

The conceptual simplicity of value-dependent types is a significant selling point: for not much work, we can significantly boost the expressiveness of the type system. However, there are several shortcomings: we highlight three of them here, discussing many others throughout the paper.

An axiom- and annotation-heavy programming style. Consider writing a type for a sorted list of integers: one would like to write $\mathsf{x{:}list\ int}\{\mathsf{sorted\ x}\}$, 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 $\mathsf{l{:}list\ int} \to \mathsf{b{:}bool}\{\mathsf{b{=}sorted\ l}\}$, 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. §3.4 shows how we specify and prove Quicksort in F*—now without code duplication.

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. Most value-dependent refinement type systems do not have a core language in which to safely write such proofs. Old F* did have a sub-language of proof terms, but this language, lacking support for any form of induction, was too impoverished for real proving. Without a fallback, some developments rely on a patchwork of tools to complete a proof, e.g., the miTLS effort uses a combination of F7, EasyCrypt and Coq, with careful manual checking of the properties proven in each tool—managing this complexity is overwhelming, and ultimately, leads to a less trustworthy formal artifact. Constructive proofs built semi-interactively are now feasible in F*: one of our largest developments to date is a formalization of the metatheory of System F$_\omega$ 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: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. In §4, we show how refinements now integrate with effects to enable functional correctness proofs of effectful programs.

***On the redesign of* F*** We report on the new design and implementation of F* that remedies these (and several other) shortcomings. Our goal is a language with (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; and (5) easy interoperability with existing ML dialects (in particular, F# and OCaml), and deployable on multiple platforms.

Our main contributions are as follows.

(1) 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. Unlike previous works that use kinds to separate effects or define a single catch-all effect, we give a semantic treatment to effects and structure them as a fine-grained, user-defined lattice of monads (Section 2).

(2) To ensure the core language of pure functions is normalizing, we have a new way of doing semi-automatic semantic termination proofs based on well-founded relations. We show how this core language can be used for both programming and proving interesting examples (Section 3).

(3) We discuss effectful programming in Section 4. Our type system infers the least effect for a program fragment. We prove that our program logic for effectful programs is sound, in the partial correctness sense. A novelty is that with logical proofs, we can prove that intensionally effectful programs are observationally pure—generalizing previous constructions (Launchbury and Peyton Jones 1994).

(4) We discuss a new open-source implementation of F*, itself programmed almost entirely in F*, that bootstraps into F# and OCaml. Two key elements of the new typechecker are type inference and SMT encoding—for lack of space, we cover these topics only lightly. We summarize example programs verified in F* to date (Section 5).

(5) We present a formalization of μF*, a core fragment of F*, distilling the main ideas of the language. We prove type soundness (which covers partial correctness of the program logic for the impure part); and, additionally, termination of the Pure fragment (Section 6).

Contents The paper presents F* using a series of programming & verification examples. An extended version including the definitions and proofs for μF* and the definitions of a larger calculus capturing all the features of F* are available online `http://fstar-lang.org/papers/icfp2015`. An online tutorial and binary packages for major platforms are also available from `http://fstar-lang.org`.

2. The high-level structure of F*

We begin by discussing a few, general organizing principles of F*.

Like prior versions, the type system of F* extends a core based on Girard's (1972) System F$_\omega$ (i.e., higher-rank polymorphism, type operators and higher kinds), with inductive type families, dependent function types, and refinement types. F*, the subject of this paper, both simplifies and generalizes the older design.

First, our new design discards the multiple base kinds of the old system in favor of a more standard, single base kind (called Type). Next, rather than relying on ad hoc restrictions on the kinds to enforce logical consistency, the central organizing principle of F* has the familiar structure of a type-and-effect system, adapted to work with dependent types. While simplifying the overall system, we also gain in expressiveness in several ways: most notably, whereas prior versions of F* only provided *value dependency* (Swamy et al. 2013a), the syntactic class of values no longer have any special status in the new type system; we allow dependency on arbitrary pure computations.

Like ML, F* is a call-by-value language, incorporating a number of primitive effects. Being primitive, we need to pick beforehand the effects of interest: we choose, non-termination, state, exceptions and IO. However, our design should apply equally to other choices of effects, e.g., one might incorporate non-determinism or concurrency.

Following Moggi (1989), we observe that such a language has an inherently monadic semantics. Every computation has a *computation type $m\ t$*, for some effect m, while functions have arrow types with effectful co-domains, e.g., fun x $\rightarrow$ e has a type of the form $t \rightarrow m\ t'$. Traditionally, the effect m is left implicit in type systems for ML; or, when treated explicitly, like Moggi, one may pick a single effect (or category, depending on the setting) in which to interpret all computations. Rather then settling on just a single effect, F*, in a style reminiscent of Wadler and Thiemann (2003), is parameterized by a join semi-lattice of effects, each element denoting some subset of all the primitive effects of the language.

By default, F* is configured with the following effect lattice:

$$\begin{tikzcd}
\mathsf{PURE} \arrow[r] & \mathsf{DIV} \arrow[r] \arrow[dr] & \mathsf{STATE} \arrow[r] & \mathsf{ALL} \\
 & & \mathsf{EXN} \arrow[ur] &
\end{tikzcd}$$

At the bottom, we have PURE, which classifies computations that are pure, total functions. The effect DIV is for computations that may diverge (i.e., they may not terminate), but are otherwise pure. STATE is for computations that may read, write, allocate, or free references in the heap; EXN is for code that may raise exceptions; ALL-computations may have all the effects mentioned so far, as well as IO. We consider other effect lattices elsewhere in the paper, although the bottom of the lattice is always PURE.

We view the language of PURE computations as a logic, using it to write specifications and proofs. The type-and-effect system ensures that the PURE-terms are always normalizing, even though the rest of the program may be effectful. Thus, we achieve with a fairly standard type-and-effect system what others have done with other, non-standard means, e.g., Aura (Jia et al. 2008) and prior

M	M.Post t	M.Pre
PURE	t → Type	Type
DIV	t → Type	Type
STATE	t → heap → Type	heap → Type
EXN	either t exn → Type	Type
ALL	either t exn → heap → Type	heap → Type

Table 1. Signatures of the default effects.

versions of F⋆ (Swamy et al. 2013a) use a system of kinds, while Zombie (Casinghino et al. 2014) uses a novel *consistency classifier* to separate pure and impure code. As we will see, our style of a type-and-effect system has a number of benefits, including promoting better reuse of library code, and greater flexibility in refining the effects in the language, e.g., prior systems only distinguish pure and effectful code, whereas our effect lattice allows for finer distinctions.

Unlike other systems (e.g., (Wadler and Thiemann 2003)), effects in F⋆ are not merely syntactic labels. Instead, each effect is equipped with a predicate-transformer semantics, precisely describing the logical behavior of that effect. In addition to providing a semantic foundation for our language, the semantics of effects naturally yields a program logic with a weakest pre-condition calculus, which is essential for computing verification conditions by typing. The main typing judgment for F⋆ has the following form:

$$\Gamma \vdash e : \mathsf{M}\ \mathsf{t}\ \mathsf{wp}$$

meaning that in a context Γ, *for any property* post dependent on the result of an expression e and its effect, if wp post is valid, then (1) e's effects are delimited by M; and (2) e returns a t satisfying post, or diverges, if permitted by M. We emphasize that the well-typedness of e depends on the validity of the formula wp post—in §3, e.g., we give a PURE function that fails to terminates when its precondition is not met; in §5.1, we discuss our use of an SMT solver to automatically discharge proof obligations.

As such, each effect M is indexed by a result type t and a predicate transformer wp : M.WP t that maps an (effect- and result-type-specific) post-condition post : M.Post t to an (effect-specific) pre-condition wp post : M.Pre. For each effect M, the type of predicate transformers M.WP t forms a monad, i.e., each M.WP is equipped with two combinators, M.return and M.bind satisfying the usual monad laws. This is the basic structure of a so-called *Dijkstra monad*, first proposed by Swamy et al. (2013b) and developed further by Jacobs (2014) for just a single effect; here, we generalize the construction to work with a lattice of effects. As such, for each M' greater than or equal to M in the lattice, we require a function M.lift_M' : WP M a → WP M' a, a monad morphism that must commute with the binds and returns in the expected way—we say that M is a *sub-effect* of M'. We expect the set of lifts to form a lattice and write M ⊔ M' for the least uper bound of two effects.

The lattice and monadic structure of the effects are relevant throughout the type system, but nowhere as clearly as in (T-Let), the rule for sequential composition, which we illustrate below.

$$\frac{\begin{array}{ccc}\Gamma \vdash e_1 : \mathsf{M}_1\ t_1\ \mathsf{wp}_1 & \Gamma, x{:}t_1 \vdash e_2 : \mathsf{M}_2\ t_2\ \mathsf{wp}_2 & \mathsf{M} = \mathsf{M}_1 \sqcup \mathsf{M}_2 \\ \mathsf{wp}'_1 = \mathsf{M}_1.\mathsf{lift\_M}\ \mathsf{wp}_1 & \mathsf{wp}'_2 = \mathsf{M}_2.\mathsf{lift\_M}\ \mathsf{wp}_2 & x \notin FV(t_2)\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathsf{M}\ t_2\ (\mathsf{M.bind}\ \mathsf{wp}'_1\ (\mathsf{fun}\ \mathsf{x} \to \mathsf{wp}'_2))}$$

The sequential composition at the level of programs is captured semantically by the sequential composition of predicate transformers, i.e., by M.bind. (We will see the role of M.return in §3.4.) To compose computations with different effects, M_1 and M_2, we lift them to M, the least effect that includes them both. Since the lifts are morphisms, we get the expected properties of associativity of sequential composition and lifting—the specific order in which we apply lifts is irrelevant to the programmer. The type system is designed to infer the least effect for a computation.

