



Secure Distributed Programming with Value-Dependent Types

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

► **To cite this version:**

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, et al.. Secure Distributed Programming with Value-Dependent Types. *J. Funct. Program.*, Cambridge University Press, 2013, 23 (4), pp.402-451. <hal-00939188>

HAL Id: hal-00939188

<https://hal.inria.fr/hal-00939188>

Submitted on 11 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Secure Distributed Programming with Value-Dependent Types

NIKHIL SWAMY¹ JUAN CHEN¹ CÉDRIC FOURNET¹
PIERRE-YVES STRUB² KARTHIKEYAN BHARGAVAN³ JEAN YANG⁴
Microsoft Research¹ MSR-INRIA² INRIA³ MIT⁴
(e-mail: {nswamy, juanchen, fournet}@microsoft.com pierre-yves@strub.nu,
karthikeyan.bhargavan@inria.fr, jeanyang@csail.mit.edu)

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. Our language 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.

We have programmed and verified nearly 50,000 lines of F^* including new schemes for multi-party sessions; a zero-knowledge privacy-preserving payment protocol; a provenance-aware curated database; a suite of web-browser extensions verified for authorization properties; a cloud-hosted multi-tier web application with a verified reference monitor; the core F^* typechecker itself; and programs translated to F^* from other languages such as F7 and JavaScript.

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), and for proving authentication and authorization properties in cryptographic protocols (Gordon & Jeffrey 2003; Fournet *et al.* 2007; 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 *et al.* 2008;

2 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

Vaughan *et al.* 2008; Jia & Zdancewic 2009), Fine (Swamy *et al.* 2010; Chen *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 & 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 existing 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 50,000 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 their interaction while still providing a single unified language to specify, program, verify, and deploy secure distributed systems.

The kind \star is for general programming; its terms may diverge and exhibit other effects, such as state, exceptions and I/O. The kind P identifies a universe of pure, total functions; P terms are used mainly in the construction of proofs. The kind A is for affine (used-at-most-once), stateful resources; it is used to model and reason about effects in a modular style. The kind E is for specifications; it includes all the kinds above, that is, we define a sub-kinding relation $P <: E$, $\star <: E$, and $A <: E$ to promote the reuse of code and specifications. We use E to control the selective erasure of proof terms, when these proofs are 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. In such cases, we express specifications using types that reside exclusively in kind E .

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 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 $x:t\{\phi\}$ 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 formal models for many cryptographic patterns and primitives, such that evidence for ghost refinement properties can be constructed and communicated using cryptographic materials, such as digital signatures; (2) they benefit from a powerful subtyping relation: $x:t\{\phi\}$ is a subtype of t ; and $x:t\{\phi\}$ is a subtype of $x:t\{\psi\}$ when ϕ implies ψ ; this *structural subtyping* is convenient to write and verify higher-order programs; (3) they can 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 (Lahiri *et al.* 2011; Borgstrom *et al.* 2011). Relying on its kind system, F^* allows the free use of affine values within specifications, while still guaranteeing that affine values are used at most once elsewhere in the code. In §5, we exploit this feature extensively in implementing a new, flexible approach proposed by Deniérou & 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 & 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. §2.6 shows examples with a simple modal authorization logic, and with an *ad hoc* logic for database provenance. Logic parametricity enables us to work 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 & 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

4 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

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 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. So, we remain with value dependency in F^* , while acknowledging that it is less expressive than having expressions in types. Nevertheless, this loss of expressiveness has not hindered the construction and verification of the programs we have built. In place of type-level reduction of expressions, F^* provides a type-conversion relation that is parameterized by a theory of logical equivalence. This allows types such as `array (1 + 1) int` to be converted to `array 2 int`, where the terms `1 + 1` and `2` are deemed equal in a theory of integer arithmetic.

Compiler implementation. We have implemented a compiler for F^* based on our prior work on a compiler for Fine. The F^* compiler still 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 various libraries and applications, programming and verifying nearly 50,000 lines in F^* . Our applications include secure implementation for multi-party session protocols; a zero-knowledge privacy-preserving payment protocol; a provenance-aware curated database; a suite of 17 web-browser extensions verified for authorization properties; a cloud-hosted multi-tier web application with a verified reference monitor; a self-certified core F^* typechecker; an operational semantics for Javascript; and various programs translated to F^* from other languages such as F7 and JavaScript.

The F^* source release, the formal Coq development, and the programming examples are available from <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 using a call-by-value evaluation strategy, but the static semantics is significantly more complex. The examples below, together with those in §5, 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 the sub-language 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 $P <: E$, to construct a model of a high-integrity database with precise 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:  $\forall a::\star. l1:\text{list } a \rightarrow (l2:\text{list } a * (x:a \rightarrow \text{Mem } \times l2 \rightarrow \text{Mem } \times l1))$ 
```

This type is polymorphic, of the form $\forall a::k. t$ where k is the kind of the abstracted type variable—kinds are ascribed to types using double colons. Here, variable a 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 we 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 $x:t \rightarrow t'$ where the formal parameter of type t is named x , and is in scope in t' , the type of the result. When the function is not dependent, we simply write $t \rightarrow t'$. The result type of `tail` shows a *concrete refinement* type, also called subset types or Σ -types (Sozeau 2007). This type takes the form of a dependent pair $x:t * t'$, with a first component of type t named x and in scope in t' , the type of the second component. Here, the type $l2:\text{list } a * (x:a \rightarrow \text{Mem } \times l2 \rightarrow \text{Mem } \times l1)$ states that the tail $l2$ contains at most the elements of the input list $l1$. Intuitively, the second component carries a proof of the logical formula $\forall x:a. \text{Mem } \times l2 \implies \text{Mem } \times l1$. The predicate $\text{Mem } \times l2$ 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, exceptions, etc) 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 such that the terms typed within 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 $a::k \Rightarrow k'$, where a binds a k -kinded formal type

6 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

parameter in the kind k' of the constructed type. Type constructors can also be applied to values; such constructors have kinds of the form $x:t \Rightarrow k$ where x names the formal argument, a value of type t in scope in the kind k . Below, the kind of `Mem` says that it is a dependent type constructor that constructs a type of kind P from a type a , a value x of type a , and a value l of type `list a`. (When $x:t$, we write `Mem x l` instead of `Mem t x l`; we also use familiar list notations, writing `[]` for the empty list and `hd::tl` to cons an element `hd` to the front of a list `tl`.)

```
type Mem :: a :: *  $\Rightarrow$  a  $\Rightarrow$  list a  $\Rightarrow$  P =
  | Mem_hd : x:a  $\rightarrow$  tl:list a  $\rightarrow$  Mem x (x::tl)
  | Mem_tl : x:a  $\rightarrow$  y:a  $\rightarrow$  tl:list a  $\rightarrow$  Mem y tl  $\rightarrow$  Mem y (x::tl)
```

Inductive types defined in P -kind are required to be positive. That is, given a constructor $C : t_1 \rightarrow \dots \rightarrow t_n \rightarrow T$ for a type T of kind P , the constructed type T may occur in each t_i only in positive positions. Violating positivity easily leads to logical inconsistency, as illustrated below.

```
type Fix = MkFix : (Fix  $\rightarrow$  unit)  $\rightarrow$  Fix
let loop' (me:Fix) = match me with MkFix f  $\rightarrow$  f me
let loop = loop' (MkFix loop')
```

The type `Fix` is not positive; it is used to define `loop`, a divergent term, which clearly should not be used as a proof term. On the other hand, requiring all datatypes to be positive would be too restrictive. For example, when modeling dynamically typed programs, it is common to use the following `dyn` datatype, where the `Fun` constructor contains a negative occurrence of the defined type.

```
type dyn = Int : int  $\rightarrow$  dyn | String : string  $\rightarrow$  dyn | Fun : (dyn  $\rightarrow$  dyn)  $\rightarrow$  dyn
```

Thus, we require positivity for P and allow recursive datatypes elsewhere. In addition to positivity, we place certain restrictions (§3.2) on the elimination rules for P -kinded types to ensure totality.

A function type $x:t \rightarrow t'$ inherits the kind of its range type—it has P kind when $t':P$. Thus, in the type of `tail`, the type $x:a \rightarrow \text{Mem } x \ l2 \rightarrow \text{Mem } x \ l1$ is in P (since the range is in P), ensuring that any well-typed term at that type is a total function, and hence a valid proof term. Theorem 3 (Strong normalization) justifies this reasoning. In contrast, the type of `tail` itself has kind $*$, since the pair in its range has kind $*$. Thus, `tail` is not required to be a total function. Indeed, as shown by its implementation below, `tail` may raise an exception when called with an empty list. However, when called with a non-empty list `hd::tl`, it returns a pair containing `tl` and a total function (i.e., a proof term) witnessing the refinement formula.

```
let tail l = match l with
  | []  $\rightarrow$  raise (Error "Can't take the tail of an empty list")
  | hd::tl  $\rightarrow$  let proof (x:a) (pf:Mem x tl) = Mem_tl hd x tl pf in (tl, proof)
```

Typechecking this code requires typing the body of the `proof` function at the type `Mem x l`. From the type of `Mem_tl` and the types of the bound variables, it is easy to compute the type of the body as `Mem x (hd::tl)`. To prove that this latter type is convertible to `Mem x l`, the F^* typechecker implicitly uses the equations induced on `l` by pattern matching, such as `l = hd::tl` in the second branch above.

Programming explicitly with proof terms for non-trivial program properties quickly becomes impractical. Fine, a predecessor of F^* , provided a feature that allowed construct-

ing such proof terms automatically. This worked by first calling Z3, an SMT solver, to try to decide refinement properties, and then by building and typechecking a proof term 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, Fine provided a custom proof kernel. These proof kernels can be expressed in F^* as well. More generally, the P -fragment allows F^* programmers to define custom logics and to build and typecheck sound proof terms for these logics within the language itself. This is illustrated further in §2.6.

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 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 application, we use ghost refinement types to specify pre- and post-conditions on the interface provided by the browser, and we verify access control properties of extensions by typing them against this specification. The listing below illustrates this approach on a tiny program; §6 reports our results for compiling a suite of 17 such extensions in a type-preserving style to .NET bytecode.

```

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  (* Sample extension policy (trusted) *)
8  assume ∀e. EltTagName e "head" ⇒ CanRead e
9  (* Sample extension code (untrusted) *)
10 let read e = if tagName e = "head" then innerText e else ""

```

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; only trusted code may include assumptions. 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 data constructors. Instead, we use them for specifications, as in the types of `innerText` and `tagName`, our two library functions from nodes to strings.

The type of `innerText` has the form $x:t\{\phi\} \rightarrow t'$, 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' . Its 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

8 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

string t to the argument e , and clients may derive facts using this property and any other property in scope, such as our 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 policy assumption; the post-condition `EltTagName e t` for the value t returned by `tagName e`; and the equation `t="head"` from the equality test to derive `CanRead e`, the pre-condition of `innerText`, and thus authorize the call. Using this approach, once typechecked, untrusted extension code need not be examined—only the policy and the annotations on the DOM API are trusted. Next, we discuss two features of ghost refinements that are suitable in our example.

Ghost refinements and erasure. The type $x:t\{\phi\}$ is a subtype of t and the values of these two types share the same runtime representation. This makes specifications using ghost refinements lightweight, inasmuch as they do not require modifications to underlying code and data. For example, we did not need to modify, or even wrap, the DOM implementation above to verify its client 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 that inhabits $x:t\{\phi\}$, our type system ensures that the formula $\phi[v/x]$ 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 \mathcal{A} . We think of \mathcal{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 large); ghost refinements fit the bill nicely.

Proof-irrelevance and P vs. E -kind. The distinction between P and E 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, so that they may 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—§2.6 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 the P fragment of F^* within 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 is robust even when composed with arbitrary attacker code. For example, Guha *et al.* (2011) applied 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

Ghost refinements have been used in F7 to verify implementations of various security protocols against cryptographic assumptions (Bhargavan *et al.* 2010; Fournet *et al.* 2011). This section presents a small fragment of a library for public key cryptography in a new style, relying on features of F* not available in F7 such as higher-kinded and indexed types; the F* distribution includes a more complete library together with several programs that rely on cryptography. Our 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 `DSig` that provides an interface to work with digital 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 purpose of these signatures, Alice can convince Bob of some additional property ϕ of the message `m`, e.g., that the message originated in Alice's file system, or that it represents a genuine mail from Alice. The signature is useful inasmuch as ϕ is *not* an intrinsic property of the contents of `m`; a constructive logical proof of ϕ in this setting may be nonsensical.

```

module DSig
type dsig = bytes (* type of digital signatures *)
type prin (* name of a principal *)
type sk :: prin  $\Rightarrow$  a :: *  $\Rightarrow$  (a  $\Rightarrow$  E)  $\Rightarrow$  *
type pk :: prin  $\Rightarrow$  a :: *  $\Rightarrow$  (a  $\Rightarrow$  E)  $\Rightarrow$  *
val sign:  $\forall a :: *, \phi :: a \Rightarrow E. p:prin \rightarrow sk\ p\ a\ pred \rightarrow x:a\ \{\phi\ x\} \rightarrow dsig$ 
type Says :: prin  $\Rightarrow$  E  $\Rightarrow$  E
val verify:  $\forall a :: *, \phi :: a \Rightarrow E. p:prin \rightarrow pk\ p\ a\ \phi \rightarrow x:a \rightarrow dsig \rightarrow r:bool\ \{r=true \implies Says\ p\ (\phi\ x)\}$ 

```

`DSig` provides a representation type `dsig` for digital signatures, here just an alias for `bytes`, the type of concrete byte arrays. 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. For simplicity, we omit functions for generating keys and accessing their representation.

A private key of type `sk Alice a ϕ` belongs to the principal `Alice:prin`, who can use it to sign values `m` of type `a` that satisfy the property $\phi\ m$. Given such a key, the function `sign` allows clients to construct a `dsig` value by signing a message `x:a`, with a pre-condition that requires that the formula $\phi\ x$ be derivable when `sign` is called. Public keys are complementary: given a key of type `pk Alice a ϕ` , the function `verify` dynamically verifies the validity of a message signature; if it succeeds, then the caller knows that `Says Alice ($\phi\ x$)` holds. In the post-condition, 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. (Untrusted principals are not featured in the code above; they are used to model key compromise.)

As in F7, a *symbolic implementation* of the `DSig` library can be verified against the specification shown above, and can be proved correct with respect to a Dolev-Yao adversary (Bhargavan *et al.* 2010); a *concrete implementation* of the `DSig` library can similarly

10 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

be verified and proved correct, except with a negligible probability, with respect to a more realistic probabilistic polynomial-time adversary (Fournet *et al.* 2011). In their work, however, F7 types cannot be parametrized by predicates, so they have to simulate the ϕ parameter through a level of indirection: instead of the F* type $sk\ p\ a\ \phi$, private keys in F7 are given a type of the form $sk\ p\ a\ usage$, where $usage$ ranges over bytes. and the predicate $Pred$ is replaced by a global predicate $SignSays$, indexed by p and $usage$. The soundness of F7 verification relies on a programming convention that each key usage must be unambiguously defined by recording an assumption of the form $\forall p.v. SignSays\ p\ usage\ v \iff Says\ p\ (\phi\ v)$. This convention is not enforced automatically in F7, and hence their 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 First-order theories over logical values

Value dependency in F* precludes reasoning about programs terms simply by reduction. In general, since F* terms can have effects, these reductions would not be sound. Instead, F* allows equational reasoning about functions within a logical theory. This allows us to recover some of the expressiveness lost by value dependency restrictions. As an example, consider the interface for an abstract data type of arrays (below, on the left), designed to ensure that all array accesses are within bounds, and some sample client code (on the right).

