Self-certification: Bootstrapping certified typecheckers in F* with Coq

Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, Juan Chen

Self-Certification

Bootstrapping Certified Typecheckers in F* with Coq

Pierre-Yves Strub
MSR-INRIA
pierre-yves@strub.nu

Nikhil Swamy
Microsoft Research
nswamy@microsoft.com

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Juan Chen
Microsoft Research
juanchen@microsoft.com

Abstract

Well-established dependently-typed languages like Agda and Coq provide reliable ways to build and check formal proofs. Several other dependently-typed languages such as Aura, ATS, Cayenne, Epigram, F*, F7, Fine, Guru, PCML5, and Ur also explore reliable ways to develop and verify programs. All these languages shine in their own regard, but their implementations do not themselves enjoy the degree of safety provided by machine-checked verification.

We propose a general technique called *self-certification* that allows a typechecker for a suitably expressive language to be certified for correctness. We have implemented this technique for F*, a dependently typed language on the .NET platform. Self-certification involves implementing a typechecker for F* in F*, while using all the conveniences F* provides for the compiler-writer (e.g., partiality, effects, implicit conversions, proof automation, libraries). This typechecker is given a specification (in F*) strong enough to ensure that it computes valid typing derivations. We obtain a typing derivation for the core typechecker by running it on itself, and we export it to Coq as a type-derivation certificate. By typechecking this derivation (in Coq) and applying the F* metatheory (also mechanized in Coq), we conclude that our type checker is correct. Once certified in this manner, the F* typechecker is emancipated from Coq.

Self-certification leads to an efficient certification scheme—we no longer depend on verifying certificates in Coq—as well as a more broadly applicable one. For instance, the self-certified F* checker is suitable for use in adversarial settings where Coq is not intended for use, such as run-time certification of mobile code.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques.

General Terms Verification, Languages.

Keywords Certification, Dependent Types, Refinement Types.

1. Introduction

Well-established dependently-typed languages like Agda (Norell 2007) and Coq (2011) provide highly reliable ways to build and check formal proofs. Researchers have also developed many other dependently-typed programming languages, with different design trade-offs, such as Cayenne (Augustsson 1998), ATS (Xi 2003), Epigram (McBride 2004), Aura (Jia et al. 2008), Fable (Swamy et al. 2008), F7 (Bengtson et al. 2008), Guru (Stump et al. 2008), Fine (Swamy et al. 2010), F* (Swamy et al. 2011), PCML5 (Avijit et al. 2010), Ur (Chlipala 2010b), etc., and more are in the works, such as Trellys (Casinghino et al. 2011).

All these languages shine in their own regard, but their implementations do not themselves enjoy the degree of formal safety their authors advocate. Even with full-fledged mechanized theories in Coq, for instance, new language implementations are often *ad hoc* and disconnected from the formalized systems. One may choose to write a core typechecker for one of these languages within Coq, and possibly extract it to another language with a similar semantics (Letouzey 2008), but this involves programming compiler internals in an impoverished language of pure total functions, and a correspondingly large interactive proof obligation. Besides, Coq is not a panacea for all automated verification tasks. For example, when dealing with hostile code at run-time, more specialized, defensive verifiers may be simpler and safer. They may, for instance, fit on a smartcard or address non-functional aspects of the Coq checker, such as parsing, formatting, and error handling.

We aspire to formal certification within general-purpose programming languages and systems. To this end, we propose a bootstrapping path that allows the implementation of a core typechecker (for such a language) to be mechanically certified for correctness using an external, independent tool such as Coq.

Traditionally, bootstrapping refers to the process of compiling a compiler with itself (Hart and Levin 1962) and, more usefully, of compiling a series of increasingly complex variants of a compiler with themselves, until a fixpoint is reached (see e.g. Appel 1994). The main benefit of bootstrapping is to reduce dependencies on external languages and tools, and thus provide flexibility as the compiler evolves. For instance, one may start with a core compiler for a subset of the language on a simple architecture, then gradually add language features, compiler optimizations, and target architectures. Another benefit is to provide reasonably complex sample code for testing, and thus gain confidence in the resulting compiler. At first sight, the benefits of bootstrapping for formal verification are limited, due to incompleteness results. Instead, a common architecture is to separate a trusted, minimal *core verifier* that just checks the proofs produced (and often already verified) by more complex, extensible, interactive tools.

The main contribution of this paper is a new, general method named *self-certification* (as opposed to self-validation; see e.g. Barras 2010 for Coq) that allows a typechecker for a suitably-expressive typed language to be certified for correctness. In combination with a mechanized theory that ensures type safety, self-certification guarantees that all programs accepted by the typechecker will be safe, without further use of external formal tools.

We implement this technique for F*, a dependently-typed programming language for the .NET platform primarily used to verify the security of distributed programs and protocols. Briefly, to self-certify F*, we carry out the following tasks:

(1) We program a core typechecker for F* in F*, while making use of all of the conveniences that F* provides for the compiler-writer (e.g., partiality, effects, implicit conversions, proof automation, libraries, etc.). Our typechecker takes a program and its candidate type and returns a detailed typing derivation—or reports an error.

(2) We equip it with a specification (using F⋆ types) strong enough to ensure that it returns (at most) valid typing derivations. Thus, informally, our typechecker is correct by typing.

(3) We run our typechecker on itself: we thus produce a (large) certificate for its own typing, which we export to Coq.

(4) We import and typecheck this certificate within Coq and apply the F⋆ metatheory (also mechanized in Coq) to formally deduce first that our typechecker is correct, and then that its termination implies the safety of any program it accepts.

(5) Once certified in this manner, the F⋆ typechecker is emancipated from Coq. We can use it independently of Coq, with essentially the same formal guarantees for the programs it accepts. Going further, we can erase the production of typing-derivation certificates, treated as irrelevant proofs.

Self-certification leads to an efficient certification scheme—after bootstrapping, we no longer depend on verifying certificates in Coq—as well as a more broadly applicable one. The self-certified F⋆ checker is suitable in adversarial settings where Coq is not intended for use, for instance to verify and then load mobile code at runtime. We plan to use F⋆ to certify reference implementations of security-critical infrastructure, such as TLS (Dierks and Rescorla 2008). Meanwhile, we may already certify properties previously checked by typing for security protocols coded in F7 or F⋆ for secure multiparty sessions (Bhargavan et al. 2009), identity management systems (Bhargavan et al. 2010), and data protection APIs (Acar et al. 2010) by running our self-certified typechecker on programs much larger than itself.

