Dependent Types and Multi-Monadic Effects in $F^\star$

Nikhil Swamy[1] Cătălin Hriţcu[2] Chantal Keller[1,3] Aseem Rastogi[4]
Antoine Delignat-Lavaud[2,5] Simon Forest[2,5] Karthikeyan Bhargavan[2]
Cédric Fournet[1,3] Pierre-Yves Strub[6] Markulf Kohlweiss[1] Jean-Karim Zinzindohoue [2,5]

[1]Microsoft Research [2]Inria [3]MSR-Inria [4]UMD [5]ENS Paris [6]IMDEA Software Institute

Abstract

We present $F^\star$, a new language that works both as a proof assistant as well as a general-purpose, verification-oriented, effectful programming language.

In support of these complementary roles, $F^\star$ is a dependently typed, higher-order, call-by-value language with *primitive* effects including state, exceptions, divergence and IO. Although primitive, programmers choose the granularity at which to specify effects by equipping each effect with a *monadic*, predicate transformer semantics. $F^\star$ uses this to efficiently compute weakest preconditions and discharges the resulting proof obligations using a combination of SMT solving and manual proofs. Isolated from the effects, the core of $F^\star$ is a language of pure functions used to write specifications and proof terms—its consistency is maintained by a semantic termination check based on a well-founded order.

We evaluate our design on more than 55,000 lines of $F^\star$ we have authored in the last year, focusing on three main case studies. Showcasing its use as a general-purpose programming language, $F^\star$ is programmed (but not verified) in $F^\star$, and bootstraps in both OCaml and F#. Our experience confirms $F^\star$'s pay-as-you-go cost model: writing idiomatic ML-like code with no finer specifications imposes no user burden. As a verification-oriented language, our most significant evaluation of $F^\star$ is in verifying several key modules in an implementation of the TLS-1.2 protocol standard. For the modules we considered, we are able to prove more properties, with fewer annotations using $F^\star$ than in a prior verified implementation of TLS-1.2. Finally, as a proof assistant, we discuss our use of $F^\star$ in mechanizing the metatheory of a range of lambda calculi, starting from the simply typed lambda calculus to F_ω and even $\mu F^\star$, a sizeable fragment of $F^\star$ itself—these proofs make essential use of $F^\star$'s flexible combination of SMT automation and constructive proofs, enabling a tactic-free style of programming and proving at a relatively large scale.

1. Introduction

Proving and programming are inextricably linked, especially in dependent type theory, where constructive proofs are just programs. However, not all programs are proofs. Effective programmers routinely go beyond a language of pure, total functions and use features like non-termination, state, exceptions, and IO—features that one does not usually expect in proofs. Thus, while Coq (The Coq development team) and Agda (Norell 2007) are functional programming languages, one does not typically use them for general-purpose programming—that they are implemented in OCaml and Haskell is a case in point. Outside dependent type theory, verification-oriented languages like Dafny (Leino 2010) and WhyML (Filliâtre and Paskevich 2013) provide good support for effects and semi-automated proving via SMT solvers, but have logics that are much less powerful than Coq or Agda, and only limited support (if at all) for higher-order programming.

We aim for a language that spans the capabilities of interactive proof assistants like Coq and Agda, general-purpose programming languages like OCaml and Haskell, and SMT-backed semi-automated program verification tools like Dafny and WhyML. This language would provide the nearly arbitrary expressive power of a logic like Coq's, but with a richer, effectful dynamic semantics. It would provide the flexibility to mix SMT-based automation with interactive proofs when the SMT solver times out (not uncommonly when working with rich theories and quantifiers). And it would support idiomatic higher-order, effectful programming with the predictable, call-by-value cost model of OCaml, but with the encapsulation of effects provided by Haskell.

Although such a language may seem beyond reach, several research groups have made significant progress, targeting various pieces of this agenda. For example, with Hoare Type Theory, Nanevski et al. (2008) extend Coq with support for interactive proofs of imperative programs. With Trellys and Zombie, Casinghino et al. (2014) design new dependently typed languages for interactive proving and programming while accounting for just a single effect, non-termination. With $F^\star$, Swamy et al. (2013a) provide SMT-based automated proving for an ML-like programming language, but lack the ability to do interactive proofs. Still, as far as we are aware, currently no tool enables the mixture of proving and general-purpose programming with the degree of automation that we desire.

Building on this prior work, we present $F^\star$, a new candidate in pursuit of this goal, that straddles the threefold roles of programming language, program-verification tool, and proof assistant. We use $F^\star$ to write effectful programs; to specify them (to whatever extent necessary) within its functional core using dependent and refinement types; and to verify them using an SMT solver that automatically discharges proofs. Where proof obligations exceed the capabilities of SMT solving, interactive proofs can be provided within the language. Full verification is not mandatory in $F^\star$—the language encourages a style in which programs are verified incrementally. Programs with ML types are easily type-checked syntactically, while more precise specifications demand deeper proofs. After type-checking, $F^\star$ programs can be compiled to OCaml or F# for execution.

Validating $F^\star$'s capabilities for programming, we have bootstrapped it using about 20,500 lines of $F^\star$ (in addition to a few platform-specific libraries in OCaml and F#). We have also used $F^\star$ to verify key parts of other complex, effectful programs, such as the cryptographic protocols underlying the TLS-1.2 standard (Dierks and Rescorla 2008). Evaluating $F^\star$ as a proof assistant, we have formalized several lambda calculi, and have even used it to mechanize part of the metatheory of $\mu F^\star$, a sizable fragment of $F^\star$. While it is premature to claim that $F^\star$ is simultaneously a replacement for, say, Coq, OCaml and Dafny, our initial experience is encouraging—we know of no other language that supports semi-automated proving and general-purpose programming to the same extent as $F^\star$. Next, we summarize a few key features of the language.

Primitive effects in a lattice of monads Enabling F⋆ to play its varied roles is a design that structures the language around an extensible lattice of monadic effects. F⋆'s runtime system provides primitive support for all the effects provided by its compilation targets. Although available primitively, programmers can specify the semantics of each effect using several monads of weakest-precondition predicate transformers.

The granularity at which to model effects is the programmer's choice, as long as (1) a distinguished PURE monad isolates pure computations from all other effects; (2) a monad GHOST encapsulates purely specificational computations (for erasure); and, (3) a monad ALL provides a semantics to all the primitive effects together. Within these bounds, the programmer has the freedom to refine the effects of the language as she sees fit, arranging them in a join semi-lattice. By default, the lattice F⋆ provides is shown below (with an implicit top element, $\top$).

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

PURE computations are at the very bottom. Using the PURE monad, programmers write pure, recursive functions. This monad forms F⋆'s logical core. It soundness depends crucially on a novel semantic termination check based on a well-founded order. The DIV effect is for possibly divergent code; STATE for stateful computations; and EXN for programs that may raise exceptions. Each edge in the lattice corresponds to a monad morphism. Using these morphisms, the F⋆ type-checker implicitly lifts specifications in one monad to another. Other arrangements of the effects are possible (e.g., splitting readers and writers from STATE) depending on the needs of an application, as long as the semantics of each user-defined effect is compatible with the semantics of ALL the effects.

Expressive specifications with dependent, refinement types Specifications in F⋆ are expressed using dependent types—types indexed by arbitrary total expressions, with type-level computation defining an equivalence relation on types. In addition to predicate transformers, programmers use indexed inductive types and refinement types (types of the form x:t{ϕ}, the sub-type of t restricted to those expressions e:t that validate the logical formula ϕ). Refinement types provide a natural notion of proof irrelevance and promote code re-use via subtyping.

Type-and-effect inference, with semi-automated proving Given a program and a user-provided specification, F⋆ infers a type and effect for it, together with a predicate transformer that fully captures the semantics of that computation. It then generates proof obligations to show that the specification is compatible with the inferred predicate transformer. These proof obligations can be discharged semi-automatically using a combination of SMT solving and user-provided proof terms, the latter using the GHOST monad.

Summary of contributions This paper reports on the following technical advances:

(1) We present the design of a new programming language, F⋆, with a dependent type-and-effect system, based on a new, extensible, multi-monadic predicate-transformer semantics (introduced in §2, and covered throughout).

(2) To ensure that F⋆'s core language of pure functions is normalizing, we employ a novel semi-automatic semantic termination checker based on a well-founded relation (§3).

(3) We illustrate the expressiveness and flexibility of F⋆'s multi-monadic design using a series of programming examples, showcasing an encoding of *hyper-heaps*, a novel model of the heap that provides lightweight support for separation and framing for stateful verification (§5).

(4) We have formalized a core calculus μF⋆: a substantial fragment of F⋆, distilling the main ideas of the language. We prove syntactic type soundness, which implies partial correctness of the program logic. Additionally, we use logical relations to prove consistency and weak normalization of a fragment of μF⋆ with only pure computations (§6).

(5) We have developed a full-fledged open source implementation of F⋆ in F⋆, and report on our experience using it. As a proof assistant, we use F⋆ to formalize several lambda calculi, including μF⋆ (§3 and §6). As a programming language, in addition to its use as the implementation language of F⋆, we report on its use in the re-design and verification of key portions of an existing implementation of TLS-1.2 (§5). In all cases, the expressiveness of F⋆'s type system, the flexibility afforded by its user-configurable effects, the semantic termination check, and the proof automation helped make verification feasible at scale.

Supplementary material The F⋆ toolchain is open source, and binary packages are available for all major platforms. We provide an interactive editor mode in addition to the batch-mode compiler. An extensive interactive, online tutorial presents many examples and discusses details of the language beyond the limits of this paper. An online appendix of this paper includes the definitions and proofs for μF⋆ and the definitions of a larger calculus capturing all the features of F⋆. By necessity, the examples in this paper are greatly simplified versions of larger F⋆ developments available online. All of this material is available in non-anonymous form in a supplement associated with our submission—we refer to it throughout the paper.

2. Dijkstra monads, generalized in F⋆

One point of departure for the design of F⋆ is the work of Swamy et al. (2013b), who propose the *Dijkstra monad* as a way of structuring and inferring specifications for higher-order stateful programs. In this section, we briefly review their proposal, note several shortcomings, and discuss how these are alleviated by F⋆'s generalized notion of a lattice of Dijkstra monads.

We intend for this section to serve as a high-level introduction to the design of F⋆. While the details are also important, we suggest that a reader not already familiar with monads and dependent types pay attention mainly to the high-level points in the prose.

2.1 Background: A single Dijkstra monad

Dijkstra (1975) defines the semantics of a program in terms of its weakest pre-condition, a function that transforms a predicate on the outcome of a computation to a predicate on that computation's input. In the context of a dependently-typed language, Swamy et al. (2013b) observe that these weakest pre-condition predicate transformers form a monad at the level of types (rather than at the level of computations).

To illustrate this point, consider the semantics of stateful computations that may raise exceptions. The outcome of such a computation is a possibly exceptional result and a final state, whereas its input is an initial state. Weakest pre-conditions for such computations, as usual, transform predicates on the outcome (aka post-conditions) to predicates on the input (aka pre-conditions).

Using the notation of F⋆ (which we introduce formally later), we can express these weakest pre-conditions as follows, where state is the type of the program state; either a string represents either a normal result Inl (v:a) or an error Inr (msg:string); and Type is the kind of types (and propositions). We define WP a, the signature of a weakest pre-condition predicate transformer for stateful, exceptional computations that may return a-typed results, i.e., wp:WP a is a function that transforms a post-condition predicate q:Post a into a pre-condition p:Pre. It may be useful to some readers to think of WP a as a continuation monad.

```
Post (a:Type) = either a string → state → Type
Pre = state → Type
WP (a:Type) = Post a → Pre
```

Viewing WP a as a monad, Swamy et al. define two combinators return and bind. The weakest pre-condition of a pure computation returning x:t is return t x—to prove any post, it suffices to prove post (Inl x) s, for the normal result x and the (unchanged) initial state of the computation.

```
return (a:Type) (x:a) : WP a = fun (post:Post a) (s:state) → post (Inl x) s
```

The weakest pre-condition of the sequential composition of two computations is bind t1 t2 wp1 wp2: when run in s0, if the first computation produces state s1 and either (1) raises an exception Inr msg, in which case one must prove the post-condition immediately; or (2) returns normally with Inl v, in which case one runs the second computation with v and s1, proving the post-condition of its result.

```
bind (a:Type) (b:Type) (wp1:WP a) (wp2 : (a → WP b)) : WP b =
  fun (post:Post b) (s0:state) → wp1 (fun x s1 →
    match x with
      | Inr msg → post (Inr msg) s1
      | Inl v → wp2 v post s1) s0
```

Swamy et al. relate a computation to its semantics by introducing computation types M t wp, where M is itself a monad parameterized by its result type t (as usual) and additionally indexed by wp: WP t, its monadic weakest-precondition predicate transformer, i.e., M is a monad-indexed monad. Informally, in a total correctness setting, given a computation e : M t wp, and a post-condition q, if e is run in a state s satisfying wp q s, then e produces a result v and state s' satisfying q v s'. This technique is reminiscent of the parameterized monad of Atkey (2009) and the Hoare monad of Nanevski et al. (2008), who use computation types H p t q to describe computations with pre-condition p, t-typed result, and post-condition q.

