Secure Distributed Programming with Value-Dependent Types

Nikhil Swamy Juan Chen Cédric Fournet Pierre-Yves Strub Karthikeyan Bhargavan Jean Yang
Microsoft Research Microsoft Research Microsoft Research MSR-INRIA INRIA MIT

Microsoft Research Technical Report
MSR-TR-2011-037

This is paper is a draft. It is an extended version of a shorter paper of the same title. It differs from the short paper in the following regards:

(1) A new section 4.5 describing an application of F* to programming privacy-friendly smart meters using a new zero-knowledge protocol.

(2) Appendix A includes the complete definitions of F* and a manual proof of its soundness. The statement of the soundness result in the main body of the paper is updated (with more details) to match the statement in Appendix A. Note, the manual soundness proof is included simply as a matter of record. The reference proof of F*'s soundness is a mechanized in Coq and is available at http://research.microsoft.com/apps/pubs/default.aspx?id=147425.

(3) Appendix B formalizes a typed embedding of RCF in F*.

We expect to make further revisions to this paper to include an improved proof strategy for the normalization of F*'s P-fragment in a forthcoming Appendix C. As opposed to our older proof strategy that translated P terms to CiC, we plan now to translate to an intermediate calculus, dCiC, and prove dCiC to be normalizing.
Distributed applications are difficult to program reliably and securely. Dependently typed functional languages promise to prevent broad classes of errors and vulnerabilities, and to enable program verification to proceed side-by-side with development. However, as recursion, effects, and rich libraries are added, using types to reason about programs, specifications, and proofs becomes challenging.

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

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

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

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

Although these languages are successful in many aspects, for large-scale distributed programming, we desire languages that (1) feature general programming constructs like effects and recursion; (2) support various styles of proofs and evidence, ranging from cryptographic signatures to logical proof terms; (3) produce proofs that can be efficiently communicated between agents in the system.

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

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

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

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

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

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

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

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

Refinements on affine state. Prior work has shown the usefulness of affine types in reasoning about programs that use mutable state (Borgstrom et al. 2011; Lahirii et al. 2011). Relying on its kind system, F⋆ freely allows the use of affine values within its specifications and proofs. In particular, recursion is available explicitly in $F_*$ terms that may be effectful and divergent, and restriction to affine values is available at most once elsewhere in the code. In §4.4, we exploit this feature extensively in implementing a new, flexible approach (Denielou and Yoshida 2011) to enforce protocols on multi-party session types.

Prior systems that integrate substructural and dependent types (e.g., Fine and Linear LF by Cervesato and Pfenning 2002) disallow refinements to speak directly about affine values, and have to rely instead on various encodings to work around this limitation, which is unsuitable for source programming.

Automation and logic parametricity. Proof automation is critical for developing large-scale programs. F⋆ is carefully designed to be parametric in the logic used to describe programming properties and their proofs. §2.5 shows examples with a simple modal authorization logic and an ad hoc logic for database provenance. Logic parametricity enables working with custom authorization logics and, importantly, makes it easy to integrate F⋆ with SMT solvers for logics extended with specific theories. Thus, program verification in F⋆ benefits from significant automation—our implementation uses the Z3 SMT solver (de Moura and Bjørner 2008) and scales up to large programs and specifications. Languages like Aura, PCML5, Coq, and Agda commit to a specific logic, limiting their flexibility. This limitation is significant since diverse logics are used and even designed when reasoning about security policies and properties—see Chapin et al. (2008) for a recent survey.
Metatheory. We establish several properties of F*. First, we prove the soundness of F* in terms of progress and preservation. From this, we derive a safety property for ghost redefinitions called global deducibility. Next, we show that the P-fragment of F* is consistent by giving a typed embedding of this fragment into CiC and proving that the translation is a simulation. We also give a typed embedding of a core subset of RCF (the core calculus of F7) into F*. The subject reduction result (modulo the admission of a few standard lemma) has been mechanized in Coq—we plan to continue to develop our Coq formalization to include the other results.

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

Compiler implementation. We have implemented a compiler for F* based on our prior work on a compiler for Fine. The F* compiler accepts both Fine and F7 programs as input. To validate this feature, we typecheck and compile a large F7 library implementing symbolic cryptography (Bhargavan et al. 2010). Our compiler translates F* to RCoC, a dependently typed dialect of .NET bytecode. This translation is considerably more efficient than the one we used for Fine. Due to the use of ghost refinements and the availability of polymorphic kinds, bytecode emitted by the F* compiler is an order of magnitude (in some cases 45x) smaller than the bytecode emitted by the Fine compiler.

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

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

2. F* by example

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

We organize our presentation around the new kind system of F*.

We start with simple programs that use P-kind and F*’s 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 data-provenance properties.

2.1 Concrete refinement types and total proof terms

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

```
val tail : ∀α : α. {l : list α} ⇒ Mem α x l → Mem α x (l \<\> Mem α x l)
```

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

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

Concrete refinements and constructive proofs. Also called subset types or Σ-types (Sozeau 2007), concrete refinements in F* are desugared into dependent pairs (as is usual in, say, Coq). The type below shows this desugaring, where the type Mem ⋆ x t denotes a dependent pair, where x names the t-typed first component and is bound in t, the type of the second component.

```
val tail : {l : list α} ⇒ Mem ⋆ x (l \<\> Mem α x l) → Mem ⋆ x (l \<\> Mem α x l)
```

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

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

Using P-kind, we define Mem, an inductive type that axiomatizes list membership in constructive style. Its kind is of the form α:α. k = k’, where α binds a k-kind formal type parameter in the kind k’ of the constructed type. Type constructors can also be applied to values; such constructors have kinds of the form xk = k where x names the formal argument, a value of type t, and is in scope in k. Below, the kind of Mem says that it is a dependent type constructor that constructs a type of kind P from a type α, a value x of type α, and a value l of type list α. (When xα, we write Mem x instead of Mem x1.)

```
type Mem : {l : list α} ⇒ Mem ⋆ x l = Mem ⋆ x l
```

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

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

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

```plaintext
type pf :: P -> P =
| Exec_middle: pf (Or α (Not α)) |
| Not_elim: pf (Not α) -> pf α -> pf β |
| Not_in-nil: xα -> pf (Not (Mem x Nil)) |
| ... (\ many constructors omitted ) |
| Unit pf: α -> pf α |
| Bind_pf: pf a -> (a -> pf β) -> pf β |
```

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

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

```plaintext
let tail = function [] -> [] | hd::tl -> tl
```

2.2 Ghost refinements for lightweight specifications

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

We illustrate the use of ghost refinements for the problem of verifying clients of libraries, where the libraries are authored separately and are non-modifiable. In recent work, Guha et al. (2011) consider programming secure web browser extensions using F*. For this scenario, we use ghost refinement types to specify pre- and post-conditions on the interface provided by the browser, and use the specifications to verify access control properties of extensions.

The listing below illustrates this approach on a tiny program—§5 reports new results for compiling a suite of 17 such extensions in a type-preserving style to .NET bytecode.

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

```plaintext
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 -> string[EltTagName e t]
7 ( ≈ Extension policy )
8 assume Y e. EltTagName e "head" => CanRead e
9 ( ≈ Extension code (untrustedly )
10 let read e = if tagName e = "head" then innerText e else
```

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

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

Ghost refinements and erasure.

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

Semantics of ghost refinement derivability.

For every value \( v: t \) that inhabits \( x: Φ \) our type system ensures that the formula \( Φ[t/v] \) 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 \( Φ \). We think of \( Φ \) as a monotonically increasing log of events and formulas that are assumed during evaluation of the program. Formally, a call to tagName e reduces to t and has the effect of adding the formula EltTagName e t to the log. For values given ghost refinement types, there may be no concrete proof at run time to witness the derivability of the refinement formula. Indeed, when working with libraries like the DOM, explicit proof terms witnessing DOM invariants seem both infeasible and undesirable (as they may be very voluminous); ghost refinements fit the bill nicely.

Proof-relevance and P vs. E-kind.

The distinction between \( P \) and E-kinds in F* may, at first, seem reminiscent of the distinction between Type and Prop in a system like Coq. The proof terms for concrete refinements in Coq are often from the Prop universe, indicating that they are computationally irrelevant (and so can be erased during code extraction). In contrast, concrete refinements in F* are accompanied by P-kinded proof terms, which are computationally relevant. We view proofs as useful runtime entities that carry important information. We choose to make proofs explicit and useful—§2.5 demonstrates a novel way of using concrete proof terms to construct precise provenance trails in a curated database. As such P-kind is closer to Coq’s Type. Indeed, our embedding
of F*’s P fragment in CiC translates P-kindred 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-kind 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-kind types may be uninhabited. Instead, the log-based semantics of ghost refinements makes trust assumptions in external code formal and explicit, and allows the definition of security properties for code that are robust even when code is composed with arbitrary attacker code. For example, Guha et al. (2011) used the log-based semantics to prove a robust safety property that ensures that verified extensions are authorization-safe even when composed with arbitrary untrusted JavaScript on a web page.

