Secure Distributed Programming with Value-dependent Types

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, Jean Yang

Secure Distributed Programming with Value-Dependent Types

Nikhil Swamy[1] Juan Chen[1] Cédric Fournet[1] Pierre-Yves Strub[2] Karthikeyan Bhargavan[3] Jean Yang[4]

Microsoft Research[1] MSR-INRIA[2] INRIA[3] MIT[4]

Abstract

Distributed applications are difficult to program reliably and securely. Dependently typed functional languages promise to prevent broad classes of errors and vulnerabilities, and to enable program verification to proceed side-by-side with development. However, as recursion, effects, and rich libraries are added, using types to reason about programs, specifications, and proofs becomes challenging.

We present F*, a full-fledged design and implementation of a new dependently typed language for secure distributed programming. Unlike prior languages, F* provides arbitrary recursion while maintaining a logically consistent core; it enables modular reasoning about state and other effects using affine types; and it supports proofs of refinement properties using a mixture of cryptographic evidence and logical proof terms. The key mechanism is a new kind system that tracks several sub-languages within F* and controls their interaction. F* subsumes two previous languages, F7 and Fine. We prove type soundness (with proofs mechanized in Coq) and logical consistency for F*.

We have implemented a compiler that translates F* to .NET bytecode, based on a prototype for Fine. F* provides access to libraries for concurrency, networking, cryptography, and interoperability with C#, F#, and the other .NET languages. The compiler produces verifiable binaries with 60% code size overhead for proofs and types, as much as a 45x improvement over the Fine compiler, while still enabling efficient bytecode verification.

To date, we have programmed and verified more than 20,000 lines of F* including (1) new schemes for multi-party sessions; (2) a zero-knowledge privacy-preserving payment protocol; (3) a provenance-aware curated database; (4) a suite of 17 web-browser extensions verified for authorization properties; and (5) a cloud-hosted multi-tier web application with a verified reference monitor.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Syntax and Semantics

General Terms Security, Verification, Languages, Theory

Keywords Security type systems, refinement types

1. Introduction

Distributed applications are difficult to program reliably and securely. To address this problem, researchers have designed new languages with security verification in mind. Early work in this space developed *ad hoc* type systems targeting verification of specific security idioms, including systems for information flow controls, starting with Volpano et al. (1996); for proving authentication properties in cryptographic protocols (Gordon and Jeffrey 2003); and, more recently, for protocols that use zero-knowledge proofs (Backes et al. 2008). More general type systems for security verification have also been proposed, e.g., Fable (Swamy et al. 2008), F7 (Bengtson et al. 2008; Bhargavan et al. 2010), Aura (Jia and Zdancewic 2009; Jia et al. 2008; Vaughan et al. 2008), Fine (Chen et al. 2010; Swamy et al. 2010), and PCML5 (Avijit et al. 2010). All these languages use various forms of dependent types to reason about security, following a long tradition of dependent typing for general-purpose theorem proving and program verification, e.g., Coq (Bertot and Castéran 2004) and Agda (Norell 2007).

Although these languages are successful in many aspects, for large-scale distributed programming, we desire languages that (1) feature general programming constructs like effects and recursion, which, while invaluable for building real systems, make it hard to formally reason about programs, specifications, and proofs; (2) support various styles of proofs and evidence, ranging from cryptographic signatures to logical proof terms; (3) produce proofs that can be efficiently communicated between agents in the system.

This paper presents F*, a full-fledged design and implementation of a new dependently-typed programming language that addresses all these challenges. F* subsumes both F7 and Fine. Unlike prior languages, F* provides arbitrary recursion while maintaining a logically consistent core, resolving the tension between programmability and consistency by restricting the use of recursion in specifications and proofs; it enables modular reasoning about state and other effects and allows specifying refinement properties on affine values; it supports proofs of refinement properties using a mixture of cryptography and logical proof terms; and it allows selective erasure and reconstruction of proofs to reduce the overhead of communicating proofs.

By compiling to verifiable .NET bytecode, F* provides access to libraries for concurrency, networking, cryptography, and interoperability with C#, F#, and other .NET languages. We have formalized the metatheory of F*, and mechanized a significant part of the metatheory in Coq. We have developed a prototype compiler for F* (35,000 lines of F#), and used F* to program and verify more than 20,000 lines of code. We believe F* is the first language of its kind with such a scale of implementation and evaluation.

Next, we give an overview of F* and our main contributions.

A novel kind system. A central feature of F* is its kind system, which tracks several sub-languages—for terms, proofs, affine resources, and specifications. This kind system controls the interaction between those, while still providing a single unified language to specify, program, verify, and deploy secure distributed systems.

We use kinds $\star$ for program terms and P for proofs. Program terms that may be effectful and divergent reside in the universe of $\star$ types. Within $\star$, P identifies a universe of pure, total functions; P terms are used (mainly) in the construction of proof terms. Additionally, we use A, the kind of affine (used-at-most-once), stateful resources, to model and reason about effects in a modular style. Finally, the kind E is for types that may have no inhabitants at all. We use E to control the selective erasure of proof terms, when these proofs are either impossible to construct in a distributed setting (e.g., due to cryptography or due to the design of legacy libraries); when the presence of a proof term would curtail expressiveness (e.g., when speaking of properties of affine values); or when proof terms would be too voluminous to construct. As such, E identifies a sub-language that plays a purely specificational role. To improve code reuse, we provide a sub-kinding relation, $P <: \star <: E$, with A unrelated to the others.

Two flavors of refinements. Refinement types are commonly used to specify program properties. In contrast with prior languages, F* features both *concrete* and *ghost* refinements; §2 illustrates the need for both for secure distributed programming. To reason about the security of distributed applications, with minimal trust between components, explicit proofs of security sometimes need to be communicated and checked at runtime.

Concrete refinements are pairs representing a value and a proof term serving as a logical evidence of the refinement property, similar to those in Coq and Fine. One novelty of F* is that it assigns a special kind *P* for proof terms, and restricts types and proof terms in the *P* universe to guarantee logical consistency.

Ghost refinements are used to state specifications for which proof terms are not maintained at run time. Ghost refinements have the form `x:t{`ϕ`}` where `x` is a value variable, `t` is a type, and ϕ is a logical formula, itself represented as a type that must have kind *E* and may depend on `x` and other in-scope variables. Ghost refinements are similar to those of F7; they are smoothly integrated in F* using the *E* kind. Ghost refinements provide the following benefits: (1) they enable precise symbolic models for many cryptographic patterns and primitives, and evidence for ghost refinement properties can be constructed and communicated using cryptographic constructions, such as digital signatures; (2) they benefit from a powerful subtyping relation: `x:t{`ϕ`}` is a subtype of `t`; this structural subtyping is convenient to write and verify higher-order programs; (3) they provide precise specification to legacy code without requiring any modifications; and (4) when used in conjunction with concrete refinements, they support *selective erasure* and *dynamic reconstruction* of evidence, enabling a variety of new applications and greatly reducing the performance penalty for runtime proofs.

Refinements on affine state. Prior work has shown the usefulness of affine types in reasoning about programs that use mutable state (Borgstrom et al. 2011; Lahiri et al. 2011). Relying on its kind system, F* freely allows the use of affine values within its specifications fragment, while still guaranteeing that affine values are used at most once elsewhere in the code. In §4, we exploit this feature extensively in implementing a new, flexible approach (Deniélou and Yoshida 2011) to enforce protocols on multi-party session types. Prior systems that integrate substructural and dependent types (e.g., Fine and Linear LF by Cervesato and Pfenning 2002) disallow refinements to speak directly about affine values, and have to rely instead on various encodings to work around this limitation, which is unsuitable for source programming.

Automation and logic parametricity. Proof automation is critical for developing large-scale programs. F* is carefully designed to be parametric in the logic used to describe programming properties and their proofs. §2.5 shows examples with a simple modal authorization logic and an *ad hoc* logic for database provenance. Logic parametricity enables working with custom authorization logics and, importantly, makes it easy to integrate F* with SMT solvers for logics extended with specific theories. Thus, program verification in F* benefits from significant automation—our implementation uses the Z3 SMT solver (de Moura and Bjørner 2008) and scales up to large programs and specifications. Languages like Aura, PCML5, Coq, and Agda commit to a specific logic, limiting their flexibility. This limitation is significant since diverse logics are used and even designed when reasoning about security policies and properties—see Chapin et al. (2008) for a recent survey.

Metatheory. We establish several properties of F*. First, we prove the soundness of F* in terms of progress and preservation. From this, we derive a safety property for ghost refinements called global deducibility. Next, we show that the *P*-fragment of F* is consistent by giving a typed embedding of this fragment into CiC and proving that the translation is a simulation. We also give a typed embedding of a core subset of RCF (the core calculus of F7) into F*. The subject reduction result (modulo the admission of a few standard lemma) has been mechanized in Coq—we plan to continue to develop our Coq formalization to include the other results.

Since our *P*-fragment is strongly normalizing, one might imagine extending F* to permit arbitrary *P*-terms to index types. However, term reduction in types, particularly with dynamic assumptions and affinity, poses a significant challenge for the metatheory. We remain with value dependency in F*, while acknowledging that it is less expressive than having expressions in types. However, value dependency is a practical design choice that has not hindered the construction and verification of the programs we have built.

Compiler implementation. We have implemented a compiler for F* based on our prior work on a compiler for Fine. The F* compiler accepts both Fine and F7 programs as input. To validate this feature, we typecheck and compile a large F7 library implementing symbolic cryptography (Bhargavan et al. 2010).

Our compiler translates F* to RDCIL, a dependently typed dialect of .NET bytecode. This translation is considerably more efficient than the one we used for Fine. Due to the use of ghost refinements and the availability of polymorphic kinds, bytecode emitted by the F* compiler is an order of magnitude (in some cases 45x) smaller than the bytecode emitted by the Fine compiler.

Experimental evaluation on a large suite of examples. We have implemented several libraries and applications in F*, verifying a large corpus of code for various properties. Prominent among our examples are (1) secure implementations of multi-party sessions protocols; (2) a prototype of a new zero-knowledge privacy scheme; (3) a provenance-aware curated database; (4) a suite of 17 web-browser extensions verified for authorization properties; and (5) an Azure-hosted multi-tier web application with a reference monitor verified for stateful authorization properties.

Due to space constraints, we leave the complete semantics of F*, manual proofs of the metatheory, and additional examples to a technical report, which, along with the compiler source code, the Coq development, and all our example programs, can be found at `http://research.microsoft.com/fstar`.

2. F* by example

This section introduces F* informally. The syntax of F* is based loosely on OCaml, F# and related languages in the ML family—notations specific to F* are primarily in support of its more expressive type and kind language. The dynamic semantics is also in the spirit of ML, in call by value, but the static semantics is significantly more complex. The examples in this section, together with those in §4, are intended to motivate and exercise its main features.