An F⋆ primer We briefly review F⋆, a variant of ML with a similar syntax and dynamic semantics but with a more expressive type system. F⋆ enables general-purpose programming with recursion and effects; it has libraries for concurrency, networking, cryptography, and interoperability with other .NET languages. After typechecking, F⋆ is compiled to .NET bytecode, with runtime support for proof-carrying code. F⋆ has been applied mostly to secure distributed programming; it subsumes F7 and Fine, two prior languages using types for security verification. It has been used to program and verify more than 30,000 lines of code, including security protocols, web browser extensions, and distributed applications. Its main typechecker, compiler, and runtime support are coded in F#.

The main technical novelty of F⋆ is a kind system to keep track of different fragments of the language and control their interaction. By default, program terms (with kind $\star$) may have arbitrary effects. Within these programs, proof terms (with kind *P*) remain pure and terminating, and thereby provide a logically consistent core. Besides, affine resources (with kind *A*) may be used for modular reasoning on stateful properties. Finally, ghost terms (with kind *E*) may be used only in specifications, to selectively control the erasure of proof terms, when the proofs are irrelevant or unavailable (for instance, when calling legacy code or using cryptography).

F⋆ is also parametric in the logic used to describe program properties and proofs. This provides support for custom logics (e.g., for expressing authorization policies) and enables integration with SMT solvers, which is important for proof automation when developing large programs. Thus, F⋆ calls Z3 (de Moura and Bjørner 2008) during typechecking, to prove logical properties specified as type refinements and to decide type equivalences.

The type system of F⋆ has 4 main mutually-recursive judgments and 67 rules. Its theory has been formalized in Coq, showing substitutivity and subject reduction. (By design, F⋆ provides only value dependencies and is inadequate for this sort of interactive theorem proving.)

Typechecking F⋆ in F⋆ Figure 1 outlines our method for self-certifying F⋆, which involves developments in F#, in F⋆, and in Coq. The rest of the paper provides a more precise, syntactic account of the developments in F⋆ and Coq.

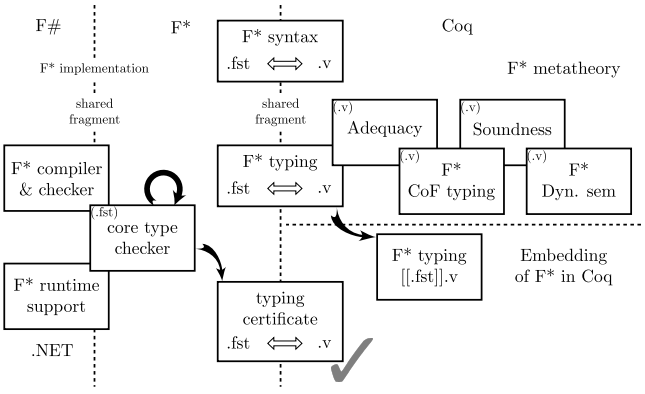

Figure 1. Self-Certification for an F⋆ checker written in F⋆

Its central component is our new core typechecker, programmed in F⋆ using mutual recursion and exceptions for error-handling, state for name generation, as well as external .NET functions for I/O. Including the type definitions for the language and its type system, as well as basic library functions (mostly operations on lists and other functional data structures), our candidate certified program has 5,139 LOCs.

We adapt the F⋆ compiler and main typechecker (in F#, on the left-hand-side) so that, after parsing and typing a given source program, it passes a typed abstract syntax tree for core typechecking certification, using a datatype shared between F# and F⋆. This F# code is significantly larger (35,000 LOCs). Thus, for a given F⋆ program, we can call the F⋆ compiler and obtain a typechecking certificate for (a desugared variant of) this program. In particular, we can call the core typechecker on itself and—after fixing any errors—produce its own typechecking certificate. Core typechecking is fast (a few minutes), but exporting the self-typechecking certificate in a format suitable for Coq verification then takes several hours, and still yields 7 GB of Coq source files.

Two theories of F⋆ in Coq To relate the F⋆ and Coq developments, we crucially rely on a well-identified fragment of F⋆ (tracked by its *P* kind) that coincides with a fragment of the inductive types of Coq. To this end, we build tools that take F⋆ types and values and emit them in the concrete syntax of Coq. In the picture, we label these definitions with both a '.fst' for F⋆ and a '.v' for Coq. Thus, the definitions of the F⋆ syntax, and (with some care) its core type system, are specified using value-dependent mutually-recursive inductive types in the shared fragment. This enables us, in particular, to emit a typing certificate in F⋆ and typecheck it as a valid typing certificate in Coq. We intend to formally reason about our F⋆ typechecker running on an F⋆ program within Coq, so we emit surface definitions (for the typechecker and its type), embedded definitions (for its program parameter and its type, both typed as surface F⋆ values), and embedding functions between the two. Conversely, we prove that, at least for the inductive types we use to represent typing derivations, the existence of an embedded value implies the existence of a surface value (Theorem 4).

Our shared definition of the F⋆ type system is convenient for programming and typechecking, but not for developing its metatheory. Instead, we use another, cofinite definition of the type system (based on the same F⋆ syntax), we prove subject-reduction for the cofinite system (Theorem 2), and we prove the adequacy of the shared system with regards to the cofinite system (Theorem 1).

With these results in hand, we can complete our bootstrapping within Coq. We load and typecheck the 7 GB self-typing certificate—despite many optimizations, this task still takes several

machine-weeks—then we apply adequacy and subject reduction on complete runs of the typechecker applied to its own embedding as an F* term and its candidate type. This yields a valid embedded typing certificate for the embedded typechecker. Finally, we apply our Theorem 4 relating embedded and surface values and obtain a valid proof of its typing (Theorem 5).

Limitations Our path to certification depends on the correctness of the following components:

(1) The Coq kernel. We use a custom build of Coq with two extensions: a compact representation of strings for typechecking our typing certificates, and SSREFLECT for the metatheory and all interactive proofs.

(2) Some of our tools for parsing, translating, and embedding specifications between F* and Coq. Except for the concrete syntax, these are almost identities. The situation is similar to the use of notations within Coq—careful printing and manual inspection is required to interpret the formal theorem statements (Pollack 1998), although their proofs can be safely ignored.