No parametric polymorphism on effects. For easy reference, Table 1 shows the signature of the effects in our default lattice. (In the table, either is the standard tagged union type.) In each case, M.WP t = M.Post t → M.Pre. We explain these signatures in detail in the coming sections, starting with the PURE effect in the next section. Before proceeding, we point out an important design choice: F⋆ lacks parametric polymorphism over effects. We choose this design for two reasons. First, as should be evident already from the table, since the signature of an effect has a kind that depends on the effect itself, writing effect-parametric types requires kind polymorphism and kind operators. Furthermore, obtaining a practically usable language with effect polymorphism, even for languages with much simpler type systems, is still an area of active research, starting from the late '80s (Lucassen and Gifford 1988) and continuing vigorously at present (e.g., in works by Swamy et al. (2011) and Leijen (2014)). The lack of effect polymorphism keeps inference tractable, and the loss in expressiveness is somewhat offset by the presence of sub-effects.

3. Programming and proving with pure functions

Pure expressions are given the computation type PURE t wp, where wp: PURE.Post t → PURE.Pre. A pure post-condition is a predicate on t-typed results, while a pure pre-condition is simply a proposition, i.e., PURE.Post t = t → Type and PURE.Pre = Type. As shown below, to prove a property post about a pure result x, one must just prove post x, and the sequential composition of pure computations involves the functional composition of their predicate transformers.

```
PURE.return a (x:a) = fun (post:PURE.Post a) → post x
PURE.bind a (wp1:PURE.WP a) (wp2:a → PURE.WP b) =
      fun (post:PURE.Post b) → wp1 (fun x → wp2 x post)
```

As an example, consider the following term (where l:list t):

```
List.hd l : PURE t wp
where wp (post:PURE.Post t) = ∃hd. l=hd::_ ∧ post hd
```

This example illustrates that purity (which includes totality and termination) can be conditional. To prove that the term is well-typed, we need to prove the validity of wp post for some given pure post-condition, such as post = fun x → True. This amounts to showing that ∃hd. l=hd::_, which ensures that List.hd l does not fail because of a non-exhaustive case-analysis.

For terms that are unconditionally pure, we introduce Tot, an abbreviation for the special case of the PURE effect defined below:

```
effect Tot (t:Type) = Pure t (fun post → ∀x. post x)
```

When writing specifications, it is often convenient to use traditional pre- and post-conditions instead of predicate transformers. Accordingly, we also introduce the abbreviations below, with keywords requires and ensures only for readability:

```
effect Pure (t:Type) (requires (p:PURE.Pre)) (ensures (q:PURE.Post t))
      = PURE t (fun post → p ∧ ∀x. q x ⟹ post x)
```

Notations: Lambda abstractions are introduced with the notation fun (b_1) ... (b_n) → ε, where the b_i range over binding occurrences for variables, and the body ε ranges over both types and expressions. Binding occurrences come in two forms, x:t for binding an expression variable at type t; and a:k, for a type variable at kind k. Each of these may be preceded by an optional #-mark, indicating the binding of an implicit parameter. In lambda abstractions, we generally omit annotations on bound variables (and the enclosing parentheses) when they can be inferred, e.g., we may write fun x → x + 1 or fun (#a:Type) (x:a) → x. Function types and kinds are written b → σ, where σ ranges over computation types m t and kinds k—note the lack of enclosing parentheses; as we will see, this convention leads to a more compact notation when used with refinement types. The variable bound by b is in scope to the right of the arrow. When the co-domain does not mention the formal parameter, we may omit the name of the parameter. For example, we may write int → m int or #a:Type → Tot (a → Tot a).

We use the Tot effect by default in our notation for curried function types: on all but the last arrow, the implicit effect is Tot.

$$\mathsf{b}_1 \to \ldots \to \mathsf{b}_n \to \mathsf{M}\ \mathsf{t}\ \mathsf{wp} \triangleq \mathsf{b}_1 \to \mathsf{Tot}\ (\ \ldots \to \mathsf{Tot}\ (\mathsf{b}_n \to \mathsf{M}\ \mathsf{t}\ \mathsf{wp}))$$

So, the polymorphic identity function has type #a:Type → a → Tot a.

The language of logical specifications is included within the language of types. However, as illustrated above, we use standard syntactic sugar for the logical connectives $\forall, \exists, \land, \lor, \Longrightarrow$, and $\Longleftrightarrow$. The appendix shows how we encode these in types. We also overload these connectives for use with boolean expressions—F* automatically coerces booleans to Type as needed.

3.1 Refinement types and structural subtyping

While the verification machinery of F* is now founded on effects equipped with predicate transformers, it is often more convenient to specify properties as refinement types. Hence, F* retains the refinement types of its prior versions: a refinement of a type t is a type x:t$\{\phi\}$ inhabited by expressions e : Tot t that additionally validate the formula $\phi[e/x]$. For example, F* defines the type nat = x:int$\{x \geq 0\}$. Using this, we can write the following code:

```
val factorial: nat → Tot nat
let rec factorial n = if n = 0 then 1 else n ∗ factorial (n-1)
```

Unlike subset types or strong sums $\Sigma x{:}t.\phi$ in other dependently typed languages, F*'s refinement types x:t$\{\phi\}$ come with a subtyping relation, so, for example, nat <: int; and n:int can be implicitly refined to nat whenever $n \geq 0$. Specifically, the representations of nat and int values are identical—the proof of $x \geq 0$ in x:int$\{x \geq 0\}$ is never materialized. As in old F*, this is convenient in practice, as it enables data and code reuse as well as automated reasoning.

A new subtyping rule allows refinements to better interact with function types and effectful specifications, further improving code reuse. For example, the type of factorial declared above is equivalent by subtyping to the following refinement-free type:

```
x:int → PURE int (fun post → x ≥ 0 ∧ ∀y. y ≥ 0 ⟹ post y)
```

We also introduce syntactic sugar for mixing refinements and dependent arrows, writing x:t$\{\phi\} \to \sigma$ for x:(x:t$\{\phi\}) \to \sigma$.

Refinement types are more than just a notational convenience: nested refinements within types can be used to specify properties of unbounded data structures, and other invariants. For example, the type list nat describes a list whose elements are all non-negative integers, and the type ref nat describes a heap reference that always contains a non-negative integer.

3.2 Indexed type families

Aside from arrows and primitive types like int, the basic building blocks of types in F* are recursively defined indexed datatypes. For example, we give below the abstract syntax of the simply typed lambda calculus in the style of de Bruijn (we only show a few cases).

```
type typ = | TUnit : typ | TArr: arg:typ → res:typ → typ
type var = nat
type exp = | EVar : x:var → exp | ELam : t:typ → body:exp → exp ...
```

The type of each constructor is of the form $\mathsf{b}_1 \to \ldots \to \mathsf{b}_n \to \mathsf{T}\ \tau_1\ \ldots\ \tau_m$, where T is type being constructed. This is syntactic sugar for $\mathsf{b}_1 \to \ldots \to \mathsf{b}_n \to \mathsf{Tot}\ (\mathsf{T}\ \tau_1\ \ldots\ \tau_m)$, i.e., constructors are total functions.

Given a datatype definition, F* automatically generates a few auxillary functions: for each constructor, it provides a *discriminator*; and for each argument of each constructor, it provides a *projector*. For example, for typ, we obtain the following two discriminators and two projectors.

```
let is_TUnit = function TUnit → true | _ → false
let is_TArr = function TArr _ _ → true | _ → false
let TArr.arg (t:ty{is_TArr t}) = match t with TArr arg _ → arg
let TArr.res (t:ty{is_TArr t}) = match t with TArr _ res → res
```

The standard prelude of F* defines the list and option types, as usual. F* supports the standard syntactic sugar for lists, and it will be clear from the context when we make use of projectors and discriminators for these types.

In contrast with a system like Coq, F* does not generate induction principles for datatypes—they may not even be inductive, since F* allows non-positive definitions. Instead, the programmer directly writes fixpoints and general recursive functions, and a semantic termination checker ensures consistency.