We organize our presentation around the new kind system of F*. We start with simple programs that use *P*-kind and F*'s sublanguage of total functions to construct proof terms for concrete refinements. Next, we discuss *E*-kind and its use in two different scenarios with ghost refinements—first, when giving specifications to legacy libraries where the construction of explicit proof terms is impractical; and, second, when verifying implementations of cryptographic protocols, where the construction of proof terms is simply impossible. We then turn to *A*-kind, which, in conjunction with *E*-kind, can specify and verify properties of stateful computations. We conclude the section with an example that exploits the interaction between *P*-kind and *E*-kind, via the sub-kinding relation $P <: \star <: E$, to construct a model of a high-integrity database with precise data-provenance properties.

2.1 Concrete refinement types and total proof terms

Consider a partial specification for a very simple program, `tail`, that returns the tail of a list:

```
val tail: ∀α::⋆. l1:list α → {l2:list α | ∀x. Mem x l2 ⟹ Mem x l1}
```

This type is polymorphic, of the form $\forall\alpha$::k. t where k is the kind of the abstracted type variable—kinds are ascribed to types using double colons. Here, α has kind $\star$, the kind given to types that admit arbitrary recursion and effects, i.e., the standard kind of fully-applied types in an ML-like system. Following ML, by default we omit explicit quantifiers for prenex-quantified type variables, and omit type applications when they can be determined by the context.

The rest of the type of tail shows a dependent function, of the form x:t → t' where the formal parameter x of type t is named, and is in scope in t'. When the function is not dependent, we simply write t → t'. The range type of tail shows our syntax for concrete refinements, {x:t | ϕ }, where ϕ is itself a type representing a logical formula, in which the name x is bound. Here, the formula states that the tail l2 contains at most the elements of l1.

Concrete refinements and constructive proofs. Also called subset types or Σ-types (Sozeau 2007), concrete refinements in F⋆ are desugared into dependent pairs (as is usual in, say, Coq). The type below shows this desugaring, where the type x:t ∗ t' denotes a dependent pair, where x names the t-typed first component and is bound in t', the type of the second component.

```
val tail: l1:list α → (l2:list α ∗ (x:α → Mem x l2 → Mem x l1))
```

When desugaring formulas, quantifiers are desugared to dependent functions, and implications to non-dependent functions. The predicate Mem x l2 is itself a type, which we show below. As such, concrete refinements are represented as pairs of the underlying value, and a proof term witnessing the validity of the refinement formula.

A total sub-language for proof terms. We must be careful when representing quantifiers and implication with function arrows. For logical consistency, we require the function arrows that represent the type of proof terms to be total, whereas arrows used in the rest of the program (where we certainly want to use arbitrary recursion) can be partial. Thus, we need to ensure that potential divergence in the program never leaks into fragments of a program used for building proof terms. We achieve this by introducing a kind P, a subkind of $\star$, where types residing in P are guaranteed to be total.

Using P-kind, we define Mem, an inductive type that axiomatizes list membership in constructive style. Its kind is of the form α::k ⇒ k', where α binds a k-kinded formal type parameter in the kind k' of the constructed type. Type constructors can also be applied to values; such constructors have kinds of the form x:t ⇒ k where x names the formal argument, a value of type t, and is in scope in k. Below, the kind of Mem says that it is a dependent type constructor that constructs a type of kind P from a type α, a value x of type α, and a value l of type list α. (When x:α, we write Mem x l instead of Mem α x l.)

```
type Mem :: α::⋆ ⇒ α ⇒ list α ⇒ P =
  | Mem_hd : x:α → tl:list α → Mem x (Cons x tl)
  | Mem_tl : x:α → y:α → tl:list α → Mem y tl → Mem y (Cons x tl)
```

Inductive types defined in P-kind are required to be strictly positive, and we place other restrictions (§3.2) on the elimination rules for P-kinded types to ensure totality. A function type x:t → t' inherits the kind of its range type—it has P kind and is a total function when t'::P. Theorem 2 (Logical consistency of P) provides a well-typed embedding of terms residing in the P-fragment into CiC, ensuring that terms in the P-fragment are valid proof terms.

Non-constructive proofs, automation, and logic parametricity. Programming explicitly with proof terms for non-trivial program properties quickly becomes impractical. The F⋆ implementation provides automation by calling Z3, an SMT solver, to try to decide refinement properties, and then constructing proof terms from the deduction traces reported by Z3. However, since Z3 (and many other automated provers) use classical logics, the proof terms produced in this manner are not constructive.

To support non-constructive proof terms (and more generally, to permit custom logics), F⋆ allows to define custom proof kernels. A specific proof kernel (the type pf below), is defined in the standard library of F⋆ and axiomatizes a classical logic, extended with axioms for commonly used data types such as list.

```
type pf :: P ⇒ P =
  | Exc_middle: pf (Or α (Not α))
  | Not_elim: pf (Not α) → pf α → pf β
  | Not_in_nil: x:α → pf (Not (Mem x Nil))
  | ... (* many constructors omitted *)
  | Unit_pf: α → pf α
  | Bind_pf: pf α → (α → pf β) → pf β
```

Each constructor in this kernel is an axiom in the logic, such as the excluded middle; introduction and elimination forms for the standard logical connectives (where Or, Not etc. are also type constructors); and axioms like Not_in_nil, which are necessary for reasoning in a non-constructive setting, where exhaustiveness arguments for inductive types are inapplicable. The pf kernel also includes constructors Unit_pf and Bind_pf that allow proofs to be composed monadically. This is particularly useful since F⋆ is a call-by-value language—Bind_pf lets us compose proofs simply by constructor application, without triggering evaluation.

Using this approach, F⋆ translates the type of tail to the type below. Relying on automated proof extraction from Z3, programmers write the code shown below, and the F⋆ compiler inlines the proof term and compiles (while carrying proofs) to RDCIL (§5), our dependently typed .NET bytecode. Concrete refinements in this form are based on a similar construct in Fine. Through the use of P-kind (which Fine lacks), F⋆ proof terms are logically consistent evidence of refinement properties.

```
val tail: l1:list α → (l2:list α ∗ pf (x:α → pf (Mem x l2) → pf (Mem x l1)))
let tail = function [] → [] | hd::tl → tl
```

2.2 Ghost refinements for lightweight specifications

Concrete refinements have a long tradition and a well-understood theory. However, as discussed below, we find them inappropriate for use in some scenarios. As an alternative, F⋆ also provides ghost refinements, based on a construct of F7, and integrates them with the other features of the system, notably higher kinding, quantification over predicates, and refinements for substructural state.

We illustrate the use of ghost refinements for the problem of verifying clients of libraries, where the libraries are authored separately and are unmodifiable. In recent work, Guha et al. (2011) consider programming secure *web browser extensions* using F⋆. For this scenario, we use ghost refinement types to specify pre- and post-conditions on the interface provided by the browser, and use the specifications to verify access control properties of extensions. The listing below illustrates this approach on a tiny program—§5 reports new results for compiling a suite of 17 such extensions in a type-preserving style to .NET bytecode.

We aim to enforce a policy that untrusted extensions (line 10) only read data from the header of a web page and not the body. This policy is specified using an assumption at line 8, which states, informally, that extensions hold the CanRead e privilege on DOM nodes e, for which the property EltTagName e "head" is derivable. Unlike the Mem predicate in §2.1 (which has P kind), EltTagName and CanRead construct erasable, or E-kinded, types. Erasable types are generally uninhabited and have no constructors. Instead, we use them for specifications, as in the types of innerText and tagName.

```
1 (* Fragment of DOM API *)
2 type elt
3 type EltTagName :: elt ⇒ string ⇒ E
```

```
4  type CanRead :: elt ⇒ E
5  val innerText: e:elt{CanRead e} →string
6  val tagName: e:elt →t:string{EltTagName e t}
7  (* Extension policy *)
8  assume ∀e. EltTagName e "head" ⟹ CanRead e
9  (* Extension code (untrusted) *)
10 let read e = if tagName e = "head" then innerText e else ""
```

The type of innerText has the form x:t{ϕ} →t', where the formula ϕ is a ghost refinement applied to the formal parameter x:t, and x is in scope in both ϕ and t'. The refinement CanRead e is a pre-condition indicating that clients must hold the CanRead e privilege before calling the function. Analogously, the post-condition of tagName relates the returned string t to the argument e, and clients may derive facts using this property and any other assumption (e.g., the policy assumption at line 8). For example, at the call to innerText in the **then**-branch at line 10, the F⋆ checker (and Z3) uses the property that, for the value t returned by tagName e, we have EltTagName e t from the post-condition; t="head" from the equality test; and using the policy assumption, we can derive CanRead e, the pre-condition of innerText and authorize the call. Using this approach, once type checked, untrusted extension code need not be examined—only the policy (and the annotations on the DOM API) are trusted.

Two key properties of ghost refinements make them well-suited for use in our scenario. We discuss these below.

Ghost refinements and erasure. The type x:t{ϕ} is a subtype of t and the values of these two types have the same representation. This makes specifications using ghost refinements lightweight, since they do not require modifications to underlying code and data. For example, we did not need to modify or even wrap the DOM implementation in order to verify code in this style. Furthermore, the subtyping relation lifts naturally into the structure of function types, promoting reuse in higher order libraries.

Semantics of ghost refinement derivability. For every value v:t that inhabits x:t{ϕ} our type system ensures that the formula ϕ[v/x] is derivable. The definition of derivability is subtle and is made precise in §3. However, intuitively, derivability is a logical entailment relation defined relative to a context of dynamic assumptions $\mathscr{A}$. We think of $\mathscr{A}$ as a monotonically increasing *log* of events and formulas that are assumed during evaluation of the program. Formally, a call to tagName e reduces to t and has the *effect* of adding the formula EltTagName e t to the log. For values given ghost refinement types, there may be no concrete proof at run time to witness the derivability of the refinement formula. Indeed, when working with libraries like the DOM, explicit proof terms witnessing DOM invariants seem both infeasible and undesirable (as they may be very voluminous); ghost refinements fit the bill nicely.

Proof-irrelevance and P vs. E-kind. The distinction between P and E-kinds in F⋆ may, at first, seem reminiscent of the distinction between Type and Prop in a system like Coq. The proof terms for concrete refinements in Coq are often from the Prop universe, indicating that they are computationally irrelevant (and so can be erased during code extraction). In contrast, concrete refinements in F⋆ are accompanied by P-kinded proof terms, which are computationally *relevant*. We view proofs as useful runtime entities that carry important information. We choose to make proofs explicit and useful—§2.5 demonstrates a novel way of using concrete proof terms to construct precise provenance trails in a curated database. As such P-kind is closer to Coq's Type. Indeed, our embedding of F⋆'s P fragment in CiC translates P-kinded types to types that reside in Coq's Type universe.