2.3 Ghost refinements and indexed types for cryptography

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

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

```plaintext
module Crypto

type dsig = bytes (* type of digital signatures *)

type prin (* name of a principal *)

val rsa_sign : ∀a:α; β:α ⇒ E. p:prin → sk p β → x:α {x β} → dsig

val rsa_verify : ∀a:α; β:α ⇒ E.

  p:prin → pk p β → x:α → dsig → r:bool {rtrue} ⇒ Says p (β x)

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

A private key of type sk Alice αPred belongs to the principal Alice:prin, who can use it to sign values m of type α that satisfy Pred m. The function rsa_sign allows clients to construct a dsig value by signing a message x:α, where the ghost refinement guarantees that the formula β x is derivable when rsa_sign is called. Public keys are complementary: rsa_verify verifies a signature using the public key pk p β and, if it succeeds, the caller knows that Says p (β x) is derivable.

As in F7, an abstract implementation of the Crypto library can be verified against the specification shown above, and can be proved correct with respect to a Dolev-Yao adversary. However, in F7, types cannot be parameterized by predicates, so predicate parameters are instead simulated through a level of indirection. Instead of the F* type sk p uPred, private keys in F7 are given a type of the form sk p uusage, and the predicate Pred is replaced by a global predicate SignSays indexed by p and usage. Verification relies on a programming convention that each key usage must be unambiguously defined by recording an assumption of the form yp,usage:∀. SignSays p usage v ⇔ Pred v. This convention is not enforced automatically in F7, and hence this style can lead to logical inconsistencies. In contrast, F* types are more concise, and require fewer dynamic assumptions and no programming discipline beyond typing.

2.4 Ghost refinements and affine-indexed types for state

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

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

Linear maps are a data structure that equips a Floyd-Hoare logic (using a classical assertion logic) with a form of local reasoning in the style of separation logic. Instead of modeling the heap of a program as a single monolithic map H:map α β from α-kindred locations to β-kindred values, the linear maps methodology advocates partitioning the heap H into several fragments H1, . . . , Hn where the Hi have disjoint domains. Each Hi is a linear map of type lin α β, and the disjoint domain condition ensures that modifications to Hi leave all the other Hi unmodified. This allows to formulate a kind of frame rule for programs that use linear maps. Since the assertion logic remains classical, linear map programs can be automatically verified using classical provers and SMT solvers.

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

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

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

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

val write : x: α → γ : β → m1: lin a β {inDomain x m1}
→ (m2: lin a β {Update m1 x y m2})

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

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

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

2.5 Selective erasure using concrete and ghost refinements

The preceding discussion of ghost refinements may lead the reader to believe that they are always to be preferred to refinements with concrete proof terms. This section illustrates that concrete proof terms are useful too, particularly when one is allowed to compute over these terms, to store them, and to communicate them over the network. The example discussed here is an excerpt from a larger program that models a database of scientific experiments, where each record contains a proof term indicating the provenance of the experiment and its “validity”, according to some custom notion of validity. The full example brings together several elements, including that the returned experiment is indeed valid.

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

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

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

(∗ Erasing outbound proofs ∗)

let readExp xid : option (exp partial) =
  match lookup xid with
  | Some (xid, r, pf) → Some (xid, r, Prune xid)
  | None → None

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

Finally, on requests to insert new records in the database, we can reconstruct proofs. The function expand below traverses the structure of a proof tree, and expands Prune nodes by looking them
up in the database. The database maintains an invariant that each record in the database has a full profile 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) : option (proof full a) = if c = full then Some pf else match pf with
   | Prune r xid → lookupProof r xid
   | AndElim1 c1 pf → (match expand c1 pf with
                     | Some pf' → Some (AndElim1 full pf')
                     | _ → None)
   ...

Inserting a new record in the DB

let insertExp (r:expsetup) (c:bool) (pf: proof c (Valid r)) =
   match (expand c pf) with
   | Some pf' → Some (insert (None, r, pf'))
   | _ → None

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

3. Formalizing F*

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

3.1 Syntax

The syntax of F* is shown below. Values include variables, lambda abstractions over values and types, and fully applied n-ary data constructors. The value $v'$ is a technical device used to prove the soundness of affine typing—$\ell$ is an identifier drawn from a class of names distinct from term and type names. We use the notation $\bar{\pi}$ to stand for a finite sequence of elements $a_1,\ldots,a_n$, for arbitrary $n$: $\bar{\pi}$ is a sequence $(\pi_1,\ldots,\pi_n)$. We adopt a (partially) let-normalized view of the expression language $e$, in particular requiring function arguments to stand always be values—this is convenient when using value-dependent types, since it ensures that expressions never escape into the level of types. The only other non-standard expression form is $\text{assume } \phi$, which has the effect of adding a formula to the log and is explained in §3.3.

Types are ranged over by meta-variables $\ell$ and $\phi$—we use $\phi$ for types that stand for logical formulas. Types include variables $a$, constants $T$, dependent functions ranging over values whose domain may be values $(x:T \rightarrow t)$ or types $(\forall a:\kappa.T : \ell)$. We use the notation $\eta$ to stand for a finite sequence of elements $a_1,\ldots,a_n$, for arbitrary $n$: $(\eta_1,\ldots,\eta_n)$ is a sequence $(\eta_1,\ldots,\eta_n)$. We adopt a (partially) let-normalized view of the expression language $e$, in particular requiring function arguments to stand always be values—this is convenient when using value-dependent types, since it ensures that expressions never escape into the level of types. The only other non-standard expression form is $\text{assume } \phi$, which has the effect of adding a formula to the log and is explained in §3.3.

Kinds $\kappa$ include the four base kinds $\star$, $P$, $A$, and $E$—we distinguish the first three of these as concrete kinds, since they classify inhabitable types. As at the type level, we have kinds for dependent function spaces whose range are types and whose domain may be either values $(x:T \Rightarrow \kappa)$ or types $(\forall a:\kappa \Rightarrow \kappa')$. 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.

Syntax: kinds, types, expressions

$$
\begin{array}{llllll}
\kappa & ::= & P \star & E | & A & \text{kinds} \\
\phi, \ell & ::= & \alpha | & T | & x:T \rightarrow t & | & \forall \alpha: \kappa.T : \ell & \text{types} \\
\phi, \ell & ::= & \lambda x.t & | & \text{match } v \text{ with } D \alpha x & \text{expressions} \\
\phi & ::= & \text{assume } \phi & | & \text{ref } v : \ell & \text{values} \\
\end{array}
$$

Kinds $\kappa$ include the four base kinds $\star$, $P$, $A$, and $E$—we distinguish the first three of these as concrete kinds, since they classify inhabitable types. As at the type level, we have kinds for dependent function spaces whose range are types and whose domain may be either values $(x:T \Rightarrow \kappa)$ or types $(\forall a:\kappa \Rightarrow \kappa')$. 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 $\Sigma$ are finite lists of inductively defined types. Each inductive definition $T: \kappa \Rightarrow \Sigma$ introduces a type constructor $T$ of kind $\kappa$ and all its constructors $D_1, D_2,\ldots,D_n$. For simplicity, we do not include mutually inductive types, although these are supported by our implementation and their addition to the formalism poses no significant difficulties. We do not need a fixpoint form in the expression language since inductive types allow us to encode recursive functions. To show that terms given $P$-kinded types are strongly normalizing, a well-formedness condition on signatures imposes a positivity constraint on inductive definitions for $P$-kinded types. An additional constraint on signatures is that they must contain a declaration $\text{unit} : \Sigma [\{() : \text{unit}\}]$ for the unit type and its one value $()$. Finally, typing environments $\Gamma$ track in-scope value variables $x$ with type $T$, type variables $\alpha$ with kind $\kappa$, and equivalences between values ($v_1 = v_2$) and types ($t_1 = t_2$) introduced when checking $\text{match expressions}$. We discuss the log $\alpha|\ell$ in conjunction with the dynamic semantics in §3.3. Briefly, $\alpha|\ell$ maintains a set of facts introduced by the dynamic assumption of ghost refinements, and also a set of names $\ell$ used to track affine values.

3.2 The $\mathbb{F}^*$ type system

Figure 1 shows key rules from each judgment in the type system. Well-formedness of kinds. The judgment $\Sigma, \Gamma \vdash \kappa : b$ states that $\kappa$ is well-formed and produces types of base kind $b$ (when $b = B$) or type constructors for $b$-kinded types. The rule (OK-TK) shows a key enhancement of $\mathbb{F}^*$ over prior languages, e.g., Fine or Linear LF. Types can be constructed from affine values ($b_1 = A$), so long

Draft – September 17, 2011

8
as the type constructed is purely specificalional \((b_2 = E)\). As illustrated in \(\frac{2.4}{\&} 4\), this improves the expressiveness of affine typing significantly, because the rules parameterized by affine types are themselves affine. Although our formalism allows higher-kinds like \(\times \Rightarrow \ast\), such kinds cannot be compiled to the type system of the .NET bytecode language and are currently rejected by our compiler. However, re-targeting \(\bowtie\) to a platform with a more flexible typing discipline would lift this restriction.

**Kinding for value-indexed and affine types.** The judgment \(\Sigma \vdash t : \kappa\) states that type \(t\) has kind \(\kappa\). The rules shown here reveal a few subtle aspects of the type system, starting with affine typing and affine-value indexing. The rule \((K-A)\) shows how the modal operator coerces the kind of a type \((K-TV)\) allows a type function \(t\) to be applied to a value \(v\); these type functions are introduced either (K-Fun) or as type constructors \(T\) in the signature.

Next, recall that we wish to allow affine values to be freely used at the type level, since specifications should not consume affine resources. For this reason, unlike prior languages, the values passed to type functions may use affine assumptions in the context \(\Gamma\) indicated on the turnstile. When the mode is \(\epsilon\) (indicating that the term being typed occurs at the type-level, effectively as \(\kappa\)) with \(\kappa\) indicated on the turnstile. When the mode is \(\epsilon\), the term being typed occurs during the type-level, effectively as \(\kappa\).
of \((K-\phi)\), i.e., inhabitable types. \((K-Arr)\) handles dependent function arrows, which (as seen in §2.1) can be used to represent both quantified formulas in the logic as well as term-level function abstractions. A function arrow is \(P\)-kinded if its range type is \(P\)-kinded. Finally, \((K-\text{Sub})\) uses a sub-kinding judgment in its premise. Sub-kinding in \(F^*\) is defined by the judgment \(S; \Gamma \vdash_\kappa \phi \iff \kappa\). Sub-kinding is based on the relation \(P; \ast \kappa; \ast \kappa\) among base kinds, with \(A \kappa \iff A\). Only these base relations are lifted into the structure of arrow kinds, with co- and contravariance as usual. \((K-\text{Sub})\) includes premises to ensure that sub-kinding preserves well-formedness of kinds. We conjecture that these premises can be eliminated in favor of lemmas establishing that sub-kinding never introduces ill-formed kinds. We have yet to prove it, so we include these premises to facilitate our formal proof of well-formedness of kinds produced by derivations.

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

**Affine typing in two modes.** \((T-XA)\) is typical of affine typing systems. To use an affine assumption \(x\), we require \(x\) to be present in the affine environment \(X\). In addition, \((T-X)\) provides two exceptions to \((T-XA)\): first, as is standard, we can use non-affine assumptions without requiring them to be present in \(X\). Second, when the mode is \(\ast\), 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. \((T-WkN)\) provides weakening for the affine context. Finally, \((T-Abs)\) is a standard rule with one subtlety that the introduced function type is tagged with the affine modality (using \(Q(X, \cdot \cdot \cdot \rightarrow \cdot \cdot \cdot)\)) if the function closure captures an affine assumption. \((T-\ell)\) is also related to affine typing, but, values \(\ell\) are used solely to state that type soundness result—we discuss this rule in conjunction with the dynamic semantics.

**Ghost refinements and subtyping.** The rules \((T-As)\), \((T-V)\), and the subtyping judgments introduce ghost refinement types. \((T-As)\) introduces a unit refined with the assumed formula \(\phi\)—no logical evidence is produced for \(\phi\). To justify this rule, the dynamic semantics of this expression adds a formula to the dynamic log \(\ell\). \((T-V)\) allows value \(\ell\) to be refined with the formula \(\phi\) when \(\phi\) is derivable: \(S; \Gamma \vdash \phi\). The context \(\Gamma\) includes bindings \(x : t \iff \nu\) which allow the derivability relation to use information about \(\nu\); however, for kind-correctness, we require the kind of the introduced formula \(\phi\) to not rely on the introduced equality. Ghost refinements have for kind-correctness, we require the kinding of the introduced formula \(\phi\) to be closed under substitution of free variables in \(\Gamma\); be allowed to use structural rules such as weakening, duplication and permutation; and be insensitive to the presence of derivable equality assumptions in the context \((S; \Gamma, \nu \vdash \phi \iff \Gamma \vdash \phi)\). Pragmatically, we often plug in a decision procedure for first order logic with additional theories, as implemented by the Z3 theorem prover—the ability to use structural rules (e.g., weakening) in the logic, enabled by our restrictions on the use of affine indices in types, makes it easy to support automation. Formally, we also exploit logic parametricity to provide an embedding of \(F^*\)’s formal core, RCF, into \(F^*\), plugging into \((T-V)\) RCF’s entailment relation (which, unlike \(F^*\), includes the basic first-order connectives and equality, each satisfying their usual introduction and elimination forms).

**Consistency of the \(P\)-universe** requires placing restrictions in various parts of the type system. For an expression \(\ell\) to reside in \(P\), \(\epsilon\) must be free of non-\(P\) expressions, since those may diverge; the last premise of \((T-Let)\) enforces this property. We disallow discriminating on values that reside in \(\ast\) or \(A\) when constructing propositions in the branches of a match—the last premise of \((T-Match)\) shows this. We also have constraints on the well-formedness of inductive type \(T\) (recall that we use \(\langle \cdot; \cdot \rangle\) to mean \(\langle \cdot_1 : \cdot_2, \ldots, \cdot_{n-1} : \cdot_{n-1} \rangle\)). In \((WFS-D)\), the first premise ensures that the kind \(\kappa\) of \(T\) is well-formed; the second and third premises ensure that the arguments of \(D\) are well-formed; the fourth premise ensures that the constructed type matches the kind expected for \(T\); and the final premise imposes two key restrictions: (1) constructors with affine arguments must construct affine results—this is unrelated to totality of \(P\)-functions; (2) a positivity constraint on inductive \(P\)-kinded types. We use a relatively simple version of positivity, excluding the constructed type \(T\) in negative position in any argument of \(D\).

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

### 3.3 Dynamic semantics: logical effects and affine names

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

\[
\text{E-Log} \quad (\text{\( \alpha \)'s; assume} \ \phi) \rightarrow_{S} (\text{\( \alpha \)'s; \( \phi \); (}))
\]

Foreshadowing our safety condition for ghost refinements (Corollary 1), we show that, when a well-typed program \( \vdash x t (\phi) \) reduces to a value, that is \( (\text{\( \alpha \)'s} \rightarrow_{S} \epsilon) \), its refinement formula \( \phi \) is derivable from the signature and all the accumulated logical effects: \( S; \text{Facts(\( \alpha \)'s)} \models \phi[v/x] \). This is in contrast to our soundness result for proof terms, expressions that reside in the S-efects:

In a distributed program, the log is an idealized global view of the logical state of all participants. Ghost refinements accompanied by cryptographic evidence (in the form of digital signatures) enable speaking about this distributed state.

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

\[
\begin{align*}
S; \text{Facts(\( \alpha \)'s)} \vdash t : A & \quad \text{fresh} \ \ell \quad (S; \text{Facts(\( \alpha \)'s)}; \text{LiveNames(\( \alpha \)'s)} \vdash v : t) \\
\text{E-New} & \quad (S; \text{Facts(\( \alpha \)'s)}; \text{LiveNames(\( \alpha \)'s)} \vdash (\text{\( \alpha \)'s}; v) \rightarrow_{S} (\text{\( \alpha \)'s}; \ell; v')) \\
\text{E-Kill} & \quad (S; \ell; \text{\( \alpha \)'s}; E[v']) \rightarrow_{S} (S; \ell; \text{\( \alpha \)'s}; E[v])
\end{align*}
\]

### 3.4 Metatheory of \( F^{\star} \)

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

**Theorem 1** (Type soundness). For all \( S \), \( \text{\( \alpha \)'s} \), \( \Gamma \), \( X \), \( X' \), \( e \), and \( t \) such that

1. \( S \vdash \text{\( \alpha \)'s} \Rightarrow \Gamma; \chi \),
2. \( \vdash S; \Gamma; X \ \text{wf} \), and
3. \( \vdash S; \Gamma; X' \vdash e : t \) where \( X' \subseteq X \),

then either \( e \) is a value, or there exist \( \text{\( \alpha \)'s}, \text{\( e \)'s}, \Gamma', \) and \( X'' \) such that

1. \( S \vdash \text{\( \alpha \)'s} \Rightarrow \Gamma'; X'' \),
2. \( X'' = (X' \setminus \text{DeadNames(\( \alpha \)'s)}) \cup \text{LiveNames(\( \alpha \)'s \setminus \text{\( \alpha \)'s})} \),
3. \( \exists \text{\( \alpha \)'s}, \Gamma' \vdash \Gamma, \Gamma' \), and
4. \( \vdash S; \Gamma; X'' \vdash e : t \).

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

**Corollary 1** (Safety for ghost refinements). For all \( S \), \( \text{\( \alpha \)'s} \), \( \Gamma \), \( \phi \), \( t \), \( \text{\( \alpha \)'s}, \Gamma', \) and \( v \) such that \( S \vdash \text{\( \alpha \)'s} \Rightarrow \Gamma; X \), \( S; \Gamma; X \vdash e : x t(\phi) \), and \( \text{\( \alpha \)'s}; e \rightarrow_{S} (\text{\( \alpha \)'s}; v) \), there exists \( \Gamma' \) such that \( S \vdash \text{\( \alpha \)'s} \Rightarrow \Gamma'; \ldots \) and \( S; \Gamma' \models \phi[v/x] \).

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

In the statement of the theorem below, \( (S; \Gamma; \vdash e : t \Rightarrow \tau) \) is the translation that marks non-\( \text{\( \lambda \)} \)-lambdas in \( F^{*} \)'s term \( e \) and translates them to a CiC term \( \tau \). (\( \Rightarrow \)) is the reduction of \( \text{\( \lambda \)}\)-terms; and \( (\neg\neg \rho, \delta) \) is term conversion in CiC.

**Theorem 3** (Well-typed translation of RCF). Given \( E \vdash [A] = ([\alpha'e], [E] = S; \Gamma, \) and \( E \vdash \phi \), we have \( S; \Gamma, \vdash e : t \) where \( E \vdash [E] = t \). Additionally \( A \rightarrow \alpha' \text{ if and only if } ([\alpha'e] \rightarrow \phi \rightarrow ([\alpha'e']) \) and \( E \vdash [A'] = ([\alpha'e'], \).

### 3.5 Coq formalization

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

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

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

4. Secure multi-party sessions in F*

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

4.1 Multi-party sessions

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

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

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

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

4.2 A session API in F*

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

To begin with, we ignore the values (cart, cc, a, p) passed in the session and aim to control the sequence of messages a session participant can send and receive. Using affine types, we can define a type for a role process, type `role0:E → A`, where the parameter of the role is a type describing an automaton. The types used to define these automata are purely specification—thay 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>role0</code></td>
<td>Represents a role process</td>
</tr>
<tr>
<td><code>Send0</code></td>
<td>Sends a message to a role process</td>
</tr>
<tr>
<td><code>Recv</code></td>
<td>Receives a message from a role process</td>
</tr>
</tbody>
</table>

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 k, then an integer y greater than k, and then concludes. (In concrete syntax, we write `fun` `t` for a type-level function `A x . t`, `fun` `_` for `t` ignores its type argument.)