Types can be indexed by both pure terms and other types. For example, we show below (just two rules of) an inductive type that defines the typing judgment of the simply-typed lambda calculus.

```
type env = var → Tot (option typ)
val extend: env → typ → Tot env
let extend g t y = if y=0 then Some t else g (y − 1)
type typing : env → exp → typ → Type = ...
| TyUn : #g:env → typing g EUnit TUnit
| TyLam : #g:env → #t:typ → #e1:exp → #t':typ →
          typing (extend g t) e1 t' → typing g (ELam t e1) (TArr t t')
```

Refinements and indexed types work well together. Notably, pattern matching on datatypes comes with a powerful exhaustiveness checker: one only needs to write the reachable cases, and F* relies on all the information available in the context, not just the types of the terms being analyzed. For example, we give below an inversion lemma proving that the canonical form of a well-typed closed value with an arrow type is a λ-abstraction with a well-typed body. The indexing of d with emp, combined with the refinements on e and t, allows F* to prove that the only reachable case for d is TyLam. Furthermore, the equations introduced by pattern matching allow F* to prove that the returned premise has the requested type.

```
let emp x = None
let value = function ELam _ _ | EVar _ | EUnit _ → true | _ → false
val inv_lam: e:exp{value e} → t:typ{is_TArr t} → d:typing emp e t →
   Tot (typing (extend emp (TArr.arg t)) (ELam.body e) (TArr.res t))
let inv_lam e t (TyLam premise) = premise
```

3.3 Semantic proofs of termination

As in any type theory, the soundness of our logic relies on the normalization of pure terms. We provide a fully semantic termination criterion based on well-founded partial orders. This is in sharp contrast with the type theories underlying systems like Coq, which rely instead on a syntactic "guarded by destructors" criterion. As has often been observed (e.g., by Barthe et al. 2004, among several others), this syntactic criterion is brittle with respect to simple semantics-preserving transformations, and hinders proofs of termination for many common programming patterns. Worse, syntactic checks interact poorly with other aspects of the logic, leading to unsoundnesses when combined with seemingly benign axioms.[1]

The type system of F* is parameterized by the choice of a well-founded partial order ($\prec$) : #a:Type → #b:Type → a → b → Type, over all terms (pronounced "precedes"). We provide a new rule for typing fixpoints, making use of this well-founded order to ensure that the fixpoint always exists, as shown below:

$$\text{(T-Fix)}\ \frac{\mathsf{t}_f = \mathsf{y{:}t} \to \mathsf{PURE}\ \mathsf{t'}\ \mathsf{wp} \qquad \Gamma \vdash \delta : \mathsf{Tot}\ (\mathsf{y{:}t} \to \mathsf{Tot}\ \mathsf{t''}) \\ \Gamma, \mathsf{x}{:}t, \mathsf{f}{:}(\mathsf{y{:}t}\{\delta \mathsf{y} \prec \delta \mathsf{x}\} \to \mathsf{PURE}\ \mathsf{t'}\ \mathsf{wp}) \vdash \mathsf{e} : \mathsf{PURE}\ \mathsf{t'}\ \mathsf{wp}}{\Gamma \vdash \mathsf{let\ rec}\ \mathsf{f}^{\delta}{:}\ \mathsf{t}_f = \mathsf{fun}\ \mathsf{x} \to \mathsf{e} : \mathsf{Tot}\ \mathsf{t}_f}$$

[1] https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html

When introducing a recursive definition of the form let rec f^{δ} : (y:t → PURE t' wp) = fun x → e, we type the expression e in a context that includes x:t and f at the type y:t{δy ≺ δx} → PURE t' wp, where the decreasing metric δ is any pure function. Intuitively, this rule ensures that, when defining the *i*-th iterate of f x, one may only use previous iterates of f defined on a strictly smaller domain. We think of δ as a decreasing metric on the parameter, which F* picks by default (as shown below) but which can also be provided explicitly by the programmer (as shown in §3.3.1). F* also provides general recursion for impure functions; the typing rule for this is standard and does not require the use of a termination metric.

We illustrate rule (T-Fix) for typing factorial (defined above). The body of factorial is typed in a context that includes n:nat and factorial: m:nat{m ≺ n} → Tot nat, i.e., in this case, F* picks δ=id. At the recursive call factorial (n-1), it generates the proof obligation n-1 ≺ n. The implementation of F* instantiates the relation ≺ with the well-founded partial order below (which includes the usual ordering on nat) and easily dispatches this obligation.

3.3.1 The default well-founded ordering

F* instantiates its well-founded partial order as follows:

(1) Given i, j : nat, we have i ≺ j ⟺ i < j. The negative integers are not related by the ≺ relation.

(2) Elements of the type lex_t are ordered lexicographically, as detailed below.

(3) The sub-terms of an inductively defined term precede the term itself, that is, for any pure term *e* with inductive type T≠lex_t, if e=D $e_1 \ldots e_n$ we have e_i ≺ e. for all i.

A larger relation would increase the expressiveness of our termination checker. For instance, we plan to add f x ≺ f when f is a total function and multi-set orders, although, so far, we have not found these necessary for our examples.

For lexicographic orderings, F* includes in its standard prelude the following inductive type (with its syntactic sugar):

```
type lex_t = LexTop : lex_t | LexCons: #a:Type → a → lex_t → lex_t
where %[v1;...;vn]@ ≜ LexCons v1 ... (LexCons vn LexTop)
```

For well-typed pure terms v, v1, v2, v1', v2', the ordering on lex_t is the following:

(2a.) LexCons v1 v2 ≺ LexCons v1' v2', if and only if, either v1 ≺ v1'; or v1=v1' and v2 ≺ v2'.

(2b.) If v:lex_t and v ≠ LexTop, then v ≺ LexTop.

For functions of several arguments, one aims to prove that a metric on some subset of the arguments decreases at each recursive call. By default, F* chooses the metric to be the lexicographic list of all the non-function-typed arguments in order. When the default does not suffice, the programmer can override it with an optional decreases annotation, as we will see below.

3.3.2 Parallel substitutions: A non-trivial termination proof

Consider the simply typed lambda calculus from §3.2. It is convenient to equip it with a *parallel substitution* that simultaneously replaces a set of variables in a term. Proving that parallel substitutions terminate is tricky—e.g., Adams (2006); Benton et al. (2012)all give examples of ad hoc workarounds to Coq's termination checker. Figure 1 shows a succinct, complete development in F*.

Before looking at the details, consider the general structure of the function subst at the end of the listing. The first three cases are easy. In the ELam case, we need to substitute in the body of the abstraction but, since we cross a binder, we need to increment the indexes of the free variables in all the expressions in the range of

```
1  type presub = {
2    sub:var → Tot exp; (* the substitution itself *)
3    renaming:bool; (* an additional field for the proof; morally ghost *)
4  } (* sub invariant: if the flag is set, then the map is just a renaming *)
5  type sub = s:presub{s.renaming ⟹ (∀ x. is_EVar (s.sub x))}
6  let sub_inc : sub = {renaming=true; sub=(fun y → EVar (y+1))}
7  let ord_b = function true → 0 | false → 1 (* an ordering on booleans *)
8  val subst : e:exp → s:sub → Pure exp (requires true)
9              (ensures (fun e' → s.renaming ∧ is_EVar e ⟹ is_EVar e'))
10             (decreases %[ord_b (is_EVar e); ord_b (s.renaming); e])
11 let rec subst e s = match e with
12   | EUnit → EUnit
13   | EVar x → s.sub x
14   | EApp e1 e2 → EApp (subst e1 s) (subst e2 s)
15   | ELam t body →
16     let shift_sub : var → Tot (e:exp{s.renaming ⟹ is_EVar e}) =
17       fun y → if y=0 then EVar y else subst (s.sub (y-1)) sub_inc in
18     ELam t (subst body ({s with sub=shift_sub}))
```

Figure 1. Parallel substitutions on λ-terms

the substitution—of course, incrementing the free variables is itself a substitution, so we just reuse the function being defined for that purpose: we call subst recursively on body, after shifting the range of the substitution itself, using shift_subst.

Why does this function terminate? The usual argument of being structurally recursive on e does not work, since the recursive call at line 17 uses s.sub (y-1) as its first argument, which is not a sub-term of e. Intuitively, it terminates because in this case the second argument is just a renaming (meaning that its range contains only variables), so deeper recursive calls will only use the EVar case, which terminates immediately. This idea was originally proposed by Altenkirch and Reus.

To formalize this intuition in F*, we instrument substitutions sub with a boolean flag renaming, with the invariant that if the flag is true, then the substitution is just a renaming (lines 1–5). Notice that given a nat → Tot exp, it is impossible to decide whether or not it is a renaming; however, by augmenting the function with an invariant, we can prove that substitutions are renamings as they are defined. Using this, we provide a decreases metric (line 10) as the lexical ordering %[ord_b (is_EVar e); ord_b (s.renaming); e]).

Let us now consider the termination of the recursive call at line 17. If s is a renaming, we are done; since e is not an EVar, and s.sub (y -1) is, the first component of the lexicographic ordering strictly decreases. If s is not a renaming, then since e is not an EVar, the first component of the lexicographic order may remain the same or decrease; but sub_inc is certainly a renaming, so the second component decreases and we are done again.

Turning to the call at line 18, if body is an EVar, we are done since e is not an EVar and thus the first component decreases. Otherwise, body is a non-EVar proper sub-term of e; so the first component remains the same while the third component strictly decreases. To conclude, we have to show that the second component remains the same, that is, subst_shift is a renaming if s is a renaming. The type of subst_shift captures this property. In order to complete the proof we finally need to strengthen our induction hypothesis to show that substituting a variable with a renaming produces a variable—this is exactly the purpose of the ensures-clause at line 9.

3.4 Intrinsic and extrinsic proofs on pure definitions

Prior systems of refinement types, including old F*, the line of work on liquid types (Rondon et al. 2008), and the style of refinement types used by Freeman and Pfenning (1991), only support type-based reasoning about programs, i.e., the only properties one can derive about a term are those that are deducible from its type.

For example, in those systems, given id: int → int, even though we may know that id=fun x → x, proving that id 0 = 0 is usually not possible (unless we give id some other, more precise type). This limitation stems from the lack of a fragment of the language in which functions behave well logically—int → int functions may have arbitrary effects, thereby excluding direct reasoning. Specifically, given id:int → int, we cannot prove that 0 has type x:int{x=id 0}. In older versions of F* (which only permitted value dependency) the type is just not well-formed, since id 0 is not a value; with liquid types, functions in refinements are uninterpreted, so although the type is well-formed, the proof still does not go through.