E-kind in F⋆ plays a role more similar to proof irrelevance in Coq. However, the semantics of E-kinded types and ghost refinements is considerably different. Not only are proofs for ghost refinements irrelevant, these proofs may not be constructible at all and E-kinded types may be uninhabited. Instead, the log-based semantics of ghost refinements makes trust assumptions in external code formal and explicit, and allows the definition of security properties for code that are robust even when code is composed with arbitrary attacker code. For example, Guha et al. (2011) used the log-based semantics to prove a robust safety property that ensures that verified extensions are authorization-safe even when composed with arbitrary untrusted JavaScript on a web page.

2.3 Ghost refinements and indexed types for cryptography

Refinement types (in the style of ghost refinements) have been used in F7 to verify implementations of cryptographic protocols. This section presents a small fragment of a library for public key cryptography implemented in a new style (see our website for a more complete library together with several client programs that use cryptography). This style is enabled by features of F⋆ not available in F7, specifically, the integration of refinements with higher-kinded and indexed types. This example also illustrates the need for ghost refinements. As we will see, it is infeasible to construct concrete proof terms (whether constructive or not) to justify the soundness of cryptographic evidence.

The listing below shows the signature of a module Crypto that provides an interface to work with public key signatures. Informally, signatures provide a means for a party in a protocol to communicate a value and a property of its local environment to a remote party. For example, Alice can sign a message m and send it to Bob, and, if Bob trusts Alice, Bob can conclude that the message originated with Alice. Additionally, given a prior agreement on the usage of keys, Alice can convince Bob of some additional property Pred of the message m, e.g., that the message originated in Alice's file system. The property Pred need not be an intrinsic property of the contents of m—a constructive proof of Pred m in this setting may be nonsensical.

```
module Crypto
type dsig = bytes (* type of digital signatures *)
type prin (* name of a principal *)
type sk :: prin ⇒ α::⋆ ⇒ (α ⇒ E) ⇒ ⋆
val rsa_sign : ∀α::⋆, β::α⇒ E. p:prin →sk p αβ→x:α{β x} →dsig
type pk :: prin ⇒ α::⋆ ⇒ (α ⇒ E) ⇒ ⋆
type Says :: prin ⇒ E⇒ E
val rsa_verify : ∀α::⋆,β::α⇒ E.
  p:prin →pk p αβ→x:α →dsig →r:bool{r=true ⟹ Says p (β x)}
```

Crypto provides a type dsig for digital signatures, here just an alias for bytes. It also exposes an abstract type prin for principal identifiers, and the type constructors sk and pk are for secret keys and public keys respectively.

A private key of type sk Alice αPred belongs to the principal Alice:prin, who can use it to sign values m of type α that satisfy Pred m. The function rsa_sign allows clients to construct a dsig value by signing a message x:α, where the ghost refinement guarantees that the formula βx is derivable when rsa_sign is called. Public keys are complementary: rsa_verify verifies a signature using the public key pk p αβ and, if it succeeds, the caller knows that Says p (βx) is derivable. The predicate Says p ϕ is the usual lifting of a proposition ϕ into a modality Says, similar to forms used in a variety of modal authorization logics (Chapin et al. 2008). Intuitively, Says p ϕ is weaker than ϕ, and the two coincide when principal p is trusted.

As in F7, an abstract implementation of the Crypto library can be verified against the specification shown above, and can be proved correct with respect to a Dolev-Yao adversary. However, in F7, types cannot be parametrized by predicates, so predicate parameters are instead simulated through a level of indirection. Instead of the F⋆ type sk p αPred, private keys in F7 are given a type of the form sk p αusage, and the predicate Pred is replaced by a global predicate SignSays, indexed by p and usage. Verification

relies on a programming convention that each key usage must be unambiguously defined by recording an assumption of the form ∀p,usage,v, SignSays p usage v ⟺ Pred v. This convention is not enforced automatically in F7, and hence this style can lead to logical inconsistencies. In contrast, F⋆ types are more concise, and require fewer dynamic assumptions and no programming discipline beyond typing.

2.4 Ghost refinements and affine-indexed types for state

F⋆ is designed to enable reasoning about effectful programs, whether the effects be in the form of non-termination, state, or I/O. We have seen how *P*-kind serves to control non-termination and we consider I/O in §4. This section looks at how F⋆ uses affine types, in combination with *E*-kinded types, to reason about state.

One innovation of F⋆ is that it permits indexing types with affine values, allowing properties to be stated about affine values without having them be consumed immediately. We illustrate this feature by showing how to program with *linear maps*, a data type proposed by Lahiri et al. (2011) to verify heap-manipulating programs. For space reasons, we do not show a client program using linear maps—a complete example is available in the F⋆ distribution.

Linear maps are a data structure that equips a Floyd-Hoare logic (using a *classical assertion logic*) with a form of local reasoning in the style of separation logic. Instead of modeling the heap of a program as a single monolithic map H:map $\alpha\,\beta$ from α-typed locations to β-typed values, the linear maps methodology advocates partitioning the heap H into several fragments $H_1, \ldots, H_n$ where the H_i have disjoint domains. Each H_i is a linear map of type lin $\alpha\,\beta$, and the disjoint domain condition ensures that modifications to H_i leave all the other H_j unmodified. This allows to formulate a kind of frame rule for programs that use linear maps. Since the assertion logic remains classical, linear map programs can be automatically verified using classical provers and SMT solvers.

```
type lin :: ⋆⇒ ⋆⇒ A
type Select :: α::⋆ ⇒ β::⋆ ⇒ lin α β⇒ α⇒ β⇒ E
type Update :: α::⋆ ⇒ β::⋆ ⇒ lin α β⇒ α⇒ β⇒ lin α β⇒ E
type InDomain :: α::⋆ ⇒ β::⋆ ⇒ α⇒ lin α β⇒ E
```

The listing above defines an abstract type of linear maps and several predicates to model their properties. The kind of lin:: ⋆⇒ ⋆⇒ *A* introduces the fourth base kind in F⋆: the kind *A* of affine types. To enforce the disjoint domains invariant on linear maps, Lahiri *et al.* require that linear maps be neither copied nor aliased. This corresponds directly to the use of affinity in F⋆: values of affine type can be used *at most once*.

The types Select and Update correspond to standard predicates implemented by SMT solvers like Z3: Select l x y states that the map l at the location x contains the value y; Update l x y m states that m is like the map l, except at the location x where it contains the value y. We omit the standard axiomatization of these predicates. Linear maps are partial maps and include a domain. The predicate InDomain x l has the obvious meaning.

Two operations to read and write locations in maps are shown below. (Other operations are omitted). The read function reads a location x out of a map m1 (when InDomain x m1), and returns the value y:β stored at x. Since m1 is affine, read threads m1 back to the caller as m2, where the refinement on m2 shows that it is unchanged. The write function is similar, and in both cases the Select and Update predicates specify the appropriate post-conditions.

```
val read: x:α →m1:lin α β{InDomain x m1}
      →(y:β ∗ (m2:lin α β{m1=m2 && Select m1 x y}))

val write: x:α →y:β →m1:lin α β{InDomain x m1}
      →(m2:lin α β{Update m1 x y m2})
```

Predicates on affine values. While seemingly unremarkable, by refining affine values, the types shown above are a significant advance over prior languages that have included substructural and dependent types. For example, in systems like Fine and Linear LF (Cervesato and Pfenning 2002), types are required to be free of affine (or linear) indices, i.e., type constructors of kind t ⇒ k, where t::*A* are forbidden. There are several reasons for this restriction in prior systems. Most prominently, expressing properties of affine values using concrete refinements requires constructing proof terms where, simply by using an affine resource in a proof term, the resource is consumed. While there are ways to work around this restriction (Borgstrom et al. 2011), they involve relatively complex whole-program transformations.

A key innovation of F⋆ is to use the *E*-kind to allow stating properties on affine values directly. Specifically, since *E*-kinded predicates have no runtime significance, indexing these predicates with affine values does not consume them—in F⋆, kinds of the form t ⇒ *E* are permitted, even when t::*A*. In our example, we use affine indexes on *E*-kinded types to state pre- and post-conditions using ghost refinements. However, when modeling linear maps programs, the dynamic log of assumptions (unlike when modeling DOM programs and cryptography) is constant, so F⋆'s metatheory guarantees that refinement formulas in pre- and post-conditions are derivable from the axiomatization of linear maps alone.

We defer further discussion of affine indexed types until §4, where we use affine indexes with higher-rank *E*-kinded types to model concurrent, message passing programs.

2.5 Selective erasure using concrete and ghost refinements

The preceding discussion of ghost refinements may lead the reader to believe that they are always to be preferred to refinements with concrete proof terms. This section illustrates that concrete proof terms are useful too, particularly when one is allowed to compute over these terms, to store them, and to communicate them over the network. The example discussed here is an excerpt from a larger program that models a database of scientific experiments, where each record contains a proof term indicating the *provenance* of the experiment and its "validity", according to some custom notion of validity. The full example brings together several elements, including the use of cryptography with a simple modal logic to authenticate experimental observations. For brevity, we focus just on one aspect: selective erasure and reconstruction of proofs, which may be implemented for both efficiency and confidentiality reasons.

Each experiment record in our database is given the type exp b (for some boolean parameter b, explained shortly). The record contains an optional primary key field xid; a field r:expsetup that defines what ingredients were used in the experiment; and, importantly, a proof term of type proof b (Valid r), where the proof term contains evidence recording the relationship of this experiment to other experiments in the database, i.e., the proof term reflects the provenance of the experiment.

```
type expsetup = list {reagent:string; quantity:int}
type Valid :: expsetup ⇒ E
type exp (b:bool) = (xid:option int ∗ r:expsetup ∗ proof b (Valid r))
```

The type proof b t represents a value from a proof kernel defining a custom logic tailored to this specific application—another example of F⋆'s logic parametricity. We show a selection of the constructors from this kernel below.

```
let full, partial = true, false
type proof :: bool ⇒ E⇒ P=
| AndIntro: ... | AndElim1 : ... | ...
| ChemicalVolcano: proof full (Valid[{reagent="(NH4)2Cr2O7"; ...}])
| Combine: r1:expsetup →r2:expsetup →r3:expsetup →b:bool
        →proof b (And (Union r1 r2 r3) (And (Valid r1) (Valid r2)))
        →proof b (Valid r3)
```

```
| Prune: r:expsetup{Valid r} → xid:int → proof partial (Valid r)
```

The interplay between ghost and concrete proof terms is central in this example—it enables proof terms to be *selectively erased* and later *reconstructed*. This allows us to maintain compact, yet detailed and reliable provenance trails. The type proof full (Valid r) represents a fully explicated proof of Valid r, with no selective erasure applied. In contrast, values of type proof partial (Valid r) may have been partially erased—these values are not guaranteed to carry a complete provenance for the experiment setup r.