```haskell
role0 (Send0 int (fun (c 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 analogous 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 parameterized by a store value (of type `st`); automata types (`Send`, `Recv`) are indexed by binary predicates δ on `st` values that define the allowed changes to the store during the next step. The function `Send` allows a client to send a message `m` and update the store from `s0` to `s1` given that the current role process is a `Send` and that the stores satisfy the predicate δ attached to `Send`—client programs calling `Send` have to prove `δ s0 s1` for some specific instantiation of δ, and our type checker uses Z3 to assist with the proof of such ghost refinement properties.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Send</code></td>
<td>Sends a message to a role process</td>
</tr>
<tr>
<td><code>Recv</code></td>
<td>Receives a message from a role process</td>
</tr>
</tbody>
</table>
```

The code below shows how we can use this session API to model the website role (w) in the example session of §4.1. The type `msg` defines the set of messages and the type `store` is the type of the distributed store (including, for 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.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msg</code></td>
<td>Defines the set of messages</td>
</tr>
<tr>
<td><code>store</code></td>
<td>Defines the distributed store</td>
</tr>
</tbody>
</table>
```

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.
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 \texttt{role proc}_w \texttt{init store}_w$. It may then call the \texttt{receive} function (the counterpart of \texttt{send}, not shown here) to receive a \texttt{Checkout} message but cannot call \texttt{send}; subsequently, it may call \texttt{send} with either a \texttt{Reject} or a \texttt{Verify} message, but not both.

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

### 4.3 Custom cryptographic protocols for session consistency

Distributed applications typically run in an untrusted environment, where the network and one or more of the session participants may be under the control of malicious adversaries. In this scenario, cryptographic mechanisms, such as digital signatures, can be used 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 $v$ Declined, to prevent a malicious $w$ from double-crossing $c$.

Bhargavan et al. (2009) show how to systematically use cryptographic evidence as proof of session compliance. They compile multi-party sessions to efficient custom cryptographic protocols that exchange and check a minimal number of digital signatures to ensure global session consistency. Their compiled protocols use session types and cryptography in the style of \F: 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 \S2.3 and the sessions API shown above. In our example session, the \texttt{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 \texttt{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 \texttt{Decline} message and not an \texttt{Approve}. The resulting type for the \texttt{recv Abort} function (a specialization of the generic receive function in our API) is as follows:

```f
let recv Abort:
  st c: store \Rightarrow role (RecvCompleteOrAbort st c) \Rightarrow (st c: store = role Done st c)\{ Unchanged st c st c' \land Says st c w (st w. Aborted st c w st w' \land Unchanged st w st c') \land Says st c v (st v. Declined st c v st v' \land 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 \texttt{Auth}. The function returns an (unchanged) store $st c'$ and a new (completed) role handle. The $E$-kinded predicates \texttt{Aborted} and \texttt{Declined} represent the session states at the other roles. For example, \texttt{Declined p st} means that the principal $p$, playing role $v$ previously sent a \texttt{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 $\texttt{Aborted st c w st w}$ and $\texttt{Declined st c v st v}$ 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 \texttt{Abort} unless $v$ sent a \texttt{Decline}. Note that the post-condition is a ghost refinement that is proved here using a combination of cryptographic evidence and \F typechecking.

### 4.4 Encoding advanced session constructs in \F

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

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

The \texttt{Fork} automation (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 \texttt{Join} automation enables the child role to transfer control back to the parent; and \texttt{Await} represents a parent role process waiting for its children to complete.

We illustrate \texttt{Fork} and its use in an application below, where we elide the store for simplicity (and so use \texttt{role 0} instead of \texttt{role}).