With its semantic treatment of effects, F* now supports direct reasoning on pure terms, simply by reduction. For example, F* proves List.map (fun x → x + 1) [1;2;3] = [2;3;4], given the standard definition of List.map with no further annotations—as expected by programmers working in type theory.[2] The typing rule below enables this feature by using monadic returns. In effect, having proven that a term e is pure, we can lift it wholesale into the logic and reason about it there, using both its type t and its definition e.

$$\text{(T-Ret)}\ \frac{\Gamma \vdash e : \mathsf{Tot\ t}}{\Gamma \vdash e : \mathsf{PURE\ t\ (PURE.return\ t\ e)}}$$

The importance of being able to reason directly about definitions is hard to overstate. Lacking this ability, prior versions of F* encouraged an axiom- and annotation-heavy programming style. The reader may wish to compare the F* proof of Quicksort developed next with the analogous proof in F*-v0.7, available at `http://rise4fun.com/FStar/UsSR`.

Verifying Quicksort Consider the following standard definition of Quicksort—we will verify its total correctness in a few steps, illustrating one style of semi-automatic proving in F*.

```
open List
let rec quicksort f = function [] → []
  | pivot::tl → let hi, lo = partition (f pivot) tl in
                append (quicksort f lo) (pivot::quicksort f hi)
```

The functions partition and append are defined as usual in the List library, with the types shown below. The main thing to note is that they are both total functions.

```
val partition: #a:Type → (a → Tot bool) → list a → Tot (list a ∗ list a)
val append: #a:Type → list a → list a → Tot (list a)
```

First, we need to write a specification against which to verify quicksort, starting with sorted f l, which decides when l is sorted with respect to the comparison function f; and count x l which counts the number of occurrences of x in l. We also define a type total_order on binary boolean functions.

```
val sorted: #a:Type → (a → a → Tot bool) → list a → Tot bool
let rec sorted f = function x::y::tl → f x y && sorted f (y::tl) | _ → true

val count: #a:Type → a → list a → Tot nat
let count x = function
  | [] → 0
  | hd::tl → if hd=x then 1 + count x tl else count x tl
let mem x tl = count x tl > 0

type total_order (a:Type) (f: (a → a → Tot bool)) =
   (∀ a. f a a) (∗ reflexivity ∗)
   ∧ (∀ a1 a2. (f a1 a2 ∧ f a2 a1) ⟹ a1 = a2) (∗ antisymmetry ∗)
   ∧ (∀ a1 a2 a3. f a1 a2 ∧ f a2 a3 ⟹ f a1 a3) (∗ transitivity ∗)
   ∧ (∀ a1 a2. f a1 a2 ∨ f a2 a1) (∗ totality ∗)
```

[2] An implementation detail is that F* delegates reasoning about the combination of reduction and conversion to an SMT solver, rather than relying on custom-built reduction machinery.

With these definitions in hand, we can write our specification of Quicksort: if f is a total order, then quicksort f l returns a permutation of l that is sorted according to f.

```
val quicksort: #a:Type → f:(a → a → Tot bool){total_order a f}
  → l:list a → Tot (m:list a{sorted f m ∧ (∀ i. count i l = count i m)})
                     (decreases (length l))
```

However, without some more help, F* fails to verify the program—the error message it reports is shown below (only the variables have been renamed). The position it reports refers to the parameter lo of the first recursive call to quicksort, meaning that F* failed to prove that the function terminates.

```
Subtyping check failed;
  expected type lo:list a{%[length lo] << %[length l]};
  got type (list a) (qs.fst(99,19-99,21))
```

We need to convince F* that, at each recursive call, the lengths of lo and hi are smaller than the length of the original list. We also need to prove that all the elements of lo (resp. hi) are smaller than (resp. greater or equal to) the pivot; that appending sorted list fragments with the pivot in the middle produces a sorted list; and that the occurrence counts of the elements are preserved.

In prior systems, one would have to re-type-check the definitions of append and partition to prove these properties, which is extremely non-modular. Instead, F* allows one to prove lemmas about these definitions, after the fact—a style we call *extrinsic* proof, in contrast with *intrinsic* proofs, which work by enriching the type and definition of a term to prove the property of interest. In this style, a lemma is any unit-returning Pure function; we provide the following sugar for it.

```
effect Lemma (requires p) (ensures q) =
     Pure unit (requires p) (ensures (fun _ → q))
```

We give below a simple extrinsic proof that append sums occurrence counts. In general, F* does not attempt proofs by induction automatically—instead, the user writes a fixpoint, setting up the induction skeleton, and relies on F* to prove all cases.

```
val app_c: #a:Type → l:list a → m:list a → x:a → Lemma (requires True)
   (ensures (count x (append l m) = count x l + count x m))
let rec app_c l m x = match l with
  | [] → () | hd::tl → app_c tl m x
```

With an extrinsic proof, we gain modularity but lose (some) convenience: if we had instead a (non-modular) intrinsic proof giving the type l:list a → m:list a → Tot (n:list a{∀x. count x n = count x l + count x m}) to append, then every call to happen would yield the property. With an extrinsic proof, to use the app_c property of append l m, we need to explicitly call the lemma, e.g., to complete the proof of quicksort, we would have to pollute its definition with a call to lemma, which is less than ideal.

Bridging the gap between extrinsic and intrinsic proofs. To have the best of both worlds, we would like to automatically apply extrinsically proved properties to refine the types of existing terms. Accordingly, F* allows Lemmas to be decorated with *SMT patterns*, as illustrated below on two further lemmas in the proof of quicksort:

```
val partition_lemma: #a:Type → f:(a → Tot bool) → l:list a → Lemma
   (requires True)
   (ensures (∀ hi lo. (hi, lo) = partition f l
         ⟹ (length l = length hi + length lo
          ∧ (∀ x. (mem x hi ⟹ f x) ∧ (mem x lo ⟹ not (f x))
               ∧ (count x l = count x hi + count x lo)))))
   [SMTPat (partition f l)] (∗ Automation hint ∗)
let rec partition_lemma f = function _::tl → partition_lemma f tl | _ → ()
```

```
val sorted_app_lemma: #a:Type → f:(a → a → Tot bool){total_order a f}
                        → l1:list a{sorted f l1} → l2:list a{sorted f l2}
                        → p:a → Lemma
    (requires (∀ y. (mem y l1 ⟹ not (f p y)) ∧ (mem y l2 ⟹ f p y)))
    (ensures (sorted f (append l1 (p::l2))))
    [SMTPat (sorted f (append l1 (p::l2)))] (* Automation hint *)
let rec sorted_app_lemma f l1 l2 p = match l1 with
    | [] → () | hd::tl → sorted_app_lemma f tl l2 p
```

The statements of these lemmas are detailed, but self-explanatory and in both cases, the proofs are one-line inductions. Adding those ad hoc properties to the intrinsic types of functions like partition and append would be completely inappropriate—this is especially true in the case of sorted_app_lemma, a property of append highly specialized for the proof of quicksort.

The main point to call out here is the use of SMTPat annotations as automation hints. In the case of partition_lemma, the hint instructs F⋆ (and the underlying solver) to apply the property for any well-typed term of the form partition f l. For sorted_app_lemma, the hint is more specific: the verifier can use the property for any well-typed occurrence of sorted f (append l1 (p::l2)). With these lemmas and the addition of [SMTPat (count x (append l m))] to append_count, F⋆ automatically completes the proof of quicksort.

This proof style is reminiscent of ACL2 (Kaufmann and Moore 1996) but, unlike first-order, untyped ACL2, the underlying mechanism based on SMT solving and pattern-based quantifier instantiation (dating from the Stanford Pascal verifier of Luckham et al. 1979) works well with higher-order dependently typed programs.

The exact mixture of extrinsic and intrinsic proof to use is a matter of taste and experience. A rule of thumb is to prove compactly stated, generally useful properties intrinsically, and to prove the rest extrinsically. For example, our proof of the quicksort function itself is intrinsic, since it is easy to state and generally useful. One the other hand, sorted_app_lemma is best proved extrinsically.

Another constraint is that termination must always be proven intrinsically; this differs from Zombie (Casinghino et al. 2014), which provides a method of doing extrinsic termination proofs. While the extrinsic termination proofs are elegant in principle, in practice, as one of the authors of Zombie says in private communication, the proofs (which themselves must be proven terminating intrinsically) "repeat the same kind of recursion that the original function performed, with a lot of extra equational reasoning, so they can get quite long". For example, for a 25-line implementation of mergesort in Zombie, the proof of termination alone (not functional correctness) is nearly 300 lines long.[3] In contrast, the flexibility of F⋆'s termination check and the automation it provides often keeps these proofs fairly simple. Finally, as we discuss next, intrinsic proofs are the only option for programs with effects.

4. Specifying and verifying programs with effects

F⋆ provides primitive support for non-termination, state, exceptions and IO. In principle, one could configure F⋆ to use the powerset lattice over these effects, e.g., we could include an effect TotST for the total correctness of stateful code.[4] Or one could choose to split reading, writing, and (de-)allocation into separate atomic effects and take the powerset over this larger set. Or, still finer, one could split operations on separate heap regions (down to singleton cells) into separate atomic effects. Effect inference in F⋆ enables such fine-grained effect tracking. The effect of a term is automatically computed as the least upper-bound of the effects of its subterms. To escape this discipline of ever-increasing effects, F⋆ also supports coercions that can be used to forget the effects of an expression when they cannot be observed by the context.