2.2 Some limitations of a single Dijkstra monad

The Dijkstra monad has several benefits, e.g., type inference is built into the weakest pre-condition calculus. However, we observe that using just a single monad for all computations also has significant downsides. Using a single monad to describe all computations is akin to using a single type to describe all values. A uni-effect system, arguably adequate from a semantic perspective, is too coarse for practical purposes, particularly in a verification-oriented language.

Non-modular specifications. With just a single monad, even in effect-free code, or in code that only uses some effects, one must write specifications that mention all the effectful constructs. For example, with only a single monad at one's disposal, even a pure computation $1+2$ is specified as M int (fun post h → post (Inl 3) h), i.e., one explicitly states that $1+2$ returns 3 without raising an exception and does not modify the state. Similarly, the computation !x, when x:ref t would be typed as M t (fun post h → post (Inl (h[x])) h) meaning that it returns the value of x dereferenced in the current heap; that it does not raise an exception; and that it leaves the state unmodified. This is cumbersome from a notational perspective and non-modular. While the notational overhead may be minimized by adopting various abbreviations, the non-modularity is pervasive: establishing that !x does not modify the state and raises no exceptions requires a logical proof about its predicate transformer. Worse, while one can prove (via its predicate transformer) that $1+2$ does not mutate the state and does not raise exceptions, that it does not read the state is not evident from its specification. Indeed, to prove that it does not read the state would require moving to a richer logic, using, for example, separation logic, or a logic of program equivalence. Likewise, proving that !x does not internally modify the state before restoring it is also difficult.

Combinatorial explosion of VCs. Consider sequentially composing n computations $e_1, \ldots, e_n$. When all these computations are typed in a single monad M of state and exceptions, the verification condition (VC) built by repeated applications of bind contains n control paths, rather than just one. In the worst case where each subcomputation may indeed raise an exception, one cannot do much better. Unfortunately, even in the common case where, say, many of the e_i are exception-free, using a single monad produces VCs with a number of paths equal to the worst case. When combined with conditionals and exception handlers, this results in an exponential explosion of VCs, even for simple, pure code. Proving that many of these paths are infeasible requires building and then performing logical proofs over needlessly enormous VCs.

2.3 Multiple Dijkstra monads in F⋆

We would prefer instead to type a computation e in a monad suited specifically to the effects exhibited by e, and no others. For example, pure expressions like $1+2$ should be typed using the PURE monad, whose predicate transformers make no mention of exceptions or state; !x in, say, a Reader monad which makes no mention of exceptions or the output state. With multiple monads, specifications are compact and modular; infeasible paths in verification conditions are pruned at the outset without needless logical proof; and many properties (e.g., state independence) can be established with simple syntactic arguments. Of course, multiple monads are a strict generalization: when syntactic arguments are insufficient, one can always fall back on detailed logical proofs. F⋆'s lattice of Dijkstra monads enables all of this, as described next.

Rather than committing to a single Dijkstra monad at the outset, F⋆ provides a lattice of such monads, each describing the semantics of some subset of all the effects provided by the language. For the moment, as in the previous section, we focus on state and exceptions as the only effects (returning to non-termination later). We define three Dijkstra monads, PURE.WP, STATE.WP, EXN.WP, and show how they can be combined piecewise to produce ALL.WP, a single monad (identical to the monad WP defined in §2.1) that captures the semantics of all the effects together.

PURE.WP To define the semantics of pure computations, we introduce (below) a Dijkstra monad PURE.WP. A weakest pre-condition for pure computations with an a-typed result transforms pure post-conditions (predicates on a) to pre-conditions (propositions). The semantics of returning a value requires simply proving the post-condition of the value; and sequential composition of pure computations is just function composition of their WPs. The main point of distinction is that PURE.WP makes no mention of any of the effects.

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

STATE.WP The predicate transformer semantics of stateful functions is captured by STATE.WP below, which, as always, transforms post-conditions to pre-conditions. Stateful post-conditions relate the result of a computation to the final state; while pre-conditions are predicates on the input state. Notice there is nothing about exceptions. The combinator return t x shows how to return a value as a stateful computation—the state is unchanged. Meanwhile, bind defines the semantics of sequential composition by threading the state through. In addition to the combinators below, we also give semantics for the primitives for reading, writing and allocating state—we leave that for a later section.

```
STATE.Post a = a → state → Type
```

```
STATE.Pre = state → Type
STATE.WP a = STATE.Post a → STATE.Pre
STATE.return a (x:a) (post:STATE.Post a) = fun s → post x s
STATE.bind a b (wp1:STATE.WP a) (wp2: a → STATE.WP b) : WP b =
  fun (post:STATE.Post b) s0 → wp1 (fun x s1 → wp2 x post s1) s0
```

EXN.WP For exceptions, post-conditions are predicates on exceptional results, while pre-conditions are just propositions. The semantics of exceptional computations is just as in §2.1, except with no mention of state. To complete the semantics of exceptions, one would also provide a semantics for raise and exception handlers.

```
EXN.Post a = either a string → Type
EXN.Pre = Type
EXN.WP a = EXN.Post a → EXN.Pre
EXN.return a (x:a) (post:EXN.Post a) = post (Inl x)
EXN.bind a b (wp1:EXN.WP a) (wp2: a → EXN.WP b) : WP b =
  fun (post:EXN.Post b) → wp1 (fun x → match x with
        | Inr msg → post (Inr msg)
        | Inl v → wp2 v post)
```

Combining effects, piecewise To describe how effects compose, we specify morphisms among the monads. The morphisms define a partial order on the effects; for coherence, we require this order to form a join semi-lattice. For instance, to combine pure and stateful computations, we define:

```
PURE.lift_state a (wp:PURE.WP a) : STATE.WP a =
    fun (post:STATE.Post a) s → wp (fun x → post x s)
```

To combine pure functions with exceptions, we define:

```
PURE.lift_exn a (wp:PURE.WP a) : EXN.WP a =
    fun (post:EXN.Post a) s → wp (fun x → post (Inl x))
```

When combining state and exceptions, one usually has two choices, depending on whether the state is propagated or reset when an exception is raised. However, since exceptions and state are primitive in F⋆, we do not have the freedom to choose. In the primitive semantics of F⋆, as is typical, when an exception is raised, the state is preserved and propagated, rather than being reset—the monad ALL.WP (exactly the monad from §2.1) captures this primitive semantics. To combine state and exceptions, we define the two morphisms below:

```
STATE.lift_all a (wp:STATE.WP a) : ALL.WP a =
    fun (post:ALL.Post a) s → wp (fun x s' → post (Inl x) s') s

EXN.lift_all a (wp:EXN.WP a) : ALL.WP a =
    fun (post:ALL.Post a) s → wp (fun x → post x s)
```

F⋆ requires these lift functions to be monad morphisms, and it is easy to check that they satisfy the morphism laws, i.e., that the returns, binds and lifts commute in the expected way.

2.4 A lattice of monad-indexed monads for computations

The type system of F⋆ extends a core based on Girard's (1972) System F_ω (i.e., higher-rank polymorphism, type operators and higher kinds), with inductive type families, dependent function types, and refinement types.

F⋆ is a call-by-value language. Following Moggi (1989), we observe that such a language has an inherently monadic semantics. Every expression has a *computation type* M t wp, for some effect M, while functions have arrow types with effectful co-domains, e.g., fun x → e has a dependent type of the form x:t → M t' wp, where the formal parameter x is in scope to the right of the arrow. Traditionally, the effect M is left implicit in type systems for ML; but, in F⋆, the computation type M t wp ties a computation to its semantic interpretation as a predicate transformer, i.e., its wp. We introduce a computation type constructor M for each Dijkstra monad, e.g., PURE for PURE.WP, EXN for EXN.WP etc.

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 in the initial configuration, then (1) e's effects are delimited by M; and (2) e returns a t-typed result and a final configuration satisfying post, or diverges, if permitted by M.

The lattice on the Dijkstra monads induces a lattice on the computation-type constructors—we have $\mathsf{M} \sqsubseteq \mathsf{M}'$ whenever we have a morphism M.lift_M' between M.WP and M'.WP. Every two elements M and M' are guaranteed to have a least upper-bound, but if the upper bound happens to be the implicit ⊤ element, we reject the program—this means that effects M and M' cannot be composed. We write $\mathsf{M} \sqcup \mathsf{M}'$ for the partial function computing the non-⊤ least upper-bound of two computation-type constructors.

The type system of F⋆ is designed to infer the least effect for a computation, if one exists. The lattice and monadic structure of the effects are relevant throughout the type system, but nowhere as clearly as in (T-Let), the (derived) rule for sequential composition, which we illustrate below.

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

The sequential composition of computations is captured semantically by the sequential composition of predicate transformers, i.e., by M.bind. (We will see the role of M.return in §3.2.) To compose computations with different effects, M_1 and M_2, we lift them to M, the least non-⊤ effect that includes them both. Since M is unique, the effect computed for the program is unambiguous—this would not be the case if we used only, say, a partial order instead of a join semi-lattice on the effects. Since the lifts are morphisms, we get the expected properties of associativity of sequential composition and lifting—the specific placement of lifts is semantically irrelevant.

The next three sections present F⋆ in detail via examples of pure, divergent, ghost and stateful computations—we leave detailed examples of exceptions to the supplement (§B.1).

3. Purity and divergence

F⋆ treats divergence differently than it does all other effects. Whereas the semantics of effects like state are given using predicate transformers, the semantics of divergence is built in to the language. In essence, given a predicate transformer like STATE.WP, one can read its semantics in either a total- or partial-correctness setting—a programmer-provided attribute specifies which. By default, only the PURE and GHOST monads are interpreted in a total-correctness semantics; the other effects implicitly include divergence and are interpreted in a partial-correctness setting.

To control the use of divergence, the language provides two constructs for building recursive computations. The first, is for fixpoints in PURE and GHOST; the second for general-recursive computations in any of the partial correctness monads. In this section, we focus on the PURE monad, its fixpoint construct, and other core features of F⋆ including refinement and indexed types. We illustrate how these features are used for both programming and proving in the PURE monad, F⋆'s logical core; we also give an example of divergence in the DIV monad.

For our examples, we present fragments of the metatheory of a tiny lambda calculus. Although tiny, this is representative of many calculi for which we have mechanized soundness proofs in F⋆. For example, §A.2.2 in the supplement illustrates how the proof techniques sketched here scale to our formalization of μF⋆. We start, however, with a brief overview of F*'s concrete syntax and summarize the main typing features it provides.

3.1 Basic F*

F* adopts a stratified syntax: expressions are classified by types which are, in turn, classified by kinds. The syntax of expressions is essentially the same as F# or Caml-light, with some minor differences that we point out as necessary. The main innovation of F* is in its language of types, whose syntax is shown below. In this section, we point out the main typing features and provide a brief summary of their semantics.

expr.	e	::=	fun (b) → e | . . . (mostly ML-like syntax)
type	t,ϕ,wp	::=	a | fun (b) → t | t t | t e | b → m t | T | x:t{ϕ}
kind	k	::=	Type | b → k
binder	b	::=	x:t | a:k | #x:t | #a:k

Lambdas, binders and applications The syntax fun (b_1) ... (b_n) → t introduces a type-level lambda abstraction, where the b_i range over binding occurrences for variables. This syntax also applies for expression-level functions, except the body is an expression e rather than a type t. 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. Applications are written using juxtaposition, as usual. A type can be applied to another type or to a pure expression.

Logical specifications The language of logical specifications ϕ and predicate transformers wp is included within the language of types. We use standard syntactic sugar for the logical connectives $\forall, \exists, \wedge, \vee, \Longrightarrow$, and $\Longleftrightarrow$, which can be encoded in types. We also overload these connectives for use with boolean expressions—F* automatically coerces booleans to Type as needed.