```

type nat = i:int{i ≥ 0}
type array :: * ⇒ *
logic val Len : array a → nat
val anew: l:nat → init:a → x:array a {Len x=l}
val aget: x:array a → i:nat{i < Len x} → a
val aset: x:array a → i:nat{i < Len x} → a → unit

let rev (len:nat) (a:array b {Len a=len}) =
  let rec aux (i:nat{i ≤ len/2 ∧ len ≠ 0}) =
    let j = len - i - 1 in
    let tmp = aget a i in
    aset a i (aget a j); aset a j tmp;
    if i + 1 > len / 2 then () else aux (i + 1)
  in if len=0 then () else aux 0

```

The abstract type `array a` represents a mutable array containing a -typed values. We allocate an array using `anew`, providing a length and an initial value. The functions `aget` and `aset` read and write the i th cell in the array. The client program `rev` reverses the array `a` by swapping the contents of cells that are equidistant from the center of the array.

To ensure that clients only access array cells within bounds, we introduce an uninterpreted function `Len:array a → nat`. The **logic** qualifier preceding its declaration indicates that `Len` can only be used in the refinement logic, and has no operational semantics otherwise. It is illegal to call `Len` when evaluating a program—thus, `rev` takes two parameters, a `len:nat` value and the array `a`, where the refinement on `a` relates `len` to `Len a` in the logic. On the other hand, our interface may additionally provide an ordinary function `val length: x:array a → l:nat {Len x=l}`. By making use of these “logical values”, and interpreting them in the logic with suitable axioms, we can simulate type-level reduction of expressions by equational rewriting of logic values.

The result type of `anew` records with a ghost refinement that the length of the new array x is l . The other two functions require that `i:nat`, the array index, be strictly less than the length of the array. Accordingly, we use a logic equipped with a theory of integer inequalities and arithmetic, i.e., the infix type constructors $(<)::int \Rightarrow int \Rightarrow E$, $(\leq)::int \Rightarrow int \Rightarrow E$, etc. are predicates in the refinement logic, with the usual interpretation as integer inequality.

In order to typecheck `rev`, we make use of F^* 's standard prelude, which includes the following declarations for arithmetic primitives.

```

logic val (+) : int → int → int
logic val (−) : int → int → int
logic val (*) : int → int → int
logic val (/) : int → int → int
val (+) : x:int → y:int → z:int{z = x + y}
val (−) : x:int → y:int → z:int{z = x − y}
val (*) : x:int → y:int → z:int{z = x * y}
val (/) : x:int → y:int{y ≠ 0} → z:int{z = x / y}

```

We have four interpreted arithmetic functions in the logic, and four corresponding primitive operators in F^* , each primitive operator being specified through the use of the corresponding logic function. The duplication may seem somewhat redundant, but it is a one-time cost for primitive operators, it can be hidden under suitable syntactic sugar, and it can be adapted to match the theories implemented by Z3.

To typecheck `aget a j` within `rev`, for instance, the F^* typechecker must prove $j < \text{Len } a$. Without any type-level reduction, the reasoning proceeds modulo theory by first showing $j = \text{len} - i - 1$ in the refinement logic; noting $0 \leq i + 1 \leq \text{len}$; and concluding $0 \leq j < \text{len}$.

2.5 Ghost refinements and affine-indexed types for state

The example above shows a weak, partial specification for arrays, capturing their size but not their mutable content, but F^* also has mechanisms to support stronger specifications and verify functional correctness. For instance, one can encapsulate effectful code within a monad, and write explicit stateful, pre- and post-conditions—this is the methodology of Nanevski *et al.* (2008); §7 discusses how a similar methodology can be applied to F^* .

In this section, we illustrate the use of affine types, in combination with E -kinded types and logical values, to reason about stateful programs. In particular, we show how to program with *linear maps*, a data type proposed by Lahiri *et al.* (2011) to verify heap-manipulating programs. One innovation of F^* (which is a key enabler of this encoding) is that it permits indexing types with affine values, allowing us to state properties about affine values without consuming them. 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. Rather than modeling the heap of a program as a single monolithic map $H:\text{heap}$ from locations of type `ref a` to values of type `a`, the linear maps methodology advocates partitioning the heap H into several fragments H_1, \dots, H_n where the fragments have disjoint domains. Each H_i is a linear map of type `lin`, and the disjoint domain condition ensures that modifications to H_i leave all the other H_j unmodified. This allows formulating 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 standard provers and SMT solvers.

The listing below shows a fragment of code used to model a `heap`, a map from references of type `ref a` to values of type `a`. In addition to abstract types `heap` and `ref`, we provide two logical values `Select` and `Update`. We use the `assume` construct to provide axioms that interpret `Select/Update` in the theory of functional arrays, in the style of McCarthy (1962).

```

type heap :: *
type ref :: * ⇒ *
logic val Select : ref a → heap → a
logic val Update : ref a → a → heap → heap

```

12 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

```
assume  $\forall h \times v. \text{Select } \times (\text{Update } \times v h) = v$ 
assume  $\forall h \times y v. \text{not } (\times=y) \implies \text{Select } y (\text{Update } \times v h) = \text{Select } y h$ 
```

Heaps are total maps from references to values. To partition them into disjoint fragments, we need to keep track of the set of references in the domain of each fragment. For this, we provide another abstract type, `locset`, with a signature (below) that includes the set operations with their usual interpretation.

```
type locset
logic val Empty: locset
logic val Singleton: ref a  $\rightarrow$  locset
logic val ( $\cup$ ): locset  $\rightarrow$  locset  $\rightarrow$  locset
logic val ( $\setminus$ ): locset  $\rightarrow$  locset  $\rightarrow$  locset
type ( $\in$ ): ref a  $\implies$  locset  $\implies$  E
```

We define the type `lin` as a record of a `heap` and a `locset`, which 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 directly corresponds to affinity in F^* : values of affine type can be used *at most once*.

```
type lin :: A = {map:heap; domain:locset}
```

We show below the types of four operations on linear maps. The first, `newlin`, allocates a new linear map with an empty domain. The `read` function reads a location `x:ref a` out of a map `m1` (such that `x` is in the domain of `m1`), and returns the value `y:a` stored at `x`. Since `m1` is affine, `read` threads `m1` back to the caller as `m2`, with a refinement that states that it is unchanged. The `write` function is similar; in both cases the `Select` and `Update` predicates specify the appropriate post-conditions. A fourth function, `transfer`, allows moving a reference `x` from the domain of one linear map into another, while preserving the disjoint domains invariant.

```
val newlin: unit  $\rightarrow$  m:lin{m.domain=Empty}
val read: x:ref a  $\rightarrow$  m1:lin{x  $\in$  m1.domain}  $\rightarrow$  (y:a * m2:lin{m2=m1 && y=Select x m1.map})
val write: x:ref a  $\rightarrow$  y:a  $\rightarrow$  m1:lin{x  $\in$  m1.domain}  $\rightarrow$  (u:unit * m2:lin{m2.domain=m1.domain  $\wedge$ 
m2.map=Update x y m1.map})
val transfer: m1:lin  $\rightarrow$  m2:lin  $\rightarrow$  x:ref a{x  $\in$  m1.domain}
 $\rightarrow$  (m1':lin * m2':lin{m1'.map=m1.map  $\wedge$ 
m1'.domain=m1.domain  $\setminus$  Singleton x  $\wedge$ 
m2'.map=Update x (Select x m1.map) m2.map  $\wedge$ 
m2'.domain=m2.domain  $\cup$  Singleton x})
```

Predicates on affine values. While seemingly unremarkable, by refining affine values, the types above are a significant advance over prior languages with substructural and dependent types. In systems like Fine and Linear LF (Cervesato & Pfenning 2002), for instance, types are required to be free of affine (or linear) indices, i.e., type constructors of kind $t \Rightarrow k$, 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, hence consuming those affine resources. 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 freely state properties on affine values. Since E -kinded predicates have no runtime significance, indexing them with affine values

does not consume them—in F^* , kinds of the form $t \Rightarrow E$ 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. When modeling linear maps programs, the dynamic log of assumptions is constant (unlike when modeling DOM programs and cryptography) so the metatheory of F^* 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 §5, where we use them with higher-rank E -kinded types to model concurrent, message passing programs.

2.6 Selective erasure using concrete and ghost refinements

Our overview of ghost refinements may lead the reader to believe that they are always to be preferred to refinements with concrete proof terms. This section finally 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 on just one aspect, the selective erasure and reconstruction of proofs, which may be required both for efficiency and for confidentiality. This example is related to work by Guts *et al.* (2009), who show how to build cryptographic audit trails that can be verified by independent third parties. Also related is the work of Vaughan *et al.* (2008), who argue that logs of logical evidence could also build audit trails. However, neither consider the combination of mobile logical and cryptographic proofs, augmented with selective erasure and reconstruction.

Each experiment recorded in the database is given the type `exp b`, where `b` is a boolean, 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 a concrete proof term, of type `proof b (Valid r)`, that contains evidence recording the relationship of this experiment to others in the database. Intuitively, this proof term reflects the provenance of the experiment.

```

type expsetup = list {reagent:string; quantity:int}
type Valid :: expsetup  $\Rightarrow$  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 instance of logic parametricity in F^* . We show a selection of the constructors from this kernel below.

```

let full, partial = true, false
type proof :: bool  $\Rightarrow$  E  $\Rightarrow$  P =
| AndIntro: ... | AndElim1 : ... | ...
| ChemicalVolcano: proof full (Valid[{reagent="(NH4)2Cr2O7"; ...}])
| Combine: r1:expsetup  $\rightarrow$  r2:expsetup  $\rightarrow$  r3:expsetup  $\rightarrow$  b:bool
            $\rightarrow$  proof b (And (Union r1 r3) (And (Valid r1) (Valid r2)))
            $\rightarrow$  proof b (Valid r3)
| Prune: r:expsetup{Valid r}  $\rightarrow$  xid:int  $\rightarrow$  proof partial (Valid r)

```

14 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

The interplay between ghost and concrete proofs 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 the 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`.