Neither fine-grained effect tracking nor effect coercions were possible in old F⋆, or for that matter in frameworks based on approaches like HTT (Nanevski et al. 2008), which use a single "catch-all" effect to capture all impure operations in the language.

The rest of this section provides some details. For brevity, we say little about the EXN effect, covering exceptions in ALL only. The online version of the paper provides more details about EXN.

4.1 The DIV effect

The specification of every effect in the F⋆ prelude includes its signature, the functions required by the signature (e.g., bind and return on predicate transformers), and a flag that indicates whether the effect includes non-termination. Given an effect lattice, we require that the effects that *exclude* non-termination be downward closed. By default, only the PURE effect excludes non-termination, and the DIV effect is the least in the lattice that includes non-termination.

Aside from the non-termination flag, the signature of the DIV effect is identical to the one of PURE effect given in §3, except that specifications in DIV are interpreted in a partial-correctness semantics. We use the abbreviations Dv and Div, which are to to DIV what Tot and Pure are to PURE.

We may use DIV when a termination proof of a pure function requires more effort than the programmer is willing to expend, and, of course, when a function may diverge intentionally.

For example, we give below the top-level statement of progress and preservation for our simply typed lambda calculus, showing only show the signatures of typecheck and typed_step.[5]

```
val typecheck: env → exp → Tot (option typ)
val typed_step : e:exp{is_Some (typecheck emp e) ∧ not(value e)}
                  → Tot (e':exp{typecheck emp e' = typecheck emp e})
val eval : e:exp{is_Some (typecheck emp e)}
           → Dv (v:exp{value v ∧ typecheck emp v = typecheck emp e})
let rec eval e = if value e then e else eval (typed_step e)
```

Recursive functions with the DIV effect need not respect the well-founded ordering of F⋆, and indeed may diverge. Expressions typed in the PURE effect (such as value e and typed_step e above) are implicitly promoted to the DIV effect, as needed. This promotion relies on *sub-effecting* according to the effect lattice. Intuitively, a function proven totally correct can also be used in a partial correctness context. Accordingly, F⋆ applies an identity lifting defined in the prelude:

```
PURE.lift_DIV (a:Type) (wp:PURE.WP a) : DIV.WP a = wp
```

4.2 The STATE effect

F⋆ provides primitive support for mutable heap references, including dynamic allocation and deallocation. We have yet to implement a runtime system that actually reclaims memory on deallocation, but the language is designed to accommodate it soundly.

A stateful post-condition STATE.Post t is a predicate relating the t-typed result of a computation to its final state; a stateful pre-condition is a predicate on the initial state. States (h, of type heap) range over abstract partial maps from references to their contents, with operations sel, and upd behaving according to the usual McCarthy (1962) axioms, and with a predicate, has h x, to indicate that x is in the domain of h.

[3] https://code.google.com/p/trellys/source/browse/trunk/zombie-trellys/test/Sort.trellys

[4] Our implementation can support TotST for references to first-order values. With a first-order store, our PURE normalization results should carry over easily to TotST (although we have not yet proven it). Totality for programs with higher order store is future work.

[5] With more work, we can also prove that evaluation in the simply typed lambda calculus terminates; an example in our test suite does it using hereditary substitutions.

We give below specifications of the stateful primitives to read and write references. For example, to update a reference using r := v, one must prove that the initial heap h0 has the reference r; in return, after the update, h0 evolves to upd h0 r v.

```
val (!) : #a:Type → r:ref a → STATE a (fun post h0 →
                                   has h0 r ∧ post (sel h0 r) h0)

val (:=) : #a:Type → r:ref a → v:a → STATE unit (fun post h0 →
                                   has h0 r ∧ post () (upd h0 r v))
```

The monadic combinators for STATE.WP t are shown below. As expected, returning a pure value leaves the heap unchanged, and sequential composition at the level of programs corresponds to function composition of predicate transformers.

```
STATE.return a x post h = post x h
STATE.bind a b wp1 wp2 post = wp1 (fun x → wp2 x post)
```

We define abbreviations to work more directly with the STATE effect and two-state specifications. The computation type ST a (requires pre) (ensures post) (modifies s) is the type of a stateful computation which when run in an initial heap h0 that satisfies pre h0, either diverges or produces an a result v:a and heap h1 satisfying post h0 v h1, where on their shared domain, h1 differs from h0 only in locations in the set of references s. We use ST to write specifications for alloc and free.

```
val alloc : #a:Type → v:a → ST (ref a) (requires (fun _ → True))
 (ensures (fun h0 r h1 → not(has h0 r) ∧ has h1 r ∧ sel h1 r=v))
 (modifies {}}

val free: #a:Type → r:ref a → ST unit (requires (fun h0 → has h0 r))
 (ensures (fun h0 r h1 → not (has h1 r))) (modifies {r})
```

When lifting from DIV to STATE, we use the following combinator, indicating that DIV computations never touch the heap.

```
DIV.lift_STATE a wp p h = wp (fun x → p x h)
```

The* ALL *effect As usual, when combining effects, we have to be careful—not all effects commute with one another. Post-conditions in the ALL monad have signature either a exn → heap → Type, where either is the tagged union type and exn is the standard (extensible) datatype of exceptions. In lifting from STATE (resp. EXN) to ALL, we specify the usual ordering of ML, with STATE following EXN, and thus we have:

```
STATE.lift_ALL a wp p = wp (fun x → p (Inl x))
EXN.lift_ALL a wp p h = wp (fun r → p r h)
```

4.3 Forgetting effects, logically

The effect of a term is the least upper-bound of the effects of its subterms. However, we would like to be able to relax this discipline when the effect of a term is unobservable to its context. For example, consider computations that use state locally, or those that handle all exceptions that may be raised: it would be convenient to treat such computations purely. Of course, this must be done with caution: there are two points to consider, related to intensionality.

(1) Termination in F* is an intensional property, i.e., termination proofs must be done intrinsically, relying on the definition of the term and not just its observational behavior. Thus, there is no way to start with a computation that has the divergence effect, prove that it is observationally equivalent to a term that terminates, and thereby forget its divergence effect. Every other effect is treated extensionally, and as we will see below, we provide a means of forgetting those effects, modulo a logical guard. For example, given a program that is proven totally correct while using state and exception, it may be possible to prove that it has no observable effect at all.

(2) The other intensional element of F* is its treatment of the PURE effect. As discussed in §3.4, we provide direct reasoning on the *definitions* of PURE functions. Definitional reasoning in this style does not apply to code that is only proven to be observational effect-free, as it it unclear how to reason in the logic about programs that internally throw exceptions, use state, and perhaps even non-determinism and concurrency. (In principle, one may interpret some of these effects using pure functions, but this is not currently supported.)

With these points in mind, our strategy for forgetting effects involves defining a second effect lattice, identical in structure to the first. For each effect M in the first lattice, we have an effect M† in the second, the extensional counterpart of (the intensional effect) M. The effects observable in code with effect M† are only those in M, although internally, M† computations may use other effects of the language. To combine the two lattices, we give identity liftings from each M to its counterpart M†. Additionally, we provide forgetful coercions from M† to M'†, as long as the non-termination effect is not forgotten.

For example, consider a base lattice with PURE, DIV, TotST, STATE, and ALL where TotST is the effect of total, stateful computations. These effects are ordered as shown below. Our construction first copies this lattice, introducing identity liftings (shown in dotted arrows below) from each M to its extensional counterpart M†. Finally, we have coercions (shown in squiggly arrows) to (1) forget exceptions from ALL† to STATE†; (2) forget state from STATE† to DIV†; and (3) forget state from TotST† to PURE†.

$$\begin{tikzcd}
 & \mathsf{TotST}\dagger \arrow[dl, squiggly] \arrow[dr] & & \\
\mathsf{PURE}\dagger \arrow[ur] \arrow[dr] & & \mathsf{STATE}\dagger \arrow[r] \arrow[dl, squiggly] & \mathsf{ALL}\dagger \arrow[l, squiggly, shift left] \\
 & \mathsf{DIV}\dagger \arrow[ur] & & \\
 & \mathsf{TotST} \arrow[uuu, dotted, bend right] \arrow[dr] & & \\
\mathsf{PURE} \arrow[uuu, dotted] \arrow[ur] \arrow[dr] & & \mathsf{STATE} \arrow[uuu, dotted] \arrow[r] & \mathsf{ALL} \arrow[uuu, dotted] \\
 & \mathsf{DIV} \arrow[uuu, dotted, bend left] \arrow[ur] & &
\end{tikzcd}$$

Some points in this lattice are potentially uninteresting. For example, ALL† is degenerate—it differs in no meaningful way from ALL. Sometimes, it may be worthwhile to distinguish, say, DIV and DIV†, e.g., computations in DIV can be run on platforms that offer limited heap storage. Nevertheless, in many cases, a programmer may not wish to distinguish between M and M† at all—although the language insists on at least distinguishing PURE and PURE†. Thus, a simpler lattice, with nearly the same expressive power, would use just the †-versions of each effect and PURE.