Computation types Computation types m t have the form M t $\tau_1 \ldots \tau_n$, where M is an effect constructor, t is the result type, and each τ_i is a type or an expression. For primitive effects, computation types have the shape M t wp, where the index wp is a predicate transformer. We also use a number of derived forms. For example, the primitive computation-type PURE (t:Type) (wp:PURE.WP t) has two commonly used derived forms, shown below. For terms that are unconditionally pure, we introduce Tot:

```
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—the abbreviation Pure defined below enables this.

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

For better readability, we write Pure t (requires p) (ensures q) $\triangleq$ Pure t p q; "requires" and "ensures" are semantically insignificant.

Arrows Function types and kinds are written b → σ, where σ ranges over kinds k and computation types m t—note the lack of enclosing parentheses on b; 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. 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.

b_1 → ... → b_n → M t wp $\triangleq$ b_1 → Tot (... → Tot (b_n → M t wp))

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

Inductive types 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 b_1 → ... → b_n → T τ_1 ... τ_m, where T is type being constructed. This is syntactic sugar for b_1 → ... → b_n → Tot (T τ_1 ... τ_m), i.e., constructors are total functions.

Given a datatype definition, F* automatically generates a few auxiliary functions: for each constructor C, it provides a *discriminator* is_C; and for each argument a of each constructor, it provides a *projector* C.a. We also use syntactic sugar for records, tuples and lists, all of which are encoded as datatypes. Unlike Coq, F* does not generate induction principles for datatypes. Instead, as we will see in §3.3, 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 an inductive type that defines the typing judgment of the simply-typed lambda calculus. The TyVar case shows discriminators and projectors in action, and also illustrates refinement types in F*, which we discuss next.

```
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 =
| TyLam : #g:env → #t:typ → #e1:exp → #t':typ →
            typing (extend g t) e1 t' → typing g (ELam t e1) (TArr t t')
| TyApp : #g:env → #e1:exp → #e2:exp → #t11:typ → #t12:typ →
            typing g e1 (TArr t11 t12) → typing g e2 t11 →
            typing g (EApp e1 e2) t12
| TyVar : #g:env → x:var{is_Some (g x)}
          → typing g (EVar x) (Some.v (g x))
```

Refinement types A refinement of a type t is a type x:t{ϕ} inhabited by expressions e : Tot t that additionally validate the formula $\phi[e/x]$. For example, F* defines the type nat = x:int{x ≥ 0}. Using this, we can write the following code:

```
let abs : int → Tot nat = fun n → if n < 0 then −n else n
```

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

```
x:int → Pure int (requires true) (ensures (fun y → y ≥ 0))
```

We also introduce syntactic sugar for mixing refinements and dependent arrows, writing x:t{ϕ} → σ for x:(x:t{ϕ}) → σ.

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.

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
```

3.2 Intrinsic vs. extrinsic proofs

F*'s refinement types are more powerful than prior systems of refinement types, including F* (Swamy et al. 2013a), the line of work on liquid types (Rondon et al. 2008), and the style of refinement types used by Freeman and Pfenning (1991), that 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 the aforementioned systems, this type may not even be well-formed, since id 0 is not necessarily effect-free. In those systems, one can ask the question whether id 0 : x:int{x=0}—the type x:int{x=0} is well-formed, since it does not contain any potentially effectful expressions. Still, given id:int → int, prior refinement type systems fail to prove id 0 : x:int{x=0}. One would have to enrich the type of id to x:int → y:int{x=y} to conclude the proof—we call the style in which one enriches the type of a function as part of its definition "intrinsic" proving.

With its semantic treatment of effects, F* 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. This style of "extrinsic" proof allows proving lemmas about pure functions separately from the definitions of those functions. F* also provides a mechanism to enrich the type of a function extrinsically, i.e., after proving a lemma about a function, we can use F*'s subtyping relation to give the function a more precise type.

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.

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

We discuss in detail the tradeoffs between intrinsic and extrinsic proofs, and transitioning between them, in §B.2 of the supplement.

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 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 the $\prec$ order to ensure that the fixpoint always exists, as shown below:

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

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 $\prec$ 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.

We illustrate rule (T-Fix) for typing factorial:

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

The body of factorial is typed in a context that includes n:nat and factorial: m:nat{m $\prec$ 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 $\prec$ n. Given the definition of the $\prec$ relation below (which includes the usual ordering on nat), F* easily dispatches this obligation.

Our style of termination proofs is in 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.

3.3.1 The built-in well-founded ordering

The F* type-checker relies on the following $\prec$ ordering:

(1) Given i, j : nat, we have i $\prec$ j $\Longleftrightarrow$ i < j. The negative integers are not related by the $\prec$ 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 $\mathsf{e}_1 \ldots \mathsf{e}_n$ we have $\mathsf{e}_i \prec$ e. for all i.

(4) For any function f : x:t → Tot t' and v:t, f v $\prec$ f.

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 $\prec$ LexCons v1' v2', if and only if, either v1 $\prec$ v1'; or v1=v1' and v2 $\prec$ v2'.

(2b.) If v:lex_t and v ≠ LexTop, then v $\prec$ 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.

As an illustration of the flexibility of F*'s termination check, §A.3 of the supplement shows how to encode accessibility predicates (Bove 2001), a technique that encompasses a wide range of

```
1 type presub = {
2   sub:var → Tot exp; (∗ the substitution itself ∗)
3   renaming:bool; (∗ an additional field for the proof; made ghost in Sec. 4 ∗)
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

termination arguments. Programmers can use this to define their own well-founded orders for custom termination arguments. While this illustrates the power of F*'s termination check, we found that the detour via accessibility predicates is very rarely needed (as opposed to Coq, for instance).

3.3.2 Parallel substitutions: A non-trivial termination proof

Consider the simply typed lambda calculus from §3.1. 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); Schäfer et al. (2015) 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 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 subterm 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 (1999).

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). This field is computationally irrelevant; in §4, we'll see how to use F*'s ghost monad to ensure that it can be erased. 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]).

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.

Such lexicographic orderings are used at scale not just in our definitions but also in our proofs. For instance, in the type soundness proof for μF* (§6) substitution composition, the substitution lemma, and preservation all use lexicographic orderings.

3.4 Divergence in the DIV effect

The predicate transformer DIV.WP is identical to PURE.WP, except its semantics is read in a partial-correctness setting. Accordingly, a computation with effect DIV may not terminate. The non-termination is safely encapsulated within the monad, ensuring that the logical core remains consistent. We use the abbreviations Dv, which is to DIV as Tot is to PURE.

```
effect Dv (a:Type) = DIV a (fun post → ∀x. post x)
```

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 a strongly typed, but potentially divergent evaluator for simply typed lambda calculus programs—the type guarantees that the type of the term being reduced is preserved. The evaluator is defined using typecheck and typed_step, a type-checker and single-step reducer—we only show their signatures. Of course, with more effort, one can also prove that an evaluator for the simply typed lambda calculus is normalizing. We provide several such proofs in the supplement, e.g., using hereditary substitutions (§A.4).

```
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)
```

When defining computations in one of the partial-correctness effects, F* allows the use of a general-recursive variant of the let rec form F* does not check that recursive calls respect the well-founded ordering.

4. Translucent abstractions with GHOST

Leveraging its lattice of effects, F* uses a monad GHOST to encapsulate computationally irrelevant code. Using this feature, we revisit the example of Figure 1 and show how to mark specification-only parts of the program for erasure. In particular, we redefine the type presub as shown below:

```
type presub = { sub: var → Tot exp; renaming: erased bool }
```

The field renaming is now typed as an erased bool, meaning that its value is irrelevant to all non-GHOST code, and hence safe to erase.

We define GHOST (a:Type) (wp:GHOST.WP a) to be an abstract alias of the PURE (a:Type) (wp:PURE.WP a), i.e., the predicate transformer semantics of GHOST computations is identical to that for PURE computations (interpreted in the total correctness sense), except the GHOST monad is a distinct point in F⋆'s effect lattice. We provide a morphism, PURE.lift_GHOST, an identity from PURE.WP a to GHOST.WP a, but none in the other direction. Type-level expressions are expected to be GHOST computations—pure computations are implicitly lifted to GHOST when used at the type level. Computations with GHOST effect cannot be composed directly with any non-ghost computations. In essence, specification-only computations are isolated from computationally relevant code. For convenience, we define the abbreviation G, which is to GHOST as Tot is to PURE.

```
effect G (a:Type) = GHOST (a:Type) (fun p → ∀x. p x)
```

When combined with the other abstraction features provided F⋆, the encapsulation of specifications provided by the GHOST monad can be used for targeted erasure within computations. For example, F⋆'s standard library includes the module Ghost below, which provides an abstract type erased a—the private qualifier hides the definition of erased a from clients of the module, while within the module, erased a simply unfolds to a. The only function we provide to destruct the erased type is reveal, which is marked with the G effect—meaning it can only be used in specifications. As such, the erased a type is opaque to clients and any total expression returning an erased t can safely be erased to ().

```
module Ghost
private type erased (a:Type) = a
val reveal: #a:Type → erased a → G a
let reveal x = x
val erase: #a:Type → x:a → Tot (e:erased a{reveal e = x})
let erase x = x
```

Importantly, the abstraction of erased a is not completely opaque. Within specifications, the abstraction is "translucent"—using reveal, one can extract the underlying a-typed value, as in the revised type sub below. To construct erased values, we use the erase function, as in the initializer of the renaming field below. The rest of the code in Figure 1 is unchanged, except that every use of s.renaming in the specifications is wrapped with a call to reveal. We plan to implement a procedure to automatically insert calls to reveal within specifications, along the lines of the bool-to-Type coercion that we already insert automatically.

```
type sub = s:presub{reveal s.renaming ⟹ (∀ x. is_EVar (s.sub x))}
let sub_inc : sub = {renaming=erase true; sub=(fun y → EVar (y+1))}
```

5. Specifying and verifying stateful programs

We now turn to some examples of verified stateful programming. A primary concern that arises in this context is the manner in which the heap is modeled—specific choices in the model have a profound impact on the manner in which programs are specified and verified, particularly with respect to (anti-)aliasing properties of heap references. We show how to instantiate F⋆'s STATE monad, picking different representations for the state and discussing various tradeoffs. A new contribution is a region-inspired, structured model of memory that we call *hyper-heaps*. Illustrating the use of hyper-heaps, we present an example adapted from our ongoing work on verifying an implementation of TLS-1.2.

5.1 A simple model of the heap

F⋆'s dynamic semantics provides state primitively, where the state is a map from heap references, locations ℓ : ref t, to values of type t. To model this, F⋆'s standard library provides a type heap, with the following purely specificational (i.e., ghost) functions. The functions sel and upd obey the standard McCarthy (1962) axioms, as well as has (upd h r v) s = (r=s || has h s). Using the function has, we define dom, the set of references in the domain of the heap. We trust that this model is a faithful, logical representation of F⋆'s primitive heap.

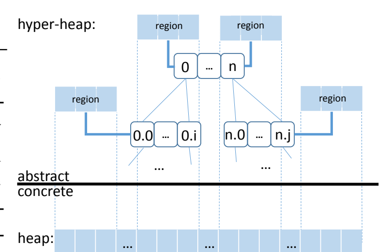
```
val sel: #t:Type → heap → ref t → G t
val upd: #t:Type → heap → ref t → t → G heap
val has: #t:Type → heap → ref t → G bool
```

When defining the STATE monad in §2.3, we left the representation of the type state unspecified. One obvious instantiation for state is heap, using which one can provide signatures for the stateful primitives to dereference, mutate, and allocate references. Although feasible—we have written a fair amount of code using heap as our model of memory—we find the style wanting, for the following reason. A common verification task is to prove that mutations to a data structure a do not interfere with the invariants of another structure b whose references are disjoint from the references of a. Stating and proving this property using just the heap type is heavy: we need to state a quadratic number of inequalities between the references of a and b, and we must reason about all of them using a quadratic number of proof steps. Our supplementary material (§A.5) provides a detailed example illustrating the problem, which is not unique to our system but also affects tools like Dafny (Leino 2010) that adopt a similar, flat memory model.

This "framing" problem for stateful verification has been explored in depth, not least by the vast literature on separation logic. Rather than moving to separation logic (which could, we speculate, be encoded within F⋆'s higher-order logic, at the expense of giving up on SMT automation), we address the framing problem by adopting a richer, structured model of memory, called hyper-heaps and described next.

5.2 Hyper-heaps

Hyper-heaps provide an abstraction layer on top of the concrete, flat heap provided by the F⋆ runtime. Like separation logic, hyper-heaps provide a memory model that caters well to the common case of reasoning about mutations to objects that reside in disjoint regions of memory. The basic structure provided by hyper-heaps is illustrated alongside. At the bottom, we have the concrete heap with a flat arrangement of heap cells (corresponding to references). The abstraction layer above partitions these heap cells into several disjoint regions—the disjointness of heap cells between regions is a key invariant of the abstraction. By allocating references from disjoint objects in separate regions, the invariant guarantees that all their references are pairwise distinct. Beyond the disjointness invariant, hyper-heaps provide a tree-shaped hierarchy of regions, by associating with each region a region identifier, a path from a distinguished root to a specific region associated with a particular fragment of the heap. The hierarchical structure supports allocating an object v in some region 0, and its disjoint sub-objects in regions 0.0 and 0.1; by allocating another object v' in a set of regions rooted at region 1, our disjointness invariant ensures that all the references in v and v' are pairwise distinct.



Formalizing this in F⋆, we define below the type of hyper-heaps, hh, which maps region identifiers rid to disjoint heap fragments—the type map t s is a map from t to s with functions msel, mupd, mhas, and mdom with a semantics analogous to the corresponding functions on the heap type. Note that rid is an erased type; it has no computational content. We define the type of hyper-heap references: rref r a is a reference to an a-value residing in region r—the index r is ghost. We also define some utilities to easily read and write rrefs.