```f
let go ps : role 0 Done =
  let client = startClient ps in
  let client, children = fork ps client in
  let children =
    map (fun (q, child) ->
      let child = send 0 Request child in
      let Response, child = receive 0 child in
      (q, child)) children
  join ps client children
```

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

## 5. Implementation and Measurement

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

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

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

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

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

### 5.1 Benchmarks and Measurements

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

<table>
<thead>
<tr>
<th>Bench.</th>
<th>LOC</th>
<th>SC</th>
<th>Trans</th>
<th>TC</th>
<th>Sq</th>
<th>TQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authac</td>
<td>39</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IFlow</td>
<td>119</td>
<td>0.8</td>
<td>0.4</td>
<td>0.5</td>
<td>25</td>
<td>18</td>
</tr>
<tr>
<td>Automation</td>
<td>117</td>
<td>0.3</td>
<td>0.2</td>
<td>0.3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>HealthWeb</td>
<td>130</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>Lookout</td>
<td>502</td>
<td>2.4</td>
<td>2.4</td>
<td>2.4</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>CostRM</td>
<td>704</td>
<td>2.7</td>
<td>2.5</td>
<td>2.5</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>ProoLab</td>
<td>10694</td>
<td>20.8</td>
<td>258.3</td>
<td>14.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HealthwebEnh</td>
<td>766</td>
<td>8.0</td>
<td>8.5</td>
<td>5.8</td>
<td>54</td>
<td>83</td>
</tr>
<tr>
<td>HigherOrder</td>
<td>150</td>
<td>1.0</td>
<td>1.5</td>
<td>1.6</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>HigherOrderFoldr</td>
<td>108</td>
<td>2.3</td>
<td>2.8</td>
<td>2.8</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>Permission</td>
<td>251</td>
<td>4.3</td>
<td>4.5</td>
<td>3.5</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>IFlow_state</td>
<td>203</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>Provenance</td>
<td>221</td>
<td>1.5</td>
<td>1.5</td>
<td>0.8</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>Browser ext</td>
<td>785</td>
<td>3.5</td>
<td>3.3</td>
<td>3.3</td>
<td>89</td>
<td>55</td>
</tr>
<tr>
<td>DynSessions</td>
<td>211</td>
<td>0.7</td>
<td>0.5</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Example</th>
<th>LOC</th>
<th>SC</th>
<th>Trans</th>
<th>TC</th>
<th>SQ</th>
<th>TQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>CryptoLib</td>
<td>1330</td>
<td>50.5</td>
<td>426</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyManager</td>
<td>608</td>
<td>55.6</td>
<td>287</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AuthRPC</td>
<td>232</td>
<td>67.9</td>
<td>335</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SessionLib</td>
<td>32</td>
<td>0.4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>126</td>
<td>1.5</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>131</td>
<td>1.3</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metering</td>
<td>111</td>
<td>0.6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. HigherOrder and HigherOrderFoldr implement higher-order library functions that iterate over lists. Permission implements a stateful API of collections and iterators that guarantee that the collection underlying an iterator is never modified while an iteration is in progress. IFlow_state provides an information-flow tracking library for stateful programs. Provenance is a larger version of the curated database in §2.5. Browser_exts is a suite of 17 browser extensions, verified for authorization and information flow properties. DynSessions is the example of §4, including fork/join parallelism. F* and F* source typechecking times are roughly equivalent. F* translation is faster than Fine because there are fewer proofs to translate. Conversely, typechecking RDCIL code with refinement types is slower than checking DCIL proofs, but in view of the many advantages of F*, such as smaller bytecode and more expressive types, we find this tradeoff worthwhile.

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

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

6. Related work

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

References

A. Soundness
\[
\begin{align*}
\text{Figure 2. Static semantics of } F^* \text{ (kind well-formedness, kinding of types, expression typing)}
\end{align*}
\]

\[
\begin{align*}
\text{Figure 3. Static semantics of } F^* \text{ (sub-kinding, sub-typing, kind and type-conversion)}
\end{align*}
\]
Figure 4. Static semantics of $F^*$ (well-formed signatures, environments and runtime configurations)

Figure 5. Dynamic semantics of $F^*$
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, i.e., for all $S, \Gamma, x, t, \phi$, such that $\vdash S; \Gamma, x: t(\phi)$ wf;
   then $S; \Gamma, x: t(\phi) \models \phi$.
2. Compatible with type equivalence, i.e., for all $S, \Gamma, \phi, \phi'$, such that $\vdash S; \Gamma, \phi$ wf, and $S; \Gamma \models \phi \equiv \phi'$;
   then for fresh $x$, $S; \Gamma, x: \text{unit}(\phi) \models \phi'$.
3. Closed under value substitution, i.e., for all $S, \Gamma, x, t, \Gamma'$ if $\vdash S; \Gamma, x: t, \Gamma'$ wf;
   then $S; \Gamma, x: t, \Gamma' \models \phi \Rightarrow \forall v. S; \Gamma, x : t \\models \phi[v/x]$.
4. Closed under elimination of derivable value equality assumptions, i.e., for all $S, \Gamma, \alpha::k, t$ if $\vdash S; \Gamma, \alpha::k, t$;\: ok;
   then $S; \Gamma, \alpha::k, \Gamma' \models \phi \Rightarrow \exists S; \Gamma, \Gamma[t/\alpha] \models \phi/t/\alpha$.
5. Closed under weakening, i.e., for all $S, \Gamma, t_1, t_2$ if $\vdash S; \Gamma, t_1, t_2$ wf and $S; \Gamma_1, \Gamma_2$ wf;
   then $S; \Gamma_1, \Gamma_2 \models \phi \Rightarrow S; \Gamma_1, \Gamma_2 \models \phi$.
6. Closed under elimination of derivable value equality assumptions, i.e., for all $S, \Gamma, v, v', \Gamma'$ if $\vdash S; \Gamma, v = v', \Gamma'$;\: ok and $S; \Gamma \models v \equiv v'$;
   then $S; \Gamma, v = v', \Gamma' \models \phi \Rightarrow \exists S; \Gamma, \Gamma' \models \phi$.
7. Closed under elimination of derivable type equality assumptions, i.e., for all $S, \Gamma, t, t', \Gamma'$ if $\vdash S; \Gamma, t = t', \Gamma'$;\: ok and $S; \Gamma \models t \equiv t'$;
   then $S; \Gamma, t = t', \Gamma' \models \phi \Rightarrow \exists S; \Gamma, \Gamma' \models \phi$.
8. Derives only well-kinded formulas, i.e., for all $S, \Gamma, \phi$ such that $\vdash S; \Gamma, \phi$ wf; if $S; \Gamma \models \phi$ then $S; \Gamma \vdash \phi \equiv E$.

Note: We use ASCII syntax throughout the proofs.

For the most part, the transliteration from math fonts to ASCII should be obvious, e.g., we use $G$ instead of $\Gamma$ etc. We point out a few exceptions below.

Note, we write $\exists t$ to mean $t$ or $t$.

Note, we write $m ::= \text{type level} | \text{term level}$ instead of $m ::= \cdot | \varepsilon$.

Note, we write $! t$ to mean $t$.

LEMMA 2 (Well-typed prefixes, well-formed). For all $S_1, S_2, G_1, G_2, X_1, X_2$, all the following are valid:

1. $\vdash S_1, S_2, G_1, G_2, X_1, X_2$ wf $\Rightarrow$ $\vdash S_1, S_2, G_1, G_2, X_1$ wf
2. $\vdash S_1, S_2, G_1, G_2, -$ wf $\Rightarrow$ $\vdash S_1, S_2, G_1, -$ wf
3. $\vdash S_1, S_2, -$; - wf $\Rightarrow$ $\vdash S_1, -$; -wf

Proof Easy induction on the definition of well-formedness.

LEMMA 3 (Canonical forms).

For all $S, G, X, T, t, ts, vs, v, \phi$, such that $\vdash S; G, X$ wf;
all of the following are valid:

(CF1) $S; G, X \vdash v : ?(x:t)$ $\rightarrow$ $t_2$ $\Rightarrow$ exists $y$. $v=y$ or exists $e$, $v =$ \textbackslash{}x::t.e or exists $l$. $v = \langle\textbackslash{}x::t.e\rangle^\times$

(CF2) $S; G, X \vdash v : \langle\textbackslash{}\alpha::k.e\rangle$ $\Rightarrow$ exists $y$. $v=y$ or exists $e$, $v =$ \textbackslash{}\alpha::k.e or exists $l$. $v = \langle\textbackslash{}\alpha::k.e\rangle^\times$

(CF3) $S; G, X \vdash v : \langle T ts vs \rangle$ $\Rightarrow$ exists $y$. $v=y$ or exists $D$ $ts'$ $vs'$, $v=D$ $ts'$ $vs'$ or exists $l$. $v = \langle D$ $ts'$ $vs'\rangle^\times$

Proof By induction on the structure of the typing derivation.

Note that (T-D) requires the constructed type to have the shape $T \equiv v$.

LEMMA 4 (Transitivity of subtyping and subkinding). For all $S, G, t_1, t_2, t_3$, such $\vdash S; G$ wf
and $S; G \vdash t_1 :: t_2$ and $S; G \vdash t_2 :: t_3$ $\Rightarrow$ $S; G \vdash t_1 :: t_3$

For all $S, G, k_1, k_2, k_3$, such $\vdash S; G$ wf
and $S; G \vdash k_1 :: k_2$ and $S; G \vdash k_2 :: k_3$ $\Rightarrow$ $S; G \vdash k_1 :: k_3$

Proof By construction, using ST-Trans and SK-Trans.

LEMMA 5 (Inversion lemmas).

(INV1) For all $S, G, v x t \phi$,
$\vdash S; G$ wf and $S; G \vdash v : x t(\phi)$ $\Rightarrow$ $S; G \models \phi[v/x]$

(INV2) For all $S, G, v x t_1 t_2 k$,
$\vdash S; G$ wf and
S;G |- ?(x:t1 -> t2) :: k  => S;G, x:t1 |- t2 :: k

(INV3) For all S G 'a k t k',
| S;G |- S;G wf and
| S;G |- ?('a::k.t) :: k'  =>  exists k''. S;G, 'a::k |- t :: k''

(INV4) For all S G X x t t' e,
| S;G |- S;G wf and
| S;G;X |- \x.t e :: Q(X, t -> t')  =>  S;G, x:t;X,x |- e : t'

(INV5) For all S G X 'a k e t,
| S;G |- S;G wf and
| S;G;X |- /'a::k.e :: Q(X, '/a::k.t)  =>  S;G,'a::k;X |- e : t

(INV6) For all S G X D ts vs t,
| S;G |- S;G wf and
| S;G;X |- D ts vs : t  =>  exists T ts', vs'. t=Q(X', T ts' vs')

(INV7) For all S G x t1 k b,
| S;G |- S;G wf and
| S;G |- x:t1 => k :: ok(b)  =>  S;G, x:t1 |- k :: ok(b)

(INV8) For all S G 'a k1 k2 b,
| S;G |- S;G wf and
| S;G |- 'a::k1 => k2 :: ok(b)  =>  S;G, 'a::k1 |- k2 :: ok(b)

(INV9) For all S G X v l t m,
| S;G |- S;G;X wf and m=Term_level
| S;G;X |-^m (v^l) : t  =>  S;G;X' |-^m v : t

Proof By induction on the structure of the typing derivation, using transitivity of subtyping to collapse multiple applications of subtyping.

□

Lemma 6 (Progress). For all S A G X e t m,
(A1) S |- A wf
(A2) A ==> G;X
(A3) |- S;G;X wf
(A4) S;G;X |-^m e : t where m=. (term level)
=> exists v, e=v or exists A' e', (A; e) -->_S (A'; e')

Proof By induction on the structure of (A4). Interesting cases follow:

Cases (T-X) and (T-XA): (e=x)
Impossible. From the definition of (A ==> G;X), since all y in dom(G) are fresh, we have x not in dom(G).

Case (T-Drop): Induction hypothesis.

Case (T-Abs), (T-D), (T-TAbs), (T-Val): Already values, satisfying LHS of disjunct.


Case (T-App): (e=v1 v2)
Inversion gives us S;G X1 |- v1:x:t1 -> t2

We split on the three cases of the canonical forms lemma (CF1)

Sub-case 1:
v1 is a variable; impossible, as in (T-X) and (T-XA).

Sub-case 2:
v1 is a lambda term, and we can step using (E-Beta)

Sub-case 3:
v1 is of the form (v1')^l. From the definition of destruct contexts, we need to step using (E-Kill).
Inverting (T-App), in the first premise, we have (eventually) an application of (T-Box), and hence l in X.
From the definition of (A ==> G;X) we have (l in X <===> l in A)
Thus (E-Kill) is applicable and we can take a step.

Case (T-TApp): (e = v t)
    Similar to (T-App)

Case (T-Let): (e = (let x = e1 in e2))
    If e1 <> v then, by the induction hypothesis it steps to e1' and e steps using (E-Ctx)
    Otherwise, e1=v and we can step using (E-Let)

Case (T-BoxTerm): Already a value, satisfying LHS of disjunct.