(3) Z3. We see the certification of SMT deductions as an important, independent issue. We plan to adapt prior work on predecessors of F* (Chen et al. 2010) to extract valid F* proof terms from Z3 deductions. We may also rely on other verified SMT solvers (Armand et al. 2010; Moskal 2008).

(4) The F* back-end compiler and its .NET runtime with regards to our formal dynamic semantics. This trust assumption is rather large. On the other hand, we run our certified programs on the same platform, so we may as well trust it for typechecking. Still, in future work, it would be interesting to compile and run F* using a smaller TCB.

We should also note that we certify only the *partial* correctness of our typechecker. This is consistent with our usage of F*, which already relies on Z3, an incomplete prover. Pragmatically, bootstrapping and many other F* examples provide adequate coverage for completeness.

Related Work To our knowledge, our self-certification approach for bootstrapping verified typecheckers is new.

Self-verification Milawa/ACL2 (Davis 2009) provides an interesting example of verification bootstrapping for a first-order proof system, starting from a minimal checker (coded in Lisp) and gradually adding self-verified features, but does not formalize or certify its core checker. Jitawa (Myreen and Davis 2011) further provides a concrete x86 runtime for Milawa, verified using HOL4.

Coq implementation with extraction Several large developments of certified programs have been achieved using Gallina, the specification language of Coq, with or without extraction, notably for certified compilers (Chlipala 2010a; Leroy 2011). In comparison, our method does not rely on extraction.

Languages embedded in Coq Another method to obtain a reliable implementation is to embed a language within a trusted verifier. Hence, Nanevski et al. (2008) develop Ynot, a language for verified imperative programming, as a Coq library. While this method is arguably simpler than building a separate self-certified compiler from scratch, it also retains some of the limitations of its host language. For example, using Ynot to certify hostile code carries the same pitfalls as a direct deployment of Coq. Additionally, while Ynot, like F*, provides support for recursion and effects, it retains Coq's interactive proof style with explicit conversions.

Reflection Many recent works show how to efficiently verify large certificates in Coq using reflection (Armand et al. 2010; Keller and Werner 2010). We use reflection in our metatheory. However, as explained in §7, the soundness of self-certification limits the extent to which reflection can be used when checking certificates. Nevertheless, §7.2 describes how even our limited use of reflection yields significant reductions in the size of certificates.

Contents §2 introduces our method and main notations by certifying an F* typechecker for the simply-typed λ-calculus (STLC). §3 reviews the main features of F* and its type system. §4 presents our two theories of F* in Coq and discusses the sharing of inductive definitions and values between F* and Coq. §5 continues our example and shows how to certify an STLC typechecker in F*. §6 presents the main development of the paper, self-certifying the core typechecker for F*. §7 provides implementation details and experimental results. §8 concludes and discusses future work. A link to the latest core typechecker, its specification, and its meta-theory in Coq is available from the F* website at `http://research.microsoft.com/fstar`.

2. Warming up: a certifying typechecker for the simply-typed λ-calculus

We begin by introducing self-certification and illustrate its key elements in a simple setting: the problem of formally certifying the type safety of STLC programs. We describe three solutions with increasing degrees of emancipation:

- We may construct typing certificates for each λ-term in an *ad hoc* manner, for instance by programming a typechecker for STLC in ML, and then verify the typing certificates it produces using Coq. This approach is illustrated in Figure 2 and discussed in the remainder of this section.
- We may implement a similar STLC typechecker in F*, and give this program an F* specification that ensures it only constructs valid STLC typing derivations. We then build a single F* typing certificate for the typechecker program and check this certificate in Coq. We run Coq just once, rather than for each λ-term we wish to typecheck. We discuss this approach in §5.
- We may implement an F* core typechecker in F*, and give this program a specification that ensures it only constructs valid F* typing derivations. We then build a certificate for the core typechecker and check this certificate in Coq. This allows us to run Coq once for all F* programs. Once certified, we may run our core typechecker to certify any other program, including our STLC checker, without running Coq anymore. We discuss this approach in §6.

2.1 Formalizing STLC in Coq

A first step to certify the safety of STLC terms is to build a formal model of STLC in Coq. This model involves defining a specification of STLC syntax and typing judgments. We give below our sample Coq definitions for the syntax of STLC.

```
Definition Lbvar := string.
Inductive Ltyp : Type :=
| LTyp_unit : Ltyp
| LTyp_fun : Ltyp → Ltyp → Ltyp.

Inductive Lexpr : Type :=
| LExpr_unit : Lexpr
| LExpr_var : Lbvar → Lexpr
| LExpr_fun : Lbvar → Ltyp → Lexpr → Lexpr
| LExpr_app : Lexpr → Lexpr → Lexpr.
Definition Lenv := seq (Lbvar * Ltyp).
```

These inductive types are just ML datatypes, so we can share our specification of the syntax between the formal development in Coq and our STLC implementation in ML. As such, these definitions

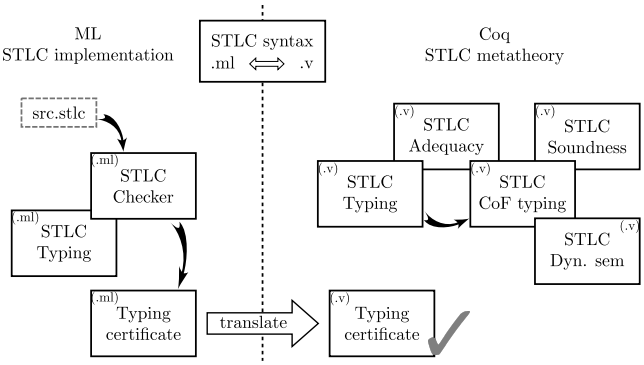

Figure 2. A traditional certifying typechecker for the STLC

are depicted in Figure 2 by the box labeled "STLC syntax[.ml|.v]". When sharing such definitions, we prove a theorem that for every value of the shared type in one language, there exists a corresponding value of the shared type in the other language.

Next, we define the typing judgments of the STLC as an inductive type in Coq—we give below the rule for functions ($\lambda x{:}t.e$).

```
Inductive LTyping : Lenv →Lexpr →Ltyp →Type := ...
| LTyping_fun : forall g x t e t',
      LTyping ((x,t)::g) e t'
    →LTyping g (LExpr_fun x t e) (LTyp_fun t t')
```

The inductive type `LTyping` is the reference specification of the type system, depicted by the box labeled "STLC typing.v". Our definitions are given in the style used in the rest of the paper (notably for the F* type system), which is convenient for use in the implementation of a typechecker in a real compiler. For instance, we use concrete strings to represent STLC variables.