```
type rid = erased (list nat)
type hh = m:map rid heap{∀ {r1,r2} ⊆ mdom m. r1≠r2
```

```
                                ⟹ dom (msel m r1) ∩ dom(msel m r2)=∅}
private type rref (r:rid) (a:Type) = ref a
let hsel #a #r hh (l:rref r a) = Heap.sel (msel hh r) l
let hupd #a #r hh (l:rref r a) v = mupd hh r (Heap.upd (msel hh r) l v)
```

Next, we provide signatures for stateful operations to create a new region, to allocate a reference in a region, and to dereference and mutate a reference.

```
val new_region: r0:rid → STATE rid (fun post hh →
    ∀r h0. not (mhas hh r) ∧ extends r r0 ⟹ post r0 (mupd hh r h0))
val alloc: #t:Type → r:rid → v:t → STATE (rref r t) (fun post hh →
    ∀l. fresh hh l ⟹ post l (hupd hh l v))
val (!): #t:Type → #r:rid → l:rref r t → STATE t (fun post hh →
    post (hsel hh l) hh)
val (:=): #t:Type → #r:rid → l:rref r t → v:t → STATE unit (fun post hh →
    post () (hupd hh l v))
```

Hyper-heaps are a strict generalization of heaps. One can always allocate all objects in the same region, in which case the hyper-heap structure provides no additional invariants. However, making use of the hyper-heap invariants where possible makes specifications much more concise. As it turns out, they also make verifying programs much more efficient. On some benchmarks, we have noticed a speedup in verification time of more than a factor of 20 when using hyper-heaps, relative to heap—we explain below why.

First, without hyper-heaps, consider a computation f () run in a heap h0 and producing a heap h1 related by modifies {x1,...,xn} h0 h1, meaning that h1 differs from h0 at most in x1 ... xn (and in some new references). Next, consider Q = fun h → P (sel h y1) ... (sel h ym), such that Q h0 is true. In general, to prove Q h1 one must prove a quadratic number of inequalities, e.g., to prove sel h1 y1 = sel h0 y1 requires proving y1 ∉ {x1,...,xn}.

However, if one can group references that are generally read and updated together into regions with a common root, one can do much better. For example, moving to hyper-heaps, suppose we place all the x1, ..., xn in region rx. Suppose y1,...,ym are all allocated in some region ry. Now, given two hyper-heaps hh0 and hh1 related by modifies {rx} hh0 hh1, consider proving the implication P (hsel hh0 y1) ... (hsel hh0 yn) ⟹ P (hsel hh1 y1) ... (hsel hh1 ym). Expanding the definition of hsel, it is easy to see that to prove this, we only need to prove that msel hh0 ry = msel hh1 ry, which involves proving that rx and ry do not overlap. Having proven this fact once, we can simply rewrite all occurrences of the sub-term msel hh0 ry to msel hh1 ry everywhere in our formula and conclude immediately—in an SMT solver, such rewrites are immediate via unification. Thus, in such (arguably common) cases, what initially required proving a number of inequalities quadratic in the number of references is now quadratic in the number of regions—in this case, just one. Of course, in the degenerate case where one has just one reference per region, this devolves back to the performance one would get without regions at all. However, typically the number of regions is much smaller than the number of references they contain. The use of region hierarchies serves to further reduce the number of region identifiers that one refers to, making the constants smaller still.

Despite this structured heap model, we run programs using the flat heap provided by the runtime system of our target language. We emphasize that hyper-heaps are just implemented as a library in F⋆—other heap models can be used instead, so long as they can be realized on a flat heap. The metatheory of F⋆ discussed in §6 identifies sufficient conditions for a user-defined memory model to be realized on the flat heap provided primitively—we show that hyper-heaps meet those conditions.

5.3 Hyper-heaps in action: Stateful authenticated encryption

The example of this section is derived from the verification of two key modules in an implementation of TLS 1.2. Our starting point is miTLS, an implementation by Bhargavan et al. (2013). We have redesigned, implemented and verified in F⋆ five exemplary modules from miTLS. These modules cover message formatting and the core stateful record encryption scheme. Verifying the tedious but error-prone message parsing code is important because bugs in this code can open the door to attacks like HeartBleed that rely on malformed messages. Our implementation and idealization of stateful encryption is new and demonstrates a more natural stateful coding style compared to miTLS.

miTLS is itself a verified implementation of TLS with detailed proofs of functional correctness, authentication, and confidentiality. However, the implementation and verification of miTLS is complicated by several limitations. It is verified using a patchwork of SMT-based proofs in F7 (Bhargavan et al. 2010), Coq proofs where F7 is inadequate, code reviews, and manual arguments. Because of the variety of tools and techniques used, the overall proof is hard to follow and maintain. Lacking support for full dependent types, a weakest pre-condition calculus, and refinement type inference, F7 programs are both axiom-heavy and annotation-heavy, and require a careful coding discipline to prevent inconsistencies. Large parts of miTLS are also purely functional, since F7 does not support stateful verification, sometimes leading to unnatural, inefficient code.

In re-designing and verifying a few exemplary modules in F⋆, we already observe substantial improvements over miTLS. For example, we largely eliminate the use of axioms in the modules we verified. By relying on inference, we reduced the type annotations for message-processing code by roughly 50%. We also proved handshake log integrity and properties like partial inverses between parsing and marshalling messages directly in F⋆, without relying on Coq. By using stateful models of cryptography, we avoid miTLS's informal code-review arguments about the linear uses of keys. Additionally, our verified model of the ideal functionality of AEAD_GCM is a first—its proof was omitted in miTLS.

We plan to further our verification effort, to first complete proofs for the full TLS 1.2 stack, and then extend it to the upcoming version 1.3 of the protocol. We also plan to continue to make pervasive use of stateful verification to make the code more natural and efficient. We believe that the reduced annotation burden, uniform proof methodology, and runtime efficiency of F⋆ will ease the task of maintaining a verified TLS stack as the standard evolves.

In the remainder of this section, we distill some essential elements from our verification of two modules in the core stateful, transport encryption scheme—the focus is on modeling and verifying their ideal functionality. The full development, with further subtleties, is available in the supplementary materials.

At a high level, one of the guarantees provided by the TLS protocol is that the messages are received in the same order in which they were sent. To achieve this, TLS builds a stateful, authenticated encryption scheme from a (stateless) "authenticated encryption with additional data" (AEAD) scheme (Rogaway 2002). Two counters are maintained, one each for the sender and receiver. When a message is to be sent, the counter value is authenticated using the AEAD scheme along with the rest of the message payload and the counter is incremented. The receiver, in turn, checks that the sender's counter in the message matches hers each time a message is received and increments her counter.

Cryptographically, the ideal functionality behind this scheme involves associating a stateful log with each instance of a encryptor/decryptor key pair. At the level of the stateless functionality, the guarantee is that every message sent is in the log and the receiver only accepts messages that are in the log—no guarantee is provided regarding injectivity or the order in which messages are received. At the stateful level, we again associate a log with each key pair and here we can guarantee that the sends and receives are in injective, order-preserving correspondence. Proving this requires relating the contents of the logs at various levels, and, importantly, proving that the logs associated with different instances of keys do not interfere. We sketch the proof in F*.

We start with a few types provided by the AEAD functionality.

```
module AEAD
type encryptor = Enc : #r:rid → log:rref r (seq entry) → key → encryptor
and decryptor = Dec : #r:rid → log:rref r (seq entry) → key → decryptor
and entry = Entry : ad:nat → c:cipher → p:plain → basicEntry
```

An encryptor encapsulates a key (an abstract type whose hidden representation is the raw bytes of a key) with a log of entries stored in the heap for modeling the ideal functionality. Each entry associates a plain text p, with its cipher c and some additional data ad:nat. The log is stored in a region r, which we maintain as an additional (erasable) field of Enc. The decryptor is similar. It is worth pointing out that although AEAD is a stateless functionality, its cryptographic modeling involves the introduction of a stateful log. Based on a cryptographic assumption, one can view this log as ghost.

On top of AEAD, we add a Stateful layer, providing stateful encryptors and decryptors. StEnc encapsulates an encryption key provided by AEAD together with the sender's counter, ctr, and its own log of stateful entries, associates plain-texts with ciphers. The log and the counter are stored in a region r associated with the stateful encryptor. StDec is analogous.

```
module Stateful
type st_enc = StEnc : #r:rid → log: rref r (seq st_entry) → ctr: rref r nat
              → key:encryptor{extends (Enc.r key) r} → st_enc
and st_dec = StDec : #r:rid → log: rref r (seq st_entry) → ctr: rref r nat
              → key:decryptor{extends (Dec.r key) r} → st_dec
and st_entry = StEntry : c:cipher → p:plain → st_entry
```

Exploiting the hierarchical structure of hyper-heaps, we store the AEAD.encryptor in a distinct sub-region of r—this is the meaning of the extends relation. By doing this, we ensure that the state associated with the AEAD encryptor is distinct from both log and ctr. By allocating distinct instances k1 and k2 in disjoint regions, we can prove that using k1 (say decrypt k1 c) does not alter the state associated with k2. In this simplified setting with just three references, the separation provided is minimal; when manipulating objects with sub-objects that contain many more references (as in our full development), partitioning them into separate regions provides disequalities between their references for free.

Encryption To encrypt a plain text p, we call Stateful.encrypt, shown below. It calls AEAD_GCM.encrypt with its current counter as the additional data to associate with this message, and obtains a cipher text c. Then, we increment the counter, and return c. In the ideal cryptographic functionality, we formally model this by also associating c with p and recording it by snoc'ing it to the log.

```
let encrypt (StEnc log ctr key) p =
  let c = AEAD_GCM.encrypt key !ctr p in
  log := snoc !log (StEntry c p); ctr := !ctr + 1; c
```

Main invariant The main invariant of these modules is captured by the predicate st_inv (e:st_enc) (d:st_dec) (hh:hh). It states that the log at the AEAD level is in point-wise correspondence with the Stateful log, where the additional data at each entry in the AEAD log is the index of the corresponding entry in the Stateful log. Additionally, the encryptor's counter is always the length of the log, and the decryptor's counter never exceeds the encryptor's counter. In addition, we have several technical heap invariants: the stateful encryptor and decryptor are in the same region; they share the same log at both levels, but their counters are distinct.

Decryption To try to decrypt a cipher c with a stateful key d, we need to first prove that d satisfies the invariant. Then, decrypt d c calls AEAD_GCM with the current value of the counter. If it succeeds, we increment the counter.