We give below the typed interface to our database. The database content `db` is simply a list of experiments with `full` proofs. It supports operations to `insert` new experiments (returning a freshly generated key); to `lookup` using the primary keys; and, using `lookupProof` 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 → optionP (proof full (Valid r))

```

(Since `proof full (Valid r)` has kind P , we use a variant of the option type in the result of `lookupProof`, with a constructor `optionP :: P ⇒ *` instead of the standard `option :: * ⇒ *`.)

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 a requestor that trusts the database can still 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 apply the `Prune` constructor, we must prove that `r` has 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.

Conversely, 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 a) : optionP (proof full a) =
  if c=full then SomeP pf
  else match pf with
  | Prune r xid → lookupProof r xid
  | AndElim1 c1 pf → (match expand c1 pf with
    | SomeP pf' → SomeP (AndElim1 full pf')
    | _ → NoneP)
  | ...
(* Inserting a new record in the DB *)
let insertExp (r:expsetup) (c:bool) (pf: proof c (Valid r)) =
  match (expand c pf) with
  | SomeP 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 `SomeP pf`, then `pf` is indeed a valid full proof in the P -fragment, F^* 's logically consistent fragment of total functions.

3 Syntax and semantics of 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, computations, 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 in Figure 1, starting with our meta-variable convention. We use α to range over type variables a and value variables x . We have two forms of value constructors D : we use C for data constructors (like `None` and `Some`) and L for term constructors in the logic (like `logic val Select` and `logic val Update` of §2.5). We use the notation \bar{t} to stand for a finite sequence of elements t_1, \dots, t_n , for arbitrary n ; $(\bar{t})_k$ is a sequence a_1, \dots, a_{k-1} . We also let τ range over both types t and values v .

Values v include variables x , lambda abstractions over values and types, and fully applied polyadic data constructors applied to a sequence of values and types. The value v^ℓ is a

16 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

Meta-variables	
$\alpha ::= a \mid x$	type and value variables
$D ::= C \mid L$	data constructor and logic value
$\tau ::= v \mid t$	type or value
$v ::= x \mid \lambda x:t.e \mid \Lambda a::\kappa.e \mid D \bar{\tau} \mid v^\ell$	values
$e ::= v \mid e v \mid e t \mid \mathbf{let} \ x = e \ \mathbf{in} \ e' \mid \mathbf{match} \ v \ \mathbf{with} \ C \ \bar{\alpha} \rightarrow e \ \mathbf{else} \ e' \mid$ $\mathbf{ref} \ t \ v \mid v_1 := v_2 \mid !v \mid \mathbf{raise} \ v \mid \mathbf{try} \ e \ \mathbf{with} \ x.e \mid \mathbf{assume} \ \phi$	expression
$t, \phi ::= a \mid T \mid x:t \rightarrow t' \mid \forall a::\kappa.t \mid t v \mid t t' \mid \lambda x:t.t' \mid \Lambda a::\kappa.t \mid x:t\{\phi\} \mid \imath t$	types
$c ::= \star \mid P \mid A$	concrete kinds
$b ::= c \mid E$	base kinds
$\kappa ::= b \mid x:t \Rightarrow \kappa \mid a::\kappa \Rightarrow \kappa'$	kinds
$S ::= \cdot \mid L:t \mid T::\kappa\{\overline{C:t}\} \mid S, S'$	signature
$\Gamma ::= \cdot \mid x:t \mid a::\kappa \mid v_1 = v_2 \mid t_1 = t_2 \mid \Gamma, \Gamma'$	type environment

Fig. 1. Syntax of F*: values, expressions, types, kinds, signatures, environments, and contexts

technical device used to prove the soundness of affine typing— ℓ is an identifier drawn from a class of names distinct from value and type names.

We adopt a (partially) let-normalized view of the expression language e , in particular requiring function arguments (in $e v$) to always be values—this is convenient when using value-dependent types, since it ensures that expressions never escape into the level of types. Most of the constructs are standard, including value and type applications; let-bindings; pattern matching; operations to allocate, read and write references; and exception raising and handling. The only 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 a ; constants T ; dependent functions ranging over values whose domains may be values ($x:t \rightarrow t'$) or types ($\forall a::\kappa.t$); 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** ($x:t$) $\rightarrow t'$) and types to types ($\Lambda a::\kappa.t$, concretely written **fun** ($a::\kappa$) $\rightarrow t$); ghost refinements $x:t\{\phi\}$; and, finally, coercions to affine types $\imath t$. This modal operator $\imath \cdot$ serves to qualify the type of a closure that captures an affine assumption; we include $\imath t$ in the formalism to avoid duplicating the rules for function arrows, but concretely we write affine function types as $x:t \gg t'$ and $\forall a::\kappa \gg t$ instead of $\imath(x:t \rightarrow t')$ and $\imath(\forall a::\kappa.t)$.

Kinds κ include the four base kinds \star , P , A , and E —we distinguish the first three of these as *concrete kinds*, since they are the minimal kinds of types that are inhabited. As at the type level, we have kinds for dependent function spaces whose ranges are types and whose domains may be either values ($x:t \Rightarrow \kappa$) or types ($a::\kappa \Rightarrow \kappa'$). Stratifying the language into terms, types, and kinds allows us to place key restrictions (discussed below) that facilitate automated verification, and 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 logic value declarations and datatype definitions. A logic value declaration introduces a logical value constructor, $L : t$. The interpretation of this function symbol (if any) is provided by axioms introduced using **assume**. Each datatype definition $T::\kappa\{\overline{C:t}\}$ introduces a type constructor T of kind κ and all its value constructors

$C_1:t_1, \dots, C_n:t_n$. For simplicity, we do not include mutually recursive types here, although these are supported by our implementation as well as in our Coq formalization. We do not need a fixpoint form in the expression language since (as illustrated in §2.1) recursive datatypes 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 definitions for P -kinded types to ensure they are inductive. An additional constraint on signatures is that they must contain a declaration `unit :: * {():unit}` for the `unit` type and its one value `()`; the `ref :: * ⇒ *` type constructor; and a definition of the datatype for exceptions, `exn`.

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.

3.2 The F^* type system

We now present the F^* type system, which takes the form of several mutually recursive judgments. The three main judgments defined well-formedness rules for kinds, kinding rules for types, and typing rules for values and expressions. In conjunction with these, we provide two subsumption judgments, one each for a sub-kinding and a sub-typing relation, as well as two conversion judgments, equating kinds and types (respectively) that are related by reduction or by equations in the environment. Finally, we have several auxiliary judgments for the well-formedness of signatures, environments and runtime configurations.

Well-formedness of kinds. Figure 2 defines the judgment $S; \Gamma \vdash \kappa \text{ ok}(b)$, which states that κ is well-formed and is the kind of a constructor of b -kinded types. The rule (OK-b) is for base kinds. As in all other judgments, we require the leaves of a derivation to ensure that the environment is well-formed, using the judgment $\vdash S; \Gamma \text{ wf}$, discussed shortly. 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 specifiational ($b_2 = E$). As illustrated in §2.5 and §5, 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 easily be compiled to the type system of the .NET bytecode language and are currently rejected by our compiler if the target platform is .NET. However, F^* programs can also be compiled to JavaScript (Fournet *et al.* 2013a), with full type erasure, in which case this form of higher kinding is allowed.

Kinding of types. The judgment $S; \Gamma \vdash t :: \kappa$ (also in Figure 2) states that type t has kind κ . The rules (K-a) and (K-T) are straightforward. The rule (K-A) shows how the modal operator coerces the kind of a type. (K- \rightarrow) handles dependent function arrows, which (as seen in §2.1) can be used to represent both quantified formulas in the logic and term-level function abstractions; a function arrow is P -kinded if its range type is P -kinded. The rule for type functions (K- \forall) is similar. (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. Value-to-type functions are introduced either using (K- λ) or as type

18 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

$$\begin{array}{c}
\boxed{S; \Gamma \vdash \kappa \text{ ok}(b)} \quad \text{OK-b} \frac{\vdash S; \Gamma \text{ wf}}{S; \Gamma \vdash b \text{ ok}(b)} \\
\text{OK-KK} \frac{S; \Gamma \vdash \kappa_1 \text{ ok}(b_1) \quad b_2 \in \{A, E\} \text{ if } b_1 = A \quad S; \Gamma, a::\kappa_1 \vdash \kappa_2 \text{ ok}(b_2)}{S; \Gamma \vdash a::\kappa_1 \Rightarrow \kappa_2 \text{ ok}(b_2)} \\
\text{OK-TK} \frac{S; \Gamma \vdash t::b_1 \quad b_2 = E \text{ if } b_1 = A \quad S; \Gamma, x:t \vdash \kappa \text{ ok}(b_2)}{S; \Gamma \vdash x:t \Rightarrow \kappa \text{ ok}(b_2)} \\
\boxed{S; \Gamma \vdash t :: \kappa} \quad \text{K-a} \frac{\vdash S; \Gamma \text{ wf}}{S; \Gamma \vdash a :: \Gamma(a)} \quad \text{K-T} \frac{\vdash S; \Gamma \text{ wf}}{S; \Gamma \vdash T :: S(T)} \quad \text{K-A} \frac{S; \Gamma \vdash t :: \star}{S; \Gamma \vdash !t :: A} \\
\text{K-}\rightarrow \frac{S; \Gamma \vdash t :: c \quad S; \Gamma, x:t \vdash t' :: c' \quad b = P \text{ if } c' = P \text{ and } b = \star \text{ otherwise}}{S; \Gamma \vdash x:t \rightarrow t' :: b} \quad \text{K-}\forall \frac{S; \Gamma \vdash \kappa \text{ ok}(b) \quad S; \Gamma, a::\kappa \vdash t :: c \quad b = P \text{ if } c = P \text{ and } b = \star \text{ otherwise}}{S; \Gamma \vdash \forall a::\kappa.t :: b} \\
\text{K-}\phi \frac{S; \Gamma \vdash t :: c \quad S; \Gamma, x:t \vdash \phi :: E}{S; \Gamma \vdash x:t\{\phi\} :: c} \quad \text{K-}\lambda \frac{S; \Gamma \vdash x:t \Rightarrow \kappa \text{ ok}(b) \quad S; \Gamma, x:t \vdash t' :: \kappa}{S; \Gamma \vdash \lambda x.t.t' :: x:t \Rightarrow \kappa} \\
\text{K-Tv} \frac{S; \Gamma \vdash t :: x:t' \Rightarrow \kappa \quad S; \Gamma, \cdot \vdash^e v : t'}{S; \Gamma \vdash t v :: \kappa[v/x]} \quad \text{K-}\Lambda \frac{S; \Gamma \vdash a::\kappa \Rightarrow \kappa' \text{ ok}(b) \quad S; \Gamma, a::\kappa \vdash t' :: \kappa'}{S; \Gamma \vdash \Lambda a::\kappa.t :: a::\kappa \Rightarrow \kappa'} \\
\text{K-tt} \frac{S; \Gamma \vdash t :: a::\kappa \Rightarrow \kappa' \quad S; \Gamma \vdash t' :: \kappa}{S; \Gamma \vdash t t' :: \kappa'[t'/a]} \quad \text{K-}\prec \frac{S; \Gamma \vdash t::\kappa \quad S; \Gamma \vdash \kappa \prec: \kappa' \quad S; \Gamma \vdash \kappa' \text{ ok}(b)}{S; \Gamma \vdash t :: \kappa'} \\
\boxed{S; \Gamma \vdash \kappa_1 \prec: \kappa_2} \quad \text{SK-Ref} \frac{S; \Gamma \vdash \kappa \equiv \kappa'}{S; \Gamma \vdash \kappa \prec: \kappa'} \quad \text{SK-Trans} \frac{S; \Gamma \vdash \kappa \prec: \kappa_1 \quad S; \Gamma \vdash \kappa_1 \prec: \kappa'}{S; \Gamma \vdash \kappa \prec: \kappa'} \\
\text{SK-P-E} \frac{}{S; \Gamma \vdash P \prec: E} \quad \text{SK-}\star\text{-E} \frac{}{S; \Gamma \vdash \star \prec: E} \quad \text{SK-A-E} \frac{}{S; \Gamma \vdash A \prec: E} \\
\text{SK-TK} \frac{S; \Gamma \vdash t' \prec: t \quad S; \Gamma, x:t' \vdash \kappa \prec: \kappa'}{S; \Gamma \vdash x:t \Rightarrow \kappa \prec: x:t' \Rightarrow \kappa'} \quad \text{SK-KK} \frac{S; \Gamma \vdash \kappa'_1 \prec: \kappa_1 \quad S; \Gamma, a::\kappa'_1 \vdash \kappa_2 \prec: \kappa'_2}{S; \Gamma \vdash a::\kappa_1 \Rightarrow \kappa_2 \prec: a::\kappa'_1 \Rightarrow \kappa'_2} \\
\boxed{S; \Gamma \vdash \kappa \equiv \kappa'} \quad \text{KE-Ref} \frac{}{S; \Gamma \vdash \kappa \equiv \kappa} \quad \text{KE-Prod} \frac{S; \Gamma \vdash t \equiv t' \quad S; \Gamma, x:t \vdash \kappa \equiv \kappa'}{S; \Gamma \vdash x:t \Rightarrow \kappa \equiv x:t' \Rightarrow \kappa'} \\
\text{KE-Sym} \frac{S; \Gamma \vdash \kappa' \equiv \kappa}{S; \Gamma \vdash \kappa \equiv \kappa'} \quad \text{KE-ProdK} \frac{S; \Gamma \vdash \kappa_1 \equiv \kappa'_1 \quad S; \Gamma, a::\kappa_1 \vdash \kappa_2 \equiv \kappa'_2}{S; \Gamma \vdash a::\kappa_1 \Rightarrow \kappa_2 \equiv a::\kappa'_1 \Rightarrow \kappa'_2}
\end{array}$$

Fig. 2. Well-formedness of kinds, well-kinded types, sub-kinding, and kind conversion

constructors T in the signature. They are eliminated using (K-Tv), which allows a type function t to be applied to a value v . Type-to-type functions are introduced using (K- Λ) and eliminated using (K-tt). Both elimination forms are dependent, i.e., the co-domain depends on the value or type argument. The rule (K-Tv) is worth closer study. 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, 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 expression typing judgment.

Finally, Figure 2 shows a subsumption rule ($K-<$) and the judgement $S; \Gamma \vdash \kappa <: \kappa'$. The latter is a reflexive and transitive relation, which treats all concrete kinds as a sub-kind of E , but otherwise unrelated to one another. This differs from an earlier formulation of the system presented by Swamy *et al.* (2011), which included $P <: \star$. While that is feasible, and allows more code re-use between \star and P (e.g., we could simply re-use the `option` type for proofs, instead of a separate `optionP` type in §2.6), as we discuss in §4.3, the current formulation enables a significantly cleaner proof of strong normalization for the P -fragment. Given our experience programming, to date, some 50,000 lines of code in F^* , we concluded that the penalty we pay in terms of code duplication by forbidding $P <: \star$ is worthwhile for the improved metatheoretical development. Another difference is that we now include $A <: E$, an improvement over our prior work to promote re-use of specifications between affine and non-affine values.

The rest of the rules in the sub-kinding relation are homomorphic with respect to the sub-typing ($S; \Gamma \vdash t <: t'$, discussed shortly) and sub-kinding relations. Kind conversion, $S; \Gamma \vdash \kappa \equiv \kappa'$ is a straightforward equivalence relation. Technically, ($K-<$) 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. Figure 3 defines the judgment $S; \Gamma; X \vdash^m e : t$, which states that expression e has type t , under signature S , environment Γ , and an affine environment $X ::= \cdot \mid \ell \mid \alpha \mid X, X'$ where X, X' denotes disjoint union. A well-formedness condition on contexts requires that all variables in X be also 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. We find it syntactically convenient to allow X to contain value names x , affine labels ℓ as well as type names a , although affinity restrictions do not apply to type names. We discuss each of the rules in detail, next.

Affinity. (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 . (T-X) provides two alternatives to rule (T-XA): first, as usual, 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-Drop) provides weakening for the affine context. (T-Box) is for typing values that have been tagged with an affine label—we discuss it in detail after presenting the dynamic semantics.