Although natural for programming, the reference type system is not ideal for conducting metatheory. To this end, we give a second specification of the type system that uses cofinite quantification in a (named) locally nameless style to represent variables (Aydemir et al. 2008)—see also §4.2. This specification is depicted by the box labeled "STLC CoF typing.v". We also prove an adequacy result in Coq (the box "STLC Adequacy.v") that relates the two specifications. Adequacy states that, for every valid derivation in the reference system, there is a corresponding derivation in the cofinite system. We may then develop a standard cofinite theory for STLC, including a dynamic semantics and a type safety theorem.

2.2 Programming a certifying STLC typechecker in ML

Given a metatheory in Coq, a common approach to building a certified typechecker would be to implement it as a Coq function and to prove that it constructs valid typing derivations. The function can then be extracted to ML and run. This task is trivial for STLC, but for more complex languages (e.g., ML itself, or F*), implementing the typechecker within Coq can be quite challenging: Coq functions must be pure and total, whereas typechecking routinely uses general recursion; state for environments, unification, and symbol generation; and exceptions for error handling. Additionally, for this approach to work, one must prove the typechecking function correct in Coq, which involves interactive proofs on a large program.

To sidestep these difficulties, we implement the type checker in a general-purpose programming language with first-class support for features like effects that are hard to handle in Coq. As shown in the following sections, we can both use these features for programming our typechecker and formally ensure that it is correct.

To begin with, we program the typechecker in ML as a partial function stlc_typing given the signature below.

```
type LTypingML = ...
 | LTyping_fun of Lenv * string * Ltyp * Lexpr * Ltyp * LTypingML
val stlc_typing: Lenv →Lexpr →Ltyp →LTypingML
```

The type LTypingML shown above is an ML data type for representing STLC typing derivations. This type is similar but much less precise than our reference specification `LTyping`—there exist values in LTypingML that do not correspond to valid reference typings. Using just this ML type as a specification, we have no way of showing that `stlc_typing` computes valid typing derivations. Still, whenever the ML term stlc_typing g e t computes a derivation d:LTypingML, we can attempt to translate d into a Coq value of type `LTyping`. Within Coq, if d typechecks, then from the rest of the metatheory we may conclude that the STLC program e is type safe.

We need not trust the programs that produce the derivation d and translate it from ML to Coq—all we care about is that the resulting certificate is a valid typing for the source term e. We do, however, care that the translated certificate is a valid typing for the program e, and not some other term. However, since the source term is a type Lexpr shared between ML and Coq, the translation from the abstract syntax of our ML implementation to Coq is simply the identity.

The method we have described so far is standard for a certifying compiler—many such instances exist in the literature (Colby et al. 2000; Morrisett et al. 1999). However, it requires running Coq on every compilation run. It is preferable to certify the STLC checker itself, that is, prove that any certificate it produces will always be typeable in Coq. Given such a proof, we would not have to run Coq on each compilation, and, furthermore, unless there were some other need for it, we need not produce the derivation value. To achieve this, §5 and §6 illustrate how to program the typechecker in F*, a call-by-value language (like ML) with higher-order functions, state, effects, and recursion, but with a verification system based on dependent typing that approaches Coq in expressive power.

3. A review of F*

Before proceeding towards self-certification, we briefly present the syntax and semantics of F*; we refer to Swamy et al. (2011) for a complete presentation and examples. The syntax is shown below.

Values, expressions, types, kinds, signatures, and environments

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

Values include variables, λ-abstractions over values and types, and fully applied n-ary data constructors. The value v^ℓ is a technical device used to prove the soundness of affine typing and can be ignored for our present purposes. The notation $\bar{a}$ stands for a finite sequence of elements $a_1, \ldots, a_n$ for arbitrary n. F* adopts a (partially) let-normalized view of the expression language e, in particular requiring function arguments to always be values—this is convenient when using value-dependent types, since it ensures that expressions never escape into the level of types. The only other non-standard expression form is **assume** ϕ, which is used in the semantics of ghost refinements and is not discussed further—we do not make use of it in our core typechecker.

Types are ranged over by meta-variables t and ϕ—we use ϕ for types that stand for logical formulas. Types include variables α, constants T, dependent functions ranging over values whose domain may be values ($x{:}t \to t'$) or types ($\forall\alpha{::}\kappa.t$), types applied to

values ($t\ v$) and to types ($t\ t'$), type-level functions from values or types to types ($\lambda x{:}t.t'$ and $\Lambda\alpha{::}\kappa.t$), ghost refinements $x{:}t\{\phi\}$, and finally coercions to affine types $¡t$. This last modal operator serves to qualify the type of a closure that captures an affine assumption.

Kinds κ include the four base kinds $\star$, P, A, and E discussed previously—F* distinguishes the first three of these as *concrete kinds*, since E is used to characterize *erasable* specifications. As at the type level, F* has kinds for dependent function spaces whose range are types and whose domain may be either values ($x{:}t \Rightarrow \kappa$) or types ($\alpha{::}\kappa \Rightarrow \kappa'$). Stratifying the language into terms, types, and kinds allows F* to place key restrictions (discussed below) that facilitate automated verification, and to compile efficiently to .NET. However, stratification does come at a cost—several pieces of technical machinery are replicated across the levels.

Signatures S are finite lists of mutually recursive datatypes. Each definition $\overline{T{::}\kappa}\{\overline{D{:}t}\}$ introduces a set of type constants $T_1{::}\kappa_1, \ldots, T_n{::}\kappa_n$ and all their data constructors $D_1{:}t_1, \ldots, D_m{:}t_m$. We do not need a fixpoint form in the expression language since these types allow us to encode recursive functions. The addition of mutual recursion is an enhancement over our prior formalization of F*, where it was left out for simplicity. Here, we make heavy use of mutual recursion in the formalization of the type system of F* in F* and so include it in our core theory.