The constructors in the kernel include axioms for basic connectives and axioms like ChemicalVolcano which state validity of some well-known experiments. Axioms like Combine allow new valid experiments to be constructed from other valid ones. The most interesting constructor is Prune, which allows a ghost refinement of the validity of an experiment (r:expsetup{Valid r}) to be traded for a concrete proof term for the validity. To allow proofs to be reconstructed, Prune takes an extra argument, xid:int, the primary key of a record in the database that holds the complete provenance for r.

Now we can give a typed interface to our database (below). The database db is simply a list of experiments with full proofs. It supports operations to insert new experiments (returning an autogenerated private key); to lookup using the primary keys; and to look up just the provenance trail of a particular experiment setup, using a primary key for the experiment.

```
type db = list (exp full)
val insert: exp full → int
val lookup: xid → option (exp full)
val lookupProof: r:expsetup → xid:int → option (proof full (Valid r))
```

We implement a client-facing interface to the database that wraps the basic lookup and insert operations. On outbound request, we lookup an experiment by its primary key. But, rather than communicate a (potentially large) proof term with explicit provenance to the requestor, we erase the proof (using Prune) and send only a partial proof to the caller, recording the primary key xid in the proof term for later reconstruction. In our full implementation, rather than simply sending a Prune node, we send an authenticated proof term, signed under a key for the database, so that the requestor can conclude that the returned experiment is indeed valid.

```
(* Erasing outbound proofs *)
assume ∀(b:bool) (r:expsetup) (pf:proof b (Valid r)). b=full ⟹ Valid r
let readExp xid : option (exp partial) =
  match lookup xid with
    | Some (xid, r, pf) → Some (xid, r, Prune r xid)
    | None → None
```

To use the Prune constructor, we have to prove that r has the type r:expsetup{Valid r}. Although pf is full proof of Valid r, we cannot use pf directly to derive ghost refinement formulas. To connect concrete and ghost refinements, we introduce the assumption above. Given the soundness of the proof kernel, this assumption is admissible, and the type of Prune ensures that the database program never introduces partial proofs for experiments that do not have a valid provenance trail. Despite the fact that the Valid r type has no inhabitants, the introduction of this assumption does not lead to logical inconsistency. Formally, assumptions are simply recorded as effects in the log, and do not produce values that can be destructed, say, via pattern matching.

Finally, on requests to insert new records in the database, we can reconstruct proofs. The function expand below traverses the structure of a proof tree, and expands Prune nodes by looking them up in the database. The database maintains an invariant that each record in the database has a full proof and thus a fully explicated provenance trail, ensured via type soundness.

```
(* Reconstructing proof terms on inbound requests *)
let rec expand (c:bool) (pf:proof c α) : option (proof full α) =
  if c=full then Some pf
  else match pf with
    | Prune r xid → lookupProof r xid
    | AndElim1 c1 pf → (match expand c1 pf with
                        | Some pf' → Some (AndElim1 full pf')
                        | _ → None)
    | ...
(* Inserting a new record in the DB *)
let insertExp (r:expsetup) (c:bool) (pf: proof c (Valid r)) =
  match (expand c pf) with
    | Some pf' → Some (insert (None, r, pf'))
    | _ → None
```

The function expand is, in effect, a partial, effectful proof-search procedure. Despite the use of non-termination and effects, the type system guarantees that if this function terminates and returns Some pf, then pf is indeed a valid full proof in the P-fragment, F*'s logically consistent fragment of total functions.

3. Formalizing F*

This section presents the syntax and the semantics of F*. We focus on five main themes: (1) the stratification into expressions, types and kinds with the ability to describe functional dependences at each level; (2) the use of kinds to isolate sub-languages for proofs, specifications, and affinity; (3) relating logical effects described using ghost refinements to propositions witnessed by proof terms; (4) logic parametricity, allowing us to plug-in proof kernels and automated decision procedures for the logics they define; and (5) the consistency of a core universe of propositions, via strong normalization, and the ability to program over its values, to support applications with mobile proofs and selective erasure.

3.1 Syntax

The syntax of F* is shown below. Values include variables, lambda abstractions over values and types, and fully applied n-ary data constructors. The value v^ℓ is a technical device used to prove the soundness of affine typing—ℓ is an identifier drawn from a class of names distinct from term and type names. We use the notation $\overline{a}$ to stand for a finite sequence of elements $a_1,\ldots,a_n$, for arbitrary n; $(\overline{a})_k$ is a sequence $a_1,\ldots,a_{k-1}$. We adopt a (partially) let-normalized view of the expression language e, in particular requiring function arguments to always be values—this is convenient when using value-dependent types, since it ensures that expressions never escape into the level of types. The only other non-standard expression form is **assume** ϕ, which has the effect of adding a formula to the log and is explained in §3.3.

Types are ranged over by meta-variables t and ϕ—we use ϕ for types that stand for logical formulas. Types include variables α, constants T, dependent functions ranging over values whose domain may be values ($x{:}t \to t'$) or types ($\forall\alpha{::}\kappa.t$, written `'a::k -> t` in the concrete syntax), types applied to values ($t\ v$) and to types ($t\ t'$), type-level functions from values to types ($\lambda x{:}t.t'$, concretely written `fun (x:t) -> t'`), ghost refinements $x{:}t\{\phi\}$, and finally coercions to affine types ¡t. This modal operator serves to qualify the type of a closure that captures an affine assumption; we include ¡t in the formalism to avoid duplicating the rules for function arrows, but concretely we write affine functions as `x:t >> t'` and `'a::k >> t` instead of ¡($x{:}t \to t'$) and ¡($\forall\alpha{::}\kappa.t$).

Kinds κ include the four base kinds $\star$, P, A, and E—we distinguish the first three of these as *concrete kinds*, since they classify inhabitable types. As at the type level, we have kinds for dependent function spaces whose range are types and whose domain may be either values ($x{:}t \Rightarrow \kappa$) or types ($\alpha{::}\kappa \Rightarrow \kappa'$). Stratifying the language into terms, types, and kinds allows us to place key restrictions (discussed below) that facilitate automated verification, and

Syntax: expressions, types, kinds, signatures, environments

v	::=	$x \mid \lambda x{:}t.e \mid \Lambda\alpha{::}\kappa.e \mid D\ \bar{t}\ \bar{v} \mid v^{\ell}$	values
e	::=	$v \mid e\ v \mid e\ t \mid$ **assume** $\phi \mid$ **let** $x = e$ **in** e'	terms
	|	**match** x **with** $D\ \bar{\alpha}\ \bar{x} \to e$ **else** e'	
ϕ, t	::=	$\alpha \mid T \mid x{:}t \to t' \mid \forall\alpha{::}\kappa.t \mid t\ v \mid t\ t'$	types
	|	$\lambda x{:}t.t' \mid x{:}t\{\phi\} \mid \text{¡}t$	
c	::=	$\star \mid P \mid A$	concrete kinds
b	::=	$c \mid E$	base kinds
κ	::=	$b \mid x{:}t \Rightarrow \kappa \mid \alpha{::}\kappa \Rightarrow \kappa'$	kinds
S	::=	$\cdot \mid T{::}\kappa\{\overline{D{:}t}\} \mid S, S'$	signature
Γ	::=	$\cdot \mid x{:}t \mid \alpha{::}\kappa \mid v_1 = v_2 \mid t_1 = t_2 \mid \Gamma, \Gamma'$	type env.
$\mathscr{A}$	::=	$\cdot \mid \phi \mid \ell \mid \hat{\ell} \mid \mathscr{A}, \mathscr{A}'$	dynamic log

to compile efficiently to .NET. However, stratification does come at a cost—several pieces of technical machinery are replicated across the levels.

Signatures S are finite lists of inductively defined types. Each inductive definition $T{::}\kappa\{\overline{D{:}t}\}$ introduces a type constructor T of kind κ and all its constructors $D_1{:}t_1, \ldots, D_n{:}t_n$. For simplicity, we do not include mutually inductive types, although these are supported by our implementation and their addition to the formalism poses no significant difficulties. We do not need a fixpoint form in the expression language since inductive types allow us to encode recursive functions. To show that terms given P-kinded types are strongly normalizing, a well-formedness condition on signatures imposes a positivity constraint on inductive definitions for P-kinded types. An additional constraint on signatures is that they must contain a declaration $\mathsf{unit}{::}\star\{():\mathsf{unit}\}$ for the unit type and its one value $()$. Finally, typing environments Γ track in-scope value variables (x with type t), type variables (α with kind κ), and equivalences between values ($v_1 = v_2$) and types ($t_1 = t_2$) introduced when checking **match** expressions. We discuss the log $\mathscr{A}$ in conjunction with the dynamic semantics in §3.3. Briefly, $\mathscr{A}$ maintains a set of facts introduced by the dynamic assumption of ghost refinements, and also a set of names ℓ used to track affine values.

3.2 The F* type system

Figure 1 shows key rules from each judgment in the type system.

Well-formedness of kinds. The judgment $S;\Gamma \vdash \kappa\ \mathrm{ok}(b)$ states that κ is well-formed and produces types of base kind b (when $\kappa = b$) or type constructors for b-kinded types. The rule (OK-TK) shows a key enhancement of F* over prior languages, e.g., Fine or Linear LF. Types can be constructed from affine values ($b_1 = A$), so long as the type constructed is purely specificational ($b_2 = E$). As illustrated in §2.4 and §4, this improves the expressiveness of affine typing significantly, enabling refinements on affine state. (OK-KK) is also an enhancement over Fine to allow dependences and to ensure that types parameterized by affine types are themselves affine. Although our formalism allows higher-kinds like $(\star \Rightarrow \star) \Rightarrow \star$, such kinds cannot be compiled to the type system of the .NET bytecode language and are currently rejected by our compiler. However, retargeting F* to a platform with a more flexible typing discipline would lift this restriction.

Kinding for value-indexed and affine types. The judgment $S;\Gamma \vdash t :: \kappa$ states that type t has kind κ. The rules shown here reveal a few subtle aspects of the type system, starting with affine typing and affine-value indexing. The rule (K-A) shows how the modal operator coerces the kind of a type. (K-Tv) allows a type function t to be applied to a value v; these type functions are introduced either using (K-Fun) or as type constructors T in the signature. Next, recall that we wish to allow affine values to be freely used at the type level, since specifications should not consume affine resources. For this reason, unlike prior languages, the values passed to type functions may use affine assumptions in the context Γ—the restrictions imposed by (OK-TK) ensure that such uses of affine assumptions at the type level cannot influence term-level reduction. The second premise of (K-Tv) uses the expression typing judgment, discussed shortly. This judgment has two modes ($m ::= \cdot \mid \varepsilon$) indicated on the turnstile. When the mode is ε (indicating that the term being typed occurs at the type-level, effectively as an index of an E-kinded type), affine assumptions in the context can be freely duplicated without resulting in their consumption. We discuss how this works shortly, in the context of the (T-X) rule.