Data and logic values. (T-D) types a constructor as a function application, by introducing a fresh variable x at the type of D in the context. The fourth premise ensures that the con-

20 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

$$\begin{array}{c}
\text{T-XA} \frac{\vdash S; \Gamma \text{ wf} \quad S; \Gamma \vdash \Gamma(x) :: A}{S; \Gamma; x \vdash x : \Gamma(x)} \quad \text{T-X} \frac{\vdash S; \Gamma \text{ wf} \quad S; \Gamma \vdash \Gamma(x) :: b \quad m = \varepsilon \text{ if } b = A}{S; \Gamma; \cdot \vdash^m x : \Gamma(x)} \\
\text{T-Drop} \frac{S; \Gamma; X \vdash^m e : t}{S; \Gamma; X, X' \vdash^m e : t} \quad \text{T-Box} \frac{S; \Gamma; X \setminus \ell \vdash^m v : t \quad S; \Gamma \vdash t :: A \quad \ell \in X \text{ or } m = \varepsilon}{S; \Gamma; X \vdash^m v^\ell : t} \\
\text{T-D} \frac{D : t_d \in S \quad S; \Gamma; x : t_d; X \vdash^m x \bar{\tau} : t \quad t = T _ \quad D = C \text{ or } m = \varepsilon}{S; \Gamma; X \vdash^m D \bar{\tau} : t} \\
\text{T-Abs} \frac{S; \Gamma \vdash t :: c \quad S; \Gamma, x : t; X, x \vdash^m e : t'}{S; \Gamma; X \vdash^m \lambda x : t. e : Q(X, x : t \rightarrow t')} \quad \text{T-TAbs} \frac{S; \Gamma \vdash \kappa \text{ ok}(b) \quad S; \Gamma, a :: k; X \vdash^m e : t}{S; \Gamma; X \vdash^m \Lambda a :: \kappa. e : Q(X, \forall a :: k. t)} \\
\text{T-App} \frac{S; \Gamma; X_1 \vdash^m e : t_1 \quad !t_1 = x : t' \rightarrow t \quad S; \Gamma; X_2 \vdash^m v : t'}{S; \Gamma; X_1, X_2 \vdash^m e v : t[v/x]} \quad \text{T-TApp} \frac{S; \Gamma; X \vdash^m e : t_v \quad !t_v = \forall a :: \kappa. t' \quad S; \Gamma \vdash t :: \kappa}{S; \Gamma; X \vdash^m e t : t'[t/a]} \\
\text{T-Let} \frac{S; \Gamma; X_1 \vdash^m e_1 : t_1(\kappa_1) \quad S; \Gamma, x : t_1; X_2, x \vdash^m e_2 : t_2 \quad S; \Gamma \vdash t_2 :: \kappa_2 \quad \kappa_1 = P \text{ if } \kappa_2 = P}{S; \Gamma; X_1, X_2 \vdash^m \text{let } x = e_1 \text{ in } e_2 : t_2} \\
\text{T-Match} \frac{S; \Gamma; X_1 \vdash^m v : t_v(\kappa_v) \quad \text{dom}(\Gamma') = \bar{\alpha} \quad S; \Gamma, \Gamma'; \bar{\alpha} \vdash^m C \bar{\alpha} : t_p \quad t_p \bar{\alpha} \bowtie_{\bar{\tau}} t_v : \Gamma_{eq} \quad S; \Gamma, \Gamma', \Gamma_{eq}, v = C \bar{\alpha}; X_2, \bar{\alpha} \vdash^m e_1 : t \quad S; \Gamma; X_2 \vdash^m e_2 : t(\kappa_b) \quad \kappa_v = P \text{ if } \kappa_b = P}{S; \Gamma; X_1, X_2 \vdash^m \text{match } v \text{ with } C \bar{\alpha} \rightarrow e_1 \text{ else } e_2 : t} \\
\text{T-Try} \frac{S; \Gamma; X_1 \vdash^m e_1 : t \quad S; \Gamma, x : \text{exn}; X_2 \vdash^m e_2 : t \quad S; \Gamma \vdash t :: \star}{S; \Gamma; X_1, X_2 \vdash^m \text{try } e_1 \text{ with } x. e_2 : t} \\
\text{T-Raise} \frac{S; \Gamma; X \vdash^m v : \text{exn} \quad S; \Gamma \vdash t :: \star}{S; \Gamma; X \vdash^m \text{raise } v : t} \quad \text{T-Rd} \frac{S; \Gamma; X \vdash^m v : \text{ref } t}{S; \Gamma; X \vdash^m ! v : t} \\
\text{T-Ref} \frac{S; \Gamma; X \vdash^m v : t \quad S; \Gamma \vdash t :: \star}{S; \Gamma; X \vdash^m \text{ref } t v : \text{ref } t} \quad \text{T-Wr} \frac{S; \Gamma; X_1 \vdash^m v_1 : \text{ref } t \quad S; \Gamma; X_2 \vdash^m v_2 : t}{S; \Gamma; X_1, X_2 \vdash^m v_1 := v_2 : \text{unit}} \\
\text{T-V} \frac{S; \Gamma; X \vdash^m v : t \quad S; \Gamma \vdash t :: c \quad S; \Gamma, x : t, x = v \models \phi}{S; \Gamma; X \vdash^m v : x : t\{\phi\}} \quad \text{T-Ax} \frac{S; \Gamma \vdash \phi :: E}{S; \Gamma; \cdot \vdash^m \text{assume } \phi : _ : \text{unit}\{\phi\}} \\
\text{T-Sub} \frac{S; \Gamma; X \vdash^m e : t' \quad S; \Gamma \vdash t' <: t \quad S; \Gamma \vdash t :: \kappa}{S; \Gamma; X \vdash^m e : t} \\
\text{where } \left\{ \begin{array}{l} t \bowtie_{\bar{\tau}} t' : \cdot \quad \frac{\alpha \in FV(t) \quad t[\tau/\alpha] \bar{\alpha} \bowtie_{\bar{\tau}} t' : \Gamma}{t \alpha, \bar{\alpha} \bowtie_{\tau, \bar{\tau}} t' : \alpha = \tau, \Gamma} \quad \frac{\alpha \notin FV(t) \quad t \bar{\alpha} \bowtie_{\bar{\tau}} t' : \Gamma}{t \alpha, \bar{\alpha} \bowtie_{\tau, \bar{\tau}} t' : \Gamma} \\ Q(\cdot, t) = t \quad !; t = t \quad !t = t \text{ otherwise} \\ Q(X, t) = !t \quad S; \Gamma; X \vdash^m e : t(\kappa) \triangleq S; \Gamma; X \vdash^m e : t \wedge S; \Gamma \vdash t :: \kappa \end{array} \right.
\end{array}$$

Fig. 3. Well-typed expressions: $S; \Gamma; X \vdash^m e : t$

structor is fully applied, and the last premise ensures that if the constructor is a logic value, then it is used only at the type level since logic values have no operational interpretation.

Function abstraction and application. (T-Abs) and (T-TAbs) are standard rules for value- and type-abstractions, except that the introduced function type is tagged with the affine modality (using $Q(X, x : t \rightarrow t')$) if the function closure captures an affine assumption. This is achieved using the auxiliary function Q defined at the bottom of the figure. The

corresponding elimination forms (T-App) and (T-TApp) are shown next. We split the affine context between the sub-expressions (if any), check each part, and then substitute the argument for the formal parameter in the result type. In the second premise of both rules, we use the auxiliary function $!t$ to strip an affine modality from t , if there is one.

Let-bindings. (T-Let) is a variation on the standard rules for let bindings that ensures that expressions with P -kinded types are total. Intuitively, for an expression e to reside in P , e must be free of non- P expressions, since those may diverge. (Non- P values are permissible, since these are already reduced.) Thus, the rule has a final premise to ensure that if the result type is in P , then the let-bound expression must also be in P . The third premise also serves to ensure that the let-bound variable does not escape its scope, and the first premise employs syntactic sugar (defined at the bottom of the figure) to compute the kind κ_1 of t_1 .

Pattern matching. The next rule, (T-Match), is expectedly the most complex. First, like (T-Let), we place restrictions on cross-universe elimination to ensure the consistency of the P -fragment. Specifically, the last premise disallows discriminating values that reside in \star or A when constructing P -kinded expressions in the branches of a **match**. The third and fourth premises check that the type of the pattern t_p is unifiable with the type of the scrutinee t_v , given the equations in Γ_{eq} , using the auxiliary judgment $t_p \bar{\alpha} \bowtie_{\bar{\tau}} t_v : \Gamma_{eq}$, which ensures that $t_p[\bar{\tau}/\bar{\alpha}] = t_v$ and Γ_{eq} contains equations of the form $\alpha_i = \tau_i$ for those α_i that are free in t_p . The fifth premise checks the branch e_1 in a context including the pattern-bound variables, extended with the equations in Γ_{eq} as well as one additional equation between the scrutinee and the pattern. As we will see, these equations are used in $S; \Gamma \vdash t \equiv t'$ to allow typing derivations to freely refine both type and value indices within types. In contrast, the **else**-branch is typed without any (in)equations. Note also that the affine context is split between the scrutinee and the branches.

The constraints cross-universe elimination in (T-Let) and (T-Match) are similar to those imposed by Aura on its Prop universe. 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 exclude programming on proof terms, as illustrated in §2.6, 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.

Exceptions and state. The stratification of F^* into sub-languages makes it easy to provide primitive support for arbitrary effects. We illustrate this here using exceptions and mutable state—adding other effects is similarly straightforward. The rules from (T-Try) to (T-Wr) are mostly standard; they restrict the use of these constructs to the \star -fragment and split the affine contexts between any sub-expressions. The one technical device worth noting is that reference allocation **ref** t v takes an explicit type for its value. This simplifies our dynamic semantics and soundness proof, while this type can usually be inferred in practice.

Ghost refinements. The rules (T-V), (T-Ax), and the subsumption rule (T-Sub) introduce ghost refinement types. (T-Ax) introduces a unit refined with the assumed formula ϕ ; in the refinement, $_$ stands for a fresh variable. No logical evidence is produced for ϕ . To justify

22 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

$$\begin{array}{c}
\boxed{S; \Gamma \vdash t <: t'} \quad \text{ST-RefI} \frac{S; \Gamma \vdash t \equiv t'}{S; \Gamma \vdash t <: t'} \quad \text{ST-Trans} \frac{S; \Gamma \vdash t <: t_1 \quad S; \Gamma \vdash t_1 <: t'}{S; \Gamma \vdash t <: t'} \\
\text{ST-Prod} \frac{S; \Gamma \vdash t'_1 <: t_1 \quad S; \Gamma, x:t'_1 \vdash t_2 <: t'_2}{S; \Gamma \vdash x:t_1 \rightarrow t_2 <: x:t'_1 \rightarrow t'_2} \quad \text{ST-ProdK} \frac{S; \Gamma \vdash \kappa' <: \kappa \quad S; \Gamma, a::\kappa' \vdash t <: t'}{S; \Gamma \vdash \forall a::\kappa.t <: \forall a::\kappa'.t'} \\
\text{ST-Left} \frac{S; \Gamma \vdash t <: t'}{S; \Gamma \vdash x:t\{\phi\} <: t'} \quad \text{ST-Right} \frac{S; \Gamma \vdash t <: t' \quad S; \Gamma, x:t \models \phi'}{S; \Gamma \vdash t <: x:t\{\phi'\}} \quad \text{ST-Afn} \frac{S; \Gamma \vdash t <: t'}{S; \Gamma \vdash !t <: !t'} \\
\boxed{S; \Gamma \vdash t \equiv t'} \quad \text{TE-RefI} \frac{}{S; \Gamma \vdash t \equiv t} \quad \text{TE-Sym} \frac{S; \Gamma \vdash t_2 \equiv t_1}{S; \Gamma \vdash t_1 \equiv t_2} \quad \text{TE-Afn} \frac{S; \Gamma \vdash t \equiv t'}{S; \Gamma \vdash !t \equiv !t'} \\
\text{TE-TApp} \frac{S; \Gamma \vdash t_1 \equiv t'_1 \quad S; \Gamma \vdash t_2 \equiv t'_2}{S; \Gamma \vdash t_1 t_2 \equiv t'_1 t'_2} \quad \text{TE-Prod} \frac{S; \Gamma \vdash t_1 \equiv t'_1 \quad S; \Gamma, x:t_1 \vdash t_2 \equiv t'_2}{S; \Gamma \vdash x:t_1 \rightarrow t_2 \equiv x:t'_1 \rightarrow t'_2} \\
\text{TE-ProdK} \frac{S; \Gamma \vdash \kappa \equiv \kappa' \quad S; \Gamma, a::\kappa \vdash t \equiv t'}{S; \Gamma \vdash \forall a::\kappa.t \equiv \forall a::\kappa'.t'} \quad \text{TE-Ref} \frac{S; \Gamma \vdash t \equiv t' \quad S; \Gamma, x:t \vdash \phi \equiv \phi'}{S; \Gamma \vdash x:t\{\phi\} \equiv x:t'\{\phi'\}} \\
\text{TE-Lam} \frac{S; \Gamma \vdash t_1 \equiv t'_1 \quad S; \Gamma, x:t_1 \vdash t_2 \equiv t'_2}{S; \Gamma \vdash \lambda x:t_1.t_2 \equiv \lambda x:t'_1.t'_2} \quad \text{TE-}\beta \frac{}{S; \Gamma \vdash (\lambda x:t.t') v \equiv t'[v/x]} \\
\text{TE-T}\beta \frac{}{S; \Gamma \vdash (\Lambda a::\kappa.t) t' \equiv t'[a]} \quad \text{TE-T} \frac{t = t' \in \Gamma}{S; \Gamma \vdash t \equiv t'} \quad \text{TE-V} \frac{S; \Gamma \vdash t \equiv t' \quad S; \Gamma \models v = v'}{S; \Gamma \vdash t v \equiv t' v'}
\end{array}$$

Fig. 4. Sub-typing and type-conversion

this rule, the dynamic semantics of this expression adds the formula to the dynamic log \mathcal{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 that the kinding of the introduced formula do not rely on the introduced equality.

Subsumption. The judgment $S; \Gamma \vdash t_1 <: t_2$ in Figure 4 shows a reflexive and transitive fully structural subtyping relation on types. The main interesting rules are (ST-Left), which allows a ghost refinement to be dropped on a type, and (ST-Right) which introduces a ghost refinement type. Ghost refinements have no impact on the representation of values, so they admit full structural subtyping, as illustrated by rules like (ST-Prod) which allow co- and contravariant subtyping on functions. In contrast, concrete refinements (i.e., the dependent pairs of Coq, Fine, F^* , etc.) do not enjoy structural subtyping, although, via a systematic translation to insert coercions (called *derefinement*), the Fine language provided a weaker, non-structural subtyping relation on concrete refinements.

Subtyping also includes type conversion $S; \Gamma \vdash t_1 \equiv t_2$. This equivalence 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. The rules (TE-V) and (ST-Right) yield logic parametricity for F^* . Both those rules have premises that use a judgment $S; \Gamma \models \phi$, a logic derivability relation that can be “plugged in” to the type system, as long as the relation meets a few important admissibility constraints, which we enumerate below.

Definition 1 (Admissibility of entailment relation)

The relation $S; \Gamma \models \phi$ is admissible if and only if it is

- (1) At least the identity on refined assumptions: for all S, Γ, x, t, ϕ , such that $\vdash S; \Gamma, x: (x : t\{\phi\})$ wf, we have $S; \Gamma, x: (x : t\{\phi\}) \models \phi$.
- (2) Compatible with type equivalence: for all S, Γ, ϕ, ϕ' such that $\vdash S; \Gamma$ wf and $S; \Gamma \vdash \phi \equiv \phi'$, if x is fresh, then $S; \Gamma, x: (x: \text{unit}\{\phi\}) \models \phi'$.
- (3) Closed under value substitution: for all $S, \Gamma, x: t, \Gamma'$ such that $\vdash S; \Gamma, x: t, \Gamma'$ wf, we have $S; \Gamma, x: t, \Gamma' \models \phi \Rightarrow \forall v. S; \Gamma; \cdot \vdash v : t \Rightarrow S; \Gamma, \Gamma'[v/x] \models \phi[v/x]$.
- (4) Closed under type substitution: for all $S, \Gamma, a::\kappa, \Gamma'$ such that $\vdash S; \Gamma, a::\kappa, \Gamma'; \cdot$ ok, we have $S; \Gamma, a::\kappa, \Gamma' \models \phi \Rightarrow \forall t. S; \Gamma \vdash t :: \kappa \Rightarrow S; \Gamma, \Gamma'[t/a] \models \phi[t/a]$.
- (5) Closed under weakening: for all $S, \Gamma_1, \Gamma, \Gamma_2$ such that $\vdash S; \Gamma_1, \Gamma_2$ wf and $S; \Gamma_1, \Gamma, \Gamma_2$ wf, we have $S; \Gamma_1, \Gamma_2 \models \phi \Rightarrow S; \Gamma_1, \Gamma, \Gamma_2 \models \phi$.
- (6) Closed under elimination of derivable value equality: for all $S, \Gamma, v, v', \Gamma'$ such that $\vdash S; \Gamma, v = v', \Gamma'; \cdot$ ok and $S; \Gamma \vdash v \equiv v'$, we have $S; \Gamma, v = v', \Gamma' \models \phi \Rightarrow S; \Gamma, \Gamma' \models \phi$.
- (7) Closed under elimination of derivable type equality: for all $S, \Gamma, t, t', \Gamma'$ such that $\vdash S; \Gamma, t = t', \Gamma'; \cdot$ ok and $S; \Gamma \vdash t \equiv t'$, we have $S; \Gamma, t = t', \Gamma' \models \phi \Rightarrow S; \Gamma, \Gamma' \models \phi$.
- (8) Entailing only well-kinded formulas: for all S, Γ, ϕ such that $\vdash S; \Gamma$ wf, if $S; \Gamma \models \phi$ then $S; \Gamma \vdash \phi::E$.

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 affine indices in types, makes it easy to support automation. Formally, we also exploit logic parametricity to provide an embedding of a formal core of F7 into F*, plugging its entailment relation (which, unlike F*, includes the basic first-order connectives and equality, each satisfying their usual introduction and elimination forms).

Well-formedness of environments. Finally, Figure 5 defines well-formed environments. The first judgment, $\vdash S; \Gamma; X$ wf, is straightforward—it requires the names in X to be distinct, and for $\vdash S; \Gamma$ to be derivable. The latter holds when Γ binds distinct value names x at concrete types, distinct well-kinded type names, well-typed equations, and finally, when S is well-formed. The latter holds, when S binds at least `unit`, `ref`, and `exn` and has distinct \star -kinded logic value constructors.