Case (T-BoxType): Impossible, we expect m=.

Case (T-Match): (e = match v with D ...)
    We first apply (INV6) to the fourth premise to derive t_p=?T ts vs, for some ts, vs.
    Then, using the fifth premise, we get t_v = ?T ts' vs', for some ts', vs'

    Applying (CF-3) to the first premise, we have three cases, as in (T-App).
    
    Sub-case 1:
        v is a variable; impossible, as in (T-X) and (T-XA).

    Sub-case 2:
        v is some (D1 ts' vs'): (E-M1) and (E-M2) and mutually exclusive and complete and thus can step.

    Sub-case 3:
        v1 is of the form (v1')^l. Again, as in (T-App), we can step using (E-Kill).

Case (T-Sub): Induction hypothesis.

\[ \square \]

**Lemma 7** (Weakening for type and kind conversion).

[1. Sub-typing]
For all S G G1 G2 t1 t2,
[A1] |- S;G1,G2 wf
[A2] |- S;G1,G,G2 wf
[A3] S;G1,G2 |- t1 <: t2
=> S;G1,G,G2 |- t1 <: t2

[2. Sub-kinding]
For all S G G1 G2 k1 k2,
[A1] |- S;G1,G2 wf
[A2] |- S;G1,G,G2 wf
[A3] S;G1,G2 |- k1 <: k2
=> S;G1,G,G2 |- k1 <: k2

[3. Type equivalence]
For all S G G1 G2 t1 t2,
[A1] |- S;G1,G2 wf
[A2] |- S;G1,G,G2 wf
[A3] S;G1,G2 |- t1 = t2
=> S;G1,G,G2 |- t1 = t2

[4. Kind equivalence]
For all S G G1 G2 k1 k2,
[A1] |- S;G1,G2 wf
[A2] |- S;G1,G,G2 wf
[A3] S;G1,G2 |- k1 = k2
=> S;G1,G,G2 |- k1 = k2

Proof In each case, proof proceeds by mutual induction over the structure of the assumption (A3). In the sub-typing judgments, we appeal to the admissibility constraints of the entailment relation, specifically that it is closed under weakening.

\[ \square \]

**Lemma 8** (Weakening).
[1. Typing]
For all $S \ G \ G_1 \ G_2 \ X_1 \ X_2 \ X \ e \ t \ m$,

[A1] $\vdash S;G_1,G_2;X_1,X_2$ $\text{wf}$

[A2] $\vdash S;G_1,G,G_2;X_1,X,X_2$ $\text{wf}$

[A3] $S;G_1,G_2;X_1,X_2 \vdash^m e : t$

$\Rightarrow S;G_1,G,G_2;X_1,X,X_2 \vdash^m e : t$

[2. Kinding]
For all $S \ G \ G_1 \ G_2 \ G \ t \ k$,

[A1] $\vdash S;G_1,G_2$ $\text{wf}$

[A2] $\vdash S;G_1,G,G_2$ $\text{wf}$

[A3] $S;G_1,G_2 \vdash t :: k$

$\Rightarrow S;G_1,G,G_2 \vdash t :: k$

[3. Kind well-formedness]
For all $S \ G \ G_1 \ G_2 \ G \ t \ k \ b$,

[A1] $\vdash S;G_1,G_2$ $\text{wf}$

[A2] $\vdash S;G_1,G,G_2$ $\text{wf}$

[A3] $S;G_1,G_2 \vdash k \ ok(b)$

$\Rightarrow S;G_1,G,G_2 \vdash k :: ok(b)$

Proof: By mutual induction over the structure of assumption (A3), in each case, generalizing over the tail of the context $G_2$.

Interesting cases of (1)

Cases (T-X), (T-XA):

In both cases, we can establish the conclusion by using (T-X/T-XA) in the premise of (T-Drop), to drop the additional affine assumptions in $X$, and noting that $G_1,G,G_2$ binds $x$ iff $G_1,G_2$ binds $x$.

Case (T-Abs):

From an inversion of (A3), we have

(A3.1) $S;G_1,G_2 \vdash t :: c$ and

(A3.2) $S;G_1,G_2,x:t; X,x \vdash e : t'$

One subtlety in this part of the proof:

To re-establish (A3.2), we need to use the induction hypothesis since, which is applicable since we’ve been careful to generalize on the extended environment $G_2$, and since the induction hypothesis specifically allows weakening by inserting.

However, a direct application of the IH results in the following:

(A3.2.broken) $S;G_1,G,G_2,x:t; X_1,X,X_2,X' \vdash e : t'$

However, this derivation is not always of the right shape, since when $X_1,X_2$ is empty and $X$ is not empty, then when constructing

$S;G_1,G,G_2;X_1,X,X_2 \vdash \lambda x : t.e : ?(x:t) \rightarrow t'$

with (A3.2.broken) in the premise, the qualifier on the introduced type $?(x:t) \rightarrow t'$ may differ from the qualifier on the type introduced in (A3).

To remedy this, we establish the conclusion by first showing that $S;G_1,G,G_2;X_1,X_2$ is well-formed—-it is always permissible to drop assumptions in $X$ and preserve well-formedness.

We then use the induction hypothesis to derive:

(A3.2') $S;G_1,G,G_2,x:t; X_1,X_2,x \vdash e : t'$
And the mutual induction hypothesis to derive:

\((A3.1')\) \(S;G1,G,G2 \vdash t :: c\)

Finally, we use an application of (T-Drop) in the context 
\((S;G1,G,G2; X1,X,X2)\) to discard the affine assumptions \(X\). In the 
premises of (T-Drop), we use (T-Abs) with (A3.1') and (A3.2').

Case (T-Tabs): Similar to (T-Abs).

Case (T-Sub): Appeal to weakening on type conversion judgment.

Other cases straightforward.

**Interesting cases of (2)**

Case (K-Sub): Appeal to weakening on sub-kind judgment.

Cases (K-Arr, K-Univ):

Induction hypothesis, as in in (T-Abs) above, but without the 
complications of the \(X\)-environment.

Cases (K-Tv):

Mutual induction hypothesis with (1).

Case (K-phi):

Appeal to admissibility of entailment relation, which includes a 
requirement on weakening.

**Interesting cases of (3)**

Case (OK-ES, OK-A, OK-V): All are analogous to (K-Arr) above.

---

**Lemma 9** (Value substitution for type and kind conversion).

1. **Sub-typing**

   For all \(S G1 G2 t1 t2 x t v X m\),
   
   \[
   \begin{align*}
   & [A1] \vdash S;G1,x:t,G2 wf \quad \text{and} \quad \vdash S;G1;X wf \\
   & [A2] S;G1,x:t,G2 \vdash t1 <: t2 \\
   & [A3] S;G1;X \vdash^m v : t \\
   & [A4] \exists k. \forall i. S;G1,x:t,G2 \vdash ti :: k \\
   \Rightarrow S;G1,G2[v/x] \vdash t1[v/x] <: t2[v/x]
   \end{align*}
   \]

2. **Sub-kinding**

   For all \(S G1 G2 k1 k2 x t v X m\),
   
   \[
   \begin{align*}
   & [A1] \vdash S;G1,x:t,G2 wf \quad \text{and} \quad \vdash S;G1;X wf \\
   & [A2] S;G1,x:t,G2 \vdash k1 <: k2 \\
   & [A3] S;G1;X \vdash^m v : t \\
   & [A4] \exists b. \forall i. S;G1,x:t,G2 \vdash ki :: \text{ok}(b) \\
   \Rightarrow S;G1,G2[v/x] \vdash ki[v/x] <: k2[v/x]
   \end{align*}
   \]

3. **Type equivalence**

   For all \(S G1 G2 t1 t2 x t v X m\),
   
   \[
   \begin{align*}
   & [A1] \vdash S;G1,x:t,G2 wf \quad \text{and} \quad \vdash S;G1;X wf \\
   & [A2] S;G1,x:t,G2 \vdash t1 = t2 \\
   & [A3] S;G1;X \vdash^m v : t \\
   & [A4] \exists k. \forall i. S;G1,x:t,G2 \vdash ti :: k \\
   \Rightarrow S;G1,G2[v/x] \vdash t1[v/x] = t2[v/x]
   \end{align*}
   \]

4. **Kind equivalence**

   For all \(S G1 G2 k1 k2 x t v X m\),
   
   \[
   \begin{align*}
   & [A1] \vdash S;G1,x:t,G2 wf \quad \text{and} \quad \vdash S;G1;X wf \\
   & [A2] S;G1,x:t,G2 \vdash k1 = k2 \\
   & [A3] S;G1;X \vdash^m v : t \\
   & [A4] \exists b. \forall i. S;G1,x:t,G2 \vdash ki :: \text{ok}(b)
   \end{align*}
   \]

---

Draft – September 17, 2011
Cases of (1):

Case ST-Ref1: Trivial.

Case ST-Prod:

Induction hypothesis on the first premise.

For the second premise, in order to establish that the extended environment is well-formed, we appeal to (A4).

Then, we use the induction hypothesis on the second premise.

Case ST-ProdK: Similar to ST-Prod.

Case ST-RefLeft: Induction hypothesis.

Case ST-RefRight:

Appeal to the admissibility of the entailment relation, which demands that it be closed under substitution.

Case ST-Afn: Induction hypothesis.

Case ST-Trans: Induction hypothesis, twice.

Cases of (2):

Cases (SK-Prod, SK-ProdK): Similar to ST-Prod in the previous case.

Remaining cases straightforward.

Cases of (3):

Cases (TE-Prod, TE-ProdK): Similar to ST-Prod.

Remaining cases straightforward.

Cases of (4):

Cases (KE-Prod, KE-ProdK): Similar to ST-Prod.

□

LEMMA 10 (Type substitution for type and kind conversion).

[1. Sub-typing]
For all \( S \ G1 \ G2 \ t1 t2 'a k t \):

\[
\begin{align*}
& [A1] \quad \vdash S;G1,'a::k,G2 \text{ wf} \\
& [A2] \quad S;G1,'a::k,G2 \vdash t1 <: t2 \\
& [A3] \quad S;G1 \vdash t :: k \\
& [A4] \quad \exists k'. \forall i. S;G1,'a::k,G2 \vdash ti :: k' \\
\Rightarrow & \quad S;G1,G2[t/'a]\vdash t1[t/'a] <: t2[t/'a]
\end{align*}
\]

[2. Sub-kindling]
For all \( S \ G1 \ G2 \ k1 k2 'a k t \):

\[
\begin{align*}
& [A1] \quad \vdash S;G1,'a::k,G2 \text{ wf} \\
& [A2] \quad S;G1,'a::k,G2 \vdash k1 <: k2 \\
& [A3] \quad S;G1 \vdash t :: k \\
& [A4] \quad \exists b. \forall i. S;G1,'a::k,G2 \vdash ki :: ok(b) \\
\Rightarrow & \quad S;G1,G2[t/'a]\vdash k1[t/'a] <: k2[t/'a]
\end{align*}
\]

[3. Type equivalence]
For all \( S \ G1 \ G2 \ t1 t2 'a k t \):

\[
\begin{align*}
& [A1] \quad \vdash S;G1,'a::k,G2 \text{ wf} \\
& [A2] \quad S;G1,'a::k,G2 \vdash t1 = t2 \\
& [A3] \quad S;G1 \vdash t :: k \\
& [A4] \quad \exists k'. \forall i. S;G1,'a::k,G2 \vdash ti :: k' \\
\Rightarrow & \quad S;G1,G2[t/'a]\vdash t1[t/'a] = t2[t/'a]
\end{align*}
\]
[4. Kind equivalence]
For all $S \ G \ 1 \ 2 \ t_1 \ t_2 \ 'a \ k \ t$,

[A1] $\ |- S;G1,'a::k,G2$ $\ w f$

[A2] $S;G1,'a::k,G2 \ |- \ k1 = k2$

[A3] $S;G1 \ |- \ t :: k$

[A4] exists $b$.forall $i$. $S;G1,'a::k,G2 \ |- \ ki :: ok(b)$

$\Rightarrow S;G1,G2[t/'a] |- k1[t/'a] = k2[t/'a]$

Proof Mutual induction in each case over the structure of (A2).
Similar to corresponding proof for value substitution.

Lemma 11 (Type-level empty affine context).
For all $S \ G \ X \ e \ m \ t$,

[A1] $\ |- S;G;X$ $\ ok$

[A2] $S;G;X \ |-^m e : t$

$\Rightarrow S;G;\ |-^type_level e : t$

Proof Easy induction over the structure of (A2).

Lemma 12 (Elimination of $X$-context for non-affine values).
For all $S \ G \ X \ v \ m \ t \ b$,

[A1] $\ |- S;G;X$ $\ ok$

[A2] $S;G;X \ |-^m v : t$

[A3] $S;G \ |- t :: b$, where $b < A$

$\Rightarrow S;G;\ |-^m v : t$

Proof Induction over the structure of (A2).

Case (T-X): We already have (A2) as the conclusion.

Case (T-XA): The premise contradicts (A3).

Case (T-Abs): We have that $Q(X, x:t \rightarrow t')$ is non-affine. By inversion on $Q$, we have $X=$.

Case (T-TAbs): Similar to T-Abs.

Case (T-Assume), (T-TApp), (T-App), (T-Let), (T-Match): These are non-values.

Case (T-BoxTerm): Second premise contradicts (A3).

Case (T-BoxType): Second premise contradicts (A3).

Lemma 13 (Strengthening of $X$-context for non-affine assumptions).
For all $S \ G \ X \ x \ m \ e \ t \ tx \ b$,

[A1] $\ |- S;G;X,x$ $\ ok$

[A2] $S;G;X \ |-^m e : t$

[A3] $G(x) = tx$

[A4] $S;G \ |- tx :: b$

$\Rightarrow (b<>A \ | | \ m=type_level) \Rightarrow S;G;X \ |-^m e : t$

Proof Straightforward induction on the structure of (A2), noting that (T-XA) requires $tx::A$.

Lemma 14 (Value substitution).
[1. Typing]
For all $S \ G1 \ G2 \ X1 \ X2 \ X3 \ X4 \ x \ t_1 \ e \ t \ v \ b \ m$,

[A0] $X2=x$ or $X2=.$

[A1] $\ |- S;G1,x::t1,G2;X1,X2,X3,X4$ $\ w f$ and $\ |- S; \ G1; \ X4 \ w f$

[A2] $S;G1,x::t1,G2;X1,X2,X3 \ |-^m e : t$

[A3] $S;G1;X4 \ |-^m v : ti$

[A5] $S;G1 \ |- t1 :: b$

$\Rightarrow exists \ X. \ (S;G1,G2[v/x];X \ |- e[v/x] : t[v/x] \ /
\{(X=X1,X3 \ when \ (b<>A \ | | \ X2=.) \ | | \ m=type_level) \ /\n(X=X1,X4,X3 \ otherwise)}$
For all $S \ G1 \ G2 \ x \ t1 \ t \ k \ v$,

[A1] $\vdash S;G1,x:t1,G2; . \ \text{wf}$

[A2] $S;G1,x:t1,G2 \vdash t :: k$

[A3] $S;G1; . \vdash^\text{type level} v : t1$

$\Rightarrow S;G1,G2[v/x] \vdash t[v/x] :: k[v/x]$

[3. Kind well-formedness]

For all $S \ G1 \ G2 \ x \ t1 \ k \ b \ v$,

[A1] $\vdash S;G1,x:t1,G2; . \ \text{wf}$

[A2] $S;G1,x:t1,G2 \vdash k \ ok(b)$

[A3] $S;G1; . \vdash^\text{type level} v : t1$

$\Rightarrow S;G1,G2[v/x] \vdash k[v/x] \ ok(b)$

[4. Well-formed environments]

For all $S \ G1 \ G2 \ x \ t1 \ v \ X$,

[A1] $\vdash S; G1, x:t1, G2 \ \text{wf}$ and $\vdash S; G1; X \ \text{wf}$

[A2] $S; G1; X \vdash v : t1$

$\Rightarrow \vdash S; G1, G2[v/x] \ \text{wf}$

Proof Mutual induction for (1,2,3,4) on the structure of (A2) in the first three cases, and on the length of G2 in the last case.

Let $s$ be the substitution $[v/x]$.

Cases of 1:

Case (T-X): $e=x$ and $S;G1,x:t1,G2; \vdash y : t$

We consider two sub-cases:

Sub-case ($x<y$):

In this case, we have $X1,X2,X3=.$, so our goal is to show

$S;G1,s(G2); . \vdash y : s(t)$