Ghost refinements, total functions, and sub-kinding. (K-ϕ) requires that formulas in ghost refinements be erasable (E-kinded). Formulas in ghost refinements are erased at runtime and refinements apply only to types given concrete kinds c (the first premise of (K-ϕ)), i.e., inhabitable types. (K-Arr) handles dependent function arrows, which (as seen in §2.1) can be used to represent both quantified formulas in the logic as well as term-level function abstractions. A function arrow is P-kinded if its range type is P-kinded. Finally, (K-Sub) uses a sub-kinding judgment in its premise. Sub-kinding in F* is defined by the judgment $S;\Gamma \vdash \kappa_1 <: \kappa_2$. Sub-kinding is based on the relation $P <: \star <: E$, among base kinds, with $A <: A$ only. These base relations are lifted into the structure of arrow kinds, with co- and contravariance as usual. (K-Sub) includes premises to ensure that sub-kinding preserves well-formedness of kinds. We conjecture that these premises can be eliminated in favor of lemmas establishing that sub-kinding never introduces ill-formed kinds. We have yet to prove it, so we include these premises to facilitate our formal proof of well-formedness of kinds produced by derivations.

Expression typing. The judgment $S;\Gamma;X \vdash^m e : t$ states that expression e has type t, under signature S, environment Γ, and an affine environment $X ::= \cdot \mid \ell \mid x \mid X, X'$, where X, X' denotes disjoint union of sets of names. A well-formedness condition on contexts requires all variables $x \in X$ to also be bound in Γ. The context X represents a set of available affine assumptions, and usual context splitting rules apply to X when typing the sub-terms of an expression. (As in Fine, we choose not to split Γ itself, since this complicates well-formedness of contexts in the presence of dependent types.) Finally, as mentioned above, expression typing comes in two modes, indicated on the turnstile.

Affine typing in two modes. (T-XA) is typical of affine typing systems. To use an affine assumption x, we require x to be present in the affine environment X. In addition, (T-X) provides two exceptions to (T-XA): first, as is standard, we can use non-affine assumptions without requiring them to be present in X. Second, when the mode is ε, we are typing a term at the level of types; since this does not consume the affine resource, we are free to use it even when X is empty. (T-WknX) provides weakening for the affine context. Finally, (T-Abs) is a standard rule with one subtlety that the introduced function type is tagged with the affine modality (using $Q(X, x{:}t \to t')$) if the function closure captures an affine assumption. ((T-ℓ) is also related to affine typing, but, values v^{ℓ} are used solely to state our type soundness result—we discuss this rule in conjunction with the dynamic semantics.)

Ghost refinements and subtyping. The rules (T-As), (T-V), and the subtyping judgments introduce ghost refinement types. (T-As) introduces a unit refined with the assumed formula ϕ—no logical evidence is produced for ϕ. To justify this rule, the dynamic semantics of this expression adds a formula to the dynamic log $\mathscr{A}$. (T-V) allows value v to be refined with the formula ϕ when ϕ is derivable: $S;\Gamma' \models \phi$. The context Γ' includes bindings $x{:}t, x = v$ which allow the derivability relation to use information about v; however, for kind-correctness, we require the kinding of the introduced formula to not rely on the introduced equality. Ghost refinements have no impact on the representation of values, so they admit structural subtyping. The subtyping rules include (S-ϕI), like (T-V), an in-

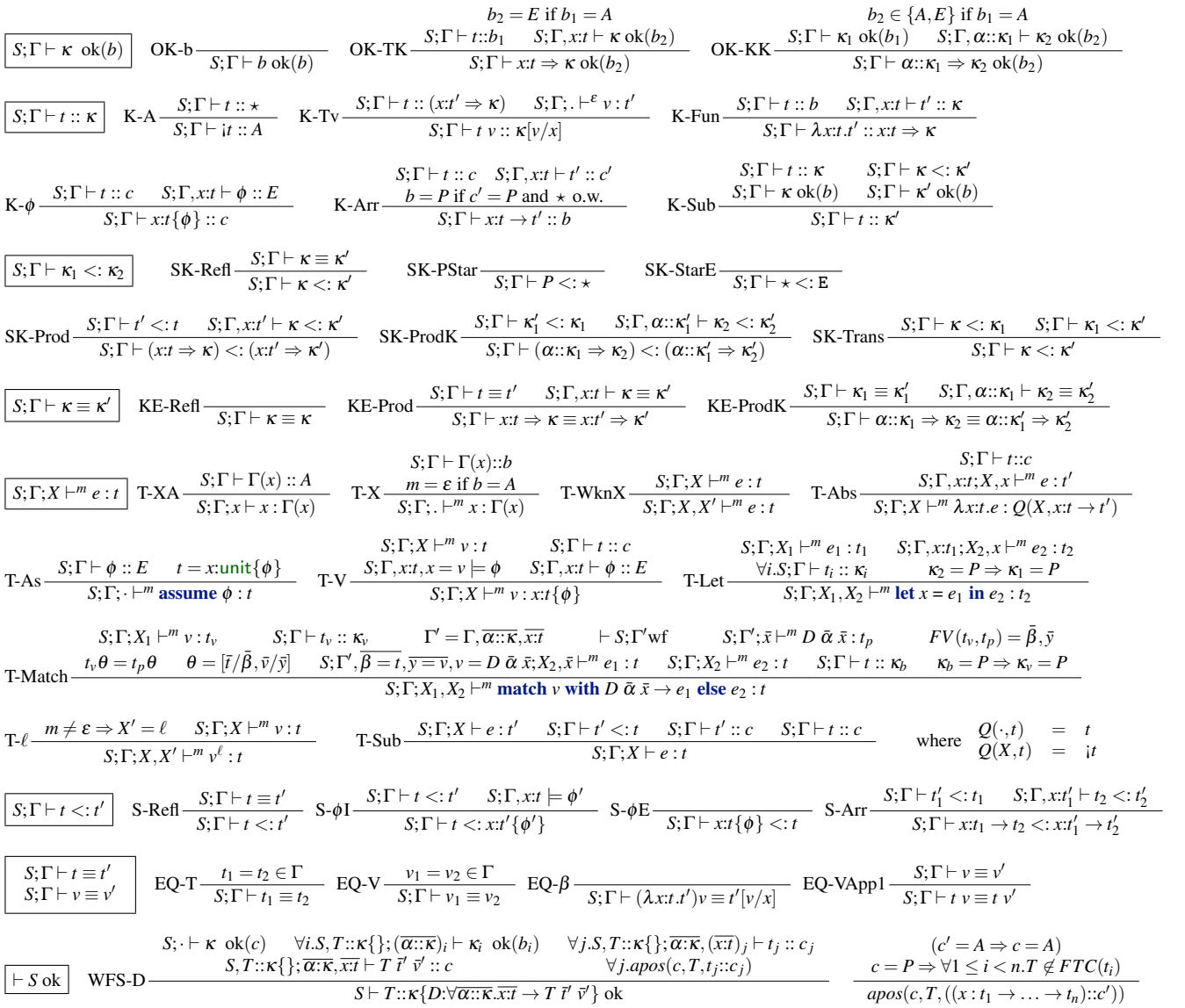


Figure 1. Selected static semantics of F*

troduction form for ghost refinements and (S-ϕE), an elimination form. (S-Arr) lifts the relation into the structure of function types—omitted rules do the same for other constructions. In contrast, concrete refinements (i.e., the dependent pairs of Coq, Fine etc.) do not enjoy this structural subtyping relation. Instead, via a systematic translation to insert coercions (called *derefinement*) inherited from Fine, F* provides programmers with a weaker, non-structural subtyping relation on concrete refinements. Subtyping also includes type conversion, an equivalence relation according to the judgment $S;\Gamma \vdash t_1 \equiv t_2$. This relation, discussed briefly in conjunction with (T-Match), converts types using equations that appear in the context, and is available everywhere within the structure of types.

Logic parametricity is embodied by (T-V) and (S-ϕI): various definitions of the derivability relation ($S;\Gamma \models \phi$) can be "plugged in" to the type system, as long as the relation meets a few important admissibility constraints. Admissible relations must at least be the identity on refined assumptions ($S; x : (x{:}t\{\phi\}) \models \phi$); be closed under substitution of free variables in Γ; be allowed to use structural rules such as weakening, duplication and permutation; and be insensitive to the presence of derivable equality assumptions in the context ($S;\Gamma, v = v \models \phi \iff S;\Gamma \models \phi$). Pragmatically, we often plug in a decision procedure for first order logic with additional theories, as implemented by the Z3 theorem prover—the ability to use structural rules (e.g., weakening) in the logic, enabled by our restrictions on the use of affine indices in types, makes it easy to support automation. Formally, we also exploit logic parametricity to provide an embedding of F7's formal core, RCF, into F*, plugging into (T-V) RCF's entailment relation (which, unlike F*, includes the basic first-order connectives and equality, each satisfying their usual introduction and elimination forms).

Consistency of the P-universe requires placing restrictions in various parts of the type system. For an expression e to reside in P, e must be free of non-P expressions, since those may diverge—the last premise of (T-Let) enforces this property. We disallow

discriminating on values that reside in $\star$ or A when constructing propositions in the branches of a **match**—the last premise of (T-Match) shows this. We also have constraints on the well-formedness of inductive type T (recall that we use $(\overline{\alpha::\kappa})_i$ to mean $\alpha_1::\kappa_1,\ldots,\alpha_{i-1}::\kappa_{i-1}$). In (WFS-D), the first premise ensures that the kind κ of T is well-formed; the second and third premises ensure that the arguments of D are well-formed; the fourth premise ensures that the constructed type matches the kind expected for T; and the final premise imposes two key restrictions: (1) constructors with affine arguments must construct affine results—this is unrelated to totality of P-functions; (2) a positivity constraint on inductive P-kinded types. We use a relatively simple version of positivity, excluding the constructed type T in negative position in any argument of D.