```
val decrypt: d:st_dec → c:cipher → ST (option plain)
 (requires (fun h → ∃e. st_inv e d h))
 (ensures (fun h0 res h1 → modifies {StDec.r d} h0 h1
  ∧ let log0, log1 = hsel h0 (StDec.log d), hsel h1 (StDec.log d) in
     let r0, r1 = hsel h0 (StDec.ctr d), hsel h1 (StDec.ctr d) in
     log0 = log1 ∧ (∃ e. st_inv e d h1) ∧
     (match res with
      | Some p → r1 = r0 + 1 ∧ p = Entry.p (index log r0)
      | _ → length log=r0 ∨ c ≠ Entry.c (index log r0))))
let decrypt (StDec log ctr key) c =
   let res = AEAD_GCM.decrypt key !ctr c in
   if is_Some res then ctr := !ctr + 1; res
```

Via the invariant, we can prove several properties about a well-typed call to decrypt d c. First, we prove that it modifies only regions that are rooted at the region of d. In this the hierarchical structure of hyper-heaps is helpful— modifies rs h0 h1 is a predicate defined to mean that h1 may differ from h0 only in regions that are rooted at one of the regions in the set rs (and, possibly, in any new allocated regions that are not present in h0). Next, we prove that the stateful logs are unmodified, and that the invariant is maintained. Finally, we prove that we return the current entry in the reader's current position in the log and then advance the position, except if there are no more entries or if the cipher is incorrect.

6. Metatheory

The supplementary materials provide a comprehensive formal definition of F* corresponding to the system we have implemented (Appendix C). However, working out the metatheory of the full language is a work in progress. Our eventual goal is a mechanized metatheory for F* in F*, and given that F* is also implemented in F*, we aim to use its verification machinery to verify the implementation as well—we are still far from this goal.

For the moment, we identify two subsets of F* called μF* (micro-F*) and pF* (pico-F*), which contain many tricky features of the full language, and study their metatheory in various ways. For μF*, we prove partial correctness for the specifications of effectful computations via a syntactic progress and preservation argument (§A.2.1). For pF*, a pure fragment of μF*, we prove weak normalization and logical consistency using logical relations (§A.2.3). Both these developments are manual proofs.

We have also mechanically checked in F* most of the progress and preservation proof for the PURE effect of μF*. (§A.2.2). This proof was developed over a period of four months by one of the authors and comprises ≈6,500 lines. In the process of mechanically checking our proof, as may be expected, we found and fixed several small bugs in our formal definitions. The style of mechanization is rather different than what is typical in tools like Coq. The proof is developed without tactics, and employs a mixture of constructive proofs (i.e., we directly write a proof term) and SMT solving. Because of the SMT solving and heavy use of termination arguments based on lexical orderings such a style of proof seems unthinkable in Coq and maybe even Agda. In the future, we plan to design an Mtac-inspired (Ziliani et al. 2013) tactic language for F*.

6.1 μF*

The language of types in μF* includes most of the syntactic forms described in §3.1. It omits implicit arguments (mainly a concern for type inference, which we do not formalize at all); refinement types (many forms of refinement can be encoded using predicate transformers); inductive datatypes (we bake-in a few constants, like int and bool); and computations that abstract over types (we do not anticipate major difficulties in adding this, since we already support dependent types and type-abstraction within types). Lacking

$$\text{(T-Var)}\ \frac{\Gamma, x:t, \Gamma' \vdash \mathsf{ok}}{\Gamma, x:t, \Gamma' \vdash x : \mathsf{Tot}\ t} \qquad \text{(T-Abs)}\ \frac{\Gamma \vdash t : \mathsf{Type} \qquad \Gamma, x{:}t \vdash e : M\ t\ \phi}{\Gamma \vdash \lambda x{:}t.\, e : \mathsf{Tot}\ (x{:}t \to M\ t\ \phi)}$$

$$\text{(T-App1)}\ \frac{x \in FV(t') \qquad \Gamma \vdash e_1 : M\ (x{:}t \to M\ t'\ \phi)\ \phi_1 \qquad \Gamma \vdash e_2 : \mathsf{Tot}\ t}{\Gamma \vdash e_1\ e_2 : M\ (t'[e_2/x])\ (\mathsf{bind}_M\ \phi_1\ \lambda\_.\phi[e_2/x])}$$

$$\text{(T-App2)}\ \frac{x \notin FV(t') \qquad \Gamma \vdash e_1 : M\ (x{:}t \to M\ t'\ \phi)\ \phi_1 \qquad \Gamma \vdash e_2 : M\ t\ \phi_2}{\Gamma \vdash e_1\ e_2 : M\ t'\ (\mathsf{bind}_M\ \phi_1\ (\lambda\_.\mathsf{bind}_M\ \phi_2\ \lambda x.\phi))}$$

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

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

Figure 2. Remaining typing rules of μF*

datatypes, we also omit the match construct at the term level, providing instead a branch-on-zero construct if0. μF* also does not cover erasure via GHOST. Despite these omissions, μF* is a substantial calculus, with dependent types and kinds; type operators; subtyping; sub-kinding; semantic termination checking; a lattice of user-defined monads; predicate transformers; user-defined heap models, and higher-order state.

Figure 2 lists all the expression typing rules of μF* that have not already been shown earlier (except the trivial rule for typing constants). We use a notation here more compact than the concrete syntax of F*. The rules for variables and λ-abstractions are unsurprising. In each case, the expression is in Tot, since it has no immediate side-effects.

Typing an application is subtle—we have two rules, depending on whether the function's result type is dependent on its argument. If it is, then only rule (T-App1) applies, since we need to substitute the formal parameter x with the argument, we require the actual argument e_2 to be pure; if e_2 were impure, the substitution would cause an effectful term to escape into types, which is meaningless. In rule (T-App2), since the result is not dependent on the argument, no substitution is necessary and the argument can be effectful. Note, in both cases, the formal parameter x can appear free in the predicate transformer ϕ of the function's (suspended) body.

Rule (T-If0) connects the predicate transformers using an ite_M operator, which we expect to be defined for each effect. For instance, for PURE it is defined as follows:

$$\mathsf{ite}_{\mathsf{PURE}}\ \phi\ \phi_1\ \phi_2\ p = \mathsf{bind}_{\mathsf{PURE}}\ \phi\ \lambda i.\ i{=}0 \Rightarrow \phi_1\ p \wedge i{\neq}0 \Rightarrow \phi_2\ p$$

Subsumption (T-Sub) connects expression typing to the subtyping judgment for computations, which has the form $\Gamma \vdash M'\ t'\ \phi' <: M\ t\ \phi$. The subtyping judgment for computations only has the (S-Comp) rule, listed in Figure 3; it allows lifting from one effect to another, strengthening the predicate transformer, and weakening the type using a mutually inductive judgment $\Gamma \vdash t <: t'$. To strengthen the predicate transformer, it uses 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 judgment $\Gamma \vdash t <: t'$ includes a structural rule (Sub-Fun) and a rule for type conversion via the logical validity judgment (Sub-Conv).

We write $\phi \Longrightarrow_M \phi'$ for the predicate transformer built from the point-wise implication of ϕ and ϕ', e.g., $\phi \Longrightarrow_{\mathsf{PURE}} \phi'$ is $\lambda p.\phi\, p \Longrightarrow \phi'\, p$. This notation extends to other connectives naturally. We also write $\mathsf{down}_M\ \phi$ for a universally quantified application of ϕ,

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

$$\text{(Sub-Conv)}\ \frac{\Gamma \models t_1 = t_2 \qquad \Gamma \vdash t_2 : \mathsf{Type}}{\Gamma \vdash t_1 <: t_2} \qquad \text{(Sub-Trans)}\ \frac{\Gamma \vdash t_1 <: t_2 \qquad \Gamma \vdash t_2 <: t_3}{\Gamma \vdash t_1 <: t_3}$$

$$\text{(S-Comp)}\ \frac{\begin{array}{cc} M \leqslant M' & \Gamma \vdash t <: t' \\ \Gamma \vdash \phi' : K'_M(t') & \Gamma \models \mathsf{down}_{M'}\ (\phi' \Longrightarrow_{M'} (\mathsf{lift}_M^{M'}\ \phi)) \end{array}}{\Gamma \vdash M\ t\ \phi <: M'\ t'\ \phi'}$$

Figure 3. Subtyping rules of μF*

e.g., $\mathsf{down}_{\mathsf{PURE}}\ \phi = \forall p.\ \phi\ p$. We also expect predicate transformers to be monotone, e.g., in the PURE monad, $\forall p_1 p_2.(p_1 \Longrightarrow p_2) \Longrightarrow \phi\ p_1 \Longrightarrow_{\mathsf{PURE}} \phi\ p_2$.

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

Monad lattice In addition to lifts being monad morphisms, our theorems rely on the following properties: the lifts should be transitively closed; should preserve validity of the down_M operator; commute over lifted connectives like $\Longrightarrow_M$; and should preserve monotonicity of predicate transformers. Additionally, the signatures of the effectful primitives like ! and := in a user-defined monad must, when lifted to ALL, match the semantics expected for these operations in the ALL monad.

Heap model abstraction We have formalized the conditions required of a user-defined heap model to ensure that it can be realized using the primitive flat heap. We define an isomorphism between a state s in such a heap model and the primitive heap H, $\Gamma \vdash s \sim \mathsf{asHeap}(H)$, and show that it is preserved by reduction.

Meta-theorems We prove a partial correctness theorem for M computations (where M is any point in the user-defined monad lattice) w.r.t. the standard CBV operational semantics of μF*. The theorem states that a well-typed M expression is either a value that satisfies ALL post-conditions consistent with its (lifted) predicate transformer, or it steps to another well-typed M expression.

Theorem 1 (Partial Correctness of M)**.** *If $\Gamma \vdash (H,e) : M\ t\ \phi$ then for all s, post such that $\Gamma \vdash s \sim \mathsf{asHeap}(H)$, $\Gamma \vdash post : \mathsf{Post}_{\mathsf{ALL}}(t)$ and $\Gamma \models \mathsf{lift}_M^{\mathsf{ALL}}\ \phi\ post\ s$, either e is a value and $\Gamma \models post\ e\ s$, or $(H,e) \to (H',e')$ such that for some $\Gamma' \supseteq \Gamma$, $\Gamma' \vdash (H',e') : M\ t\ \phi'$, $\Gamma' \vdash s' \sim \mathsf{asHeap}(H')$, and $\Gamma' \models \mathsf{lift}_M^{\mathsf{ALL}}\ \phi'\ post\ s'$.*

For PURE expressions, we prove the analogous property, but in the total correctness sense.

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 $\cdot \models p\ v$.*

Both these results assume the consistency of the validity judgment and total correctness additionally relies on the weak normalization of PURE terms. We have proved both consistency and weak normalization for pF*, a pure fragment of μF* including: dependent function types, the weakest precondition calculus, logical formulas and the validity judgment, fixpoints with metrics and our semantic termination check, a well-founded ordering on naturals, and subtyping. The termination argument uses logical relations and is similar to the arguments of System T (Harper 2015) and Trellys/Zombie (Casinghino et al. 2014).

Theorem 3 (Consistency of validity for $p\mathrm{F}^\star$). $\cdot \not\models \mathsf{false}$

Theorem 4 (Weak normalization of PURE for $p\mathrm{F}^\star$). *If* $\Gamma \vdash e : \mathsf{PURE}\ t\ \phi$*,* Γ *is consistent, and there exists a post-condition for which* ϕ *holds, then there exists a value* v *so that* $e \rightarrow^* v$*.*

7. Related work

Adding dependency to an effectful language Integrating dependent types within a full-fledged, effectful programming language has been a long-standing goal for many researchers. 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. More recently, F* (Swamy et al. 2013a) adds *value-dependent* types to an ML-like language; Rondon et al. (2008) add decidable, refinement types to OCaml and Vazou et al. (2014) adapt that work to Haskell—we have compared these prior refinement type systems to F* in §3.2. Meanwhile, monadic-F*adds a single monad to a variant of F* without refinement types—§2.1 discusses its limitations in detail. Liquid Haskell only has non-termination as an effect and for soundness requires a termination check based on the integer ordering, which is less expressive than ours. All these languages provide SMT-based automation, but do not have the ability to support interactive proofs or to carry out functional correctness proofs of effectful programs.