We have two further sub-cases:

Sub-sub-case ($y \in \text{dom}(G1)$):

From the well-formedness of $S;G1,x:t1;$. (from well-formed prefixes and (A1))
And $G1(y)=t$, we can conclude we conclude that $x$ not in $\text{FV}(t)$
Thus, $s(t)=t$, and we have:

$S;G1,s(G2); . \vdash y : s(t)$

Sub-sub-case ($y \in \text{dom}(G2)$):

We have $y \in \text{dom}(G2)$ and $G2(y)=t \Rightarrow s(G2(y))=s(t)$

Sub-case ($x=y$):

We have $X2=.$, so we need to show $S;G1,s(G2); . \vdash v : t1$

Since, from well-formed environments, we have $\text{FV}(t1) \ \subseteq \ \text{dom}(G1)$.

As a sub-goal, we construct $(A3')$ $S;G1; . \vdash v : t1$ and then apply
Weakening to conclude with the result, where in order to apply weakening, we are required to show $(\vdash S;G1,s(G2);X4 \ \text{wf})$

This follows from a use of the mutual induction hypothesis with (4).

Now, to show $(A3')$, we consider two further sub-cases:

----Sub-sub-case ($t1::b \ \land \ b \not\leftrightarrow A$):

Since, we have $(b\not\leftrightarrow A)$, we apply Lemma (Elimination of X-context for non-affine values) to $(A3)$ to get:
Here, we must have m=type_level, so we apply Lemma (Type-level empty affine context) to conclude.

Case (T-XA): Here, we have (t1::A)

The goal is \( S; G_1, s(G_2); X_4 \vdash v : t_1 \)

Which following from weakening applied to A3, where in order to apply weakening, we are required to show \( \vdash S; G_1; s(G_2); X_4 \text{ wf} \)

This follows from a use of the mutual induction hypothesis with (4).

Case (T-Abs): Inversion of (A2) gives us:

\[ X=X_1,X_2,X_3 \]
\[ G=G_1,x:tx,G_2 \]  \hspace{1cm} (A2.1)
\[ S; G \vdash ty :: c \]  \hspace{1cm} (A2.2)
\[ S; G,y:ty; X,y \vdash e : t' \]  \hspace{1cm} (A2.3)
\[ X_1=\Rightarrow t = ?(y:ty) \rightarrow t' \]  \hspace{1cm} (A2.4)
\[ X_1<\Rightarrow t = !(y:ty) \rightarrow t' \]  \hspace{1cm} (A2.5)

-------------------------------------- [T-Abs]
\[ S; G; X \vdash \lambda y:ty.e : t \]

From the mutual induction hypothesis on (A2.2), applied to

\( (A3') \ S; G_1; \vdash \text{type_level } v : t_1 \)  \hspace{1cm} (from Lemma type-level empty affine context)

we get

\( (A2.2') \ S; G_1, s(G_2) \vdash s(ty) :: c \)  \hspace{1cm} (since \( s(c) = c \))

From the induction hypothesis on (A2.3), we get

\( (A2.3') \) exists \( X_0 \). \( S; G_1, s(G_2), y: s(ty); X_0, y \vdash s(e) : s(t') \)
\hspace{2cm} where \( (X_0 = X_1, X_3 \text{ when } \ldots \) \)
\hspace{3.5cm} \( X_0 = X_1, X_4, X_3 \text{ otherwise})

We can now construct

\( (A2') \ S; G_1, s(G_2); X_0 \vdash s(\lambda y:ty.e) : ?(y:s(ty)) \rightarrow s(t') \)

using (T-Abs) with (A2.3') and (A2.4') in the premises.

We now consider two sub-cases:

Sub-case \( (X_2=x) \): Goal requires us to show that \( X_0 = X_1, X_4, X_3 \)

From \( (A2.3') \) we have \( X_0 = X_1, X_4, X_3 \), as needed.

But, we now have to get the qualifiers on the type in \( A2' \) and A2 to match up.

Sub-sub-case and \((X_1,X_4,X_3=.)\):

In this case, we are required to construct a derivation with:

\[ s(t) = !(y:s(ty)) \rightarrow s(t') \]

However, the definition of \( Q(X,t) \), (the fourth premise (A2.4) above), gives us the freedom to chose

\[ ?(y:s(t)) \rightarrow s(t') = !(y:s(t)) \rightarrow s(t') \]

as required.

Sub-sub-case and \((X_4<>.)\):
Here, $Q((X_1,X_4,X_3),t) = Q((X_1,X_2,X_3),t)$.

Sub-case ($X_2=\_.$):

Here, we must show that that $X_0=X_1,X_3$, which again we obtain from ($A2.3'$)

Since we have $(X_1,X_2,X_3 = X_1,X_3)$, the qualifiers on the result match.

Case (T-TAbs): Similar to T-Abs.

Case (T-D): Induction hypothesis on the second and third premise; signature is unchanged.

Case (T-Val):

From admissibility of entailment relation, we get that the second premise is closed under substitution.

Case (T-Sub): Induction hypothesis and substitution lemma for type conversion.

Case (T-Assume): Mutual induction hypothesis on the kinding judgment.

Case (T-App):

\[
\begin{align*}
X_1,X_2,X_3 &= X,X' \\
G &= G_1,x:tx,G_2 \\
S; G; X |- v_1 : ?(y:t) \rightarrow t' & \quad (A2.1) \\
S; G; X' |- v_1 : t & \quad (A2.2) \\
\hline
\end{align*}
\]

$\text{T-App}$

\[
\begin{align*}
S; G; X_1,X_2,X_3 |- v_1 v_2 : t'[v_2/y] & \\
\end{align*}
\]

Where our goal is, exists $X_0$, such that

\[
S; G_1,s(G_2); X_0 |- s(v_1 v_2) : s(t'[v_2/y])
\]

i.e., $S; G_1,s(G_2); X_0 |- s(v_1 v_2) : s(t')[s(v_2)/y]

Since, $x<>y$ and $y$ not in $\text{FV}(\text{range}(s))$.

We consider two sub-cases, depending on whether $t_1$, the type of $x$ is affine.

---Sub-case ($t_1:A$):

We consider two further sub-cases:

-----Sub-sub-case ($X_2=\_.$):

In this case, we have to show that $X_0=X_1,X_3$

From the induction hypothesis applied to the premises ($A2.2$) and ($A2.3$), we have

\[
\begin{align*}
(A2.2') & \quad S; G_1,s(G_2); X |- s(v_1) : ?(y:s(t)) \rightarrow s(t') \\
(A2.3') & \quad S; G_1,s(G_2); X' |- s(v_2) : s(t) \\
\end{align*}
\]

where $(X,X' = X_1,X_3)$

Thus we have the conclusion by applying (T-App) to $A2.2'$ and $A2.3'$.

-----Sub-sub-case ($X_2=x$):

Note that on splitting $X_1,X_2,X_3$ into $X,X'$, the assumption $x$ in $X_2$ goes either in $X$ or in $X'$, but not oth.

Without loss of generality, assume $x$ is in $X$. 

Draft – September 17, 2011
Applying the induction hypothesis to (A2.2) we get:

(A2.2') S; G1,s(G2); X,X4 |- s(v1) : ?(y:s(t)) -> s(t')

And, applying the induction hypothesis to (A2.3) we get:

(A2.3') S; G1,s(G2); X' |- s(v2) : s(t)

since we have (t1::A and x not in X')

From (A1), we have X, X4, X' is well-formed.

So, for the conclusion, we apply (T-App), with (A2.2') and (A2.3') in the premises.

---Sub-case (t1::b & b<>A)

In this case, from lemma (Elimination of X-context for non-affine values), we have

[A3'] S;G1; . |- v : t1

Likewise, from Lemma (Strengthening of X-context for non-affine names), we have

(A2.2') S; G1,x:t1,G2; X |- v1 : ?(y:t1 -> t2
(A2.3') S; G1,x:t1,G2; X' |- v2 : t1

where x not in X,X'

Now, using the induction hypothesis on (A2.2') and (A2.3'), using (A3') gives us

(A2.2'') S; G1,s(G2); X |- s(v) : ?(y:s(t1)) -> s(t2)
(A2.3'') S; G1,s(G2); X' |- s(v') : s(t1)

For the conclusion, we require a derivation with the type s(t2[v'/y])

Using (T-App) with (A2.2') and (A2.3') gives us the type

s(t2) [s(v')/y]

We need to show

s(t2[v'/y]) = s(t2)[s(v')/y])

which follows from the observations (from alpha renaming of y)

Case (T-TApp): Similar to T-App

Case (T-Let): Straightforward, from induction hypothesis.

Case (T-Match):

--- T-Match ---

S;G;X1,X2,X3 |- match v with
D a1..an x1..xm -> e1 : t (A2.7)
else e2

G=G1,x:tx,G2           (A2.1)
X1,X2,X3 = X,X'         (A2.2)
X3 = x1..xn            (A2.3)
S; G; X1 |- v : tv      (A2.4)
S; G, ai::ki,xj:tj; X3 |- D a1..an x1..xm : tp (A2.5)
tv = tp[vj/xj][tj'/aj]  (A2.6)
S; G, ai::ki,xj:tj, aj=tj',xj=vj, v=D a1..an x1..xm; X2,X3 |- e1 : t (A2.7)
S; G; X2 |- e2 : t      (A2.8)
As in (T-App), the affine assumption in X’, if present, either goes to X or X’. When using the induction hypothesis, we satisfy premise (A0) by using either side of the disjunct, depending on whether x in X1, X2 or neither.

We use the induction hypothesis to establish

\((A2.4')\)  \(S; G_1, s(G_2); X_1 \vdash s(v) : s(tv)\)
\((A2.5')\)  \(S; G_1, s(G_2, a_i::k_i, x_j::t_j); X_3 \vdash s(D a_1..a_n x_1..x_m) : s(tp)\)

\((A2.6')\) follows analogously, since equality is closed under substitution

The induction hypothesis on the last two premises gives us:

\((A2.7')\)  \(S; G_1, s(G_2, a_i::k_i, x_j::t_j, \ldots); X_2, X_3 \vdash s(e_1) : s(t)\)
\((A2.8')\)  \(S; G_1, s(G_2); X_2 \vdash s(e_2) : s(t)\)

We use each of these in the premises of (T-Match) for the goal.

Cases of 2:
Cases K-a, K-T: Value substitutions do not affect type variables and constants.

Case K-Univ:
For the first premise, we use the mutual induction hypothesis with (3), subst. for well-formedness of kinds

We establish the well-formedness of the extended environment in the second premise using the first premise. Then, apply the induction hypothesis to conclude.

Case K-Sub:
Induction hypothesis on the first premise.
Substitution lemma for the sub-kinding for the second premise.
Mutual induction hypothesis for the third premise.

Case K-A:
Induction hypothesis on the first premise.

Case K-Arr, K-Fun: Similar to K-Univ.

Case K-Tv:
For the second premise, we use the mutual induction hypothesis with (1), subst. for the typing judgment

Additionally, we need to show for the second premise that \(v_1[v/x]\) is a value, which is immediate from the syntax of values

Case K-Tt:
Induction hypothesis on each premise.

Case K-phi:
Admissibility of the entailment relation ensures it is closed under substitution.

Cases of 3:
Case OK-b: Trivial.

Case OK-KK: Induction hyp. on each premise.

Case OK-TK: Induction hyp. on each premise.

Cases of 4:
Substitutions map over environments.

Suffice to show well-formedness is preserved on each kind of binding in the environment.

Sub-case (x::t): Mutual induction hypothesis with (2).
Sub-case (\(\tilde{\alpha}::k\)): Mutual induction hypothesis with (3).

Sub-case (\(v_1=v_2\)): Mutual induction hypothesis with (1) on either side of the equality.

Sub-case (\(t_1=t_2\)): Mutual induction hypothesis with (2) on either side of the equality.

\[
\square
\]

**Lemma 15 (Type substitution).**

[1. Typing]

For all \(S G_1 G_2 X \tilde{\alpha} k e t ta\),

\[\begin{align*}
&A1 \vdash S;G_1,\tilde{\alpha}::k,G_2;X \text{ ok} \\
&A2 \vdash S;G_1,\tilde{\alpha}::k,G_2;X \vdash e : t \\
&A3 \vdash S;G_1 \vdash ta :: k
\end{align*}\]

\[\Rightarrow S;G_1,G_2[ta/'\tilde{\alpha}]X \vdash e[ta/'\tilde{\alpha}] : t[ta/'\tilde{\alpha}]\]

[2. Kinding]

For all \(S G_1 G_2 \tilde{\alpha} k k' ta\),

\[\begin{align*}
&A1 \vdash S;G_1,\tilde{\alpha}::k,G_2; . \text{ ok} \\
&A2 \vdash S;G_1,\tilde{\alpha}::k,G_2 \vdash t :: k' \\
&A3 \vdash S;G_1 \vdash ta :: k
\end{align*}\]

\[\Rightarrow S;G_1,G_2[ta/'\tilde{\alpha}] \vdash t[ta/'\tilde{\alpha}] : k'[ta/'\tilde{\alpha}]\]

[3. Kind well-formedness]

For all \(S G_1 G_2 \tilde{\alpha} k k' b ta i\),

\[\begin{align*}
&A1 \vdash S;G_1,\tilde{\alpha}::k,G_2; . \text{ ok} \\
&A2 \vdash S;G_1,\tilde{\alpha}::k,G_2 \vdash _i k' \text{ ok}(b) \\
&A3 \vdash S;G_1 \vdash ta :: k
\end{align*}\]

\[\Rightarrow S;G_1,G_2[ta/'\tilde{\alpha}] \vdash k'[ta/'\tilde{\alpha}] : \text{ ok}(b)\]

[4. Well-formed environments]

For all \(S G_1 G_2 \tilde{\alpha} k t,\),

\[\begin{align*}
&A1 \vdash S; G_1, \tilde{\alpha}::k, G_2 \text{ wf} \\
&A2 \vdash S; G_1 \vdash t :: k
\end{align*}\]

\[\Rightarrow \vdash S;G_1, G_2[t/'\tilde{\alpha}] \text{ wf}\]

\[
\text{Proof By mutual induction in each case on the structure of (A3).}
\]

\[
\text{Identical in structure to the value substitution lemma, but simpler, since it does not involve the added complication of managing the affine environment } X.
\]

\[
\square
\]

**Lemma 16 (Well-kinded typings, Well-formed kindings).**

[1. Typing]

For all \(S G X e t,\)

\[\begin{align*}
&A1 \vdash S;G;X \text{ wf} \\
&A2 \vdash S;G;X \vdash e : t
\end{align*}\]

\[\Rightarrow \exists c. S;G \vdash t :: c\]

[2. Kinding]

For all \(S G t k,\)

\[\begin{align*}
&A1 \vdash S;G \text{ wf} \\
&A2 \vdash S;G \vdash t :: k
\end{align*}\]

\[\Rightarrow \exists b. S;G \vdash k \text{ ok}(b)\]

\[
\text{Proof By mutual induction, in both cases on the structure of (A2).}
\]

**Cases of 1:**

**Cases (T-X, T-XA):** From well-formedness of the environment.

**Case (T-Abs):**

The extended environment is well-formed by construction and so the range type \(t'\) is well-kinded by the induction hypothesis.