The signature of the forgetful coercion from STATE† to DIV† is shown below—the coercion from TotST† to PURE† is identical.

```
1 assume val forget_ST: #a:Type → #b:(a → Type)
2   → #req:(a → heap → Type)
3   → #ens:(x:a → heap → b x → heap → Type)
4   → f:(x:a → ST† (b x) (req x) (ens x) (modifies {}))
5      {∀ x h h'. req x h ⟹ req x h'}
6   → Tot (x:a → Div† (b x) (requires (∀ h. req x h))
7         (ensures (fun (y:b x) → ∃h0 h1. ens x h0 y h1)))
```

The coercion forget_ST is a primitive in the language (as indicated by the assume keyword). Given a stateful function f, with pre-condition req, post-condition ens, and which does not mutate any existing heap reference (modifies {}); if the pre-condition req is insensitive to the contents of the heap (line 5), forget_ST coerces f to Div†, turning its pre- and post-condition into pure, heap-invariant formulae. The intuition is that if f can safely be called in any initial heap, then it cannot read, write or free any existing reference (since, from the signatures of those primitives, doing so requires proving that the current heap has those references). On the other hand, f may allocate and use references internally, but the post-condition on line 7 hides all properties of the heap that results after the execution of f—so, the freshly allocated state (if any) is inaccessible to the caller. Taking these two properties together, we may as well just forget about f's use of the heap, since a caller can never observe it. However, we cannot forget f's pre- and post-condition (req and ens) altogether, since f may require some non-heap-related property of its argument (e.g., that $x > 0$), and provide some similar non-heap-related property of its result. So, in the returned function, we retain heap-invariant versions of req and ens,

Forgetting effects, in action. Our implementation of Quicksort in §3.4 is compact, but space inefficient. We would prefer to use an imperative, in-place quicksort on arrays, sorting an immutable sequence by copying it first to an array (an abstract type, logically equivalent to a mutable reference holding a sequence), sorting it efficiently, and then copying back. Despite the linear space overhead this is still a win (by at least a logarithmic factor) over the purely functional code. Using forget_ST, we can hide this optimization as an implementation detail, as shown below.

```
val qsort_arr: #a:Type → f:tot_ord a → x:array a → ST unit
  (requires (fun h → has h x))
  (ensures (fun h0 u h1 → has h1 x ∧ Seq.sorted f (sel h1 x)
                          ∧ permutation a (sel h0 x) (sel h1 x)))
  (modifies {x})

val qsort_seq : #a:Type → f:tot_ord a → x:seq a → ST (seq a)
  (requires (fun h → True))
  (ensures (fun h0 y h1 → Seq.sorted f y ∧ permutation a x y))
  (modifies {})
let qsort_seq f x =
  let x_ar = Array.of_seq x in qsort_arr f x_ar;
  let res = to_seq x_ar in Array.free x_ar; res

val qsort: a:Type → f:tot_ord a → s1:seq a
     → Dv (s2:seq a{Seq.sorted f s2 ∧ permutation a s1 s2})
let qsort f x = forget_ST (qsort_seq f) x
```

Using forget_ST, the top-level function calls qsort_seq, but has a stateless specification that reflects the functional correctness specification of the stateful function qsort_arr.

Haskell's runST Finally, it is interesting to compare this solution with runST :: (∀s. ST s a) → a (Launchbury and Peyton Jones 1994), which, according to the Haskell documentation, returns "the value computed by a state transformer computation. The forall ensures that the internal state used by the ST computation is inaccessible to the rest of the program."

Aside from F*'s ability to specify and verify functional correctness, our forget_ST is similar in spirit to runST. In F*, we can quantify over the initial heap in the logic, rather than using higher-rank polymorphism in the types. In Haskell, the types prevent the computation from returning an STRef s a, whereas in F*, a local reference can be returned by a Div† computation, but such a reference will be inaccessible to the caller (since the caller can no longer prove that the heap has that reference). Additionally, in F*, one can prove that a computation is conditionally in Div†. For example, if x then r := 0 else () is observationally stateless if x=false.

5. A new implementation of F*

Recall our five main goals for the redesign of F* (Section 1). The previous sections describe in some detail how F* now enables proving and programming, with pure and impure code. Underlying the usability of the language are three additional goals: (1) an expressive and efficient encoding of F*'s higher-order dependently typed logic into a simpler first-order logic provided by an SMT solver; (2) type-and-effect inference; and (3) interoperability with existing ML dialects on multiple platforms.

Given the space constraints of this format, we provide just a brief overview of F*'s SMT encoding and type inference algorithm (leaving a more detailed presentation as future work), and then report on the engineering of our implementation.

5.1 Type-and-effect inference and SMT encoding

Without good type inference, F* would be unusable. As the reader may have guessed from the examples, to a first approximation, F* provides type inference based on higher-order unification. Consider the fragment forget_ST (qsort_seq f) from the example in §4.3—type inference computes instantiations for the type-level functions b, req and ens. However, classical, higher-order unification, as implemented (to varying extents) in many other proof assistants, must be adapted for use in F*. There are two complications which our implementation addresses. First, type inference and effect inference are interleaved, since associated with each of our effects is a predicate transformer, whose structure depends on the inferred types. Second, F* has refinement subtyping: as is well known, unification and subtyping do not always mix well.

The typical strategy for dealing with subtyping in a unification-based type inference algorithm is to resort to bidirectional type-checking (Pierce and Turner 2000). However, after using bidirectional type-checking for five years, F* has moved on to a style where type inference gathers all (higher-order) unification and subtyping constraints from a term, and then solves these constraints together at the top-level. Solving constraints with a holistic view of the term produces much better results, and is robust to small, semantics-preserving code transformations (e.g., η-expansion, let-binding, and argument re-ordering), whereas local type-inference is not always as robust. This style of constraint solving is feasible because, in our setting, refinements of the same underlying type form a full lattice (where the join and meet are respectively logical disjunction and conjunction).

A solution to a set of typing constraints produces a logical guard ϕ:Type which, at the top level, may for instance reflect an implication between user-provided annotations and the inferred type and logical specification of a term. To prove that ϕ is valid, we encode it as an SMT theory. Our encoding is essentially a deep embedding of the syntax of F* terms, types and kinds into SMT terms, with interpretation functions giving logical meaning to the deeply-embedded terms in the SMT solver's logic. On top of this basic deep embedding structure, we implement several optimizations, such as shallow embeddings of commonly used connectives like ∀ and ∧ , by essentially inlining the interpretation functions on certain deeply embedded terms. The other main optimization is in the encoding of recursive functions and types—we implement various strategies to control the number of unrollings of recursively defined terms and types that the solver is allowed to explore.

If the SMT solver fails to prove a goal, we translate the returned counterexample model into a meaningful error message for the user, who will typically try to break up the goal into smaller lemmas. At worst, if the SMT solver remains unsuccessful, the user still has the option of manually providing a constructive proof. On the other hand, if the solver succeeds, there remains the question of whether it (and, perhaps more importantly, our encoding) can be trusted — we are exploring a certification pipeline to address this issue, building on our prior work on self-certification by proof witnesses from SMT solvers (Armand et al. 2011).

5.2 Engineering the F* compiler

F* is an open source project hosted at `https://github.com/FStarLang`. The compiler is itself programmed in about 21,000 lines of F* code. Most of the complexity of the compiler lies in the modules implementing type inference and SMT encoding. In comparison to prior versions, our new implementation is significantly

Example	Verification Goal	Effects	F* LOC	Time(s)
RB tree	Insert correctness	PURE	327	10.4
Quicksort	Correctness	PURE	90	6.6
Counters	Stateful invariant[a]	STATE	72	5.2
Wysteria	Security[b]	STATE	179	3.6
ACL	Security[c]	ALL	232	5.2
Encrypt	Security[d]	ALL	270	5.5
$\lambda_\to$[e]	Type Soundness	PURE	1708	186.5
λ_ω		PURE	1413	22.6
System F$_\omega$		PURE	2055	55.9
$\lambda_\to$[f]	W. Normalization	PURE	260	5.2

[a] Created counters return even values, while hiding their local state

[b] An EDSL encoding a type system for secure multi-party computations, including Yao's Millionaires' Problem (Rastogi et al. 2014)

[c] File access control with dynamic access granting/revocation

[d] Secrecy of multi-key symmetric encryption scheme

[e] Five variants with different binder (named, xde Bruijn), substitution (point, parallel), and reduction (CBV, strong) schemes. Stats are cumulative.

[f] Hereditary substitutions with de Bruijn indices

Figure 2. A sampling of verified F* examples

less complex (the prior version had bloated to over 100,000 lines of F# code), even though it implements a more expressive language.

The compiler makes extensive use of effects (e.g., unification is imperative and exceptions are used heavily throughout), and is written idiomatically in a shared subset of F* and F#. We have yet to prove any deep properties about our implementation, aside from standard type safety — yet we are now well-positioned to start verifying it. Regardless, our experience developing the compiler is good validation that our new design, despite the addition of effects, retains the flavor of programming in ML at a non-trivial scale.

While we rely heavily on F# tools (such as the Visual Studio IDE) and external libraries (the .NET platform) for bootstrapping, we also offer a new F* standard library with support for lists, strings, sequences, arrays, sets, bytes, basic networking, limited I/O, and some cryptographic primitives. In many cases, these libraries are verified, providing a suite of lemmas for programmers to use in other developments.

The compiler is designed to support several backends. Currently, our main backend targets OCaml, as it requires little compilation effort and is able to produce binaries for many platforms. We rely on the OCaml Batteries package to efficiently implement the F* standard library. Calls to the F* library get rewritten by each backend to take advantage of the native representations and library functions available in the target language. To bootstrap, we first build the compiler using F#, then we run the generated binary on its own source files, emitting OCaml code through the backend, which we finally run through the OCaml compiler, targeting several platforms. With this approach, we offer binary packages compiled from OCaml for Windows, Linux and MacOS from the F* homepage.