These constraints are similar to those imposed by Aura on its Prop universe—to our knowledge, Aura is the only other language that embeds a strongly normalizing core within a language with arbitrary recursion. However, there are several important differences. First, Aura (like Coq) insists on Prop terms being computationally irrelevant, so its match rule forbids cross-universe elimination—Prop terms cannot be eliminated to construct values in Type. We explicitly wish to program over proofs, so F* permits P-to-$\star$ elimination. Next, Aura does not allow the branches of a match expression to use equality assumptions between the pattern and the scrutinized term. This makes it impossible to program on proof terms, as shown in §2.5, which make essential use of GADT-style programming patterns. In contrast, (T-Match) checks the **then**-branch e_1 in a context that includes equality assumptions induced by unifying the type of the pattern with the type of the scrutinee (the first three premises on the second line). These equality assumptions are used in $S;\Gamma\vdash t\equiv t'$ to allow typing derivations to freely refine both type and value indices within types. This feature complicates the consistency proof of F*'s P-fragment. We discuss this aspect when presenting the metatheory of F*.

3.3 Dynamic semantics: logical effects and affine names

The dynamic semantics of F* is a small-step reduction relation for a call-by-value language. It has the form $(\mathscr{A};e)\to_S(\mathscr{A};e')$ where the signature S is unvarying. The interesting part of this relation is the log $\mathscr{A}$, a non-decreasing set of logical facts ϕ and names ℓ for affine values. As mentioned in §2.2, the set of facts in $\mathscr{A}$ is used in the definition of ghost refinement derivability. Facts are added to the log using the **assume** ϕ form, which reduces as below.

$$\text{E-Log}\ \frac{}{(\mathscr{A};\textbf{assume }\phi)\to_S(\mathscr{A},\phi;())}$$

Foreshadowing our safety condition for ghost refinements (Corollary 1), we show that, when a well-typed program $e:x{:}t\{\phi\}$ reduces to a value, that is $(\cdot;e)\to_s^*(\mathscr{A};v)$, its refinement formula ϕ is derivable from the signature and all the accumulated logical effects: $S;\mathsf{Facts}(\mathscr{A})\models\phi[v/x]$. This is in contrast to our soundness result for proof terms, expressions that reside in the P-fragment, for which we obtain a more traditional logical consistency property. In a distributed program, the log is an idealized global view of the logical state of all participants. Ghost refinements accompanied by cryptographic evidence (in the form of digital signatures) enable speaking about this distributed state.

The log also tracks affine values. We aim to show that well-typed programs destruct affine values at most once. For this purpose, we instrument the dynamic semantics to tag an affine value v with a fresh name when it is introduced, recording the name in the log (E-Newℓ). Names held in the log come in two variants: names ℓ are "live", while names $\hat{\ell}$ are "dead". When v appears in a destruct position (the context E in (E-Killℓ), which includes the function position of a β redex, and the scrutinee position of a match), we require the name ℓ to be "live" in the log for reduction to not get stuck, kill the name in $\mathscr{A}$, and then proceed. This instrumentation serves as our specification of the use-at-most-once property.

$$\text{E-New}\ell\ \frac{S;\mathsf{Facts}(\mathscr{A})\vdash t::A\qquad\text{fresh }\ell\qquad S;\mathsf{Facts}(\mathscr{A});\mathsf{LiveNames}(\mathscr{A})\vdash v:t}{(\mathscr{A};v)\to_S(\mathscr{A},\ell;v^\ell)}$$

$$\text{E-Kill}\ell\ \frac{}{(\mathscr{A}_1,\ell,\mathscr{A}_2;E[v^\ell])\to_s(\mathscr{A}_1,\hat{\ell},\mathscr{A}_2;E[v])}$$

3.4 Metatheory of F*

Our first theorem is a type soundness result, stated in terms of standard progress and preservation lemmas. In addition to well-typed programs not getting stuck, this result ensures that affinely typed values are destructed at most once, and can thus be soundly implemented using destructive reads and mutation. (The theorem relies on an auxiliary judgment $S\vdash\mathscr{A}\Longrightarrow\Gamma;X$ which obtains a context $\Gamma;X$ from the dynamic log, where Γ includes the logical assumptions accumulated in $\mathscr{A}$, and live names in $\mathscr{A}$ recorded in X.)

THEOREM 1 (Type soundness). *For all S, $\mathscr{A}$, Γ, X, X', e, and t such that*

(1) $S\vdash\mathscr{A}\Longrightarrow\Gamma;X$,
(2) $\vdash S;\Gamma;X$ *wf, and*
(3) $S;\Gamma;X'\vdash e:t$ *where* $X'\subseteq X$,

either e is a value, or there exist $\mathscr{A}'$, e', Γ', and X'' such that

(1) $S\vdash\mathscr{A}'\Longrightarrow\Gamma';X''$,
(2) $X''=(X'\setminus\mathsf{DeadNames}(\mathscr{A}'))\cup\mathsf{LiveNames}(\mathscr{A}'\setminus\mathscr{A})$,
(3) $\exists\Gamma''.\Gamma'=\Gamma,\Gamma''$, *and*
(4) $S;\Gamma';X''\vdash e':t$.

From type soundness, we obtain our main safety property for ghost refinements: their formulas are derivable from the logical effects accumulated in the log.

COROLLARY 1 (Safety for ghost refinements). *For all S, $\mathscr{A}$, Γ, e, ϕ, t, $\mathscr{A}'$, Γ', and v such that $\vdash S$ ok, $S\vdash\mathscr{A}\Longrightarrow\Gamma;X$, $S;\Gamma;X\vdash e:x{:}t\{\phi\}$, and $(\mathscr{A};e)\to_S^*(\mathscr{A}';v)$, there exists Γ' such that $S\vdash\mathscr{A}'\Longrightarrow\Gamma';\_$ and $S;\Gamma'\models\phi[v/x]$.*

Theorem 2 (Logical consistency of P) shows logical consistency of the P-fragment, by translating P-terms into CiC (The Coq Development Team 2010), which is widely believed to be strongly normalizing. Intuitively, the theorem states that the translation of F*'s P-fragment into CiC is a (weak) simulation, with regard to F* and CiC's reduction relations. Since in CiC, all reduction sequences are presumed to be finite, this guarantees the absence of infinite reduction sequences for F*'s P-terms. A direct proof of strong normalization of F*'s P-fragment is likely to be challenging since it involves inductive types like CiC. Our proof strategy borrows ideas from a proof of a related property in Aura. However, unlike Aura, F* supports implicit type conversions using equalities introduced by pattern matching—we find this essential for programming over proof terms. We translate this to CiC through the use of explicit, equality-witnessing conversions in CiC.

In the statement of the theorem below, $(S;\Gamma;\cdot\vdash_p e:t\hookrightarrow\tau)$ is the translation that marks non-P-lambdas in F* term e and translates them to a CiC term τ. $(\rightsquigarrow)$ is the reduction of P-terms; and $(\longrightarrow_{\beta,\iota,\delta})$ is term conversion in CiC.

THEOREM 2 (Logical consistency of P). *For all S, Γ, p, e, e', t, τ such that $\vdash S;\Gamma$ ok$_E$ and $S;\Gamma\vdash t::P$, if $S;\Gamma;\cdot\vdash_p e:t\hookrightarrow\tau$ and $e\rightsquigarrow e'$, then there exists τ' such that $\tau\longrightarrow_{\beta,\iota,\delta}^*\tau'$ and $S;\Gamma;\cdot\vdash_p e':t\hookrightarrow\tau'$.*

Theorem 3 (Well-typed translation of RCF) is an embedding into F* of a core fragment of RCF without Public/Tainted kinds, without concurrency, and with restrictions on the use of RCF's isorecursive types. In the statement below, A is an RCF configuration; E is an RCF context; and $A \to A'$ is a single step of reduction in RCF. The notation $[\![\cdot]\!]$ means translation. The judgment $E \vdash \diamond$ means that E is well-formed. The translation is over the structure of RCF typing derivations. The theorem states, roughly, that well-typed RCF terms translate to well-typed F* terms, and that the translation is a simulation, i.e., reduction steps in RCF correspond to reductions in F*.

THEOREM 3 (Well-typed translation of RCF). *Given* $E \vdash [\![A]\!] = (\mathscr{A}; e)$, $\vdash [\![E]\!] = S; \Gamma$, *and* $E \vdash \diamond$, *we have* $S; \Gamma; . \vdash e : t$ *where* $E \vdash [\![T]\!] = t$. *Additionally* $A \to A'$ *if and only if* $(\mathscr{A}; e) \to_S (\mathscr{A}; e')$ *and* $E \vdash [\![A']\!] = (\mathscr{A}'; e')$.

3.5 Coq formalization

The proofs of the metatheory of F* were initially done by hand. As we reached completion of the hand proofs, we started a formalization of the meta-theory of F* in the Coq proof assistant, using the SSREFLECT extension (Gonthier et al. 2011). At the time of writing, we had formalized the whole calculus definition, along with a major part of its metatheory, including all the key technical lemmas up to substitutivity, and a partial proof of the type soundness result. We found a few oversights in our pencil and paper proofs, which we have since corrected. We expect to have a fully mechanized type soundness proof in the near future. A more challenging, next step is a formalization of the embedding of the propositional part of F* to CiC. We plan to use the Barras's formalization of CiC[1] as a basis.

Our formalization in Coq is noteworthy in several ways. First, this is the first time the SSREFLECT package has been used to carry out a large development in programming language metatheory. We started out attempting to use the Coq code generator Ott (Sewell et al. 2010) to help reduce the gap between the formal and informal descriptions of our type system. Although this did help in the maintenance of the two versions of the type system, we found that with the many different kinds of variables in F*, the code produced by Ott resulted in a very large, incomplete set of lemmas.

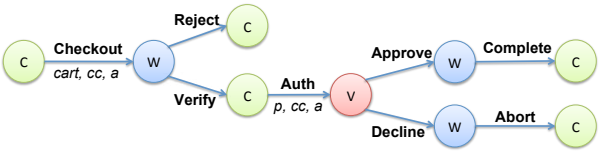
Following this experience, we have developed a new framework for metatheory using SSREFLECT, and use it on F*—a particularly challenging test case, since it involves many kinds of names and binders, with subtle differences across the levels of terms, types and kinds. Furthermore, most central judgments in the type system are mutually recursive. Despite these complications, we are happy to note that our framework has allowed us to develop short, largely automated proofs. Our experience is encouraging initial progress towards an eventual goal: a general framework, based on the reflection pattern and the theory of the pure lambda calculus, dedicated to the study of type systems. We think that the development of the F* formal meta-theory will serve as a basis of such a framework.

4. Secure multi-party sessions in F*

This section shows how to use the libraries and programming idioms in §2 to write and verify realistic distributed security applications. The combination of cryptographic evidence, ghost refinements, and affine types proves crucial for this case study and enables more precise specifications and stronger proofs than earlier results obtained using F7 (Bhargavan et al. 2009).