The conclusion follows from an application of of \((K-Arr)\)

**Case (T-D):**
From well-formed signatures, we have that \( t_\_d \) has a concrete kind \( c \).
So, the extended environment in the third premise is well-formed.
From the induction hypothesis, we conclude that \( t \) is well-kinded.

Case (T-TAbs): Similar to (T-Abs).

Case (T-TAbs): Similar to (T-Abs).

Case (T-Val):

From the admissibility of the entailment relation, we have \( \phi :: E \)
We conclude using the third premise and \( \phi :: E \) in the premises of (K-\( \phi \)).

Case (T-Assume):

Apply the premise to K-\( \phi \), noting from \( \mathcal{L} \to S \, w \, f \) that we have \( S;G |- \text{unit} :: * \)

Case (T-App):

\[
\begin{align*}
S;G;X1 |- v1 : ?x:t1 -> t2 & \quad (A2.1) \\
S;G;X2 |- v2 : t1 & \quad (A2.2) \\
\hline
S;G;X1,X2 |- v1 v2 : t2[v2/x] & \quad (A2)
\end{align*}
\]

We use the induction hypothesis on each of the premises to conclude that the types are well-kinded, i.e., we have

\[
S;G |- ?x:t1 -> t2 :: k \quad (K1)
\]

Using (INV2) on K1, we get

\[
S;G,x:t1 |- t2 :: k' \quad (K1.1)
\]

Applying the substitution lemma to (K1.1), we get

\[
S;G |- t2[v2/x] :: k'[v2/x] \quad (K1.1)
\]

Which suffices for the conclusion.

Case (T-TApp): Similar to (T-App), but using the type substitution lemma.

Case (T-Let): Immediate from the second last premise.

Case (T-Match): Immediate from the second last premise.

Case (T-BoxTerm, T-BoxType, T-Sub): Immediate from the last premise.

Cases of 2:

Case (K-a): From well-formed environment.

Case (K-T): From well-formed signature.

Case (K-Univ): Trivial. (\( b \) is always well-formed)

Case (K-Sub): From the last premise.

Case (K-A): Trivial. (\( A \) is always well-formed)

Case (K-Arr): Trivial. (\( b \) is always well-formed)

Case (K-Fun): From the first premise.
Case (K-Tv): Analogous to (T-App), using the inversion lemma (INV7) followed by the value subst. lemma.

Case (K-Tt): Analogous to (T-TApp), using the inversion lemma (INV8) followed by the type subst. lemma.

Case (K-phi): Trivial (c is always well-formed)

**Lemma 17** (Elimination of derivable value equality assumptions).

1. **Typing**
   - For all $S$, $G_1$, $G_2$, $X$, $v_1$, $v_2$, $e$, $t$, $m$,
     - $\vdash S; G_1.v_1=v_2.G_2.X$ wf
     - $S; G_1 \vdash v_1=v_2$
     - $S; G_1.v_1=v_2.G_2.X \vdash \neg \neg^m e : t$
     - $\Rightarrow S; G_1,G_2.X \vdash ^m e : t$

2. **Kinding**
   - For all $S$, $G_1$, $G_2$, $v_1$, $v_2$, $t$, $k$
     - $\vdash S; G_1.v_1=v_2.G_2$ wf
     - $S; G_1 \vdash v_1=v_2$
     - $S; G_1.v_1=v_2.G_2 \vdash t :: k$
     - $\Rightarrow S; G_1,G_2 \vdash t :: k$

3. **Kind well-formedness**
   - For all $S$, $G_1$, $G_2$, $v_1$, $v_2$, $k$, $i$, $b$
     - $\vdash S; G_1.v_1=v_2.G_2$ wf
     - $S; G_1 \vdash v_1=v_2$
     - $S; G_1.v_1=v_2.G_2 \vdash ^i \neg k \, ok(b)$
     - $\Rightarrow S; G_1,G_2 \vdash ^i k \, ok(b)$

   Proof: By mutual induction over the structure of (A3) in each case. The interesting cases appeal to the admissibility of the entailment relation and weakening of type conversion judgment.

**Lemma 18** (Elimination of derivable type equality assumptions).

1. **Typing**
   - For all $S$, $G_1$, $G_2$, $X$, $t_1$, $t_2$, $e$, $t$, $m$,
     - $\vdash S; G_1.t_1=t_2.G_2.X$ wf
     - $S; G_1 \vdash t_1=t_2$
     - $S; G_1.t_1=t_2.G_2.X \vdash \neg \neg^m e : t$
     - $\Rightarrow S; G_1,G_2.X \vdash ^m e : t$

2. **Kinding**
   - For all $S$, $G_1$, $G_2$, $t_1$, $t_2$, $t$, $k$
     - $\vdash S; G_1.t_1=t_2.G_2$ wf
     - $S; G_1 \vdash t_1=t_2$
     - $S; G_1.t_1=t_2.G_2 \vdash t :: k$
     - $\Rightarrow S; G_1,G_2 \vdash t :: k$

3. **Kind well-formedness**
   - For all $S$, $G_1$, $G_2$, $t_1$, $t_2$, $k$, $i$, $b$
     - $\vdash S; G_1.t_1=t_2.G_2$ wf
     - $S; G_1 \vdash t_1=t_2$
     - $S; G_1.t_1=t_2.G_2 \vdash ^i \neg k \, ok(b)$
     - $\Rightarrow S; G_1,G_2 \vdash ^i k \, ok(b)$

   Proof: Similar to the previous lemma.

**Theorem 4** (Subject reduction).

For all $A$, $A'$, $G$, $X$, $X'$, $e$, $t$,

(A0) $A \Rightarrow (G; X)$

(A1) $\vdash S; G.X$ wf

(A2) $S; G.X \vdash ^m e : t$ where $m=\text{term_level}$ and $X' \subseteq \text{subseqeq } X$

(A3) $\exists \_S (A', e')$

$\Rightarrow \exists S' (G', X'')$

$A' \Rightarrow G'; X''$

$X'' = (X' \setminus \text{deadNames}(A')) \cup (\text{liveNames}(A') \setminus \text{liveNames}(A))$

exists $G''$, $G' = G$, $G''$

$S; G'.X'' \vdash ^m e' : t$

Proof: By induction on the structure of the typing derivation, (A2), generalizing over the environments $G$, $X$, $X'$. 

Draft – September 17, 2011
Case (T-X), (T-XA):
Impossible, since the premise of (T-X) requires x:t in G.
But, from (A0), we have y:t in G implies either
t_=_:unit{phi} and y not in FV(e), since y is chosen fresh

Case (T-Drop): From the induction hypothesis, using X' \subseteq X in the premise.

Case (T-Abs, T-D, T-TAbs, T-Val):
If v has kind * or P it is irreducible.

If v has kind A, it can reduce using (E-New-l), i.e.
(A; v) -->_S (A,1; v^1) for fresh l

We have (A,l) ==> G; X,l
And our goal is to show S;G;X,l |- v^1 : t
We use (T-BoxTerm), with T-Drop and A3 in the premise.

Case (T-Assume):
Inversion of (A3) gives an application of (E-Log)

(A; assume phi) -->_A (A,phi; ())
where (A,phi) ==> G,x:unit{phi}; X
And our goal is to show
S; G,x:unit{phi}; X |-^m () : x:unit{phi}
For the conclusion, we apply T-Val, we (T-D) in the premise with (T-Drop) to derive
S; G,x:unit{phi}; X |-^m () : unit
using the well-formedness of S.
Finally, we must show
S; G,x:unit{phi}, ()=() |= phi
Which follows from the admissibility of the entailment relation.

Case (T-App):
S; G; X1 |- v1 : ?(x:t1) -> t2 (A2.1)
S; G; X4 |- v2 : t1 (A2.2)
------------------------------------ [T-App]
S; G; X1,X4 |- v1 v2 : t2[v2/x]

We first use the canonical forms lemma to establish either
Sub-case (v1 = y): Impossible (as in (T-X)).
Sub-case (v1 = \x:t1.e): Then, inversion of (A3) gives us an application of (E-Beta).

----------------------------- [E-Beta]
\x:t1.e v2 -->_S e[v2/x]

Applying the inversion lemma INV4 to (A2.1), we get an application of (T-Abs) with
(A2.1.1) \[ S;G,x:t1;X,x |- e : t2 \]

Now, using (A2.1.1) and (A2.2), we apply the value substitution lemma, for \( X_1=X_1, X_2=x, X_3=., X_4=X_4 \), to derive the goal

\[
\exists X'. \\
(S;G;X |- e[v2/x] : t2[v2/x] \quad \text{(/ (PreGoal))} \\
\{(X'=X1 \quad \text{when (b<>A) /} \\
(X'=X1,X4 \quad \text{otherwise})\})
\]

For the goal, we need to show
\[
S;G;X1,X4 |- e[v2/x] : t2[v2/x]
\]

We consider the two cases, (b=A) and (b<>A).

In the first case, we have \( X'=X1,X4 \) and (PreGoal) has the desired shape.

Otherwise, we have \( X'=X1 \). For the conclusion we apply (T-Drop) first, and then conclude with (PreGoal)

Sub-case (v1 = \((\lambda x:t1.e)^l\)):

From the inversion lemma (INV9), we have \( X_1 = X_1',l \) and
\[
S;G;X1' |-^m \lambda x:t1.e : (?x:t1) -> t2 \quad \text{(A2.1')}\]

Since we have \( (A \Rightarrow G; X) \) and \( X_1, X_4 \) subset \( X \), we have \( 1 \) in \( A \)

From an inversion of (A2), we have an application of E-Destruct of the form
\[
(A,1; \ ((\lambda x:t1.e)^l) v2) \to_S (A,\hat{l}; \ \lambda x:t1.e v2)
\]

Our goal is to show
\[
S;G;X1',X4 |-^m \lambda x:t1.e v2 : t2 [v2/x]
\]

We use (T-App), and use (A2.1') and (A2.2) in the premises.

Case (T-Tapp): Similar to (T-App)

Case (T-Let):

\[
S;G;X1 |-^m e1 : t1 \quad \text{(A2.1)}\\nS;G;x:t1;X4,x |-^m e2 : t2 \quad \text{(A2.2)}\\nS;G |- t2 :: k \quad \text{(A2.3)}
\]

\[----------------------------- \text{[T-Let]}\]
\[
S;G;X1,X4 |- \text{let } x = e1 \text{ in } e2 : t2
\]

Sub-case (e1 is a non-value):

Inversion of (A2) gives (E-Ctx), with
\[
(A; e1) \to_S (A'; e1')
\]

From the induction hypothesis, we have:
\[
A' \Rightarrow G';X1'
\]

and \( S;G';X1' |-^m e1' : t1 \) \quad (IH1)

and \( X1' = (X1 \setminus \text{deadNames } A') \cup (\text{liveNames}(A') \setminus \text{liveNames}(A)) \) \quad (IH2)

and \( G' = G, G'' \) \quad (IH3)
For the conclusion, we aim to show

$$S;G';X1',X4 \vdash^m \text{let } x = e1' \text{ in } e2 : t2$$

We construct this using (T-Let), with (IH1) in the first premise.

For the second premise, we use (IH3) to show that $G'$ is an extension of $G$ and use weakening on (A2.2).

For the third premise, again, we appeal to weakening.

For the conclusion, we are required to show that $X1'$ is disjoint from $X4$.

But, from the premise, we have the $X1$ is disjoint from $X4$ and from (A0) we have $X4 \subseteq \text{liveNames}(A)$

From (IH2), we get that $X1'$ is disjoint from $X4$.

Sub-case (e1 is a value v1):

Inversion of (A2) gives either (E-Let) or (E-Newl).

Sub-sub-case (E-Let):

We apply the substitution lemma as in (T-App) to obtain

$$S;G,x:t1;X1,X4 \vdash^m e2 : t2[v1/x]$$

But, from (A2.3), we have $x \notin \text{FV}(t2)$

and so $t2[v1/x] = t2$, as required.

Sub-sub-case (E-Newl):

We have $(A; \text{let } x = v1 \text{ in } e2) \rightarrow_S (A,l; \text{let } x = v1^l \text{ in } e2)$

For the conclusion, we apply T-Let,

with $(X1,l)$ and T-BoxTerm first premise, followed by A2.1

And (A2.2) and (A2.3) in the remaining premises.

Case (T-Match):

$$S;G;X1 \vdash^m v : tv \quad \text{(A2.1)}$$

$$S;G,..;X' \vdash^m D \text{ as } xs : tp \quad \text{(A2.5)}$$

$$tv=t_{p[ts/as][vs/xs]} \quad \text{(A2.6)}$$

$$\vdots \quad \text{(A2.8)}$$

$$S;G;X1,X2 \vdash^m \text{match } v \text{ with } D \text{ as } xs \rightarrow e1 \text{ else } e2 : t \quad \text{[T-Match]}$$

From the fifth and sixth premises and inversion lemma (INV6), we get $t_v=T$ $ts$ $vs$

Applying the canonical forms lemma (CF3) to the first premise, we get three sub-cases.

Sub-case ($v$ is a variable y): Impossible, as in (T-X) and (T-XA).

Sub-case ($v = (D' \ ts' \ vs')^l$) for some $D'$ $ts'$ $vs'$ $l$: 

Draft – September 17, 2011
Applying inversion lemma (INV9), we get $X_1 = X_1', 1$ and $S; G; X_1' \vdash^m D' ts' vs' : tv$ (A2.1')

Inverting (A3), we get an application of (E-Kill) and apply (T-Match), with (A2.1')

Sub-case ($v = D' ts' vs'$) for some $D' ts' vs'$:

Inverting of (A3) gives us two further sub-cases.

Sub-sub-case (A3 is E-M2):

If a step is taken to the false branch via (E-Match), then, the conclusion follows using the last premise of (A1), with (T-Drop) to introduce unused affine assumptions, if any.

Sub-sub-case (A3 is E-M1):

We have $\sigma = ([v_1/x_1], \ldots, [v_n/x_n], [t_1/a_1], \ldots, [t_m/a_m])$

The seventh premise of A1 is:

(A2.7) $S; G, a_1::k_1, \ldots, a_m::k_m,$

$x_1::t_1'..x_n::t_n',$

$\ldots a_i=t_i, \ldots,

x_j=v_j, \ldots,$

$(D t_1..tn x_1..xn=D a_1..an v_1..vn); X_2, xs \vdash e_1: t$

where the equalities ($\ldots a_i=t_i \ldots x_j=v_j \ldots$) is computed by unifying the types of $v$ and the pattern.

Then, from the repeated application of the substitution lemma, we get

$S; G, \ldots t_i=t_i \ldots v_j=v_j \ldots (D t_1..tn v_1..vn = D t_1..tn v_1..vn); X \vdash e_1: \sigma(t),$

From repeated use of the derivable type and value-equality lemmas, we arrive at

$S; G; X_1, xs \vdash e_1: \sigma(t)$

From the substitution lemmas, we get

$S; G; X_1, X_4 \vdash e_1: \sigma(t)$

Finally, from well-kindred typings lemma applied to (A2.8), we get that $\sigma(t) = t$, since $FV(t)$ does not include any of the pattern variables.

Case (T-BoxTerm): Reduction occurs via (E-Kill) and we conclude with the second premise of (A2).

Case (T-BoxType): Impossible, since $m = term_level$