Additionally, we have constraints on the well-formedness of datatypes T defined in S , according to the judgment $S \vdash T::\kappa\{\overline{C}:t\}$ ok. The main rule in the judgment is (T-Ok1), which requires the constructor to be well-typed, to have a concrete kind, and to construct a value whose type is an instance of type T . We also require the constructor to respect affinity restrictions and positivity constraints depending on the kind c of types constructed by T . This is captured by the last judgment $S; \Gamma \vdash \text{apos}(c, T, \Gamma')$. Its main rules, (APos-XP) and (APos-XA), ensure (1) that constructors with affine arguments 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 that excludes the constructed type T in negative position in any argument of D .

24 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

$$\begin{array}{c}
\boxed{\vdash S; \Gamma; X \text{ wf}} \quad \frac{\vdash S; \Gamma; X \text{ wf} \quad \alpha \notin X}{\vdash S; \Gamma; X, \alpha \text{ wf}} \quad \frac{\vdash S; \Gamma; X \text{ wf} \quad \ell \notin X}{\vdash S; \Gamma; X, \ell \text{ wf}} \quad \frac{\vdash S; \Gamma \text{ wf}}{\vdash S; \Gamma; \cdot \text{ wf}} \\
\boxed{\vdash S; \Gamma \text{ wf}} \quad \frac{S; \Gamma \vdash t :: c \quad \vdash S; \Gamma \text{ wf} \quad x \notin \Gamma}{\vdash S; \Gamma, x:t \text{ wf}} \quad \frac{S; \Gamma \vdash \kappa \text{ ok}(b) \quad \vdash S; \Gamma \text{ wf} \quad a \notin \Gamma}{\vdash S; \Gamma, a::\kappa \text{ wf}} \\
\frac{\forall i. S; \Gamma \vdash v_i : t \quad \vdash S; \Gamma \text{ wf}}{\vdash S; \Gamma, v_1 = v_2 \text{ wf}} \quad \frac{\forall i. S; \Gamma \vdash t_i :: k \quad \vdash S; \Gamma \text{ wf}}{\vdash S; \Gamma, t_1 = t_2 \text{ wf}} \quad \frac{\text{exn}::\star\{\overline{C:t}\} \in S \quad \vdash S \text{ wf}}{\vdash S; \cdot \text{ wf}} \\
\boxed{\vdash S \text{ wf}} \quad \frac{}{\vdash \text{unit}::\star\{(): \text{unit}\}, \text{ref}::\star\Rightarrow\star\} \text{ wf}} \\
\frac{\vdash S \text{ wf} \quad \{T, \overline{D}\} \cap FV(S) = \emptyset \quad S \vdash T::\kappa\{\overline{D:t}\} \text{ ok}}{\vdash S, T::\kappa\{\overline{D:t}\} \text{ wf}} \quad \frac{\vdash S \text{ wf} \quad L \notin FV(S) \quad S; \cdot \vdash t :: \star}{\vdash S, L:t \text{ wf}} \\
\boxed{S \vdash T::\kappa\{\overline{C:t}\} \text{ ok}} \quad \text{T-Ok0} \frac{S \vdash \kappa \text{ ok}(b)}{S \vdash T::\kappa\{\} \text{ ok}} \quad \text{T-OkN} \frac{S \vdash T::\kappa\{D_0:t_0\} \text{ ok} \quad S \vdash T::\kappa\{\overline{D:t}\} \text{ ok}}{S \vdash T::\kappa\{D_0:t_0, \overline{D:t}\} \text{ ok}} \\
\text{T-Ok1} \frac{S; \cdot \vdash \kappa \text{ ok}(c) \quad \text{dom}(\Gamma) = \bar{\alpha} \quad S, T::\kappa\{\}; \Gamma; \cdot \vdash C \bar{\alpha} : T \bar{\tau}(c) \quad S; \cdot \vdash \text{apos}(c, T, \Gamma)}{S \vdash T::\kappa\{C:t\} \text{ ok}} \\
\boxed{S; \Gamma \vdash \text{apos}(c, T, \Gamma')} \quad \text{APos-0} \frac{}{S; \Gamma \vdash \text{apos}(c, T, \cdot)} \quad \text{APos-T} \frac{S; \Gamma, a::\kappa \vdash \text{apos}(c, T, \Gamma')}{S; \Gamma \vdash \text{apos}(c, T, (a::\kappa, \Gamma'))} \\
\text{APos-XP} \frac{t = \Gamma_t \rightarrow t' \quad \forall 0 \leq i < |\Gamma_t|. T \notin FTC(\Gamma_t[i]) \quad S; \Gamma, x:t \vdash \text{apos}(P, T, \Gamma')}{S; \Gamma \vdash \text{apos}(P, T, (x:t, \Gamma'))} \\
\text{APos-XA} \frac{S; \Gamma \vdash t :: c' \quad c' \neq P \quad c' = A \Rightarrow c = A \quad S; \Gamma, x:t \vdash \text{apos}(c, T, \Gamma')}{S; \Gamma \vdash \text{apos}(c, T, (x:t, \Gamma'))}
\end{array}$$

Fig. 5. Well-formed signatures, environments and runtime states

3.3 Dynamic semantics: logical effects and affine names

The dynamic semantics of F^* , shown in Figure 6, is a small-step reduction relation for a call-by-value language. The semantics is given in the style of Felleisen & Hieb (1992) using evaluation contexts E and exception frames F . It has the form $(\mathcal{A}; e) \rightarrow_S (\mathcal{A}'; e')$ where the signature S is unvarying, and a runtime configuration is a pair of a runtime state \mathcal{A} and an expression e . The state \mathcal{A} maintains a set of facts introduced by the dynamic assumption of ghost refinements and a map from reference locations to values. Additionally, \mathcal{A} records affine names in two variants, ℓ or $\hat{\ell}$, which we use to track the usage of affinely typed values. The most interesting part of this relation is the way it maintains a non-decreasing set of logical facts ϕ and names ℓ for affine values in the state \mathcal{A} . As mentioned in §2.2, the set of facts in \mathcal{A} is used in the definition of ghost refinement derivability. Facts are added to the log using the **assume** ϕ form, reduced by rule (E-Log).

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 $(\cdot; e) \rightarrow_S^* (\mathcal{A}; v)$, its refinement formula ϕ is derivable from the signature and all the accumulated logical effects:

$$\begin{array}{l}
\mathcal{A} ::= \cdot \mid \phi \mid \ell \mid \hat{\ell} \mid (\ell \mapsto_t v) \mid \mathcal{A}, \mathcal{A}' \quad \text{runtime state} \\
F[\bullet] ::= \bullet \mid F v \mid F t \mid \mathbf{let} x = F \mathbf{in} e \quad \text{exception frame} \\
E[\bullet] ::= \bullet \mid E v \mid E t \mid \mathbf{let} x = E \mathbf{in} e \mid \mathbf{try} E \mathbf{with} x.e \quad \text{evaluation context}
\end{array}$$

$$\begin{array}{c}
\text{E-Ctx} \frac{(\mathcal{A}, e) \rightarrow_S (\mathcal{A}', e') \quad e \neq \mathbf{raise} _}{(\mathcal{A}, E[e]) \rightarrow_S (\mathcal{A}', E[e'])} \quad \text{E-Kill} \frac{}{(\mathcal{A}_1, \ell, \mathcal{A}_2; E[v^\ell]) \rightarrow_S (\mathcal{A}_1, \hat{\ell}, \mathcal{A}_2; E[v])} \\
\text{E-Err} \frac{}{(\mathcal{A}; F[\mathbf{raise} v]) \rightarrow_S (\mathcal{A}, \mathbf{raise} v)} \quad \text{E-Hdl} \frac{}{(\mathcal{A}; \mathbf{try}(F[\mathbf{raise} v]) \mathbf{with} x.e) \rightarrow_S (\mathcal{A}; e[v/x])} \\
\text{E-Log} \frac{}{(\mathcal{A}; \mathbf{assume} \phi) \rightarrow_S (\mathcal{A}, \phi; ())} \quad \text{E-Let} \frac{}{(\mathcal{A}; \mathbf{let} x = v \mathbf{in} e) \rightarrow_S (\mathcal{A}; e[v/x])} \\
\text{E-Beta} \frac{}{(\mathcal{A}; (\lambda x:t.e) v) \rightarrow_S (\mathcal{A}; e[v/x])} \quad \text{E-TBeta} \frac{}{(\mathcal{A}; (\Lambda \alpha::k.e) t) \rightarrow_S (\mathcal{A}; e[t/\alpha])} \\
\text{E-M1} \frac{}{(\mathcal{A}; \mathbf{match} C \bar{\tau} \mathbf{with} C \bar{\alpha} \rightarrow e_1 \mathbf{else} e_2) \rightarrow_S (\mathcal{A}; e_1[\bar{\tau}/\bar{\alpha}])} \\
\text{E-M2} \frac{C \neq C'}{(\mathcal{A}; \mathbf{match} C \bar{\tau} \mathbf{with} C' \bar{\alpha} \rightarrow e_1 \mathbf{else} e_2) \rightarrow_S (\mathcal{A}; e_2)} \quad \text{E-Rd} \frac{\ell \mapsto_t v \in \mathcal{A}}{(\mathcal{A}; !\ell) \rightarrow_S (\mathcal{A}; v)} \\
\text{E-Wr} \frac{\mathcal{A}' = \mathcal{A}[\ell \mapsto_t v]}{(\mathcal{A}; \ell := v) \rightarrow_S (\mathcal{A}'; ())} \quad \text{E-Ref} \frac{\ell \notin \text{Names}(\mathcal{A})}{(\mathcal{A}; \mathbf{ref} t v) \rightarrow_S (\mathcal{A}, \ell \mapsto_t v; \ell)} \\
\text{E-Newl} \frac{\mathcal{A} \Longrightarrow \Gamma; X \quad S; \Gamma; X \vdash v : t \quad S; \Gamma \vdash t :: A \quad \ell \notin \text{Names}(\mathcal{A})}{(\mathcal{A}; v) \rightarrow_S (\mathcal{A}, \ell; v^\ell)}
\end{array}$$

$$\begin{array}{l}
\text{where} \\
\begin{array}{ll}
\text{live}(\mathcal{A}, \ell \mapsto _) = \text{live}(\mathcal{A}, \ell) & \text{dead}(\mathcal{A}, \ell) = \text{dead}(\mathcal{A}) \\
\text{live}(\mathcal{A}, \ell) = \text{live}(\mathcal{A}, \hat{\ell}) & \text{dead}(\mathcal{A}, \hat{\ell}) = \text{dead}(\mathcal{A}, \hat{\ell}) \\
\text{live}(\mathcal{A}, \phi) = \text{live}(\mathcal{A}) & \text{dead}(\mathcal{A}, \phi) = \text{dead}(\mathcal{A}) \\
\text{live}(\mathcal{A}, \phi) = \text{live}(\mathcal{A}) & \text{Names}(\mathcal{A}) = \text{live}(\mathcal{A}), \text{dead}(\mathcal{A})
\end{array}
\end{array}$$

$$\begin{array}{c}
\boxed{\mathcal{A} \Longrightarrow \Gamma; X} \quad \frac{}{\cdot \Longrightarrow \cdot} \quad \frac{\mathcal{A} \Longrightarrow \Gamma; X \quad \text{fresh } x}{(\mathcal{A}, \phi) \Longrightarrow \Gamma, x: \mathbf{unit}\{\phi\}; X} \\
\frac{\mathcal{A} \Longrightarrow \Gamma; X}{(\mathcal{A}, \ell \mapsto_t v) \Longrightarrow \Gamma, \ell: \mathbf{ref} t; X} \quad \frac{\mathcal{A} \Longrightarrow \Gamma; X}{(\mathcal{A}, \ell) \Longrightarrow \Gamma; X, \ell} \quad \frac{\mathcal{A} \Longrightarrow \Gamma; X}{(\mathcal{A}, \hat{\ell}) \Longrightarrow \Gamma; X} \\
\boxed{S; \Gamma \vdash \mathcal{A} \text{ wf}} \quad \frac{}{S; \Gamma \vdash \cdot \text{ wf}} \quad \frac{\ell: \mathbf{ref} t \in \Gamma \quad S; \Gamma; \cdot \vdash v : t}{S; \Gamma \vdash \mathcal{A}, \ell \mapsto_t v} \quad \frac{S; \Gamma \vdash \mathcal{A} \text{ wf} \quad S; \Gamma \vdash \phi :: E}{S; \Gamma \vdash \mathcal{A}, \phi \text{ wf}} \\
\frac{S; \Gamma \vdash \mathcal{A} \text{ wf} \quad \ell, \hat{\ell} \notin \mathcal{A}}{S; \Gamma \vdash \mathcal{A}, \ell \text{ wf}} \quad \frac{S; \Gamma \vdash \mathcal{A} \text{ wf} \quad \ell, \hat{\ell} \notin \mathcal{A}}{S; \Gamma \vdash \mathcal{A}, \hat{\ell} \text{ wf}}
\end{array}$$

Fig. 6. Dynamic semantics of F^* : $(\mathcal{A}; e) \rightarrow_S (\mathcal{A}'; e')$

$S; \text{Facts}(\mathcal{A}) \models \phi[v/x]$. This is in contrast to our soundness result for proof 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 state \mathcal{A} 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-New ℓ). Names held in the log come in two variants: names ℓ are “live”, while names $\hat{\ell}$ are “dead”—the auxiliary functions $\text{live}(\cdot)$ and $\text{dead}(\cdot)$ collect these names from the state. To introduce a new name, (E-New ℓ) checks if a value has an affine type by reifying the runtime state \mathcal{A} into a typing environment (using $\mathcal{A} \Longrightarrow \Gamma; X$) and typing the value. When v appears in a destructor position—the context E in (E-Kill), which includes the function position of a β redex, and the scrutinee position of a match—reduction requires the name ℓ to be “live” in the log, kills the name in \mathcal{A} , and then proceeds. This instrumentation serves as our specification of the use-at-most-once property.

At the bottom of the figure we show the rules for typing a runtime state \mathcal{A} . This is straightforward, requiring the names in \mathcal{A} to be distinct, for the values stored in reference cells to be well-typed, and for all logged formulas to be E -kinded.

This completes our presentation of the static and dynamic semantics of F^* . Next, we develop its metatheory proving type soundness, strong normalization of the P -fragment, and discussing an embedding of RCF in F^* .

4 Metatheory of F^*

This section provides our main theorems for F^* . The proofs of its metatheory were initially conducted by hand; as we reached completion of the hand proofs, we started a formalization of F^* in the Coq proof assistant, using the SSREFLECT extension of Gonthier *et al.* (2011). We now have a mechanized proof of type soundness for the whole language defined in §3. In the process, we found a few oversights in our pencil and paper proofs, which we have since corrected.

Our formalization in Coq is noteworthy in several ways. First, it 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 of 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 applied it to 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. The type system has many rules. Furthermore, its central judgments are all mutually recursive. Despite these complications, we are happy to note that our framework has allowed us to develop short, largely automated proofs: the final development consists of 1 KLOC for the framework, 2.5 KLOC for the F^* definition, and 4 KLOC for its metatheory. Our experience is encouraging initial progress towards 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 our development of the F^* formal metatheory will serve as a basis of such a framework.

4.1 Type soundness via preservation and progress

Our first theorem is a type soundness result, stated in terms of standard progress and preservation—we refer to the formal development for the intermediate 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 the auxiliary judgment $\mathcal{A} \Longrightarrow \Gamma; X$ of Figure 6, which obtains a context $\Gamma; X$ from the dynamic log, where Γ and X collect the logical assumptions and the live names of \mathcal{A} , respectively.)

Theorem 1 (Type soundness)

For all $S, \mathcal{A}, \Gamma, X, X', e$, and t such that

- (1) $S \vdash \mathcal{A} \Longrightarrow \Gamma; X$,
- (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 \mathcal{A}', e' , and Γ' such that

- (1) $(\mathcal{A}, e) \rightarrow (\mathcal{A}', e')$,
- (2) $S \vdash \mathcal{A}' \Longrightarrow \Gamma'; X''$, and
- (3) $S; \Gamma'; X'' \vdash e' : t$.

where Γ' extends Γ and $X'' = (X' \setminus \text{dead}(\mathcal{A}')) \cup \text{live}(\mathcal{A}' \setminus \mathcal{A})$.

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 $S, \mathcal{A}, \Gamma, X, e, \phi, t, \mathcal{A}', \Gamma'$, and v such that $\vdash S$ ok, $S \vdash \mathcal{A} \Longrightarrow \Gamma; X$, $S; \Gamma; X \vdash e : x:t\{\phi\}$, and $(\mathcal{A}; e) \rightarrow_S^* (\mathcal{A}'; v)$, there exists Γ' such that $S \vdash \mathcal{A}' \Longrightarrow \Gamma'; _$ and $S; \Gamma' \models \phi[v/x]$.

4.2 Embedding RCF in F^*

To relate F^* to $F7$, one of its predecessor, our second theorem gives 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. This fragment of RCF is used, for instance, by Fournet *et al.* (2011) for modelling ideal cryptography. We refer to the technical report for the definition of RCF and its embedding, which mostly account for syntactic differences between the two languages. In the statement below, A is an RCF configuration; E is an RCF context; and $A \rightarrow A'$ is a single step of reduction in RCF. The notation $\llbracket \cdot \rrbracket$ 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 2 (Well-typed translation of RCF)

Given $E \vdash \llbracket A \rrbracket = (\mathcal{A}; e)$, $\vdash \llbracket E \rrbracket = S; \Gamma$, and $E \vdash \diamond$, we have $S; \Gamma; _ \vdash e : t$ where $E \vdash \llbracket T \rrbracket = t$. Additionally $A \rightarrow A'$ if and only if $(\mathcal{A}; e) \rightarrow_S (\mathcal{A}'; e')$ and $E \vdash \llbracket A' \rrbracket = (\mathcal{A}'; e')$.

4.3 Strong normalization of the P -fragment

One significant difference between our version of F^* and the version of Swamy *et al.* (2011) is in the strong normalization proof for its P -fragment. Our initial proof involved translating P -terms in F^* to CiC (The Coq Development Team 2010), and proving this translation a simulation. Relying on recent strong normalization results for CiC, we concluded that P -terms in F^* were also strongly normalizing. This indirect proof was, however, not without difficulties. In particular, the implicit type conversion relation of F^* , from any equations in context, were represented in the translation to CiC through the use of dependent pattern matching with explicit coercions. Completing the proof in this style involved cumbersome book-keeping. Additionally, in order for the proof to go through, we required a non-standard side condition on the rule (T-TApp) for type applications. We discuss this side condition next, illustrating it on an example. Recall that, in our previous formulation, we had the additional sub-kinding rule $P <: \star$, and consider the following program:

```

type False :: P
let badid :  $\forall a::\star. \text{unit} \rightarrow a = \text{fun } (a::\star) () \rightarrow \text{raise Error in}$ 
    badid False : unit  $\rightarrow$  False

```

We define first a P -kinded type `False` with no constructors, then a polymorphic function of type $\forall a::\star. \text{unit} \rightarrow a$ that simply raises an exception. Since the co-domain of `badid` is a \star -kinded type, the function is free to have effects like exceptions or divergence. However, if we are not careful, the rule $P <: \star$ let us instantiate `badid` to `False` and obtain a term of type $\text{unit} \rightarrow \text{False}$. Since this function has a P -kinded co-domain, we expect it to be total, but, clearly, it is not. To exclude such problematic instantiation, our earlier system includes the following variation on rule (T-TApp):

$$\text{T-TApp-Old} \frac{S; \Gamma; X \vdash^m e : t_v \quad !t_v = \forall \alpha::\kappa. t' \quad S; \Gamma \vdash t :: \kappa \quad S; \Gamma \vdash t'[t/\alpha] \equiv t'' \quad S; \Gamma \vdash t'' :: P}{S; \Gamma; X \vdash^m e t : t'[t/\alpha]} \implies S; \Gamma \vdash t_v :: P$$

The third and fourth premises of (T-TApp-Old) ensured that if, after instantiating a to t in t' , we obtain a type t'' in P , then t' must have been in P initially. The side condition excludes the problematic example listed above, since the initial type $\forall a::\star. \text{unit} \rightarrow a$ has kind \star and after instantiation it has kind P . The side condition also accounts for examples in which t' could have more than one kind, e.g., kind \star then kind P after type conversion to t'' .

While this side condition appears sufficient to establish the soundness of the P -fragment while tolerating $P <: \star$, we find this restriction on type application cumbersome. More fundamentally, this style of side condition prevented us from easily developing a direct, domain-theoretic proof of strong normalization for the P -fragment. Thus, as discussed earlier, we revised our kind hierarchy to disallow $P <: \star$ and simplify (T-TApp-Old) to (T-TApp). In the remainder of this section, we discuss our new proof of strong normalization.

The proof is decomposed in three phases. First, we define a new reduction relation on expressions which captures, at the P level, the one defined in §3.3. We then show, using the now standard technique of *reducibility candidates*, that this reduction relation is strongly normalizing on well-typed expressions of the P fragment. We conclude by transferring the strong normalization property to the dynamic semantics of F^* .

Weaker reductions Our new reduction relation is defined by pruning from the dynamic semantic of F^* all the rules not related to the P -level. The main benefit is the removal of all the rules depending on the log \mathcal{A} , which matters only at the \star and A levels, leading to a simpler relation that still captures the dynamic semantic at the P level.

Definition 2 (Weak dynamic semantic)

Figure 6, where only the rules E-Ctx, E-Let, E-Beta, E-TBeta, E-M1 and E-M2 are kept, defines the *weak dynamic semantic* of F^* . Since this semantics does not involve \mathcal{A} , we write $e \rightarrow_S e'$, or simply $e \rightarrow e'$ when S is clear from the context, if e weakly reduces to e' .

The relation \rightarrow is non-branching, hence confluent. Moreover, from type soundness, we obtain the subject reduction property:

Lemma 3 (Subject reduction)

If $S; \Gamma; X \vdash^m e : t$ and $e \rightarrow_S e'$, then $S; \Gamma; X \vdash^m e' : t$.

Proof From Theorem 1 (Type soundness), since \rightarrow does not create any new label, it is sufficient to take $\mathcal{A}' = \mathcal{A}$ and $\Gamma = \Gamma'$. \square

As stated, the weak dynamic semantic simulates F^* dynamic semantic at the P level:

Lemma 4 (P-level dynamic semantic simulation)

Assume that $S; \Gamma; X \vdash^m e : t$ with $S; \Gamma \vdash t :: \kappa$, and that $(\mathcal{A}, e) \rightarrow (\mathcal{A}', e')$. Then, $e \rightarrow e'$.

For the sake of the strong normalization proof, we also define a reduction relation at type level capturing the contraction of type level β -redexes.

Definition 5 (Type level reduction)

The type level reduction, written $\cdot \rightarrow \cdot$, is the smallest compatible (i.e. context-closed) relation such that $(\Lambda a :: \kappa.t) u \rightarrow t[u/a]$ and $(\lambda x:t.u) v \rightarrow u[v/x]$.

Type level reduction is quite standard and clearly confluent. Moreover, as a consequence of type soundness, we know that type level judgment is stable by substitution, and thus obtain a subject reduction property:

Lemma 6 (Type level subjection reduction)

If $S; \Gamma \vdash t :: \kappa$ and $t \rightarrow u$, then $S; \Gamma \vdash u :: \kappa$.

We can now move to the proof of strong normalization of the weak reduction at P -level.

Strong normalization of weak reduction For the entire section, we assume given a fixed, well-formed, inductive signature S . We write **SN** for the set of expressions that weakly strongly normalized, and **NT** for neutral expressions, defined bellow:

Definition 7 (Neutral terms)

An expression is *neutral* if it is not of the following form:

- (1) **abstraction**: $\lambda x:t.e$
- (2) **type-level abstraction**: $\Lambda a::\kappa.e$
- (3) **applied constructor**: $C \bar{\tau}$

The sets of values, expressions, types and kinds are respectively denoted by Λ_v , Λ_e , Λ_t and Λ_κ . We write \mathcal{X}^t and \mathcal{X}^e for the set of type variables and value variables. We are ready to define our notion of *reducibility candidates*:

30 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

Definition 8 (Reducibility candidates)

Let \preceq_κ be the smallest (partial) order such that

$$\begin{aligned} P \preceq E \quad \star \preceq E \quad A \preceq E \\ x:t \Rightarrow \kappa \preceq_\kappa x:t' \Rightarrow \kappa' \quad \text{whenever } \kappa \preceq_\kappa \kappa' \\ \alpha:\kappa_1 \Rightarrow \kappa_2 \preceq_\kappa \alpha:\kappa_1' \Rightarrow \kappa_2' \quad \text{whenever } \kappa_1' \preceq_\kappa \kappa_1 \text{ and } \kappa_2 \preceq_\kappa \kappa_2' \end{aligned}$$

Let \sim_κ be the equivalence relation induced by \preceq_κ . We define the sets \mathcal{R}_κ of the interpretations for the types of kind κ . The definition is by induction on the structure of κ , by case on the head symbol:

- \mathcal{R}_P is the set of all the subsets of Λ_e s.t.

$$\begin{aligned} \text{(CR1)} \quad R \subseteq \mathbf{SN} \\ \text{(CR2)} \quad \text{if } t \in R, \text{ then } u \in R \text{ whenever } t \rightarrow u \\ \text{(CR3)} \quad \text{if } t \in \mathbf{NT} \text{ and } u \in R \text{ whenever } t \rightarrow u, \text{ then } t \in R \end{aligned}$$

- $\mathcal{R}_b = \{\Lambda_e\}$ for $b \in \{\star, A\}$.
- $\mathcal{R}_E = \wp(\Lambda_e)$.
- $\mathcal{R}_{x:t \Rightarrow \kappa}$ is the set of functions from $\{\emptyset\}$ to $\mathcal{R}_{\kappa'}$ for all $\kappa' \preceq_\kappa \kappa$.
- $\mathcal{R}_{\alpha:\kappa_1 \Rightarrow \kappa_2}$ is the set of functions R from $\Lambda_e \times \mathcal{R}_{\kappa_1'}$ to $\mathcal{R}_{\kappa_2'}$ for all κ_1', κ_2' s.t. $\kappa_1 \preceq_\kappa \kappa_1'$ and $\kappa_2' \preceq_\kappa \kappa_2$, s.t. $R(e, S) = R(e', S)$ whenever $e \rightarrow e'$.

It is immediate to check that $\kappa \sim_\kappa \kappa'$ (resp. $\kappa \preceq_\kappa \kappa'$) implies $\mathcal{R}_\kappa = \mathcal{R}_{\kappa'}$ (resp. $\mathcal{R}_\kappa \subseteq \mathcal{R}_{\kappa'}$). Hence:

Lemma 9 (Candidate (in)variance)

If $\Gamma \vdash \kappa \equiv \kappa'$ (resp. $\Gamma \vdash \kappa <: \kappa'$), then $\mathcal{R}_\kappa = \mathcal{R}_{\kappa'}$ (resp. $\mathcal{R}_\kappa \subseteq \mathcal{R}_{\kappa'}$).

By a direct induction, we obtain:

Lemma 10 (Candidate invariance by substitution)

\mathcal{R}_κ is invariant by substitution on κ .

The interpretation of inductive types in the P fragment will be defined as the smallest fixpoint of a given operator over \mathcal{R}_κ that we equip a structure of a complete lattice:

Lemma 11 (Lattice structure of \mathcal{R}_κ)

\mathcal{R}_κ can be equipped with a structure of complete lattice.

Proof We define $(\mathcal{R}_\kappa, \leq_\kappa, \top_\kappa, \text{lub}_\kappa)$, by induction on κ , where \leq_κ (resp. $\top_\kappa, \text{lub}_\kappa(\mathbf{R})$) stands for the ordering over \mathcal{R}_κ (resp. the top element of \mathcal{R}_κ , the supremum of $\mathbf{R} \in \wp(\mathcal{R}_\kappa)$).

- $\leq_P = \subseteq, \top_P = \mathbf{SN}$ and $\text{lub}(\mathbf{T}) = \bigcap \mathbf{T}$.
- If $b \in \{\star, A\}$, then $\leq_b = \subseteq, \top_b = \Lambda_e$, and $\text{lub}(\mathbf{T}) = \Lambda_e$.
- $\leq_E = \subseteq, \top_E = \Lambda_e$ and $\text{lub}_E(\mathbf{T}) = \bigcap \mathbf{T}$.
- Let $R_1 : \{\emptyset\} \rightarrow \mathcal{R}_{\kappa_1}, R_2 : \{\emptyset\} \rightarrow \mathcal{R}_{\kappa_2} \in \mathcal{R}_{x:t \Rightarrow \kappa}$. Then, $R_1 \preceq_{\mathcal{R}_{x:t \Rightarrow \kappa}} R_2$ if $\kappa_1 \preceq_\kappa \kappa_2$ and $R_1(\emptyset) \preceq_{\mathcal{R}_{\kappa_2}} R_2(\emptyset)$. Moreover, $\top_{x:t \Rightarrow \kappa} = \emptyset \mapsto \top_\kappa$ and $\text{lub}_{x:t \Rightarrow \kappa}(\mathbf{R}) = \emptyset \mapsto \text{lub}_\kappa\{R(\emptyset) \mid R \in \mathbf{R}\}$.

- Let $R_1 : \mathcal{R}_{\kappa_1^{(1)}} \rightarrow \mathcal{R}_{\kappa_2^{(1)}}$, $R_2 : \mathcal{R}_{\kappa_1^{(2)}} \rightarrow \mathcal{R}_{\kappa_2^{(2)}} \in \mathcal{R}_{a::\kappa_1 \Rightarrow \kappa_2}$. Then $R_1 \preceq_{\mathcal{R}_{a::\kappa_1 \Rightarrow \kappa_2}} R_2$ if $\kappa_1^{(2)} \preceq_{\kappa} \kappa_1^{(1)}$, $\kappa_2^{(1)} \preceq_{\kappa} \kappa_2^{(2)}$, and for any $x \in \mathcal{R}_{\kappa_1^{(2)}}$, $R_1(x) \preceq_{\mathcal{R}_{\kappa_2^{(2)}}} R_2(x)$. Moreover, $\top_{a::\kappa_1 \Rightarrow \kappa_2} = (e, S) \mapsto \top_{\kappa_2}$ and $\text{lub}_{a::\kappa_1 \Rightarrow \kappa_2}(\mathbf{R}) = \text{lub}_{\kappa_2}(\{R(e, S) \mid R \in \mathbf{R}\})$.

□

We now move to the definition of the interpretation schema:

Definition 12 (Interpretation)

Let $\mathcal{R} = \bigcup_{k \in \Lambda_{\kappa}} \mathcal{R}_k$. A candidate assignment is any finite map from \mathcal{X}^{\dagger} to \mathcal{R} . We inductively define the interpretation of a well-typed t w.r.t a candidate assignment ξ and a type-level substitution θ , written $\llbracket t \rrbracket_{\xi, \theta}$. If $t :: b$ but $\neg(t :: P)$, then $\llbracket t \rrbracket_{\xi, \theta} = \Lambda_e$. Otherwise:

$$\begin{aligned}
\llbracket a \rrbracket_{\xi, \theta} &= \xi(a) \\
\llbracket T \rrbracket_{\xi, \theta} &= \mathcal{I}_T \\
\llbracket x:t \rightarrow u \rrbracket_{\xi, \theta} &= \{e \in \Lambda_e \mid \forall v \in \llbracket t \rrbracket_{\xi, \theta} \cap \Lambda_v, (e v) \in \llbracket u \rrbracket_{\xi, \theta}\} \\
\llbracket \forall a::k.u \rrbracket_{\xi, \theta} &= \{e \in \Lambda_e \mid \forall t \in \Lambda_t, \forall S \in \mathcal{R}_k, (e t) \in \llbracket u \rrbracket_{\xi_a^S, \theta_a^t}\} \\
\llbracket t v \rrbracket_{\xi, \theta} &= \llbracket t \rrbracket_{\xi, \theta}(\theta) \\
\llbracket t u \rrbracket_{\xi, \theta} &= \llbracket t \rrbracket_{\xi, \theta}(u\theta, \llbracket u \rrbracket_{\xi, \theta}) \\
\llbracket \lambda x:t.u \rrbracket_{\xi, \theta} &= \theta \mapsto \llbracket u \rrbracket_{\xi, \theta} \\
\llbracket \Lambda a::k.u \rrbracket_{\xi, \theta} &= (t, S) \in \Lambda_t \times \mathcal{R}_k \mapsto \llbracket u \rrbracket_{\xi_a^S, \theta_a^t} \\
\llbracket x:t\{\phi\} \rrbracket_{\xi, \theta} &= \llbracket t \rrbracket_{\xi, \theta}
\end{aligned}$$

The definition of the interpretation of inductive types \mathcal{I}_T is done in a standard way using an introduction based methodology. For instance, $\mathcal{I}_{\text{nat}} = F^{\infty}(\emptyset)$ where $F(X) = \{e \in \mathbf{SN} \mid e \rightarrow^* \mathbf{0}\} \cup \{e \in \mathbf{SN} \mid e \rightarrow^* \mathbf{S} e' \text{ with } e' \in X\}$. As usual, the lattice structure of the \mathcal{R}_{κ} and the strict positivity of inductive types at P level assure the well formeness of the definition of the map \mathcal{I} .

A candidate assignment ξ validates an environment Γ , written $\xi \vDash \Gamma$, if for any variable $a \in \text{dom}(\Gamma) \cap \mathcal{X}^{\dagger}$, $\xi(a) \in \mathcal{R}_{\Gamma(a)}$, and if for any equation $a = t \in \Gamma$, $\xi(a) = \llbracket t \rrbracket_{\xi, \theta}$. We say that ξ is a Γ -assignment. A type-level substitution is *adapted to a Γ -assignment ξ* if $\text{dom}(\theta) \subseteq \text{dom}(\Gamma)$ and for any variable $a \in \text{dom}(\theta)$, $a\theta \in \llbracket x\Gamma \rrbracket_{\xi, \theta}$. A pair (ξ, θ) composed of a candidate assignment and a type-level substitution is *adapted to a typing environment* if ξ is a Γ -assignment and θ is adapted to ξ .

We are now left to prove that if $\Gamma; S; X \vdash^m e : t :: \kappa$, then $t\theta \in \llbracket t \rrbracket_{\xi, \theta} \in \mathcal{R}_{\kappa}$ for any pair (ξ, θ) adapted to Γ .

Lemma 13 (Well-formness of interpretation)

If $S; \Gamma \vdash t :: \kappa$, then $\llbracket t \rrbracket_{\xi, \theta} \in \mathcal{R}_{\kappa}$ for any candidate assignment ξ adapted to Γ .

Proof By induction on $S; \Gamma \vdash t :: \kappa$, proving in parallel that if $\theta \rightarrow \theta'$, then $\llbracket t \rrbracket_{\xi, \theta} = \llbracket t \rrbracket_{\xi, \theta'}$. (This second property is always obtained by a direct application of the induction hypothesis and is not detailed). If $S; \Gamma \vdash t :: b$ but $\neg(S; \Gamma \vdash t :: P)$, then $\llbracket t \rrbracket_{\xi, \theta}^{\kappa} = \Lambda_e \in \{\Lambda_e\} = \mathcal{R}_b$. Otherwise, we do a case analysis on the last rule used:

- If $S; \Gamma \vdash a :: \Gamma(a)$, then $\llbracket a \rrbracket_{\xi, \theta} = a\xi \in \mathcal{R}_{\Gamma(a)}$ by assumption on ξ .
- If $S; \Gamma \vdash T :: S(T)$, then $\llbracket T \rrbracket_{\xi, \theta} = \mathcal{I}_T \in \mathcal{R}_{S(T)}$ by assumption on \mathcal{I} .

32 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

- Assume that $S; \Gamma \vdash \forall a :: \kappa.t :: P$ is derived using the rule $\text{K-}\forall$ from i) $S; \Gamma \vdash \kappa \text{ ok}(P)$, and ii) $S; \Gamma; a :: \kappa \vdash t :: P$. Let $R = \{e \in \Lambda_e \mid \forall t \in \Lambda_t, \forall S \in \mathcal{R}_k, (e t) \in \llbracket u \rrbracket_{\xi_a^S, \theta_a^u}\}$. We have to prove that $R \in \mathcal{R}_P$:
 - **(CR1)** Let $e \in R$. For any $u \in \Lambda_t, S \in \mathcal{R}_\kappa$, we have $e u \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$. By application of the induction hypothesis on ii), $\llbracket t \rrbracket_{\xi_a^S, \theta_a^u} \in \mathcal{R}_P \supseteq \mathbf{SN}$. Taking $u = x$, we obtain $e x \in \mathbf{SN}$, hence $e \in \mathbf{SN}$.
 - **(CR2)** Let $e \in R$ and $e' \text{ s.t. } e \rightarrow e'$. Let $u \in \Lambda_t$ and $S \in \mathcal{R}_\kappa$. Then, $e u \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$. By induction hypothesis, $\llbracket t \rrbracket_{\xi_a^S, \theta_a^u} \in \mathcal{R}_P$, and is thus stable by reduction (property **(CR3)**). Since u is normal, $e u \rightarrow e' u$ and $e' u \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$. Hence, $e' \in R$.
 - **(CR3)** Let $e \in \mathbf{NT}$. Assume that $e' \in R$ whenever $e \rightarrow e'$. Let $u \in \Lambda_t$ and $S \in \mathcal{R}_\kappa$. We have to show that $e u \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$. By induction hypothesis, $\llbracket t \rrbracket_{\xi_a^S, \theta_a^u} \in \mathcal{R}_P$. Hence, noting that $e u$ is neutral, by **(CR3)**, $e u \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$ if the reduced of $e u$ is in $\llbracket t \rrbracket_{\xi_a^S, \theta_a^u}$. Since e is neutral, $e u$ can only reduce to a term of the form $e' u$. By assumption, $e' \in R$, hence the result.
- Assume that $S; \Gamma \vdash \Lambda a :: \kappa.t :: a :: \kappa \Rightarrow \kappa'$ is derived using the rule $\text{K-}\Lambda$ from i) $S; \Gamma \vdash a :: \kappa \Rightarrow \kappa' \text{ ok}(b)$, and ii) $S; \Gamma; a :: \kappa \vdash t :: \kappa'$. We have to prove that $R = \llbracket \Lambda a :: \kappa.t \rrbracket_{\xi, \theta} \in \mathcal{R}_{a :: \kappa \Rightarrow \kappa'}$:
 - Let $(u, S) \in \Lambda_t \times \mathcal{R}_\kappa$. Then, $R(u, S) = \llbracket u \rrbracket_{\xi_a^S, \theta_a^u}$. By induction hypothesis on ii), $R(u, S) \in \mathcal{R}_{\kappa'}$.
 - Moreover, assume that $u \rightarrow u'$. Then, $R(u', S) = \llbracket u' \rrbracket_{\xi_a^S, \theta_a^{u'}}$. By induction hypothesis, since $\theta_a^u \rightarrow \theta_a^{u'}$, $\llbracket u \rrbracket_{\xi_a^S, \theta_a^u} = \llbracket u \rrbracket_{\xi_a^S, \theta_a^{u'}}$.
- Assume that $S; \Gamma \vdash t t' :: \kappa'[t'/a]$ using rule K-tt from i) $S; \Gamma \vdash t :: a :: \kappa \Rightarrow \kappa'$, and ii) $S; \Gamma \vdash t' :: \kappa$. We have to prove that $R = \llbracket t t' \rrbracket_{\xi, \theta} = \llbracket t \rrbracket_{\xi, \theta} (t' \theta, \llbracket t' \rrbracket_{\xi, \theta}) \in \mathcal{R}_{\kappa'[t'/a]}$. By application of the induction hypothesis, $\llbracket t \rrbracket_{\xi, \theta} \in \mathcal{R}_{a :: \kappa \Rightarrow \kappa'}$ and $\llbracket t' \rrbracket_{\xi, \theta} \in \mathcal{R}_\kappa$. Hence, by definition of $\mathcal{R}_{a :: \kappa \Rightarrow \kappa'}$, $R \in \mathcal{R}_{\kappa'} = \mathcal{R}_{\kappa'[t'/a]}$.
- The cases $\text{K-}\rightarrow$, $\text{K-}\lambda$ and K-tv are similar to the $\text{K-}\forall$, $\text{K-}\Lambda$ and K-tt cases.
- In the $\text{K-}\prec$ case, the result follows by direct application on the induction hypothesis, using Lemma 9 (Candidate (in)variance).
- In the $\text{K-}\phi$ case, the result follows by direct application of the induction hypothesis, noting that $\llbracket x:t\{\phi\} \rrbracket_{\xi, \theta} = \llbracket t \rrbracket_{\xi, \theta}$.

□

Lemma 14 ((In)variance of interpretation)

If $S; \Gamma \vdash t_1 \prec t_2$ (resp. $S; \Gamma \vdash t_1 \equiv t_2$), then $\llbracket t_1 \rrbracket_{\xi, \theta} \subseteq \llbracket t_2 \rrbracket_{\xi, \theta}$ (resp. $\llbracket t_1 \rrbracket_{\xi, \theta} = \llbracket t_2 \rrbracket_{\xi, \theta}$) for any assignment $\xi \models \Gamma$.

Proof By induction on $S; \Gamma \vdash t_1 \prec t_2$. □

Lemma 15 (Correctness of interpretation)

If $S; \Gamma; X \vdash^m e : t$, then $e\theta \in \llbracket t \rrbracket_{\xi, \theta}$ for any pair (ξ, θ) adapted to Γ .

Proof By induction on $S; \Gamma; X \vdash^m e : t$. By type correctness, there exists a concrete kind c s.t. $\Gamma \vdash t :: c$. Hence, $\llbracket t \rrbracket_{\xi, \theta} \in \mathcal{R}_c$. If $c \neq P$, then $\mathcal{R}_c = \{\Lambda_e\}$. Thus, $\llbracket t \rrbracket_{\xi, \theta} = \Lambda_e$ and the result easily follows. If $c = P$, we do a case analysis on the last rule used:

- If $S; \Gamma; \cdot \vdash x : \Gamma(x)$ with $\Gamma \vdash \Gamma(x) :: P$, then by **(CR1)**, $\mathbf{SN} \subseteq \llbracket \Gamma(x) \rrbracket_{\xi, \theta}$. Since θ is a type-level, $x\theta = x \in \mathbf{SN} \subseteq \llbracket \Gamma(x) \rrbracket_{\xi, \theta}$.

- If $S; \Gamma; X \vdash^m \lambda x:t. e : x:t \rightarrow t'$ is derived using T-Abs from i) $S; \Gamma \vdash t :: c$, ii) $S; \Gamma, x : t; X, x \vdash^m e : t'$, and iii) $S; \Gamma \vdash x:t \rightarrow t'$, we have to prove that $(\lambda x:t. e\theta) \in \llbracket x \rightarrow t' \rrbracket_{\xi, \theta}(\emptyset)$. Unfolding definition, this amounts to show that $e\theta \in \llbracket t' \rrbracket_{\xi, \theta}$. By inversion of iii), we have that $S; \Gamma \vdash t' :: P$. Hence the result by application of the induction hypothesis on ii).
- If $S; \Gamma; X \vdash^m \Lambda a::\kappa.t : \forall a::\kappa. t$ is derived using T-Tabs from i) $S; \Gamma \vdash \kappa \text{ ok}(b)$, ii) $S; \Gamma, \alpha :: \kappa; X \vdash^m e : t$ and, iii) $S; \Gamma \vdash \forall a::\kappa. t :: P$, we have to prove that $\Lambda a::\kappa. e\theta \in \llbracket \forall a::\kappa. t \rrbracket_{\xi, \theta}$. Let $t_\kappa \in \Lambda_t$ and $S \in \mathcal{R}_\kappa$. We have to prove that $(\Lambda a::\kappa. e\theta) t_\kappa \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^{t_\kappa}}$. By inversion of iii), we have that $S; \Gamma \vdash t :: P$. Hence, $\llbracket t \rrbracket_{\xi_a^S, \theta_a^{t_\kappa}} \in \mathcal{R}_P$. Hence, by condition (CR3), it is sufficient to show that $(e\theta)[t_\kappa/a] = (e\theta_a^{t_\kappa}) \in \llbracket t \rrbracket_{\xi_a^S, \theta_a^{t_\kappa}}$. Since $(\xi_a^S, \theta_a^{t_\kappa})$ is adapted to $\Gamma, a :: \kappa$, we can apply the induction hypothesis on ii) and conclude.
- Assume that $S; \Gamma; X_1, X_2 \vdash^m e v : t[v/x]$ is derived using T-App from i) $!t_1 = x:t' \rightarrow t$, ii) $S; \Gamma; X_1 \vdash^m e : t_1$, iii) $S; \Gamma; X_2 \vdash^m v : t'$, and iv) $S; \Gamma \vdash t[v/x] :: P$. We have to prove that $(e\theta v\theta) \in \llbracket t[v/x] \rrbracket_{\xi, \theta} = \llbracket t \rrbracket_{\xi, \theta}$. By inversion of iv), $S; \Gamma \vdash x:t \rightarrow t' :: P$. Hence, $x::t' \rightarrow t$ cannot be typed under the affine constructor i and $t_1 = x:t' \rightarrow t$. By application of the induction hypothesis on ii) and iii), we have $e\theta \in \llbracket x:t' \rightarrow t \rrbracket_{\xi, \theta}$ and $v\theta \in \llbracket t' \rrbracket_{\xi, \theta}$. Hence, $(e\theta v\theta) \in \llbracket t \rrbracket_{\xi, \theta}$ by definition of $\llbracket x:t' \rightarrow t \rrbracket_{\xi, \theta}$.
- Assume that $S; \Gamma; X \vdash^m e t : t'[t/a]$ is derived using T-TApp from i) $!t_v = \forall a::\kappa. t'$, ii) $S; \Gamma; X \vdash^m e : t_v$, iii) $S; \Gamma \vdash^m t :: \kappa$, and iv) $S; \Gamma \vdash t'[t/a] :: P$. Using a similar reasoning, we obtain that $(e\theta t\theta) \in \llbracket t' \rrbracket_{\xi', \theta'}$ with $\xi' = \xi \{x \mapsto \llbracket t\theta \rrbracket_{\xi, \theta}\}$ and $\theta' = \theta \{x \mapsto t\theta\}$. By candidate substitution, $\llbracket t' \rrbracket_{\xi', \theta'} = \llbracket t[t/a] \rrbracket_{\xi, \theta}$, hence the result.
- The case T-Sub is a direct consequence of the invariance of interpretation by subtyping.
- The cases T-Drop, T-V are done by direct application on the induction hypothesis.
- All other cases implies $c \neq P$.

□

Lemma 16 (Weak strong normalization of P)

If $S; \Gamma; X \vdash^m e : t$ with $S; \Gamma \vdash t :: P$, then $e \in \text{SN}$.

Proof Let $\xi(\Gamma)$ defined by:

$$\begin{aligned}
 \xi(\varepsilon) &= \varepsilon \\
 \xi(\Gamma, x:t) &= \xi(\Gamma) \\
 \xi(\Gamma, a::\kappa) &= \xi(\Gamma)_a^\top \kappa \\
 \xi(\Gamma, v_1 = v_2) &= \xi(\Gamma) \\
 \xi(\Gamma, a = t) &= \xi(\Gamma)_a^{\llbracket t \rrbracket_{\xi(\Gamma), \varepsilon}}
 \end{aligned}$$

Then, $(\varepsilon, \xi(\Gamma))$ is adapted to the environment Γ , and by Lemma 15 (Correctness of interpretation), $e \in \llbracket t \rrbracket_{\xi(\Gamma), \varepsilon}$. By Lemma 13 (Well-formness of interpretation), $\llbracket t \rrbracket_{\xi(\Gamma), \varepsilon} \in \mathcal{R}_P$. By condition (CR1), $\llbracket t \rrbracket_{\xi(\Gamma), \varepsilon} \subseteq \text{SN}$. Hence, $e \in \text{SN}$. □

Using our simulation property, we conclude:

Theorem 3 (Strong normalization)

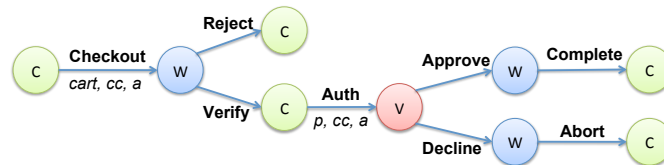
If $S \vdash \mathcal{A} \implies \Gamma; X, \vdash S; \Gamma; X$ wf, and $S; \Gamma; X' \vdash^m e : t$ with $X' \subseteq X$, then (\mathcal{A}, e) must reduce to (\mathcal{A}', v) for some \mathcal{A}' .

5 Security Programming in F*

We describe two security applications written and verified in F*. We show how the design of F* enables compact yet precise specifications that can be verified by typechecking.

5.1 Multi-party sessions

Multi-party sessions (Honda *et al.* 2008; Bhargavan *et al.* 2009; Deniérou & Yoshida 2011) 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.

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. Also related is work by Kiselyov *et al.* (2010), who add type functions to Haskell and show how these can be used to program simple two-party sessions.

5.2 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 ⇒ 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: msg:m → role0 (Send0 m k) → role0 (k msg)
```