4.1 Multi-party sessions

Multi-party sessions (Bhargavan et al. 2009; Deniélou and Yoshida 2011; Honda et al. 2008) offer a powerful method to structure and build distributed message-based applications when their message flow is fixed beforehand. Consider a 3-party session between a customer (c), a website (w), and a credit-card verifier (v), with the message flow below:



The customer initiates a Checkout session for buying some items (*cart*), billed to her credit card *cc* for the total amount *a*. The web site then either rejects the transaction outright, or asks for credit card verification. The customer is redirected to a verification server and provides a password (p) to authorize payment of a on her credit card *cc*. The verifier then either confirms or declines payment to the web site, who completes or aborts the session accordingly.

Such a session specifies a contract between component programs in a distributed application. Every program promises to play one role of the session, and in return, it expects the others to correctly play their roles. For instance, w promises not to charge more than a, and not to abort the transaction if the payment is approved: c can rest assured that if she receives an Abort message, her credit card has not been charged.

A variety of type systems have been proposed to verify that a program complies with a session role, and each type system is tailored to a specific set of session primitives and programming language features. Instead, we encode multi-party sessions as F* types. By standard F* typing, we can verify that a program correctly plays a session role. Even if some programs deviate from their role at run-time (because they have been taken over by an attacker, for example), we show how the rest of the application can protect itself by using a custom cryptographic protocol.

4.2 A session API in F*

We define a generic session API for distributed applications to enforce a multi-party session discipline. We start with a simplified version of our API, then build up to showing our model of more complex features.

To begin with, we ignore the values (*cart*, *cc*, a, p) passed in the session and aim to control the sequence of messages a session participant can send and receive. Using affine types, we can define a type for a *role process*, **type** role0::$E \Rightarrow A$, where the parameter of the role is a type describing an automaton. The types used to define these automata are purely specificational—they are given E-kind. A value of a role process type is a handle that gives a program the capability to enact the automaton. We show two simple automata types provided by our API, and a function that consumes and returns a role handle.

```
type Send0 :: 'm::⋆ ⇒ 'k::('m ⇒ E) ⇒ E
type Done :: E
val send: m:'m →role0 (Send0 m 'k) →role0 ('k m)
```

The Send0 automaton is indexed by two types—the first, a type 'm of the message to be sent by the process; the second, a type 'k representing a *continuation process*, where the process is dependent on the value of type 'm sent in the first state. Done represents a finished automaton. Using these automata, we can define the following role process type that represents a program that first sends an integer x, then an integer y greater than x, and then concludes. (In concrete syntax, we write **fun** (x:t) ⇒ t' for a type-level function $\lambda x{:}t.t'$; **fun** _ ⇒ t' ignores its type argument.)

```
role0 (Send0 int (fun (x:int) ⇒ Send0 (y:int{y > x}) (fun _ ⇒ Done)))
```

Our full API generalizes the automata types above with the notion of a global distributed store for session values; each participant maintains a local view of the store and we ensure, by typing, that

[1] http://www.lix.polytechnique.fr/~barras/proofs/sets/

these views are consistent. We show below the extended analogs of role0, Send0, and also automata types for receiving a message and for choice-points in the session graph. The type of a role process (role) is parametrized by a store value (of type 'st); automata types (Send, Recv) are indexed by binary predicates δ on 'st values that define the allowed changes to the store during the next step. The function send allows a client to send a message m and update the store from s0 to s1 given that the current role process is a Send and that the stores satisfy the predicate δ attached to Send—client programs calling send have to prove δ s0 s1 for some specific instantiation of δ, and our type checker uses Z3 to assist with the proof of such ghost refinement properties.

```
type role:: 'st::⋆ ⇒ E ⇒ 'st ⇒ A
type Send:: 'm::⋆ ⇒ 'st::⋆ ⇒ 'm ⇒ ('st ⇒ 'st ⇒ E) ⇒ 'k::('m ⇒ E) ⇒ E
type Recv:: 'm::⋆ ⇒ 'st::⋆ ⇒ 'm ⇒ ('st ⇒ 'st ⇒ E) ⇒ 'k::('m ⇒ E) ⇒ E
type Choice:: 'l::E ⇒ 'r::E ⇒ E
val send: m:'m →s0:'st →s1:'st{δ s0 s1}
        →role (Send 'm 'st m δ 'k) s0 →role ('k m) s1
```

The code below shows how we can use this session API to model the website role (w) in the example session of §4.1. The type msg defines the set of messages and the type store is the type of the distributed store (including, for this example, the names of each participant in the session, their view of the contents of the shopping cart, etc.) The process automaton involves an alternation of message send and receive, and this type uses two store update predicates (of kind store ⇒ store ⇒ E): Update_id_c_v_cart_cc_a allows initial assignments from the customer to id, c, v, cart, cc, and a; then Unchanged disallows any changes.

```
type msg = Checkout | Reject | Verify | Auth | Approve | (...)
type store = {id:nat; c:prin; w:prin; v:prin; cart:string; (...) }
type proc_w =
  Recv Checkout Update_id_c_v_cart_cc_a (fun _ ⇒ Choice
    (Send Reject Unchanged (fun _ ⇒ Done)
    (Send Verify Unchanged (fun _ ⇒ Choice
      (Recv Approve Unchanged (fun _ ⇒
            Send Complete Unchanged (fun _ ⇒ Done)))
      (Recv Decline Unchanged (fun _ ⇒
            Send Abort Unchanged (fun _ ⇒ Done))))))
```

Type soundness ensures that a well-typed program is guaranteed to comply with its declared role process. For example, a program that joins a session in role w obtains a role handle of type role proc_w init_store_w. It may then call the receive function (the counterpart of send, not shown here) to receive a Checkout message but cannot call send; subsequently, it may call send with either a Reject or a Verify message, but not both.

In earlier work, Bhargavan et al. (2009) showed how to encode multi-party sessions as refinement types in F7. However, since the F7 type system does not support generic predicate-indexed types, such as Send above, they encode the session using large session-specific logical formulas rather than types. Our use of F⋆ higher-order kinds here yields session specifications that are, in general, one-third the size of the corresponding F7 specifications. Moreover, F7 lacks affine types, and they have to prove by hand, with the help of an awkward continuation-passing style encoding, that their applications use role handles linearly.

4.3 Custom cryptographic protocols for session consistency

Distributed applications typically run in an untrusted environment, where the network and one or more of the session participants may be under the control of malicious adversaries. In this scenario, cryptographic mechanisms, such as digital signatures, can be used ensure that all honest session participants have consistent states. For example, when the client c receives an Abort message from the website w, it may demand that this message include a valid signature proving that card verifier v Declined, to prevent a malicious w from double-crossing c.

Bhargavan et al. (2009) show how to systematically use cryptographic evidence as proof of session compliance. They compile multi-party sessions to efficient custom cryptographic protocols that exchange and check a minimal number of digital signatures to ensure global session consistency. Their compiled protocols use session types and cryptography in the style of F7: without higher-order kinds, affinity, or predicate-indexed types.

We implement secure multi-party sessions in F⋆ using protocol libraries adapted from those of Bhargavan et al., but instead using the crypto library of §2.3 and the sessions API shown above. In our example session, the Abort message from w to c carries two digital signatures, one from w and one from v, each signature authenticating the last message sent by the corresponding principal and the values in its store at that time. On receiving the Abort message, c verifies these signatures and checks that they conform to the session type: in particular, that the signature from v says that v sent a Decline message and not an Approve. The resulting type for the recv_Abort function (a specialization of the generic receive function in our API) is as follows:

```
type Aborted:: w:prin ⇒ st_w:store ⇒ E
type Declined:: v:prin ⇒ st_v:store ⇒ E
val recv_Abort:
  st_c:store →role (RecvCompleteOrAbort) st_c →
  (st_c':store ∗ role Done st_c'){
    Unchanged st_c st_c' ∧
    Says st_c.w (∃ st_w. Aborted st_c.w st_w ∧ Unchanged st_w st_c') ∧
    Says st_c.v (∃ st_v. Declined st_c.v st_v ∧ Unchanged st_v st_c')}
```

The function takes the current store st_c at c and a role handle for c that must be in the state after c has sent Auth. The function returns an (unchanged) store st_c' and a new (completed) role handle. The E-kinded predicates Aborted and Declined represent the session states at the other roles. For example, Declined p st means that the principal p, playing role v previously sent a Decline when it had a store st. Hence, the post-condition says that the principals st_c.w and st_c.v (playing w, v) claim to be in the states Aborted st_c.w st_w and Declined st_c.v st_v where the stores st_w and st_v are the same as c's store st_c'. So if v is honest, then even if w is malicious, it cannot cause c to accept an Abort unless v sent a Decline. Note that the post-condition is a ghost refinement that is proved here using a combination of cryptographic evidence and F⋆ typechecking.

4.4 Encoding advanced session constructs in F⋆

The four automata types shown above are adequate to represent a wide variety of static sessions that do not use delegation or parallelism. Adding constructors for recursive sessions is straightforward. We now show how to extend our API to capture a limited form of parallelism, inspired by the dynamic multi-role session types of Deniélou and Yoshida (2011).

Distributed applications often run several instances of the same role in parallel, for scalability. For example, a web site may run several copies of a web server all connected to the same backend database. Or, a client may fork several processes that may communicate with a server in parallel, in an arbitrary order. To verify such applications we extend our sessions API with three new automata types: Fork, Join, and Await.

The Fork automaton (given below) enables a role to fork multiple instances of a child role, transfer control to them, and then wait for them to complete. These child role processes may either execute sequentially (in any order), or in parallel. Each child process is given a unique principal name which it can use when communicating with its parent or with other roles. The Join automaton enables the child role to transfer control back to the parent; and Await represents a parent role process waiting for its children to complete.

We illustrate Fork and its use in an application below, where we elide the store for simplicity (and so use role0 instead of role).

```
type Fork :: ps:list prin ⇒ 'ParentProc::E
          ⇒ 'ChildProc::(role0 (Await ps 'ParentProc) ⇒ prin ⇒ E)
          ⇒ E
let go ps : role0 Done =
  let client = startClient ps in
  let client, children = fork ps client in
  let children =
    map (fun (q, child) →
           let child = send0 Request child in
           let Response, child = receive0 child in
             (q, child)) children in
    join ps client children
```

The function go forks a number of children (indexed by a given list of principals ps). Each child sends a Request message, then receives a Response message and then joins its parent (p). Here, the variable client is a role handle that has an automaton type of the form Fork ps Done ChildRole, where the automaton ChildRole sends a Request, receives a Response and then Joins its parent. Since role handles have an affine type, the code here passes client in and out of every session operation. The variable children is given an affine list type, which guarantees that the different child processes cannot interfere with each other; in other words each Response can be accurately correlated with its corresponding Request.