Clean-slate designs The Zombie language (Casinghino et al. 2014) investigates the design of a dependently typed language that includes non-termination via general recursion. Zombie arose from a prior language, Trellys (Kimmell et al. 2012)—we focus primarily on Zombie here. Rather than using an effect system, Zombie adds a "consistency qualifier" to isolate potentially divergent programs from logical terms, with a notion of mobility that allows moving first-order types implicitly from one fragment to another. For functions, Zombie requires programmers to explicitly designate the fragment in which they belong. While our effect system with predicate transformers has a very different structure, there are also some similarities. For example, we also require function types to be explicit about the effects they may exhibit, in particular whether they include divergence or not. In addition to general recursion, Zombie provides a rule for fixpoints. Their rule (T-Ind) is similar to our (T-Fix), except instead of F*'s well-founded order, Zombie provides primitive recursion on natural numbers only. On the other hand, Zombie supports reasoning extrinsically about potentially divergent code, whereas in F*, proofs about divergent programs are carried out intrinsically, within its program logic. Zombie does not address other effects and does not provide proof automation.

Another recent clean-slate design is Idris (Brady 2013), which provides non-termination primitively and also an elegant style of algebraic effects. Brady points out that algebraic effects are preferable since they avoid some of the complications of composing effects posed by monads. In F*, we show some of these complications can be mitigated through the use of a type- and effect-system based on a lattice of monads, which automates effect composition in a modular manner. Additionally, effects in F* are supported primitively in the language, whereas in Idris, effectful programming is provided via an embedded DSL which elaborates effectful code to the underlying pure language. This has the benefit in Idris of making the effects fully extensible; the monad lattice in F* is also user extensible, but only within the bounds of what is provided primitively by the language. On the plus side, primitive effects in F* are more efficiently implemented than effects encoded in a pure language. Idris' metatheory has yet to be studied significantly—as far as we are aware, the language does not attempt to ensure that non-termination does not compromise logical consistency. Idris also lacks SMT-based proof automation.

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 in which to encapsulate imperative code—the discussion about a single monad in §2.1 applies to HTT as well. 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, many researchers have been investigating improving on Coq's syntactic termination check via more semantic approaches. Agda offers two 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 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.

Semi-automated program verifiers Software verification frameworks, such as Why3 (Filliâtre and Paskevich 2013) and Dafny (Leino 2010), also use SMT solvers to verify the logical correctness of (mostly) first-order programs. Unlike F*, they do not provide the expressiveness of dependent types and do not provide the flexibility of user-defined effects and memory models.

Memory abstractions for aliasing Our hyper-heap model is closely related to local stores in Euclid (Lampson et al. 1977). Local stores are also a partitioned heap abstraction realized on a flat heap. However, local stores lack the hierarchical scheme of hyper-heaps, which we find convenient for hiding from clients the details of the partitioning scheme used within an object. Utting (1996) describes a variation on local heaps that supports a "transfer" operation, moving references dynamically from one region to another. This may be a useful variation on hyper-heaps as well, at the cost of losing the stable, state-independent invariants obtained by pinning a reference to a (dynamically chosen) region.

Looking ahead In the past decade, several research groups have made remarkable progress in building formally verified software artifacts. One cohort of researchers mainly use interactive tools like Coq and Isabelle/HOL; another uses SMT-based tools like Dafny and F7. Despite their successes, neither approach is without difficulties, e.g., interactive provers could benefit from more automation and the ability to more freely use imperative features; users of automated tools would benefit from greater expressive power, and a way to provide interactive proofs when an SMT solver fails. F* seeks to be the bridge between these communities.

F* is a living language: it is a work in progress currently, and will continue to be for the foreseeable future. However, given the significant experience we already have had with it, we are optimistic that its design provides the flexibility and expressive power needed to satisfy the growing demand for producing formally verified software, at a cost that compares favorably with that offered by existing tools.

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*, 1999.

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

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

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.

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.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*. Mar. 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.

R. Harper. *Practical foundations for programming languages*. Cambridge University Press, second edition, 2015.

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

C. Keller and T. Altenkirch. Hereditary Substitutions for Simple Types, Formalized. In V. Capretta and J. Chapman, editors, *MSFP - Third Workshop on Mathematically Structured Functional Programming - 2010*. 2010.

G. Kimmell, A. Stump, H. D. E. III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012.

B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Not.*, 12(2):1–79, Feb. 1977.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. 2010.

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.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

P. Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, 2002.

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

S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de Bruijn Terms and parallel substitutions. To appear in ITP, 2015.

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.

The Coq development team. *The Coq proof assistant*.

M. Utting. Reasoning about aliasing. In *In The Fourth Australasian Refinement Workshop*, 1996.

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

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: a monad for typed tactic programming in coq. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 2013.

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. However, as we aim to move F⋆ forwards to the certification of larger pieces of code, we found its original design to be lacking, as discussed in §2.1 and §3.2 of our submission. MUMON is a complete re-design and re-implementation of F⋆. Henceforth, we refer to MUMON as F⋆.

A. Appendix

This appendix describes the supplementary materials in a folder `supplement` associated to this paper and gives pointers to materials that are only available in electronic form; it also provides some additional appendices that were omitted due to space constraints. Appendix A is also available as a more easily clickable `README.html` file.

A.1 Online resources

- The F⋆ website `https://www.fstar-lang.org/` gathers most resources about the language.
- The implementation of F⋆ is available on GitHub (`https://github.com/FStarLang/FStar`). The snapshot of the repository at the time of the submission deadline is available at revision 82153c717.
- An extensive interactive, online tutorial (`https://www.fstar-lang.org/tutorial/`) presents many examples and discusses details of the language beyond the limits of this paper. A non-interactive snapshot of the tutorial is available as `tutorial/tutorial.html` in the supplementary materials.
- The interactive editor mode for F⋆ is at `https://github.com/FStarLang/fstar-interactive`.

A.2 Local snapshot of the metatheory

The `metatheory` directory contains 3 files:

A.2.1 Paper proof of partial correctness for μF⋆

The paper proof of partial correctness for μF⋆ is available in `metatheory/micro-fstar.txt`.

A.2.2 Mechanized progress and preservation proofs for the PURE effect of μF⋆

The mechanized progress and preservation proofs for the PURE effect of μF⋆ is available in `metatheory/micro-fstar.fst` and `https://github.com/FStarLang/FStar/blob/master/examples/metatheory/micro-fstar.fst` on github.

A.2.3 Paper proof of normalization and consistency for pF⋆

The proof of normalization and consistency for pF⋆ is available in `metatheory/pico-fstar.txt`.

Please not that unless your browser is configured to correctly display `utf8` you will need a proper text editor to read this file.

A.3 Accessibility Predicates

The encoding of accessibility predicates is available in locally at`examples/wf.fst` and `https://github.com/FStarLang/FStar/blob/master/examples/wf/wf.fst` on github.

It deeply relies on Property (4) of the built-in well-founded ordering detailed in §3.3.1.

A.4 Hereditary substitutions

We proved the termination of a normalizer for the simply-typed λ-calculus via hereditary substitutions. This development is available locally at `examples/hereditary-substitutions/hs.fst` and on github at `https://github.com/FStarLang/FStar/blob/master/examples/hereditary-substitutions/hs.fst`.

This formalization is based on an existing Agda formalization (Keller and Altenkirch 2010). It illustrates the termination checker, and in particular the lexicographical ordering on measures applied to arguments of recursive functions.

A.5 Example for Heap Problem

See `examples/crypto/statefulEnc-multi-frame.fst` and `examples/crypto/statefulEnc-multi-hyperheap.fst` for two developments for stateful authenticated encryption, the first with a heap, the second with a hyperheap taking roughly 1m20s and 20s to verify on a typical laptop with typical load close to the deadline.

A.6 miTLS*

The `mitls` directory contains 3 exemplary miTLS modules verified in F*. We chose to include these modules because they can be read and understood in isolation, and because they illustrate the difference in the coding style between F* and F7.

A separate directory `mitls/injectivity_proof` contains the proof of injectivity of our message formats. This proof was earlier done in Coq and we have ported it to F*.

The full development of miTLS in F* is work-in-progress and you can find the latest developments as well as prior F7 proofs at `http://www.miTLS.org`.

B. Additional explanations and examples

B.1 Examples of Exceptions

See section 1a of the tutorial.

B.2 Tradeoffs between intrinsic and extrinsic proofs

Here, we develop a small, verified implementation of Quicksort. We draw attention to one style of proving in F⋆, using a flexible combination of defining functions with rich types and proving lemmas about existing pure definitions. A live version of this example is available in Section 6 of the F⋆ tutorial.

Here is the definition of Quicksort:

```
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, a refinement of binary boolean functions to total orders—unlike many prior systems of refinement types, refining pure functions in F⋆ is not a problem.

```
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 ∗)
```

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:total_order a
  → 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 of refinement types, 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 in the two lemmas below: of quicksort:

```
val partition_lem: #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_lem f = function _::tl → partition_lem 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 SMTPat annotations are 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 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". This overhead may be reduced if it were possible for Zombie to provide the kind of automation that F⋆ provides.

C. Definition of full F⋆ language

This section describes the full language implemented by F⋆. While in the future we hope to scale our proofs to this language, we have no proofs for this at the moment.

C.1 Monadic operations

Each monad M must come with a signature, as explained in §2: M.WP t ≜ M.Post t → M.Pre t; and a bunch of monadic operations. We already saw M.return and M.bind, but we actually need more for the typing judgments: they are summed up in Figure 4:

- M.⊕ lifts any binary operator ⊕ : Type → Type → Type (like ∧ or ⟹) to a similar operator for weakest preconditions;
- M.up lifts any formula to a weakest precondition;

- M.down, its converse, lifts any weakest precondition to a formula;
- M.closeE and M.closeT close term-level (respectively type-level) functions to weakest preconditions.

We will detail their use in the typing rules. We give in Figure 5 their implementations for three major points in our lattice: PURE, STATE and EXN, the exception monad.

M.return	:	α:Type $\to \alpha \to$ M.WP α
M.bind	:	α, β:Type $\to$ M.WP $\alpha \to (\alpha \to$ M.WP $\beta) \to$ M.WP β
M.$\oplus$	:	α:Type $\to$ M.WP $\alpha \to$ M.WP $\alpha \to$ M.WP α
M.up	:	α:Type $\to$ Type $\to$ M.WP α
M.down	:	α:Type $\to$ M.WP $\alpha \to$ Type
M.closeE	:	α, β:Type $\to (\beta \to$ M.WP $\alpha) \to$ M.WP α
M.closeT	:	α:Type $\to$ (Type $\to$ M.WP $\alpha) \to$ M.WP α

Figure 4. Monadic operations

The PURE monad:

PURE.return	$\triangleq$	$\lambda\ \alpha$ a p. p a
PURE.bind	$\triangleq$	$\lambda\ \beta\ \alpha$ wp f p. wp (λ a. f a p)
PURE.$\oplus$	$\triangleq$	$\lambda\ \alpha$ wp_1 wp_2 p. (wp_1 p) $\oplus$ (wp_2 p)
PURE.up	$\triangleq$	$\lambda\ \alpha\ \phi$ p. ϕ
PURE.down	$\triangleq$	$\lambda\ \alpha$ wp. $\forall$ p. wp p
PURE.closeE	$\triangleq$	$\lambda\ \alpha\ \beta$ f p. $\forall$ b:β. f b p
PURE.closeT	$\triangleq$	$\lambda\ \alpha$ f p. $\forall\ \beta$:Type. f β p

The STATE monad:

STATE.return	$\triangleq$	$\lambda\ \alpha$ a p h. p a h
STATE.bind	$\triangleq$	$\lambda\ \beta\ \alpha$ wp f p h. wp (λ a h'. f a p h') h
STATE.$\oplus$	$\triangleq$	$\lambda\ \alpha$ wp_1 wp_2 p h. (wp_1 p h) $\oplus$ (wp_2 p h)
STATE.up	$\triangleq$	$\lambda\ \alpha\ \phi$ p h. ϕ
STATE.down	$\triangleq$	$\lambda\ \alpha$ wp. $\forall$ p h. wp p h
STATE.closeE	$\triangleq$	$\lambda\ \alpha\ \beta$ f p h. $\forall$ b:β. f b p h
STATE.closeT	$\triangleq$	$\lambda\ \alpha$ f p h. $\forall\ \beta$:Type. f β p h

The EXN monad:

EXN.return	$\triangleq$	$\lambda\ \alpha$ a p. p (left a)
EXN.bind	$\triangleq$	$\lambda\ \beta\ \alpha$ wp f p. wp (λ x. ($\forall$ a. x = left a $\Longrightarrow$ f a p) $\wedge$ ($\forall$ e. x = right e $\Longrightarrow$ p (right e)))
EXN.$\oplus$	$\triangleq$	$\lambda\ \alpha$ wp_1 wp_2 p. (wp_1 p) $\oplus$ (wp_2 p)
EXN.up	$\triangleq$	$\lambda\ \alpha\ \phi$ p. ϕ
EXN.down	$\triangleq$	$\lambda\ \alpha$ wp. $\forall$ p. wp p
EXN.closeE	$\triangleq$	$\lambda\ \alpha\ \beta$ f p. $\forall$ b:β. f b p
EXN.closeT	$\triangleq$	$\lambda\ \alpha$ f p. $\forall\ \beta$:Type. f β p

Figure 5. Some implementations of monadic operations

C.2 Lattice operations

As explained in §2, the monads are organized into a lattice. We thus need operations to lift a computation that lies inside one monad M into a greater monad M': M.lift_M' : α:Type $\to$ M.WP $\alpha \to$ M'.WP α. As an example, Figure 6 gives an example of such functions for PURE, STATE and EXN.

PURE.lift_STATE	$\triangleq$	$\lambda\ \alpha$ wp p h. wp (λ a. p a h)
PURE.lift_EXN	$\triangleq$	$\lambda\ \alpha$ wp p. wp (λ a. p (left a))

Figure 6. Some implementations of lattices operations

C.3 Inductive datatypes

F* also offers the possibility to define inductive datatypes and, correspondingly, supports pattern-matching. The definitions of inductive datatypes are gathered into a signature S, which is a list of declarations of the form

$$S ::= \cdot \mid S, T{:}k\{\overline{C : t}\}$$

where a new type T of kind k is introduced, with constructors C_i of types t_i.

For conciseness, pattern-matching has two branches, one to match a constructor, and a default case: match e with C $\overline{\alpha}$ $\overline{x}$ $\to$ e' else e''.

C.4 Syntax

Now that we have presented the main two novelties of F* compared to μF*, let us see the other extensions to the syntax, presented in Figure 7. For a matter of conciseness, we do not put in the language the constructions that are used only for operational semantics, like memory locations.

bindings	b	::=	x:t | α:k
kinds	k	::=	Type | b $\to$ k
terms	τ	::=	t | e
types	t, ϕ, wp	::=	α | λ b.t | T | t τ | x:t{ϕ} | b $\to$ M t wp | ref t | exn
exprs	e, δ	::=	x | λ b.e | fix (f^d:t) x = e | C | e τ | match e with C $\overline{\alpha}$ $\overline{x}$ $\to$ e' else e'' | ref e | !e | e := e' | raise e | try e with λ x.e
measures	d	::=	ε | δ

Figure 7. Syntax of F*

The main differences with respect to μF* are:

- refinement types x:t{ϕ}, which informally are inhabited by expressions e of type t that satisfy ϕ[e/x];
- inductive datatypes, constructors and pattern-matching, as we explained above;
- parametric polymorphism: the possibility to abstract over types inside expressions $\lambda\alpha$:k.e, and the corresponding applications e t;
- exceptions of type exn thrown with raise e and caught with try e with λ x.e;
- logical connectives no longer appear in the syntax, since they are defined as empty inductive data-types: for instance, $\forall$ x:t. ϕ is syntactic sugar for Forall t (λx:t.ϕ) where Forall is an empty inductive type of kind α:Type $\to (\alpha \to$ Type) $\to$ Type. Since we are not interpreting inductive types constructively, these empty types do not break soundness and can be interpreted as their SMT counterpart during SMT encoding.

C.5 Type system

Typing judgments are parameterized by the signature S presented above, and a context Γ, which as usual is a list of bindings:

$$\Gamma ::= \cdot \mid \Gamma, b$$

There are 8 different judgments which are mutually recursive, presented in Table 1. Contrary to μF*, in which only subtyping is parameterized by a side condition, F* propagates side-conditions all the way down, until sending one single verification condition to prove to the SMT solver. Moreover, side conditions are a bit more complex: if most judgments are parameterized by a formula ϕ:Type, subtyping judgments S;$\Gamma \vdash$ t <: t' $\Longleftarrow \psi$ and S;$\Gamma \vdash$ M t wp <: M' t' wp' $\Longleftarrow \psi$ are parameterized by a predicate ψ:t $\rightarrow$ Type. Only typing computations do not have this condition, since it is already included in the wp.

Judgment	Meaning
S wf $\Longleftarrow \phi$	S is well-formed
S;Γ wf $\Longleftarrow \phi$	Γ is well-formed
S;$\Gamma \vdash$ k ok $\Longleftarrow \phi$	k is well-formed
S;$\Gamma \vdash$ b ok $\Longleftarrow \phi$	b is well-formed
S;$\Gamma \vdash$ t:k $\Longleftarrow \phi$	t has kind k
S;$\Gamma \vdash$ e:M t wp	e is a computation of type M t wp
S;$\Gamma \vdash$ t <: t' $\Longleftarrow \psi$	t is a subtype of t'
S;$\Gamma \vdash$ M t wp <: M' t' wp' $\Longleftarrow \psi$	M t wp is a sub-computation of M' t' wp'

Table 1. The judgments of F*

All these judgments are given in Figures 8, 9 and 10, and detailed below.

C.6 Well-formedness judgments

Signatures, contexts, kinds and bindings must be well-formed, which is expressed by the judgments of Figure 8. These rules are quite standard, the novelty here being the side conditions propagated all along the way, as explained before.

C.7 Kinding

Kinding judgments are presented in Figure 9. Most rules are straightforward; a first reason for the condition ϕ appears in rule (K-AppE): applying a type t to an expression e makes sense only if this expression is pure and terminating, but we do not require it to be unconditionally pure: it might be only under some pre-condition wp, which is thus propagated into the side condition. We will see more examples of this with typing rules. In other rules, side conditions are simply accumulated in the conclusions.

C.8 Typing

As for μF*, the major novelty appears in typing judgments: expressions are not just typed with types, but with computations M t wp where wp:M.WP t. This is where monadic and lattice operations play an important role. Those typing judgments are presented in Figure 10.

The rules (T-Ret), (T-Var), (T-Lam) (T-FixOmega) are the same as for μF*. The rule (T-Fix) is a bit different since the decreasing argument appears inside a refinement instead of a precondition, as explained in §3.3. The rule (T-C) lookups in the signature for the type of a constructor.

We now have two rules for application, one to apply expressions (T-AppE) and one to apply types (T-AppT). In both cases, we build the conjunctions of the preconditions together with the side conditions of the premises.

The more complex rule is for pattern-matching (T-Match). We used the following notation in the conclusion:

$$\mathsf{wp}_{\mathsf{Match}} \triangleq \left(\begin{array}{l} \mathsf{M.bind\ wp}\ (\lambda \mathsf{x}. \\ \quad (\mathsf{M.close}\ \overline{\alpha}\ \overline{\mathsf{x}}\ (\mathsf{M.up}\ (\mathsf{x} = \mathsf{C}\ \overline{\alpha}\ \overline{\mathsf{x}})\ \mathsf{M.}\Longrightarrow \mathsf{wp}'))\ \mathsf{M.}\wedge \\ \quad ((\mathsf{M.up}\ (\forall \overline{\alpha}.\ \forall \overline{\mathsf{x}}.\ \mathsf{x} \neq \mathsf{C}\ \overline{\alpha}\ \overline{\mathsf{x}}))\ \mathsf{M.}\Longrightarrow \mathsf{wp}'')) \end{array} \right)$$

We type-check e and e' as usual, whereas e' is type-checked in an environment augmented with the pattern variables. Type-checking the pattern itself is a way to ensure that it is meaningful. In the conclusion, we need to relate all the precondition; the idea is exactly the same as for (T-If0) in μF*, except that we need to close the preconditions with the pattern variables: M.close is a notation for iterating M.closeT then M.closeE.

The subtyping rule (T-Sub) is similar to μF*, except that, as we will see in more details later, the side-condition is a predicate.

Finally, the rules for stateful computations (T-Ref), (T-!) and (T-Upd) are similar to μF*, and the rules for exceptions (T-Raise) and (T-Try) follow accordingly.

C.9 Subtyping

Finally, the subtyping judgments are given in Figure 11.