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. Recall that 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 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 d 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 d attached to `Send`—client programs calling `send` have to prove $d\ s0\ s1$ for some specific instantiation of d , 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: msg:m → s0:st → s1:st {d s0 s1} → role st (Send m st msg d k) s0 → role st (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 §5.1. 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. Note, for clarity, we omit inferrable type arguments writing, for example, `role` instead of `role msg store`, `Recv` instead of `Recv msg store` etc.

```
type msg = Checkout | Reject | Verify | Auth | Approve | (...)
```

36 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

```

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 verbose, session-specific logical formulas rather than types. Our use of higher-order kinds 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.

5.3 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 to 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 sent a “Declined” message, 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.

5.4 Encoding advanced session constructs in F^*

The 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 Deniérou & 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

```

38 *N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang*

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`.

5.5 Privacy-friendly smart metering

Utility providers are deploying smart meters for billing customers for their use of electricity, gas, etc. These meters provide frequent readings (up to every minute) so that variable rates and billing policies may be applied for each time period, depending on contracts between utilities and customers. However, many customers are reluctant to disclose such detailed readings, as they leak information about their private life: ideally, the utility providers should get the aggregate fee once a month, not the series of individual readings.

Rial & Danezis (2010) develop privacy-friendly zero-knowledge cryptographic protocols for smart metering. We implement and verify their simpler, ‘fast protocol’, which applies to linear policies: for some given series of readings \vec{x} and rates \vec{p} , the monthly fee is the scalar product $z = \vec{x} \cdot \vec{p}$. Skimming over most cryptographic details, the protocol goes as follows: at the end of the month, the customer collects readings \vec{x} from the meter and rates \vec{p} from the utility provider, and it builds cryptographic evidence that he owes z without disclosing any extra information on \vec{x} . The protocol relies on homomorphic commitments: instead of signing the readings \vec{x} , the meter signs blinded commitments \vec{c} to those readings, and also gives the corresponding openings \vec{r} to the customer; the customer computes $z = \vec{x} \cdot \vec{p}$ and $r = \vec{r} \cdot \vec{p}$, then sends z , r , and the signed commitments \vec{c} to the utility provider. For each reading x_i and opening r_i , the commitment is $c_i = g^{x_i} h^{r_i}$ in some large multiplicative group. The opening r_i is uniformly at random, so x_i and c_i are statistically independent and, as long as r_i is secret, the scheme is ‘perfectly hiding’. The utility provider verifies the meter’s signature, computes $c = \prod_{i=1..N} c_i^{p_i}$, and checks $c = g^z h^r$. As long as the customer cannot effectively compute any other opening that would pass this check, the scheme is ‘computationally binding’.

Our F* implementation includes a new library supporting both Pedersen and RSA homomorphic commitments. We rely on the .NET BigInt library for multiplications and modular exponentiations on 2048-bit integers and use ghost refinements on `Commit` to keep track of committed values. We give below functions for building and verifying commitments, and a computational assumption stating that a commitment can be opened to at most one value. In addition, our library supports vectors of commitments and their operations, stating for instance that `Commit x0 r0 c0` and `Commit x1 r1 c1` imply `Commit (x1+x0) (r0+r1) (c0*c1)`.

We finally briefly explain the utility function `verify_payment`, which performs the verification steps explained above: (1) call `verify_meter_signature` to confirm the existence of genuine readings `xs` for the received commitments, line 7; (2) call `scalarExp` in `Commitment`, to compute $\prod_{i=1..N} c_i^{p_i}$ with (ghost) postcondition $\exists xs, x, r. \text{Readings } xs \ \&\& \ \text{ScalarProduct } xs \ ps \ x \ \&\& \ \text{Commit } x \ r \ c$; (3) call `verify` on the result and the received x and r . In conjunction with

the injectivity assumption on line 4, this suffices to prove the postcondition on line 10 that expresses our security goal: if b is true, then indeed $z = \vec{x}.\vec{p}$ for the readings \vec{x} issued by the meter.

```

type Commit:: num ⇒ num ⇒ exp ⇒ P
val commit: x:num → (r:num * c:exp { Commit x r c })
val verify: x:num → r:num → c:exp → b:bool { b=true ⇒ Commit x r c }
assume ∀c x0 x1 r0 r1. Commit x0 r0 c && Commit x1 r1 c ⇒ x0 = x1
(...)
val verify_meter_signature: cs:vec → dsig → b:bool
{ b=true ⇒ (∃(xs:vec),(rs:vec). Readings xs && Commits xs rs cs) }
val verify_payment:
  ps:vec → cs:vec → s: dsig → x: num → r: num → b: bool
  { b=true ⇒ (∃ (xs:vec). Readings xs && ScalarProduct xs ps x) }
let verify_payment ps cs s x r =
  let b = verify_meter_signature cs s in if b = true then
    let c = scalarExp Readings cs ps in verify x r
  else false

```

Backes *et al.* (2008) also verify authentication properties of zero-knowledge protocols, but they extend F7 with special-purpose types that represent zero-knowledge proofs. Instead, our types use the generic higher-order kinds available in F*, but not in F7.