In addition to the 21,000 lines of compiler source code in F*, our repository also contains more than 10,000 lines of verified example F* code as part of our regression suite. While this is already a significant figure, our current F* examples do not achieve the scale of the most impressive applications of its prior versions (Bhargavan et al. 2013; Fournet et al. 2013); indeed, there remains tens of thousands of lines of old F* code to port forward. Figure 2 shows a table of significant F* examples, including the verification goal, the effects used in the program, the line count (including comments), and the verification time on a workstation equipped with a Xeon E5-1620 CPU at 3.6GHz and 16GB of RAM.

$$
\begin{array}{rcl}
c & ::= & () \mid 0 \mid 1 \mid -1 \mid \ldots \mid \ell \mid\, ! \mid\, := \;\mid \mathsf{sel} \mid \mathsf{upd} \\
v & ::= & c \mid \lambda x{:}t.\, e \mid \mathsf{let\,rec}\,(f^d{:}t)\; x = e \\
e & ::= & x \mid v \mid e_1\; e_2 \mid \mathsf{if}_0\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \\
T & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{ref\;int} \mid \mathsf{heap} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{and} \mid \mathsf{or} \mid \mathsf{not} \\
 & \mid & \mathsf{impl} \mid \mathsf{forall} \mid \mathsf{forall}_k \mid \mathsf{eq} \mid \mathsf{eq}_k \\
t, \phi & ::= & \alpha \mid T \mid x : t_1 \to M\; t_2\; \phi \mid \lambda \alpha{:}k.t \mid \lambda x{:}t.t' \mid t\; t' \mid t\; e \\
k & ::= & \mathsf{Type} \mid \alpha{:}k \to k' \mid x{:}t \to k'
\end{array}
$$

Figure 3. Syntax of μF*

6. Metatheory

This section describes the formalization of μF*, a small core calculus capturing the essence of F*. μF* features dependent types and kinds, type operators, subtyping, sub-kinding, semantic termination checking, and statically-allocated first-order state. The μF* calculus only has a fixed two-point lattice of effects with only PURE and ALL, a single effect combining state and non-termination, at the top. Our main results are partial correctness for the program logic for ALL computations; weak normalization for the PURE fragment; and total correctness of the logic for PURE computations. The online appendix contains all the definitions and proofs. We also include online the comprehensive formal definitions of F* corresponding to the system we have implemented—however, its metatheory has not yet been fully studied.

Figure 3 presents the syntax of μF*. Constants (c) include unit, integers, memory locations (ℓ), operations for reading (!) and updating memory (:=) as well as corresponding symbols (sel and upd) for reasoning about memory at the logical level. Beyond constants and the lambda calculus, expressions include a recursion construct $\mathsf{let\,rec}\,(f^d{:}t)\; x = e$, where the optional metric d is an arbitrary pure function used for termination checking. We also include a form for testing whether an integer is zero. Types (t) include dependent function types ($x{:}t_1 \to M\; t_2\; \phi$) enhanced with predicate transformer specifications (ϕ), type variables (α), lambdas for type-indexed ($\lambda \alpha{:}k.t$) and expression-indexed type operators ($\lambda x{:}t.t'$), as well as the corresponding application forms ($t\; t'$ and $t\; e$). The main use of type-level lambdas and application is for representing the predicate transformers (ϕ) in function specifications. For the same purpose we include typed classical logical connectives as type constants (T); for instance $\forall x{:}t.\phi$ is represented as the type "$\mathsf{forall}\; (\lambda x{:}t.\phi)$", where $\mathsf{forall}{:}\alpha{:}\mathsf{Type} \to (\alpha \to \mathsf{Type}) \to \mathsf{Type}$. Constructively, the $\to$ type is for universal quantification, as usual.

We give μF* expressions a standard CBV operational semantics. Reduction has the form $(H, e) \to (H', e')$, for heaps H and H' mapping locations to integers. We additionally give a liberal reduction semantics to μF* types ($t \rightsquigarrow t'$) that includes CBV and CBN, and that also evaluates pure expressions ($e \to e'$). The type system considers types up to conversion.

Figure 4 lists all the expression typing rules of μF* that have not already been shown earlier (except the trivial rule for typing constants). The rules for variables and λ-abstractions are unsurprising. In each case, the expression has no immediate side-effects; we thus use Tot to mark these expressions as unconditionally pure.

Rule (T-App) is more interesting: first, the effect M of the function application is an upper-bound on the effects of computing the function e_1 and its argument e_2 as well as on the effect of executing the function body. Effects can be freely lifted from PURE to ALL using the (S-Comp) subtyping rule in Figure 5. Then, while the first two preconditions of (T-App) are standard, via the third one, $\Gamma \vdash t'[e_2/x] : \mathsf{Type}$, we ensure that if x appears in t' then e_2 is pure. This restriction on dependent type application is necessary for soundness and is much more liberal than the value-

$$\text{(T-Var)}\ \frac{\Gamma(x) = t \qquad \Gamma \vdash t : \mathsf{Type}}{\Gamma \vdash x : \mathsf{Tot}\ t} \qquad \text{(T-Abs)}\ \frac{\Gamma \vdash t : \mathsf{Type} \qquad \Gamma, x{:}t \vdash e : M\ t\ \phi}{\Gamma \vdash \lambda x{:}t.\, e : \mathsf{Tot}\ (x{:}t \to M\ t\ \phi)}$$

$$\text{(T-App)}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : M\ (x{:}t \to M\ t'\ \phi)\ \phi_1 \\ \Gamma \vdash e_2 : M\ t\ \phi_2 \qquad \Gamma \vdash t'[e_2/x] : \mathsf{Type}\end{array}}{\Gamma \vdash e_1\ e_2 : M\ (t'[e_2/x])\ (\mathsf{bind}_M\ \phi_1\ (\lambda f.\, \mathsf{bind}_M\ \phi_2\ (\lambda x.\, \phi)))}$$

$$\text{(T-If0)}\ \frac{\Gamma \vdash e_0 : M\ \mathsf{int}\ \phi_0 \qquad \Gamma \vdash e_1 : M\ t\ \phi_1 \qquad \Gamma \vdash e_2 : M\ t\ \phi_2}{\Gamma \vdash \mathsf{if}_0\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : M\ t\ (\mathsf{ite}_M\ \phi_0\ \phi_1\ \phi_2)}$$

$$\text{(T-Sub)}\ \frac{\Gamma \vdash e : M'\ t'\ \phi' \qquad \Gamma \vdash M'\ t'\ \phi' <: M\ t\ \phi \Leftarrow \phi'' \qquad \Gamma \models \phi''}{\Gamma \vdash e : M\ t\ \phi}$$

Figure 4. Remaining typing rules of μF*

$$\text{(Sub-Fun)}\ \frac{\Gamma \vdash t' <: t \Leftarrow \phi'' \qquad \Gamma, x{:}t' \vdash M\ s\ \phi <: M'\ s'\ \phi' \Leftarrow \phi'''}{\Gamma \vdash (x{:}t \to M\ s\ \phi) <: (x{:}t' \to M'\ s'\ \phi') \Leftarrow \phi'' \wedge_{M'} \forall x{:}t'.\phi'''}$$

$$\text{(Sub-Conv)}\ \frac{\Gamma \models t_1 = t_2 \qquad \Gamma \vdash t_2 : \mathsf{Type}}{\Gamma \vdash t_1 <: t_2 \Leftarrow \mathsf{true}}$$

$$\text{(S-Comp)}\ \frac{M \leqslant M' \qquad \Gamma \vdash t <: t' \Leftarrow \phi'' \qquad \Gamma \vdash \phi' : K'_M(t')}{\Gamma \vdash M\ t\ \phi <: M'\ t'\ \phi' \Leftarrow \phi'' \wedge \mathsf{down}'_M(\phi' \Rightarrow'_M \mathsf{lift}_M^{M'}\ \phi)}$$

Figure 5. Subtyping rules of μF*

dependency restriction (and corresponding requirement of A-normal form (Flanagan et al. 1993)) in old F* and similar systems, e.g., liquid types. Instead of requiring programs to be A-normal, we monadically sequence their predicate transformers in the conclusion.

Rule (T-If0) connects the predicate transformers using an ite_M operator, which is defined as follows for our two effects:

$$\begin{array}{l}\mathsf{ite}_{\mathsf{PURE}}\ \phi_0\ \phi_1\ \phi_2\ \mathsf{post} = \\ \quad \mathsf{bind}_{\mathsf{PURE}}\ \phi_0\ (\lambda i.\ i{=}0 \Rightarrow \phi_1\ \mathsf{post}\ \wedge\ i{\neq}0 \Rightarrow \phi_2\ \mathsf{post})\end{array}$$

$$\begin{array}{l}\mathsf{ite}_{\mathsf{ALL}}\ \phi_0\ \phi_1\ \phi_2\ \mathsf{post} = \\ \quad \mathsf{bind}_{\mathsf{ALL}}\ \phi_0\ (\lambda i\ h.\ i{=}0 \Rightarrow \phi_1\ \mathsf{post}\ h\ \wedge\ i{\neq}0 \Rightarrow \phi_2\ \mathsf{post}\ h)\end{array}$$

Subsumption (T-Sub) connects expression typing to the subtyping judgment for computations, which has the form $\Gamma \vdash M'\ t'\ \phi' <: M\ t\ \phi \Leftarrow \phi''$. The formula ϕ'' gathers the necessary conditions on subtyping and is required to be logically valid in environment Γ. This is captured by a separate logical validity judgment $\Gamma \models \phi''$ that gives a semantics to the typed logical constants and equates types and pure expressions up to convertibility—this is the judgment that our implementation encodes in an SMT solver. The subtyping judgment for computations only has the (S-Comp) rule, listed in Figure 5; it allows lifting a PURE computation to the ALL effect, strengthening the predicate transformer, and weakening the type using a mutually inductively defined judgment $\Gamma \vdash t <: t' \Leftarrow \phi$. This judgment includes a structural rule (Sub-Fun) and a rule for type conversion via the logical validity judgment (Sub-Conv).