A characteristic feature of the F* type system is that the kind of a type indicates whether or not values of that type are pure. In particular, functions in the universe of P-kinded types are guaranteed to be total, whereas those in the $\star$-universe may diverge or have other effects. This allows us to embed a logically consistent sublanguage (namely the P-fragment) within F*. (The meaning of the other two kinds is beyond scope for this paper.) To ensure 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. In addition, the kinding rules for function arrows are such that the kind of a function type is determined by the kind of its co-domain, i.e., the kind of $x{:}t \to t'$ is the kind of t'. Intuitively, since F* is a call-by-value language, the domain of a function consists of fully evaluated terms, and hence it is the co-domain that determines whether or not a function is total.

These and other restrictions (e.g., on cross-universe elimination) allow us to place the P-fragment of F* in close correspondence with Coq, where inductives are also positive. Specifically, P-kinded types in F* correspond to a fragment of the `Type` universe in Coq rather than `Prop`, i.e., P-terms in F* need not be computationally irrelevant. We exploit (a limited version of) this correspondence in an essential way in the main development of the paper, via our data embedding theorem (Theorem 4).

Typing judgments The F* type system contains several mutually-recursive judgments. The judgments are given with respect to typing environments Γ that track in-scope value variables (x with type t), type variables (α with kind κ), and equivalences between values ($v_1 = v_2$) and types ($t_1 = t_2$) introduced when checking **match** expressions. These equalities enable implicit conversions, a distinctive feature of F* with regard to a system like Coq, where all type conversions must be performed by equality-witnessing coercions.

The four central judgments are as follows: (1) well-formedness of kinds: $S;\Gamma \vdash \kappa\ \mathrm{ok}(b)$, where the well-formedness is indexed by a base kind b; (2) kinding of types: $S;\Gamma \vdash t :: \kappa$; (3) typing for values: $S;\Gamma;X \vdash^m v : t$, and (4) typing for expressions $S;\Gamma;X \vdash^m e : t$. Note, unlike our presentation of the STLC, F* separates value and expression typing, which are the two most interesting judgments. They state that a value v (resp. expression e) has type t, under signature S, environment Γ, and an affine environment $X ::= \cdot \mid \ell \mid x \mid X,X'$, where X,X' denotes disjoint union of sets of names. The context X represents a set of available affine assumptions, and usual context splitting rules apply to X when typing the subterms of an expression. The index m on the turnstile represents two possible modes that distinguish the typing of terms that appear at the type level from those that appear at the term level—the former are subject to less stringent restrictions on affinity.

We give only two sample rules, for value-typing λ-abstractions and for expression-typing applications, which we also use for examples in the next sections.

$$\text{WFV-Fun}\ \frac{S;\Gamma \vdash t{::}c \qquad S;\Gamma,x{:}t;X,x \vdash^m e : t'}{S;\Gamma;X \vdash^m \lambda x{:}t.e : Q(X,x{:}t \to t')}$$

$$\text{WFE-App}\ \frac{S;\Gamma;X \vdash^m e :?x{:}t \to t' \qquad S;\Gamma;X' \vdash^m v : t}{S;\Gamma;X,X' \vdash^m e\ v : t'[v/x]}$$

In WFV-Fun, the first premise requires that the argument type t have a concrete kind. The second premise types the body expression e in a context and an affine environment both extended with x (so that x may be used once if t is affine, e.g. c is A). Finally, the introduced function type is tagged with the affine modality (using $Q(X,t) = ¡t$, when X is non-empty, and t otherwise) if the function closure captures an affine assumption. In WFE-App, the two premises type e and v as a function and its argument, respectively, using disjoint parts X, X' of the affine environment. We write $?x{:}t \to t'$ to range over the types $x{:}t \to t'$ and $¡x{:}t \to t'$, thereby capturing the elimination of both standard and affine functions (respectively) in a single rule. Finally, the result type is obtained by substituting v for x in t'.

In addition to these four judgments, F* has several auxiliary judgments, including judgments for subkinding, subtyping, well-formedness of signatures and environments, and the like. We elide these here, since they do not play a central role in the remainder of our presentation. However, one judgment is worth mentioning: logic derivability.

The F* type system is parametrized by an external logic, where derivability in this logic is characterized by the judgment $S;\Gamma \models \phi$. The introduction of ghost refinements (as well as certain implicit conversions) are governed by this judgment. In effect, the value typing $S;\Gamma;X \vdash^m v : x{:}t\{\phi\}$ requires deriving $S;\Gamma \models \phi[v/x]$ in the external logic. As such, F* places various admissibility constraints on this judgment (e.g., that it be closed under substitution; that it be a congruence on equality etc.). The soundness of the F* type system is modulo the soundness of the logic meeting these conditions, and we rely on this for our embedding theorem in §4.3. This approach gives us the flexibility to use F* with an external theorem prover, like Z3, where we often instantiate the external logic in question to first-order logic with theories. However, instead of trusting the external logic, it is also possible to extract proof terms from a decision procedure from the logic and have them checked in pure F*. This is an approach followed by Chen et al. (2010) in the context of Fine, a predecessor of F*.

4. Formalizing F* in Coq

A prerequisite for self-certifying F*, and indeed certifying any F* program such as our STLC checker, is to formalize F* in Coq. There are three aspects to this formalism. First, we develop a reference specification of the F* type system. Next, we develop a cofinite version of the reference type system, prove an adequacy theorem relating the two, and prove the F* type system sound. The third and final aspect involves developing metatheory (currently by hand) that relates F* and Coq in a way that justifies sharing certain inductive types between the two systems. We consider each of these aspects in turn, returning (in §4.3) to the setting of our STLC checker for illustrative examples.

4.1 A reference specification of the F⋆ type system in Coq

Our reference specification begins with a model of F⋆ syntax. We show a fragment of this syntax below, starting with type definitions for the various kinds of names used in our formalization. We have `iname` for inductive type names; `cname` for data constructors; `vlname` for free value names; `tyname` for free type names; and `bvar` for bound (value or type) names. Each of these is represented using a string type with decidable equality, the same representation as in the F⋆ implementation of the F⋆ typechecker.

```
Definition iname := String.
Definition cname := String.
Definition vlname := String.
Definition tyname := String.
Definition bvar := String.
```