B. A well-typed embedding of RCF

This appendix describes RCF, the typed $\lambda$-calculus underlying the F7 typechecker (Bengtson et al. 2008) without concurrency, cryptographic kinds, and with restrictions on the use of recursive types. We show how to embed this fragment of RCF to F* via a type-directed translation, and show that our translation is a simulation.

B.1 Syntax of RCF

Syntax of Values, Expressions, and Types (with omitted constructs from full RCF grayed out)

<table>
<thead>
<tr>
<th>$a, b, c$</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x, y, z$</td>
<td>variable</td>
</tr>
<tr>
<td>$h ::= $</td>
<td>value constructor</td>
</tr>
<tr>
<td>$\text{inl}$</td>
<td>left constructor of sum type</td>
</tr>
<tr>
<td>$\text{inr}$</td>
<td>right constructor of sum type</td>
</tr>
</tbody>
</table>

Draft – September 17, 2011
fold

\( M, N \) ::= 

\( x \) 

value

\( () \) 

variable

\( \text{fun}(x) \to A \) 

function (scope of \( x \) is \( A \))

\( (M, N) \) 

pair

\( h M \) 

construction

\( A, B \) ::= 

\( M \) 

expression

\( M N \) 

application

\( \text{let } x = A \text{ in } B \) 

let (scope of \( x \) is \( B \))

\( \text{let } (x, y) = M \text{ in } A \) 

pair split (scope of \( x, y \) is \( A \))

\( \text{match } M \text{ with} \)

\( h x \to A \text{ else } B \) 

constructor match

\( A \cup B \) 

fork (record of dynamic assumption)

\( \text{assume } \phi \) 

assumption of formula \( \phi \)

\( \text{assert } \phi \) 

assertion of formula \( \phi \)

\( \{ \alpha \} \phi \) 

restriction (scope of \( \alpha \) is \( A \))

\( a? \) 

transmission of \( M \) on channel \( a \)

\( a! \) 

receive message off channel

\( \text{true} \triangleq \text{inl} () \) 

formula

\( \text{false} \triangleq \text{inr} () \) 

where \( p \) an n-ary predicate symbol

\( \phi ::= \text{True}, \text{False}, \phi \land \phi, \phi \rightarrow \phi \) 

type

\( \exists x, \phi, M = N, p(M_1, \ldots, M_n) \) 

type variable

\( \Pi x: T. U \) 

unit type

\( \Sigma x: T. U \) 

dependent function type (scope of \( x \) is \( U \))

\( T + U \) 

dependent pair type (scope of \( x \) is \( U \))

\( \mu \alpha T \) 

disjoint sum type

\( \{ x : T | \phi \} \) 

iso-recursive type (scope of \( \alpha \) is \( T \))

\( \text{refinement type (scope of } x \text{ is } \phi \))

\( E \vdash \cdot \) 

E is syntactically well-formed

\( E \vdash T \) 

in \( E \), type \( T \) is syntactically well-formed

\( E \vdash \phi \) 

formula \( \phi \) is derivable from \( E \)

\( E \vdash T <: U \) 

in \( E \), type \( T \) is a subtype of type \( U \)

\( E \vdash A : T \) 

in \( E \), expression \( A \) has type \( T \)

The key difference from full RCF is the omission of four rules in the subtyping judgment and in the handling of the recursive type constructor fold.

These restrictions on the typing rules of RCF come from our approach of translating built-in types like sums and products in RCF to inductive datatypes in \( F^* \). For example, we translate RCF’s union type \( T + U \) and its constructors inl and inr to instances of an inductive type in \( F^* \): \( \text{Sum} :: \star \Rightarrow \star \Rightarrow \star \) and its constructors \( \text{inl} :: \forall \alpha, \beta. \alpha \to \text{Sum} \\alpha \beta \) and \( \text{inr} :: \forall \alpha, \beta, \beta \to \text{Sum} \\alpha \beta \). We take a similar approach for RCF’s dependent pairs \( \Sigma x: T. U \) and iso-recursive types \( \mu \alpha T \).

RCF’s type system is equipped with subtyping relation on each of these three built-in type constructors. For example, RCF allows subtyping of the form \( (T + U) <: (T' + U') \), when \( T <: T' \) and \( U <: U' \). However, \( F^* \) does not permit subtyping on the indexes of an inductive type, i.e., \( \text{Sum} T U \) cannot in general be viewed as a subtype of \( \text{Sum} T' U' \), even when \( T <: T' \) and \( U <: U' \). For this reason, we consider the fragment of RCF that does not include subtyping on the built-in types \( (T + U), \Sigma x: T. U \), and \( \mu \alpha T \).

The limitation on structural subtyping for sums and pairs is not fundamental. To surmount this restriction, note that like RCF, \( F^* \) supports structural subtyping on function types. This means that we could translate sums and pairs to function types, rather than to inductive datatypes and recover structural subtyping. For example, we could translate RCF’s \( (T + U) \) to \( F^* \) as \( \forall \alpha. ([T] \to \alpha) \to ([U] \to \alpha) \to \alpha \), and structural subtyping on \( (T + U) \) naturally translates to structural subtyping on function types in \( F^* \). However, this approach requires using a kind of CPS translation, which we prefer to omit for simplicity.

A final difficulty arises from RCF’s treatment of the recursive type constructor fold. In RCF, fold is highly polymorphic and can be used to introduce any recursive type \( \mu \alpha T \), for any shape of \( T \) and at arbitrary scopes (e.g., \( T \) may even include local let-bound variables). Translating this kind of polymorphism requires the use of higher-kinded features intentionally omitted from \( F^* \), since we require \( F^* \) to be translatable to DCIL. Specifically, a canonical translation of the iso-recursive type \( \mu \alpha T \) to a single inductive datatype requires a type of the form below:

\[ \text{type} \quad \mu u : (\star \Rightarrow \star) \Rightarrow \star = \text{fold} : \forall(T :: \star \Rightarrow \star). T (\mu u T) \to \mu u T \]
Using such a type, we might translate the RCF type \( \mu \alpha \cdot (\text{unit} + (\sum_{\alpha} \text{int} \alpha)) \) to the inductive type \( \text{muT} \ (\lambda \alpha \cdot \star \ (\sum \text{unit} \ (\text{Pair} \ \text{int} \ \alpha))) \). However, types with kinds like \( \star \Rightarrow \star \Rightarrow \star \) are excluded from \( F^* \). Thus, our approach is to translate each use of \text{fold} that introduces \( \mu \alpha T \) in an RCF program to a data constructor of an inductive type corresponding to \( \mu \alpha T \). We require each of these instantiations of \text{fold} to be recorded in the RCF typing environment \( E \) and use these instantiations to direct the translation to \( F^* \).

The display below shows the syntax of typing environments extended to include binding \( \beta \), which record the available instantiations of \text{fold}. We also omit two kinds of bindings—one that is used for subtyping on recursive types (omitted in the translation, as described above), and another used to give types of channel names (omitted, since we ignore concurrency).

**Syntax of Typing Environments:**

\[
\begin{align*}
\mu & ::= \text{fold} : (T \{ \mu \alpha T / \alpha \}, \mu \alpha T) \\
\mu & ::= \text{fnfv} \ (\{ x \mid \alpha \} < : \alpha') \\
\mu & ::= \text{fold} : (T, T') = \{ \text{fold}_i \} \\
\end{align*}
\]

\( \text{instatiation of polymorphic recursive type constructor} \)

\( \text{environment entry} \)

\( \text{name of a typed channel} \)

\( \text{environment} \)

\( \text{dom}(\mu_1, \ldots, \mu_n, \mu_1, \ldots, \mu_m) = \text{dom}(\mu_1) \cup \cdots \cup \text{dom}(\mu_n) \cup \text{dom}(\mu_1) \cup \cdots \cup \text{dom}(\mu_m) \)

\( \text{dom}(\alpha) = \{ \alpha \} \)

\( \text{fold} : (T, T') = \{ \text{fold}_i \} \)

\( \text{dom}(\alpha < : \alpha') = \{ \alpha, \alpha' \} \)

\( \text{dom}(\alpha) = \{ \alpha \} \)

**Rules of Well-Formedness and Deduction:**

\[
\begin{align*}
(\text{Env Empty}) & : E \vdash T \quad \text{if } \text{types } \phi \subseteq \text{dom}(E) \\
(\text{Env Entry}) & : E \vdash \alpha \quad \text{parametric type} \\
(\text{Env Fold}) & : E, \mu \vdash \emptyset \\
(\text{Type}) & : E \vdash \alpha < : \alpha' \\
(\text{Subsum}) & : E \vdash T < : T' \\
(\text{Assume}) & : E \vdash \phi \in \text{dom}(E) \\
(\text{Assert}) & : E' \vdash \phi \\
(\text{Let}) & : E \vdash A : T, x : T \vdash B : U \\
& \quad x \notin \text{fv}(U) \\
& \quad E \vdash \text{let } x = A \text{ in } B : U \\
\end{align*}
\]

**General Rules for Values and Expressions:**

\[
\begin{align*}
(\text{Val Var}) & : E \vdash (x:T) \in E \\
(\text{Exp Subsum}) & : E \vdash A : T, E \vdash T < : T' \\
(\text{Exp Assume}) & : E \vdash \phi \in \text{dom}(E) \\
(\text{Exp Assert}) & : E' \vdash \phi \\
(\text{Exp Let}) & : E \vdash A : T, E, x : T \vdash B : U, x \notin \text{fv}(U) \\
& \quad E \vdash \text{let } x = A \text{ in } B : U \\
\end{align*}
\]

**Rules for Unit:**

\[
\begin{align*}
(\text{Sub Unit}) & : E \vdash \text{unit} < : \text{unit} \\
(\text{Val Unit}) & : E \vdash \text{(unit)} : \text{unit} \\
\end{align*}
\]

**Rules for Functions:**

\[
\begin{align*}
(\text{Sub Fun}) & : E \vdash T' < : T, E, x : T' \vdash U < : U' \\
& : E \vdash (\Pi x : T. U) < : (\Pi x : T'. U') \\
\end{align*}
\]

39
B.3 Type-directed translation

We suppose that all RCF formulas are well-typed.

Our translation begins by defining a signature $S_0$, including a set of pre-defined data types and constructors for RCF’s built-in types, and a polymorphic types for RCF’s variadic, untyped, predicate symbols.

$S_0$, a signature of pre-defined data types and data constructors

$$\begin{align*}
\text{unit} &::= * &\text{Vunit} &::= \text{unit} \\
\text{Sum} &::= * + * \Rightarrow * &\text{TSum} &::= \alpha \beta \\
&| \text{inl} : \alpha \rightarrow \text{TSum} \alpha \beta \\
&| \text{inr} : \beta \rightarrow \text{TSum} \alpha \beta \\
\text{DepPair} &::= (\alpha::*) \Rightarrow (\alpha \Rightarrow *) \Rightarrow * &\text{Tuple} &::= x : \alpha \rightarrow \beta x \rightarrow \text{DepPair} \alpha \beta \\
\text{p_1} &::= (\alpha::*) \Rightarrow \alpha \Rightarrow P
\end{align*}$$
\p_n :: (\alpha::*) \Rightarrow \alpha \Rightarrow P
\And :: P \Rightarrow P \Rightarrow P
\Not :: P \Rightarrow P
\Eq :: (\alpha::*) \Rightarrow \alpha \Rightarrow \alpha \Rightarrow P
\True :: P
\False :: P

The translation involves 5 mutually inductive judgments and a final top-level judgment that translates and RCF configuration into a F* expression and a log of accumulated assumptions.

**Translation Judgments:**

\begin{align*}
\vdash [E] = S; \Gamma & \quad \text{E is translated to a signature } S \text{ and environment } \Gamma \\
E \vdash [T] = t & \quad \text{A type } T \text{ in RCF is translated to a type } t \\
E \vdash \phi = t & \quad \text{A formula } \phi \text{ in RCF is translated to a type } t \\
E \vdash [M] : T = v & \quad \text{A value } M \text{ of type } T \text{ in RCF is translated to a value } v \\
E \vdash [A] : T = e & \quad \text{An expression } A \text{ of type } T \text{ in RCF is translated to an expression } e \\
E \vdash [A] = (\phi'; e) & \quad \text{An RCF configuration } A \text{ is translated to a log and an expression}
\end{align*}

**Lemma 19 (Admissibility of the RCF entailment relation).** RCF’s first-order logic entailment relation, extended with the requirement that all derivable formulas are well-typed, is admissible in F*.

**Theorem 5 (Well-typed translation).**

Given \( E \vdash [A] = (\phi'; e) \) and \( \vdash [E] = S; \Gamma \) and \( E \vdash \phi \).

Then \( S; \Gamma \vdash e : t \), where \( E \vdash [T] = t \).

Additionally \( A \rightarrow A' \) if and only if \( (\phi'; e) \rightarrow_S (\phi'; e') \) and \( E \vdash [A'] = (\phi'; e') \).
\textbf{Figure 6.} Type-directed translation of values, expressions, types, environments