Meta-theorems We prove the following partial correctness theorem for ALL computations stating that a well-typed ALL expression is either a value that satisfies all post-conditions consistent with its predicate transformer, or it steps to another well-typed ALL expression. For PURE expressions, we prove the analogous property, but in the total correctness sense.

Theorem 1 (Partial Correctness of ALL)**.** *If $\cdot \vdash e : \mathsf{ALL}\ t\ \phi$ then for all p and H s.t. $\cdot \vdash p : \mathsf{Post}_{\mathsf{ALL}}(t)$ and $\cdot \models \phi\ p\ \mathsf{asHeap}(H)$, either e is a value and $\cdot \models p\ e\ \mathsf{asHeap}(H)$, or $(H, e) \to (H', e')$ such that $\cdot \vdash e' : \mathsf{ALL}\ t\ \phi'$ and $\cdot \vdash \phi'\ p\ \mathsf{asHeap}(H')$.*

Theorem 2 (Total Correctness of PURE)**.** *If $\cdot \vdash e : \mathsf{PURE}\ t\ \phi$ then for all p s.t. $\cdot \vdash p : \mathsf{Post}_{\mathsf{PURE}}(t)$ and $\cdot \models \phi\ p$, we have $e \to^* v$ such that v is a value, and $\cdot \models p\ v$.*

Our total correctness result relies on a weak normalization theorem for PURE terms—this is proven using a well-founded induction on the ambient, partial ordering on terms by exhibiting an accessibility predicate. The theorem only considers the reduction of terms in a consistent context (i.e., $\Gamma \models \mathsf{false}$ is not derivable), and, as such, excludes strong reduction under binders as well. Additionally, we require the validity judgment to be consistent with respect to the ordering.

Theorem 3 (Weak normalization of PURE)**.** *If Γ is consistent, $\Gamma \vdash e : \mathsf{PURE}\ t\ \phi$, and $\exists p.\Gamma \models \phi\ p$; then e is weakly normalizing.*

7. Related work

Integrating dependent types within a full-fledged, effectful programming language is a long-standing goal. An early effort in pursuit of this agenda was Cayenne (Augustsson 1998) which integrated dependent types within a Haskell-like language. Cayenne intentionally permitted the use of non-terminating code within types, making it inconsistent as a logic. Nevertheless, Cayenne was able to check many useful program properties statically. Subsequent efforts aim to preserve the consistency of the logic in the presence of effects including but not limited to non-termination. Unsurprisingly, there are three main strands:

Clean-slate designs We have compared with Trellys/Zombie and Aura. Another notable clean-slate design is Idris (Brady 2013), which provide both non-termination and an elegant style of algebraic effects. A syntactic termination check is also provided.

Adding dependency to an effectful language This camp includes the predecessors of F*, notably Fine (Swamy et al. 2010), F7 (Bhargavan et al. 2010), and F5 (Backes et al. 2014), all of which add value-dependent types to a base ML-like language. We have already discussed liquid types. A recent variation by Vazou et al. (2014) adds liquid types to Haskell, a call-by-name language—in this setting, non-values may appear in refinements, but they are still uninterpreted. For soundness, Liquid Haskell provides a termination check, which is less expressive than ours (being based only on the integer ordering). Liquid Haskell does not have any effects.

Adding effects to a type-theory based proof assistant Nanevski et al. (2008) develop Hoare type theory (HTT) as a way of extending Coq with effects. The strategy there is to provide an axiomatic extension of Coq with a single catch-all monad (like monadic-F*, which builds on HTT) in which to encapsulate imperative code. Tools based on HTT have been developed, notably Ynot (Chlipala et al. 2009). This approach is attractive in that one retains all the tools for specification and interactive proving from Coq. On the downside, one also inherits Coq's limitations, e.g., the syntactic termination check and lack of SMT-based automation.

Non-syntactic termination checks Most dependent type theories rely crucially on normalization for consistency. Weak normalization is usually sufficient, as is the case in Coq, which is not normalizing under call-by-value; Pure F* is also only weakly normalizing—strong reductions (under binders) are not included. Coq's syntactic termination check is known to be brittle. So, many researchers have been investigating more semantic approaches to termination in type theories. Agda (which only includes pure functions) offers two non-syntactic termination checkers. The first one is based on fœtus (Abel 1998), and tries to discover a suitable lexicographic ordering on the arguments of mutually-defined functions automatically. Contrary to fœtus, our termination checker does not aim to find an ordering automatically (although well-chosen defaults mean that the user often has to provide no annotation); nonetheless, our check is far more flexible, since it is not restricted to a structural decreasing of arguments, but the decreasing of a *measure* applied to the arguments. The second one is based on sized types (Abel 2007; Barthe et al. 2004), where the size on types approximates the depth of terms. In contrast, in F*, the measures are defined by the user and are first-class citizens of the language and can be reasoned about using all its reasoning machinery. In both Agda and Coq, one can also instrument functions with *accessibility predicates*, independently proving that they are well-founded (Bove 2001). We can see F* as having a single ambient accessibility predicate on all terms, that can be augmented currently only by adding axioms. Allowing user-defined well-founded orderings while retaining good automation seems non-trivial and we leave it for future work.

Semantic model for higher-order refinements Barthe et. al. (2015) recently introduced a relational type system with higher-order refinements and showed this system sound with respect to a denotational semantics. The authors suggest that old F* and other refinement based languages would have a hard time soundly supporting semantic subtyping, because the logic does not embed any guard against non-terminating logical symbols. The non-termination monad Barthe et. al. use to avoid logical inconsistency in their setting is in a sense similar to our Tot effect.

SMT-based program verifiers Software verification frameworks, such as Why3 (Filliâtre et al. 2014) and Dafny (Leino 2013), also use SMT solvers to verify the logical correctness of first-order programs. Unlike us, they do not rely on dependent types. F*'s ability to reason about definitions by computing with an SMT solver is related to the work of Amin et al. (2014)—however, many of the details differ, e.g., we provide a means for the user to control the number of unrollings of a function, which their approach lacks.

Looking back, looking ahead Our redesign was motivated primarily by our desire to use F* to build and verify more complex software (a high-efficiency implementation of TLS is high among our priorities). Old, value-dependent F* had become too cumbersome for the task. Moving towards full dependency, using a type-and-effect system for consistency, results in a system that is (arguably) more uniform, without ad hoc restrictions based on kinds or qualifiers, but still not substantially more complex theoretically. Ultimately, we find that new F* is more expressive and pleasant for programming and proving—particularly when backed by practical type-and-effect inference. Value dependency has served us well—it is technically simple, and many of us have gotten good mileage out of it, using it to build several useful verified artifacts. But its time is up.

References

A. Abel. foetus – termination checker for simple functional programs. Programming Lab Report 474, LMU München, 1998.

A. Abel. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types*. PhD thesis, LMU München, 2007.

R. Adams. Formalized metatheory with terms represented by an indexed family of types. *TYPES*, 2006.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. *CSL*.

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

M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. *CPP*, 2011.

L. Augustsson. Cayenne—a language with dependent types. *ICFP*, 1998.

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.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. *POPL*. 2015.

N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *JAR*, 49(2):141–159, 2012.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. *POPL*, 2010.

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

A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, 2001.

E. Brady. Programming and reasoning with algebraic effects and dependent types. *ICFP*, 2013.

C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. *POPL*, 2014.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. *ICFP*, 2009.

T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

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

J.-C. Filliâtre, L. Gondelman, and A. Paskevich. The spirit of ghost code. *CAV*, 2014.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *PLDI*, 1993.

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

T. Freeman and F. Pfenning. Refinement types for ML. *PLDI*, 1991.

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI I, 1972.

B. Jacobs. Dijkstra monads in monadic computation. *CMCS*, 2014.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. *ICFP*, 2008.

M. Kaufmann and J. Moore. ACL2: an industrial strength version of Nqthm. *COMPASS*, 1996.

J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. *PLDI*, 1994.

D. Leijen. Koka: Programming with row polymorphic effect types. *MSFP*, 2014.

K. Leino. Automating theorem proving with SMT. *ITP*, 2013.

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

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *POPL*, 1988.

D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal verifier user manual. Technical report, Stanford University, 1979.

J. McCarthy. Towards a mathematical science of computation. *IFIP Congress*, 1962.

E. Moggi. Computational lambda-calculus and monads. *LICS*, 1989.

A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

B. C. Pierce and D. N. Turner. Local type inference. *TOPLAS*, 22(1):1–44, 2000.

A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. *S&P (Oakland)*, 2014.

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, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. *ESOP*, 2010.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. *ICFP*, 2011.

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.

P. Wadler and P. Thiemann. The marriage of effects and monads. *TOCL*, 4(1): 1–32, 2003.