We have four mutually inductive types, for kinds, types, values and expressions. We list a representative fragment of these types below, starting with the variant type `gvar` that we use to distinguish bound names and free names in the syntax. We have the four base kinds, where `BK_Comp` corresponds to the ⋆-kind of §3.

```
Inductive gvar (T : Type) : Type :=
  GV_Bound : bvar → gvar T | GV_Free : T → gvar T.

Inductive basekind : Type :=
  BK_Comp | BK_Prop | BK_Erase | BK_Afn.

Inductive kind : Type :=
| K_Base : basekind → kind
| K_ProdK : bvar → kind → kind → kind
| K_ProdT : bvar → typ → kind → kind
with typ : Type := ...
| T_Var : gvar tyname → typ
| T_Ind : iname → typ
| T_VApp : typ → value → typ
| T_Prod : bvar → typ → typ → typ
| T_Ref : bvar → typ → typ → typ
| T_Ascribe : typ → kind → typ
with value : Type := ...
| V_Var : gvar vlname → value
| V_Fun : bvar → typ → expr → value
| V_Const : cname → seq typ → seq value → value
| V_Ascribe : value → typ → value
with expr : Type := ...
| E_Value : value → expr
| E_App : expr → value → expr
| E_Ascribe : expr → typ → expr.
```

The abstract syntax closely follows the informal syntax of §3. The one addition is the presence of type- and kind-ascription forms at each level, e.g., we use `T_Ascribe` to give a type a kind, which make (core) typechecking more syntax-directed.

The reference type system is formalized as a 4-way mutually recursive inductive type. We show the signatures of these types below: `wfK` is the judgment for well-formedness of kinds, `wfT` is the kinding judgment for types, `value_ty` is value typing, and `expr_ty` is expression typing.

```
Inductive wfK : icompartment → environment → bindings
      → kind → basekind → Type := ...
with wfT : icompartment → environment → bindings
      → typ → kind → Type := ...
with value_ty : icompartment → environment → bindings
      → seq bvar → mode → value → typ → Type := ...
with expr_ty : icompartment → environment → bindings
      → seq bvar → mode → expr → typ → Type := ... .
```

Each judgment is given with respect to an environment that contains three compartments for binding various kind of names.

First, corresponding to signature S in §3, an `icompartment` is a sequence of records, one for each family of mutually recursive inductive types. Each record contains the names of the defined type constants, their kinds, and their constructors.

```
Record constructor : Type := mkConstructor
  { ct_name : cname; ct_type : typ }.
Record ikind : Type := mkIKind
  { it_name : iname; it_kind : kind }.
Record inductive : Type := mkIndType
  { it_ikind : seq ikind; it_constructors : seq constructor }.
Definition icompartment : Type := seq inductive.
```

Next, Γ is split into two parts. The `environment` compartment is a sequence recording free value and type names. The free names are kept separate from local `bindings` and equality assumptions induced by pattern matching. We show the definitions of both below. Separating these two compartments simplifies our adequacy proof, inasmuch as only the local names are subject to α-conversion and implicit type conversion. We include value and type equalities (`EB_VlName` and `EB_TyName`) in the environment as they become necessary in the cofinite system of §4.2—these constructors are not used in the reference system.

```
Inductive ebinding : Type :=
| EB_VlName : vlname → typ → ebinding
| EB_TyName : tyname → kind → ebinding.
| EB_VlEq : value → value → locbinding
| EB_TyEq : typ → typ → binding.
Definition environment : Type := seq ebinding.

Inductive locbinding :=
| LB_BvarTy : bvar → typ → locbinding
| LB_BvarKind : bvar → kind → locbinding
| LB_VlEq : value → value → locbinding
| LB_TyEq : typ → typ → binding.
Definition bindings := seq locbinding.
```

Value and expression typing judgments (`value_ty` and `expr_ty`) take two additional arguments, discussed in §3: a sequence of bound names that represents the affine compartment and a mode.

With these definitions in hand, we give a flavor of the `value_ty` and `expr_ty` judgments by showing the rules for λ-abstractions and applications, respectively. Note the intensional use of concrete names throughout the system.

```
| WFV_Fun : forall I G B X m t bt u x body Q,
    ConcreteKind bt → AQual X (T_Prod x t u) Q → wfT I G B t bt
  → expr_ty I G ((LB_BvarTy x t) :: B) (x :: X) m body u
  → value_ty I G B X m (V_Fun x t body) Q
```

```
| WFE_App : forall I G B X X1 X2 m e v x t u tv tres,
    Split X X1 X2 → StripQual tv (T_Prod x t u)
  → SubstTV u x v tres
  → expr_ty I G B X1 m e tv
  → value_ty I G B X2 m v t
  → expr_ty I G B X m e v tres
```

The rules use auxiliary predicates: `ConcreteKind` identifies the subclass c of kinds; `Split` specifies $X = X_1 \uplus X_2$; `Aqual` specifies $Q(X,t)$; `StripQual` specifies $?x{:}t \to t'$; and `SubstTV` specifies value substitutions in types. All of these predicates are defined as inductive types in Coq; for instance, `ConcreteKind` is defined as:

```
Inductive ConcreteKind : basekind → Type :=
| CK_Star : ConcreteKind BK_Comp
| CK_Aff : ConcreteKind BK_Afn
| CK_Prop : ConcreteKind BK_Prop.
```

One may wonder why we do not use functions to define `ConcreteKind`, `SubstTV`, etc. This is a key point related to the soundness of self-certification, discussed in detail in the coming sections. Keeping in mind that we aim to share these definitions between F⋆ and Coq, one reason for using inductive predicates uniformly is that F⋆, being a value-indexed language, does not have function applications within types.

However, our reference type system is not entirely free of function applications. For example, in defining the well-formedness of typing environments, we use the following predicate for non-membership in a sequence.

```
Inductive NotMem : A → seq A → Prop :=
| NotMem_nil : forall x, NotMem x [::]
| NotMem_cons : forall x y ys,
        x != y → NotMem x ys → NotMem x (y::ys).
```

In writing `x != y` above, we make use of reflection and decidable equality on names: the type `x != y` is a function application that reduces in Coq to the type `true=true`. We permit such decidable (dis-)equalities in our reference specification and show in §4.3 how to model them in F⋆ using ghost refinements.

4.2 F⋆ metatheory in Coq

The reference specification of the type system conforms to the informal presentation of F⋆ and to its compiler implementation. However, due to the usage of the *for-one locally nameless* representation of binders (Leroy 2007), this formulation is a poor one for conducting metatheory. As explained above, we provide a second specification of the F⋆ type system, this time making use of the *for-all representation* of binders with the *co-finite quantification* optimization (Aydemir et al. 2008). We prove a correspondence between the two, and prove a subject reduction result for the second system.