6 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 initially based on the type-preserving compiler for Fine (Swamy *et al.* 2010; 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, similarly to source F*, its security can be verified with the help of Z3.

Checking two forms of refinements in RDCIL. Ghost refinement is a 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.

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

Fig. 7. Code size (in bytes)

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.

6.1 Benchmarks and Measurements

Code size. We compile the Fine benchmarks of Chen *et al.* (2010) with the F* compiler, treating all refinement types as ghost refinements. This way, no proofs are extracted. Figure 7 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.

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.,

Benchmark	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

Fig. 8. Compilation and typechecking times

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. Figure 8 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.

`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 §2.6. `Browser exts` is a suite of 17 browser extensions, verified for authorization and information flow properties. `DynSessions` is the example of §5, 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.

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

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*.

`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 (§5), 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 & Danezis 2010).

7 Further work

We have compared F* to Fine, F7, Aura, Coq, Agda, and discussed other related work in detail throughout this paper. Another language worth discussing is Trellys (Kimmell *et al.* 2012). Like F*, Trellys aims to isolate a safe proof language to express properties about effectful computations. However, the means by which these ends are achieved are rather different. For example, rather than resort to a kind system to isolate a strongly normalizing sub-language, Trellys relies on a judgmental notion of values combined with constructs that allow discriminating on potentially divergent expressions within the proof language. This yields an interesting new proof methodology, although the design of Trellys remains preliminary. (As far as we are aware, the soundness proof of Trellys is as yet incomplete.) Also, Trellys considers only divergence as an effect, whereas F* incorporates many other primitive effects—our stratified presentation combined with the kind system makes it relatively easy to handle these features in the metatheory.

In the remainder of this section, we briefly review further work based on F* in a variety of contexts, complementing the results in this paper.

Self-certification. Strub *et al.* (2012) propose a general technique called *self-certification* that allows a typechecker for a suitably expressive language to be certified for correctness, and illustrate it using F*. Self-certification involved implementing a core typechecker for F* in approximately 5,500 lines of F* code, while using all the conveniences F* provides for the compiler-writer (e.g., partiality, effects, implicit conversions, proof automation, libraries). This core typechecker is given a specification (in F*) strong enough to ensure that it computes valid typing derivations. Running the core typechecker on itself yields a typing derivation for the core typechecker, which is exported to Coq as a type-derivation certificate. By typechecking this derivation (in Coq) and applying the F* metatheory (also mechanized in Coq; see §4), Strub *et al.* formally conclude that the typechecker is correct.

Once certified in this manner, the F^* typechecker is emancipated from Coq, i.e., programs accepted by the core typechecker are guaranteed to be formally well-typed in Coq, without having to run Coq itself.

Monadic F^* . Schlesinger & Swamy (2012) define a monadic dialect of F^* based on a monad of predicate transformers that they call the *Dijkstra state monad*. This monad provides F^* with a customizable type inference algorithm, or, equivalently, a weakest pre-condition calculus. Their methodology involves a liberal use of higher-order logic, but, when specifications are structured in their prescribed style, the resulting higher-order verification conditions can be normalized and encoded in a first-order theory, e.g., in the logic provided by an automated solver like Z3. They use monadic F^* to verify a number of programs, ranging from small classic combinators to web applications and security protocols. We emphasize, however, that the core F^* calculus remains unchanged, justifying the design choices made in this paper. As such, programmers in F^* have several complementary ways of verifying effectful programs, e.g., the linear maps of §2 for state, the session types of §5 for IO, and, also monadic F^* which can be used to precisely model a range of monadic effects including exceptions, state, IO, reactivity, etc.

Verifying JavaScript programs. Swamy *et al.* (2012) apply monadic F^* to the problem of verifying JavaScript programs. Using a new refinement of the type `dyn` (§2.1), they show how JavaScript programs translated to F^* (via a standard translation provided by Guha *et al.* (2010)) can be given precise specifications and verified in a modular manner using monadic F^* 's verification condition (VC) generator combined with its encoding of higher-order VCs in Z3. In evaluating this approach, Swamy *et al.* develop *JSVerify*, a library of JavaScript runtime primitives (e.g., operations to allocate objects, to implement JavaScript's calling convention, etc.), in all some 1,500 lines of fully verified, heavily higher-order and stateful code. In addition, they show how a collection of web-browser extensions authored in JavaScript can be translated to monadic F^* and verified there for a variety of safety properties.

Translating F^* to JavaScript. Many tools allow programmers to develop applications in high-level languages and deploy them in web browsers via compilation to JavaScript. While practical and widely used, these compilers are ad hoc. No guarantee is provided on their correctness for whole programs, nor their security for programs executed within arbitrary JavaScript contexts. Fournet *et al.* (2013b) present a compiler from (a subset of) F^* (including higher-order functions, references and exceptions) down to JavaScript, while preserving all source program properties. The main contribution of their work includes a new applicative bisimulation for F^* (yielding a powerful coinductive technique for proving equivalences on F^* programs), and they show how to use this bisimulation to prove that the translation from F^* to JavaScript is fully abstract. Providing further evidence for the suitability and expressiveness of F^* for practical program verification tasks, Fournet *et al.* use monadic F^* and the mechanically verified *JSVerify* library to facilitate the proof of full abstraction, e.g., they use the types of monadic F^* to show that the translation from F^* to JavaScript is types preserving and to maintain a heap shape invariant. The translation to JavaScript, in conjunction with the type-preserving translation to .NET mentioned in this paper, allows F^* programs to be deployed in a wide variety of settings while obtaining formal assurances of the security of the deployed code.

Relational F^* . The applicative bisimulation of Fournet et al. provides a manual proof technique for program equivalence in F^* . In order to mechanically check proofs of program equivalence, and other *hyperproperties* (Clarkson & Schneider 2010), Barthe *et al.* (2012) present a probabilistic, relational variant of F^* called RF^* . They prove the soundness of this language using a denotational semantics for RF^* , in contrast to the operational formalization used in this paper. Through careful language design, they adapt the monadic F^* typechecker to generate both classic and relational verification conditions, and to automatically discharge their proofs using Z3. Thus, they are able to benefit from the existing features of F^* , including, for example, its abstraction facilities that support modular reasoning about program fragments. They evaluate RF^* experimentally by programming a series of cryptographic constructions and protocols, and by verifying their security properties, ranging from information flow to unlinkability, integrity, and privacy.

A DSL with an authorization logic in F^* . To facilitate the easy construction and deployment of authorization protocols, Jeannin *et al.* (2013) develop $DKAL^*$, a domain specific language that embeds the DKAL authorization logic (Gurevich & Neeman 2008) within F^* . Protocol and policy designers can use $DKAL^*$'s authorization logic for expressing distributed trust relationships, and its small rule-based programming language to describe the message sequence of a protocol. Importantly, many low-level details of the protocol (e.g., marshaling formats, management of state consistency etc.) are left abstract in $DKAL^*$, but sufficient details must be provided in order for the protocol to be executable. Jeannin et al. formalize the semantics of $DKAL^*$, giving it both an operational semantics and a type system. They also present an interpreter for $DKAL^*$, programmed and mechanically verified in F^* for correctness and security.

In summary, in addition to the experimental evaluation in this paper, the aforementioned efforts have used F^* for building and verifying a wide variety of programs, ranging from typecheckers and compilers to distributed programs and cryptographic constructions. As we continue to gain experience with F^* , its core system presented in this paper has remained mostly unchanged. The most significant changes since our original presentation (Swamy *et al.* 2011) have been the revised kind hierarchy, (motivated primarily to streamline our metatheoretic development) and the addition of other effects (such as exceptions, required to embed Javascript). Thus, we conclude that the value-dependent types provided by F^* identifies a sweet spot in the wide design space of dependently typed programming languages—the language is expressive enough for a variety of practical and theoretical programming tasks, incorporating complex language features like effects, while enjoying a formal and relatively simple metatheory.

8 References

Bibliography

- Avijit, Kumar, Datta, Anupam, & Harper, Robert. (2010). Distributed programming with distributed authorization. *Tldi*.
- Backes, Michael, Hritcu, Catalin, & Maffei, Matteo. (2008). Type-checking zero-knowledge. *CCS*.

- Barthe, Gilles, Fournet, Cédric, Gregoire, Benjamin, Strub, Pierre-Yves, Swamy, Nikhil, & Beguelin, Santiago Zanella. (2012). *Probabilistic relational verification for cryptographic implementations*. Tech. rept. MSR-TR-2012-37. MSR.
- Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., & Maffei, S. (2008). Refinement types for secure implementations. *Csf*.
- Bertot, Yves, & Castéran, Pierre. (2004). *Coq'art: Interactive theorem proving and program development*. Springer Verlag.
- Bhargavan, K., Fournet, C., & Gordon, A. D. (2010). Modular verification of security protocol code by typing. *Popl*.
- Bhargavan, Karthikeyan, Corin, Ricardo, Dénielou, Pierre-Malo, Fournet, Cédric, & Leifer, James. (2009). Cryptographic protocol synthesis and verification for multiparty sessions. *Csf*.
- Borgstrom, Johannes, Chen, Juan, & Swamy, Nikhil. 2011 (Jan.). Verifying stateful programs with substructural state and hoare types. *Plpv '11*.
- Cervesato, Iliano, & Pfenning, Frank. (2002). A linear logical framework. *Inf. comput.*, **179**(1).
- Chapin, Peter C., Skalka, Christian, & Wang, X. Sean. (2008). Authorization in trust management: Features and foundations. *Acm comput. surv.*, **40**.
- Chen, Juan, Chugh, Ravi, & Swamy, Nikhil. (2010). Type-preserving compilation of end-to-end verification of security enforcement. *Pldi '10*. ACM.
- Clarkson, Michael, & Schneider, Fred. (2010). Hyperproperties. *JCS*, **18**(6).
- de Moura, Leonardo, & Bjørner, Nikolaj. (2008). Z3: An efficient SMT solver. *Tacas*.
- Dénielou, Pierre-Malo, & Yoshida, Nobuko. (2011). Dynamic multirole session types. *Popl*.
- Felleisen, Matthias, & Hieb, Robert. (1992). The revised report on the syntactic theories of sequential control and state. *Theor. comput. sci.*, **103**(2), 235–271.
- Fournet, C., Gordon, A. D., & Maffei, S. (2007). A type discipline for authorization policies in distributed systems. *Pages 31–45 of: 20th ieee computer security foundations symposium (csf'07)*.
- Fournet, Cédric, Kohlweiss, Markulf, & Strub, Pierre-Yves. (2011). Modular code-based cryptographic verification. *Pages 341–350 of: Acm conference on computer and communications security*.
- Fournet, Cedric, Swamy, Nikhil, Chen, Juan, Dagand, Pierre-Evariste, Strub, Pierre-Yves, & Livshits, Ben. (2013a). Fully abstract compilation to javascript. *In proceedings of the acm symposium on principles of programming languages (popl) (to appear)*.
- Fournet, Cedric, Swamy, Nikhil, Chen, Juan, Dagand, Pierre-Evariste, Strub, Pierre-Yves, & Livshits, Ben. (2013b). Fully abstract compilation to javascript. *In proceedings of the acm symposium on principles of programming languages (popl) (to appear)*.
- Gonthier, Georges, Mahboubi, Assia, & Tassi, Enrico. (2011). Research Report RR-6455.
- Gordon, Andrew D., & Jeffrey, Alan. (2003). Authenticity by typing for security protocols. *Journal of computer security*, **11**(4), 451–520.
- Guha, Arjun, Saftoiu, Claudiu, & Krishnamurthi, Shriram. (2010). The essence of JavaScript. *Ecoop*.
- Guha, Arjun, Fredrikson, Matthew, Livshits, Benjamin, & Swamy, Nikhil. (2011). Verified security for browser extensions. *Ieee symposium on security and privacy (oakland)*.

46 N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan and J. Yang

- Gurevich, Yuri, & Neeman, Itay. (2008). Dkal: Distributed-knowledge authorization language. *Pages 149–162 of: Csf*. IEEE Computer Society.
- Guts, Nataliya, Fournet, Cédric, & Nardelli, Francesco Zappa. (2009). Reliable evidence: Auditability by typing. *Esorics*.
- Honda, Kohei, Yoshida, Nobuko, & Carbone, Marco. (2008). Multiparty asynchronous session types. *Popl*.
- Jeannin, Jean-Baptiste, de Caso, Guido, Chen, Juan, Gurevich, Yuri, Naldurg, Prasad, & Swamy, Nikhil. (2013). Dkal*: Constructing executable specifications of authorization protocols. *In proceedings of the international symposium on engineering secure software and systems*.
- Jia, Limin, & Zdancewic, Steve. (2009). Encoding information flow in aura. *Plas*.
- Jia, Limin, Vaughan, Jeffrey, Mazurak, Karl, Zhao, Jianzhou, Zarko, Luke, Schorr, Joseph, & Zdancewic, Steve. (2008). Aura: A programming language for authorization and audit. *Icfp*.
- Kimmell, Garrin, Stump, Aaron, Eades, III, Harley D., Fu, Peng, Sheard, Tim, Weirich, Stephanie, Casinghino, Chris, Sjöberg, Vilhelm, Collins, Nathan, & Ahn, Ki Yung. (2012). Equational reasoning about programs with general recursion and call-by-value semantics. *Pages 15–26 of: Proceedings of the sixth workshop on programming languages meets program verification*. PLPV '12. New York, NY, USA: ACM.
- Kiselyov, Oleg, Jones, Simon Peyton, & chieh Shan, Chung. (2010). *Fun with type functions*. Unpub.
- Lahiri, Shuvendu K., Qadeer, Shaz, & Walker, David. (2011). Linear maps. PLPV '11. ACM.
- McCarthy, John. (1962). Towards a mathematical science of computation. *Pages 21–28 of: Ifip congress*.
- Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avraham, Govereau, Paul, & Birkedal, Lars. (2008). Ynot: dependent types for imperative programs. *Icfp*.
- Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers Institute of Technology.
- Rial, Alfredo, & Danezis, George. 2010 (nov). *Privacy-friendly smart metering*. Tech. rept. Microsoft Research.
- Schlesinger, Cole, & Swamy, Nikhil. (2012). *Verification condition generation with the dijkstra state monad*. Tech. rept. MSR.
- Sewell, Peter, Zappa Nardelli, Francesco, Owens, Scott, Peskine, Gilles, Ridge, Thomas, Sarkar, Susmit, & Strnisa, Rok. (2010). Ott: Effective tool support for the working semanticist. *Jfp*, **20**(1).
- Sozeau, Matthieu. (2007). Subset coercions in coq. *Types*.
- Strub, Pierre-Yves, Swamy, Nikhil, Fournet, Cedric, & Chen, Juan. (2012). Self-certification: Bootstrapping certified typecheckers in f* with coq. *In proceedings of the acm symposium on principles of programming languages (popl)*.
- Swamy, Nikhil, Corcoran, Brian J., & Hicks, Michael. (2008). Fable: A language for enforcing user-defined security policies. *S&P*.
- Swamy, Nikhil, Chen, Juan, & Chugh, Ravi. (2010). Enforcing stateful authorization and information flow policies in Fine. *Esop*.

- Swamy, Nikhil, Chen, Juan, Fournet, Cédric, Strub, Pierre-Yves, Bhargavan, Karthikeyan, & Yang, Jean. (2011). Secure distributed programming with value-dependent types. *Pages 266–278 of: Chakravarty, Manuel M. T., Hu, Zhenjiang, & Danvy, Olivier (eds), Icfp*. ACM.
- Swamy, Nikhil, Weinberger, Joel, Schlesinger, Cole, Chen, Juan, & Livshits, Ben. (2012). *Monadic refinement types for verifying javascript programs*. Tech. rept. MSR-TR-2012-37. MSR.
- The Coq Development Team. (2010). *Chapter 4: Calculus of Inductive Constructions*. Tech. rept.
- Vaughan, Jeffrey A., Jia, Limin, Mazurak, Karl, & Zdancewic, Steve. (2008). Evidence-based audit. *Csf*.
- Volpano, Dennis, Smith, Geoffrey, & Irvine, Cynthia. (1996). A sound type system for secure flow analysis. *Journal of computer security*, **4**(3), 167–187.