5. Implementation and Measurement

This section describes the implementation of our prototype F⋆ compiler and its performance measured on a variety of programs (about 20,000 lines of code in total), including cloud applications, cryptographic protocols, and secure browser extensions.

Compilation. The F⋆ compiler consists of about 35,000 lines of F# code and is still under active development. It is based on the type-preserving compiler for Fine (Chen et al. 2010). It takes as input an F⋆ program and typechecks the program by asking logical queries of Z3. The compiler also accepts Fine and F7 programs and translates them into F⋆. Source programs are then compiled to RDCIL, a small extension of a functional core of the .NET bytecode language CIL. Like DCIL, the target language of Fine, RDCIL extends CIL with type-level functions and value parameters (in addition to type parameters) in class declarations, to model value-dependent types in the source language. Unlike DCIL, RDCIL also supports ghost refinements. RDCIL encodes these additional type constructs as custom attributes, so RDCIL binaries can run on stock .NET virtual machines, access libraries of other .NET languages (e.g., C# and F#), and be called from those languages. RDCIL is fully typed and can be verified (with the help of Z3) for security, the same way F⋆ is verified.

Checking two forms of refinements in RDCIL**.** Ghost refinement is a new feature of F⋆ and RDCIL, not supported by Fine or DCIL. Ghost refinements in F⋆ are translated to ghost refinements in RDCIL, and the typechecker for RDCIL verifies them using Z3.

Concrete refinements are handled similarly in F⋆ and Fine. During source typechecking, the F⋆ compiler extracts proofs of concrete refinements from the SMT solver and injects them as terms in the generated RDCIL. Hence, concrete refinements in the source program are translated to a pair of a value and an explicit proof in RDCIL, which can be verified by the RDCIL type checker.

Reducing the size of generated bytecode. Explicit proofs can be costly though. Carrying proofs increases the code size by 50x for a Fine benchmark. The F⋆ compiler addresses this difficulty by relying on the RDCIL typechecker to reconstruct proofs by refinement type checking, rather than just depend on explicit proofs. As a result, RDCIL programs contain far fewer proofs compared to DCIL, and the overhead of proofs and types is only 60% for our benchmarks. The F⋆ compiler also reduces the size of generated bytecode (ignoring proofs and custom attributes for types), because higher-order dependent kinds allow more concise translation of polymorphic types and higher-order code, which are prevalent in F⋆ programs. Combining the two factors, the F⋆ compiler produces binaries an order of magnitude smaller than those produced by Fine, as much as a 45x improvement.

5.1 Benchmarks and Measurements

Code size. We compile the Fine benchmarks in (Chen et al. 2010) with the F⋆ compiler, treating all refinement types as ghost refinements. This way, no proofs are extracted. The table below shows the names of the benchmarks (column Bench.), the F⋆ code size (in bytes) with and without custom attributes for the additional types (A+ and A- respectively), and Fine size (in bytes) with and without proofs reported in (Chen et al. 2010) (Pf+ and Pf- respectively). The Fine numbers reflect only proof overhead, not attributes.

	F⋆ size		Fine size		A+/	A-/	A+/	A-/
Bench.	A+	A-	Pf+	Pf-	Pf+	Pf+	A-	Pf-
Authac	15k	12k	30k	20k	0.50	0.4	1.3	0.6
Iflow	27k	18k	840k	30k	0.03	0.02	1.5	0.6
Automaton	28k	15k	40k	20k	0.70	0.38	1.9	0.8
HealthWeb	76k	48k	2.1M	80k	0.04	0.02	1.6	0.6
Lookout	147k	81k	1.8M	120k	0.08	0.05	1.8	0.7
ConfRM	72k	51k	3.3M	110k	0.02	0.02	1.4	0.5
Total	365k	225k	8.1M	380k	0.05	0.03	1.6	0.6
ProofLib	7M	5M	51M	51M	0.14	0.1	1.4	0.1

Because of no proofs, the code size overhead is simply the custom attributes for encoding more expressive types than the CIL types. Column "A+/A-" shows that RDCIL code with those custom attributes is about 1.3x-1.9x of the code without the custom attributes, with an average 60% overhead for the custom attributes. Our current implementation simply uses compressed strings of pretty-printing types as custom attributes. A smarter encoding may further reduce the size overhead.

The RDCIL code is about an order of magnitude smaller than the DCIL code for the Fine benchmarks. Column "A+/Pf+" shows that the RDCIL code (with custom attributes) is about 3%-70% of the DCIL code (with proofs), with an average of 5%—a 20x improvement. Column "A-/Pf+" shows that the RDCIL code (without custom attributes) is about 2%-38% of the DCIL code (with proofs), with an average of 3%—indicating a 30x improvement in this configuration, although the accurate breakdown is hard to obtain because Fine numbers do not include custom attributes. Benchmarks with less proofs, e.g., Authac and Automaton, show less reduction. Column "A-/Pf-" shows that the pure code size of RDCIL is about 10% to 80% of that of DCIL, with an average of 60%—a 40% reduction because of a more expressive type language. Prooflib is purely refinement-free code. The 10x reduction in code size is entirely due to dependent higher kinds.

Compilation and typechecking times. The table below shows the time taken to typecheck and compile the Fine benchmarks as well as several new F⋆ programs we develop ourselves. For each program, it shows number of lines of source code (LOC), source parsing and checking time (SC, in seconds), compilation of F⋆ to RDCIL time (Trans), target checking time (TC), and the number of queries made to Z3 by the source checker (SQ) and target checker (TQ). All measurements were performed on a 2.67 GHz two-core Intel Core i7 CPU running Windows 7.

Bench.	LOC	SC	Trans	TC	SQ	TQ
Authac	37	0.2	0.1	0.2	1	1
Iflow	119	0.8	0.4	0.5	25	18
Automaton	117	0.3	0.2	0.3	5	4
HealthWeb	330	2.3	1.9	1.1	33	10
Lookout	502	2.4	2.4	1.9	29	33
ConfRM	704	2.7	2.5	1.8	63	21
Prooflib	10694	20.8	258.3	14.7	0	0
HealthwebEnh	766	8.0	8.5	5.8	156	83
HigherOrderIter	150	1.0	3.5	1.6	13	13
HigherOrderFoldr	108	2.3	5.8	0.9	10	6
Permission	251	4.1	4.3	5.5	29	29
Iflow_state	204	0.8	0.6	0.8	7	14
Provenance	221	1.6	1.5	0.8	22	17
Browser exts	785	3.1	3.3	3.8	89	55
DynSessions	211	0.7	0.5	0.2	0	0

HealthwebEnh is a cloud application managing an electronic medical record database, interacting with code written in ASP.NET, C#, and F#. It is about twice as big as the Fine HealthWeb benchmark, and is deployable on Microsoft Windows Azure. HigherOrderIter and HigherOrderFoldr implement higher-order library functions that iterate over lists. Permission implements a stateful API of collections and iterators that guarantee that the collection underlying an iterator is never modified while an iteration is in progress. Iflow_state provides an information-flow tracking library for stateful programs. Provenance is a larger version of the curated database in §2.5. Browser exts is a suite of 17 browser extensions, verified for authorization and information flow properties. DynSessions is the example of §4, including fork/join parallelism.

Fine and F* source typechecking times are roughly equivalent. F* translation is faster than Fine because there are fewer proofs to translate. Conversely, typechecking RDCIL code with refinement types is slower than checking DCIL proofs, but in view of the many advantages of F*, such as smaller bytecode and more expressive types, we find this tradeoff worthwhile.

Verifying cryptographic applications. Finally, we report source code verification results for several cryptographic protocol examples, many of which were previously developed for F7, and are now verified by F*.

Example	LOC	SC	SQ
CryptoLib	1530	50.5	426
KeyManager	608	55.6	287
AuthRPC	232	67.9	335
SessionLib	32	0.4	0
Commit	126	1.5	28
Forward	131	1.3	22
Metering	111	0.6	3

CryptoLib is a large F7 library implementing symbolic cryptography, which is used in all subsequent applications. KeyManager is a key management application. AuthRPC implements an authenticated RPC protocol. SessionLib is the generic API for multi-party sessions (§4), used to securely implement a two-party session Commit and a three-party session Forward. Metering is a privacy-friendly zero-knowledge cryptographic protocol for smart metering (Rial and Danezis 2010).

6. Related work

We have compared F* to Fine, F7, Aura, Coq, Agda, and discussed other related work in detail throughout this paper. Here, we briefly cover a few more topics. Guts et al. (2009) show how to build cryptographic audit trails that can be verified by independent third parties; Vaughan et al. (2008) argue that logs of logical evidence could also build audit trails. Neither consider the combination of mobile logical and cryptographic proofs, augmented with selective erasure and reconstruction. Related work on session types focuses on enforcing session compliance in the absence of malicious adversaries. Honda et al. (2008) develop special-purpose type systems for multi-party asynchronous sessions. They do not consider security or source code verification. Kiselyov et al. (2010) add type functions to Haskell and show how these can be used to program simple two-party sessions.

References

K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *TLDI*, 2010.

M. Backes, C. Hritcu, and M. Maffei. Type-checking zero-knowledge. In *CCS*, 2008.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

Y. Bertot and P. Castéran. *Coq'Art: Interactive Theorem Proving and Program Development.* Springer Verlag, 2004.

K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, 2009.

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

J. Borgstrom, J. Chen, and N. Swamy. Verifying stateful programs with substructural state and hoare types. In *PLPV '11*, Jan. 2011.

I. Cervesato and F. Pfenning. A linear logical framework. *Inf. Comput.*, 179 (1), 2002.

P. C. Chapin, C. Skalka, and X. S. Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40, 2008.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI '10*. ACM, 2010.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*, 2011.

G. Gonthier, A. Mahboubi, and E. Tassi. Research Report RR-6455, 2011.

A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, 2011.

N. Guts, C. Fournet, and F. Z. Nardelli. Reliable evidence: Auditability by typing. In *ESORICS*, 2009.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, 2008.

L. Jia and S. Zdancewic. Encoding information flow in aura. In *PLAS*, 2009.

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

O. Kiselyov, S. P. Jones, and C. chieh Shan. Fun with type functions, 2010. Unpub.

S. K. Lahiri, S. Qadeer, and D. Walker. Linear maps. PLPV '11. ACM, 2011.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Institute of Technology, 2007.

A. Rial and G. Danezis. Privacy-friendly smart metering. Technical report, Microsoft Research, nov 2010.

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *JFP*, 20(1), 2010.

M. Sozeau. Subset coercions in coq. In *TYPES*, 2007.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*, 2008.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*, 2010.

The Coq Development Team. Chapter 4: Calculus of Inductive Constructions. Technical report, 2010. URL `http://coq.inria.fr`.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*, 2008.

D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.