We give a flavor of the cofinite system, still using the rule for λ-abstraction. As in the reference system, we have a 4-way mutually inductive type for the main judgments of the type system.

```
Inductive pname : Type :=
  TermName : vlname → pname | TypeName : tyname → pname.
Definition pnames := seq pname.

Inductive coF_wfK : ... := ...
with coF_value_ty:icompartment → environment → acontext
              →mode → value → typ → Type := | ...
| CoF_WFV_Fun :
  forall I G X m t bt u x body Q (L : pnames),
   concretekind bt → aqual X (T_Prod x t u) Q → coF_wfT I G t bt
→(forall y, TermName y ∉ L →
      coF_expr_ty I ((EB_BvarTy y t)::G) ((ValueName y)::X) m
                    (subst_e body x y) (subst_t u x y))
→ coF_value_ty I G X m (V_Fun x t body) Q
```

The major difference between the two systems is in their treatment of binders. Rule `CoF_WFV_Fun` requires that the typing of `body` be insensitive to the particular choice of bound variable name. This is modeled by requiring the body to be typeable for a cofinite set of names, that is, those not in the set `L`. We type the body by substituting the bound name `x` for the chosen free name `y`. As such, in this formulation, there is no need for the local binding context—the free names in `environment` suffice. A relatively superficial difference is in the modeling of the affine environment, which in the cofinite system is a sequence of value names, whereas in the reference system, it is a sequence of bound variables.

A significant stylistic difference is that we make liberal use of functions and type-level reduction provided by Coq in the cofinite system. For example, `concretekind` is now a boolean function, given below. The cofinite system is not shared with F⋆ and is thus not subject to the value-indexed restriction imposed on the reference system.

```
Definition concretekind (b : basekind) : bool :=
  match b with
  | BK_Comp | BK_Afn | BK_Prop ⇒ true
  | BK_Erase ⇒ false
  end.
```

Our first theorem is *adequacy* of the reference system with regards to the cofinite system. It is stated below for expression typing, but of course its proof requires a 4-way mutual induction on the main judgments.

THEOREM 1 (Adequacy).

```
forall I G X m e t,
    expr_ty I G [::] X m e t
→ coF_expr_ty I G X m e t.
```

Next, we give our main subject-reduction theorem. To simplify the presentation in this paper, we consider a specialization of our subject reduction theorem to programs that, first, make no use of affinity; and, second, have no dynamic assumptions for introducing ghost refinements. We consider `reduces: icompartment → expr → expr → Type`, and inductive definition specializing the general reduction relation for F⋆.

THEOREM 2 (Subject Reduction, specialized).

```
forall I G e e' t,
   coF_expr_ty I G [::] Term_level e t
→ reduces I e e'
→ coF_expr_ty I G [::] Term_level e' t.
```

In the rest of the paper, we only rely on the following corollary, obtained by composing the adequacy theorem with subject reduction and an induction on the sequence of reduction steps—`evaluates` is a transitive closure of `reduces` followed by matching on `E_Value`.

COROLLARY 3 (Subject Reduction to Values).

```
forall I G e v t,
   expr_ty I G [::] [::] Term_level e t
→ evaluates I e v
→ coF_value_ty I G [::] Term_level v t.
```

For convenience, we use abbreviations for typing closed terms, defined as follows:

```
Definition val_ty i v t := value_ty i [::] [::] [::] Term_level v t.
Definition exp_ty i e t := expr_ty i [::] [::] [::] Term_level e t.
Definition coF_val_ty i v t :=
  coF_value_ty i [::] [::] Term_level v t.
```

4.3 Sharing definitions and embeddings

Inductive types in F⋆ are nearly as expressive as inductive types in Coq. In this section, we give an embedding result, Theorem 4, which allows us to soundly treat certain inductive types in Coq as if they were also inductive types in F⋆, and vice versa. We exploit this result to give strong specifications to typecheckers (and other programs) implemented in F⋆, ensuring, for example, that they only compute valid derivations. Theorem 4 then allows us to conclude that these derivations are also valid reference typings in Coq.

We illustrate this approach using the STLC reference specification as an example. In §2, we argued informally that the syntax of STLC terms can be shared between ML and Coq, whereas the inductive type specifying the typing judgment could not, since ML inductive types are not sufficiently precise—not so for F⋆. We can express both the syntax and the reference typing for STLC as inductives in F⋆, in a module `STLC` (outlined below) in close correspondence with our Coq definitions.

```
module STLC
type Ltyp :: P=
  | LTyp_unit : Ltyp
  | LTyp_fun : Ltyp → Ltyp → Ltyp
type Lexpr :: P=
  | LExpr_unit : Lexpr
  | LExpr_var : string → Lexpr
  | LExpr_fun : string → Ltyp → Lexpr → Lexpr
  | LExpr_app : Lexpr → Lexpr → Lexpr.
type Lenv = seq (string * Ltyp)
type LTyping :: Lenv ⇒ Lexpr ⇒ Ltyp ⇒ P= ...
  | LTyping_fun : g:Lenv →x:string →t:Ltyp →t':Ltyp →e:Lexpr
                →LTyping ((x,t)::g) e t'
                →LTyping g (LExpr_fun x t e) (LTyp_fun t t')
```

Formally, sharing between F* and Coq is justified by an embedding theorem (explained below). Pragmatically, resolving differences in concrete syntax is easily automated—we provide some basic support for this, although we often manually transcribe definitions from F* to Coq. Less superficially, we have implemented pretty printers that allow F* values and type definitions to be printed as values in the abstract syntax of F* in Coq. This is essential for importing large certificates and we use this tool extensively.

We describe the correspondence between inductive types in F* and Coq using examples from the STLC development. For every F* file, say `STLC.fst`, containing the type definitions in the STLC module, we have a Coq file, `STLC.v`, with the following elements.