For types, (S-Eq) relates convertible types, with respect to the operational semantics of the types, as described for μF*. (S-RefineIntro) introduces a refinement by adding the formula to the side condition: this is why, contrary to μF* (which does not have refinements), we need the side condition to be a predicate instead of a formula. (S-Simp) is the rule that finally relates side-conditions to the logic, and in practices discharges side-conditions to the SMT solver. (S-RefSub) allows to weaken refinements. The (S-Fun) rule defines subtyping for functions, with the interesting point (already mentioned in §3.1) that it will allow to relate refinements and precondition by strengthening the final precondition with the formulas that have been deduced by subtyping the types (the ones that might contain refinements).

For computations, the single rule is the same as for μF*.

C.10 Operational semantics

The operational semantics is a call-by-value reduction strategy really similar to the one of μF*, and thus is not presented here.

$$\boxed{\mathsf{S\ wf} \Longleftarrow \phi} \qquad \frac{}{\cdot\ \mathsf{wf} \Longleftarrow \top} \qquad \frac{\mathsf{S};\cdot \vdash \mathsf{k\ ok} \Longleftarrow \phi \quad \mathsf{S},\mathsf{T{:}k};\cdot \vdash \mathsf{t}_i : \mathsf{Type} \Longleftarrow \phi_i}{\mathsf{S},\mathsf{T{:}k}\{\overline{\mathsf{C{:}t}}\}\ \mathsf{wf} \Longleftarrow \phi \wedge (\wedge_i \phi_i)}$$

$$\boxed{\mathsf{S};\Gamma\ \mathsf{wf} \Longleftarrow \phi} \qquad \frac{\mathsf{S\ wf} \Longleftarrow \phi}{\mathsf{S};\cdot\ \mathsf{wf} \Longleftarrow \phi} \qquad \frac{\mathsf{S};\Gamma \vdash \mathsf{b\ ok} \Longleftarrow \phi}{\mathsf{S};\Gamma,\mathsf{b\ wf} \Longleftarrow \phi}$$

$$\boxed{\mathsf{S};\Gamma \vdash \mathsf{k\ ok} \Longleftarrow \phi} \qquad \frac{\mathsf{S};\Gamma\ \mathsf{wf} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash \mathsf{Type\ ok} \Longleftarrow \phi} \qquad \frac{\mathsf{S};\Gamma \vdash \mathsf{b\ ok} \Longleftarrow \phi \quad \mathsf{S};\Gamma,\mathsf{b} \vdash \mathsf{k\ ok} \Longleftarrow \phi'}{\mathsf{S};\Gamma \vdash \mathsf{b} \to \mathsf{k\ ok} \Longleftarrow \phi \wedge \phi'}$$

$$\boxed{\mathsf{S};\Gamma \vdash \mathsf{b\ ok} \Longleftarrow \phi} \qquad \frac{\mathsf{S};\Gamma \vdash \mathsf{t} : \mathsf{Type} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash \mathsf{x{:}t\ ok} \Longleftarrow \phi} \qquad \frac{\mathsf{S};\Gamma \vdash \mathsf{k\ ok} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash \alpha{:}\mathsf{k\ ok} \Longleftarrow \phi}$$

Figure 8. Well-formedness of signatures, contexts, kinds, and bindings

$$\boxed{\mathsf{S};\Gamma \vdash \mathsf{t} : \mathsf{k} \Longleftarrow \phi}$$

$$\text{(K-Var)}\frac{\mathsf{S};\Gamma\ \mathsf{wf} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash \alpha : \Gamma(\alpha) \Longleftarrow \phi} \qquad \text{(K-Lam)}\frac{\mathsf{S};\Gamma,\mathsf{b} \vdash \mathsf{t} : \mathsf{k} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash (\lambda \mathsf{b.t}) : (\mathsf{b} \to \mathsf{k}) \Longleftarrow \phi} \qquad \text{(K-Ind)}\frac{\mathsf{S};\Gamma\ \mathsf{wf} \Longleftarrow \phi}{\mathsf{S};\Gamma \vdash \mathsf{T} : \mathsf{S}(\mathsf{T}) \Longleftarrow \phi}$$

$$\text{(K-AppE)}\frac{\mathsf{S};\Gamma \vdash \mathsf{t} : (\mathsf{x{:}t'} \to \mathsf{k}) \Longleftarrow \phi \quad \mathsf{S};\Gamma \vdash \mathsf{e} : \mathsf{PURE\ t'\ wp}}{\mathsf{S};\Gamma \vdash (\mathsf{t\ e}) : \mathsf{k}[\mathsf{e}/\mathsf{x}] \Longleftarrow \phi \wedge (\mathsf{wp}\ (\lambda \mathsf{y.y} = \mathsf{e}))} \qquad \text{(K-AppT)}\frac{\mathsf{S};\Gamma \vdash \mathsf{t} : (\alpha{:}\mathsf{k} \to \mathsf{k}') \Longleftarrow \phi \quad \mathsf{S};\Gamma \vdash \mathsf{t'} : \mathsf{k} \Longleftarrow \phi'}{\mathsf{S};\Gamma \vdash (\mathsf{t\ t'}) : \mathsf{k}'[\mathsf{t'}/\alpha] \Longleftarrow \phi \wedge \phi'}$$

$$\text{(K-Ref)}\frac{\mathsf{S};\Gamma \vdash \mathsf{t} : \mathsf{k} \Longleftarrow \phi' \quad \mathsf{S};\Gamma,\mathsf{x{:}t} \vdash \phi : \mathsf{Type} \Longleftarrow \phi''}{\mathsf{S};\Gamma \vdash \mathsf{x{:}t}\{\phi\} : \mathsf{k} \Longleftarrow \phi' \wedge \phi''} \qquad \text{(K-Arr)}\frac{\mathsf{S};\Gamma \vdash \mathsf{b\ ok} \Longleftarrow \phi \quad \mathsf{S};\Gamma,\mathsf{b} \vdash \mathsf{t'} : \mathsf{Type} \Longleftarrow \phi' \quad \mathsf{S};\Gamma,\mathsf{b} \vdash \mathsf{wp} : \mathsf{M.WP\ t'} \Longleftarrow \phi''}{\mathsf{S};\Gamma \vdash (\mathsf{b} \longrightarrow \mathsf{M\ t'\ wp}) : \mathsf{Type} \Longleftarrow \phi \wedge \phi' \wedge \phi''}$$

Figure 9. Kinding judgments

$$\boxed{S;\Gamma \vdash e : M\ t\ wp}$$

$$\text{(T-Ret)}\ \frac{S;\Gamma \vdash e : \mathsf{Tot}\ t}{S;\Gamma \vdash e : \mathsf{PURE}\ t\ (\mathsf{Pure.return}\ e)} \qquad \text{(T-Var)}\ \frac{S;\Gamma\ \mathsf{wf} \Longleftarrow \phi}{S;\Gamma \vdash x : \mathsf{Tot}\ \Gamma(x)} \qquad \text{(T-Lam)}\ \frac{\begin{array}{c} S;\Gamma \vdash b\ \mathsf{ok} \Longleftarrow \phi \\ S;\Gamma, b \vdash e : M\ t\ wp \end{array}}{S;\Gamma \vdash (\lambda b.e) : \mathsf{Tot}\ (b \longrightarrow M\ t\ wp)}$$

$$\text{(T-Fix)}\ \frac{\begin{array}{c} t = y{:}t_x \longrightarrow \mathsf{PURE}\ t'\ wp \\ S;\Gamma \vdash t : \mathsf{Type} \Longleftarrow \phi \\ S;\Gamma, x{:}t_x, f{:}(y{:}t_x\{d\ y < d\ x\} \longrightarrow \mathsf{PURE}\ t'\ wp) \vdash e : (\mathsf{PURE}\ t'\ wp)[x/y] \end{array}}{S;\Gamma \vdash (\mathsf{fix}\ (f^d{:}t)\ x = e) : \mathsf{Tot}\ t} \qquad \text{(T-FixOmega)}\ \frac{\begin{array}{c} t = y{:}t_x \longrightarrow M\ t'\ wp \qquad M \neq \mathsf{PURE} \\ S;\Gamma \vdash t : \mathsf{Type} \Longleftarrow \phi \\ S;\Gamma, x{:}t_x, f{:}t \vdash e : (M\ t'\ wp)[x/y] \end{array}}{S;\Gamma \vdash (\mathsf{fix}\ (f{:}t)\ x = e) : \mathsf{Tot}\ t}$$

$$\text{(T-C)}\ \frac{S;\Gamma\ \mathsf{wf} \Longleftarrow \phi}{S;\Gamma \vdash C : \mathsf{Tot}\ S(C)} \qquad \text{(T-AppE)}\ \frac{\begin{array}{c} S;\Gamma \vdash e_1 : M\ (x{:}t_2 \longrightarrow M\ t\ wp)\ wp_1 \\ S;\Gamma \vdash e_2 : M\ t_2\ wp_2 \qquad S;\Gamma \vdash t[e_2/x] : \mathsf{Type} \Longleftarrow \phi \end{array}}{S;\Gamma \vdash (e_1\ e_2) : M\ t[e_2/x]\ ((wp_1\ M.\wedge\ (M.\mathsf{bind}\ wp_2\ (\lambda x.wp)))\ M.\wedge\ (M.\mathsf{up}\ \phi)))}$$

$$\text{(T-AppT)}\ \frac{\begin{array}{c} S;\Gamma \vdash e_1 : M\ (\alpha{:}k_2 \longrightarrow M\ t\ wp)\ wp_1 \\ S;\Gamma \vdash t_2 : k_2 \Longleftarrow \phi \end{array}}{S;\Gamma \vdash (e_1\ t_2) : M\ t[t_2/\alpha]\ wp_1\ M.\wedge\ (wp[t_2/\alpha])\ M.\wedge\ (M.\mathsf{up}\ \phi))}$$

$$\text{(T-Match)}\ \frac{\begin{array}{c} S;\Gamma \vdash e : M\ t\ wp \qquad \Gamma' = \mathit{PatVars}(S, C, \overline{\alpha}, \overline{x}) \\ S;\Gamma' \vdash C\ \overline{\alpha}\ \overline{x} : \mathsf{Tot}\ t \\ S;\Gamma,\Gamma' \vdash e' : M\ t'\ wp' \qquad S;\Gamma \vdash e'' : M\ t'\ wp'' \end{array}}{S;\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ C\ \overline{\alpha}\ \overline{x} \rightarrow e'\ \mathsf{else}\ e'' : M\ t'\ wp_{\mathsf{Match}}} \qquad \text{(T-Sub)}\ \frac{\begin{array}{c} S;\Gamma \vdash e : M\ t\ wp \\ S;\Gamma \vdash M\ t\ wp <: M'\ t'\ wp' \Longleftarrow \psi \end{array}}{S;\Gamma \vdash e : M'\ t'\ ((M'.\mathsf{up}\ (\psi\ e))\ M'.\wedge\ wp')}$$

$$\text{(T-Ref)}\ \frac{\begin{array}{c} S;\Gamma \vdash e : M\ t\ wp \\ \Delta \vDash M \geqslant \mathsf{STATE} \end{array}}{S;\Gamma \vdash \mathsf{ref}\ e : M\ (\mathsf{ref}\ t)\ ((\mathsf{STATE.lift}_M\ (\lambda p\ h.\forall l.l \notin \mathsf{dom}\ h \Longrightarrow p\ l\ (\mathsf{upd}\ h\ l\ e)))\ M.\wedge\ wp)}$$

$$\text{(T-!)}\ \frac{\begin{array}{c} S;\Gamma \vdash e : M\ (\mathsf{ref}\ t)\ wp \\ \Delta \vDash M \geqslant \mathsf{STATE} \end{array}}{S;\Gamma \vdash\ !\ e : M\ t\ ((\mathsf{STATE.lift}_M\ (\lambda p\ h.p\ (\mathsf{sel}\ h\ e)\ h))\ M.\wedge\ wp)}$$

$$\text{(T-Upd)}\ \frac{\begin{array}{c} S;\Gamma \vdash e_1 : M\ (\mathsf{ref}\ t)\ wp_1 \\ S;\Gamma \vdash e_2 : M\ t\ wp_2 \\ \Delta \vDash M \geqslant \mathsf{STATE} \end{array}}{S;\Gamma \vdash e_1 := e_2 : M\ \mathsf{unit}\ ((\lambda p\ h.p\ ()\ (\mathsf{upd}\ h\ e_1\ e_2))\ M.\wedge\ wp_1\ M.\wedge\ wp_2)}$$

$$\text{(T-Raise)}\ \frac{\begin{array}{c} S;\Gamma \vdash e : M\ t_1\ wp \\ S;\Gamma \vdash t_2 : \mathsf{Type} \Longleftarrow \phi \\ \Delta \vDash M \geqslant \mathsf{EXN} \end{array}}{S;\Gamma \vdash \mathsf{raise}\ e : M\ t_2\ ((\lambda p.p\ \mathsf{Exn})\ M.\wedge\ wp\ M.\wedge\ (M.\mathsf{up}\ \phi))}$$

$$\text{(T-Try)}\ \frac{\begin{array}{c} S;\Gamma \vdash e_1 : M\ t_1\ wp_1 \\ S;\Gamma, x{:}t_1 \vdash e_2 : M\ t_2\ wp_2 \\ S;\Gamma \vdash t_2[e_1/x] : \mathsf{Type} \Longleftarrow \phi \\ \Delta \vDash M \geqslant \mathsf{EXN} \end{array}}{S;\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{with}\ \lambda x.e_2 : M\ t_2[e_1/x]\ ((M.\mathsf{bind}\ wp_1\ (\lambda x.wp_2))\ M.\wedge\ (M.\mathsf{up}\ \phi))}$$

Figure 10. Typing judgments

$$\boxed{\mathsf{S};\Gamma \vdash \mathsf{t} <: \mathsf{t}' \Longleftarrow \psi}$$

$$\text{(S-Eq)}\ \frac{S;\Gamma \vdash \mathsf{t} \sim \mathsf{t}' \Longleftarrow \phi}{S;\Gamma \vdash \mathsf{t} <: \mathsf{t}' \Longleftarrow \lambda_.\phi}
\qquad
\text{(S-RefineIntro)}\ \frac{S;\Gamma, \mathsf{x}{:}\mathsf{t} \vdash \phi{:}\mathsf{Type} \Longleftarrow \phi'}{S;\Gamma \vdash \mathsf{t} <: \mathsf{x}{:}\mathsf{t}\{\phi\} \Longleftarrow \lambda \mathsf{x}.\phi \wedge \phi'}
\qquad
\text{(S-Simp)}\ \frac{\begin{array}{c} S;\Gamma \vdash \mathsf{t}_1 <: \mathsf{t}_2 \Longleftarrow \psi \\ S;\Gamma, \mathsf{x}{:}\mathsf{t}_1, \psi'\ \mathsf{x} \vDash \psi\ \mathsf{x}\ \mathsf{valid} \end{array}}{S;\Gamma \vdash \mathsf{t}_1 <: \mathsf{t}_2 \Longleftarrow \psi'}$$

$$\text{(S-RefSub)}\ \frac{S;\Gamma \vdash \mathsf{x}{:}\mathsf{t}\{\phi'\}{:}\mathsf{Type} \Longleftarrow \phi''}{S;\Gamma \vdash \mathsf{x}{:}\mathsf{t}\{\phi\} <: \mathsf{x}{:}\mathsf{t}\{\phi'\} \Longleftarrow (\lambda \mathsf{x}.(\phi\ \mathsf{x} \Longrightarrow \phi'\ \mathsf{x}) \wedge \phi'')}
\qquad
\text{(S-Fun)}\ \frac{\begin{array}{c} S;\Gamma \vdash \mathsf{t}' <: \mathsf{t} \Longleftarrow \phi \\ S;\Gamma, \mathsf{x} : \mathsf{t}' \vdash \mathsf{s} <: \mathsf{s}' \Longleftarrow \psi \\ \mathsf{WP} \triangleq \mathsf{M.strengthen\ s\ wp}\ (\lambda\ \mathsf{y}.\psi\ \mathsf{x} \wedge \psi\ \mathsf{y}) \\ \Phi \triangleq \lambda\ \mathsf{f}.\forall\, \mathsf{x} : \mathsf{t}'.\mathsf{wp}'\mathsf{M}'.\Longrightarrow (\mathsf{M.lift}_{M'}\ \mathsf{WP}) \end{array}}{S;\Gamma \vdash (\mathsf{x}{:}\mathsf{t} \longrightarrow \mathsf{M\ s\ wp}) <: (\mathsf{x}{:}\mathsf{t}' \longrightarrow \mathsf{M}'\ \mathsf{s}'\ \mathsf{wp}') \Longleftarrow \Phi}$$

$$\boxed{\mathsf{S};\Gamma \vdash \mathsf{M\ t\ wp} <: \mathsf{M}'\ \mathsf{t}'\ \mathsf{wp}' \Longleftarrow \psi}$$

$$\frac{\begin{array}{c} \Delta \vDash \mathsf{M} \leqslant \mathsf{M}' \\ S;\Gamma \vdash \mathsf{t} <: \mathsf{t}' \Longleftarrow \psi \\ \phi \triangleq \mathsf{wp}'\ \mathsf{M}'.\Longrightarrow (\mathsf{M.lift}_{\mathsf{M}'}\ \mathsf{wp}) \end{array}}{S;\Gamma \vdash \mathsf{M\ t\ wp} <: \mathsf{M}'\ \mathsf{t}'\ \mathsf{wp}' \Longleftarrow \lambda \mathsf{x}.(\psi\ \mathsf{x}\ \wedge\ \mathsf{M}'.\mathsf{down}\phi)}$$

Figure 11. Subtyping judgments