Surface definitions We have Coq definitions such as `Inductive Lexpr : Type := …` (listed in §2.1) that mirror STLC definitions such as LExpr. We apply this superficial translation only to the F* inductive types that can be viewed soundly as Coq inductives, namely those that meet the *P*-restriction formalized below. In addition to the basic inductive types we have seen so far, *P*-restricted types also include inductive types use ghost refinement formulas, where the refinement formulas correspond to the use of reflection for computing decidable equalities. For example, the type in F* corresponding to the definition of `NotMem` from §4.1 is NotMem, shown below.

```
type tag = D:tag
type NEq v1 v2 = _:tag{v1 != v2}
type NotMem :: α⇒ seq α⇒ P=
  | NotMem_nil : x:α →NotMem x []
  | NotMem_cons : x:α →y:α →tl:seq α
     →NEq x y →NotMem x tl →NotMem x (y::tl)
```

Whereas in Coq we use the function application `x!=y`, in F* we use an uninformative type refined with the formula `x!=y`. In general, any F* ghost refinement formula can be embedded in Coq using reflection, provided we have a corresponding function in Coq that can decide the formula. In principle, we could use this embedding to specify the entire type system of F* using ghost refinements, but this would require having a function in Coq to decide typing—which, of course, renders moot building a certified typechecker for F* in the first place. As such, we restrict our use of reflection in the reference system to trivially decidable equalities. We show how to relate these formally below.

Embedded signatures We also embed the F* inductives as terms in our Coq formalization of F*. Specifically, we generate a Coq value of type `icompartment` that represents our formalization of these F* inductive definitions in Coq. All F* inductive types can be embedded in this manner, whether they meet the *P*-restricted condition or not. For instance, for the Lexpr inductive we generate:

```
Definition iLexpr: inductive := mkIndType
  [:: mkIKind "Lexpr" (K_Base BK_Prop)]
  [:: mkConstructor "LExpr_App"
      (T_Prod "" (T_Ind "Lexpr")
        (T_Prod "" (T_Ind "Lexpr") (T_Ind "Lexpr")))); … ].
Definition iSTLC: icompartment := [:: expr ; … ].
```

Type embedding functions For each *P*-restricted inductive type T of kind $\alpha_1::\kappa_1 \Rightarrow \ldots \Rightarrow \alpha_n::\kappa_n \Rightarrow x_1{:}t_1 \Rightarrow \ldots x_m{:}t_m \Rightarrow P$, we have a Coq function `tembed_T` of type `typ` $\rightarrow\ldots\rightarrow$ `typ` $\rightarrow$ `value` $\rightarrow\ldots\rightarrow$ `value` $\rightarrow$ `typ`, for *n* `typ`s and *m* `value`s. For instance, for the Lexpr and LTyping inductives of STLC, we generate:

```
Definition tembed_Lexpr := (T_Ind "Lexpr").
Definition tembed_LTyping (g e t:value) :=
  (T_VApp (T_VApp (T_VApp (T_Ind "LTyping") g) e) t).
```

Value embedding functions For values of shared inductive types, we have functions in Coq that produce terms in the `value` inductive that represents F* values in Coq. For instance, we have a function from surface to embedded STLC expressions:

```
Fixpoint embed_Lexpr (e: Lexpr) : value :=
  match e with …
  | LExpr_App e0 e1 ⇒
   V_Const "LExpr_App" [::] [:: embed_Lexpr e0; embed_Lexpr e1 ].
```

Using these value embedding functions, we define analogs of the type embedding functions that allow us to combine the type and value embeddings. For example, we define the following wrapper around `tembed_LTyping`:

```
Definition tembed_LTyping' g e t :=
 tembed_LTyping (embed_Lenv g) (embed_Lexpr e) (embed_Ltyp t).
```

Finally, we have pretty-printers from F* values to Coq values representing F* abstract syntax. For instance, we print the F* value

```
LExpr_App (LExpr_var "x") (LEexpr_Var "y"): Lexpr
```

as the Coq value

```
LExpr_App (LExpr_var "x") (LExpr_Var "y"): Lexpr
```

To this end, the F* code generator injects a ToString method into each .NET class that represents an F* data constructor. By calling ToString on any (first-order) F* value and printing the result, we get the representation of that value as a value of type `value`, the type of F* values, in Coq. This tool is in effect an (untrusted) F* primitive function, string_of_any : $\alpha \rightarrow$ string.

Formal embedding Our theorem states that the existence of an embedded F* proof for a given inductive type implies the existence of a Coq proof for that type. We first define a restriction on inductive types such that our theorem applies. The purpose of the restriction is first to exclude non-*P*-fragment types: types that live in F*'s ⋆-universe, for instance, need not have any analog in Coq. Second, F* expressions, even in the *P*-fragment, employ constructs that do not translate directly to Coq—notably implicit conversions introduced by pattern matching, which must be made explicit in Coq. For this reason, the *P*-restriction also excludes types whose values may contain (non-value) sub-expressions, in particular λ-terms. We still allow some limited use of ghost refinements in *P*-restricted types, notably for types refined by formulas speaking about decidable, syntactic equality on F* values.

- A type t is *P*-restricted when it is a type variable whose kind is *P*-restricted; or NEq v_1 v_2; or $T\ \overline{\alpha}\ \overline{x}$ where T is *P*-restricted.
- A kind κ is *P*-restricted when it is of the form $\overline{\alpha::\kappa} \Rightarrow \overline{x{:}t} \Rightarrow P$ where each κ_i and each t_i is *P*-restricted.
- An inductive type T defined by $T::\kappa\{\overline{D{:}t}\}$ is *P*-restricted when the kind κ is *P*-restricted and each constructor $D{:}t$ has a type t of the form $\overline{\alpha::\kappa} \rightarrow \overline{x{:}t} \rightarrow T\ \overline{\alpha}\ \overline{x}$ where each κ_i and each t_i is *P*-restricted.

THEOREM 4 (Embedding). *Let S be a signature and* i *its embedding in Coq. Let* T *be a P-restricted inductive type of S with kind* $\overline{\alpha::\kappa} \Rightarrow \overline{x{:}t} \Rightarrow P$ *and let* T *be the Coq surface-level inductive type for* T. *We have*

$\forall\ \overline{(\alpha : \kappa)}\ \overline{(x : t)}$ (v: value), coF_val_ty i v (tembed_T' $\overline{\alpha}\ \overline{x}$) $\rightarrow$ T $\overline{\alpha}\ \overline{x}$.

The statement of this (meta) theorem is not expressible in Coq, since it involves quantifying over Coq inductive types, so we prove it by hand. However, we can easily express in Coq instances of the theorem for specific type constructors. We will later formally admit some of these instances for types such as `LTyping` for STLC. In future work, we anticipate mechanizing the proofs of specific instances of the data embedding theorem in Coq, rather than admitting them via the meta theorem.

Our hand proof relies on a structurally inductive function that translates F* types and values to Coq. We show that this translation preserves types. This proof, in most cases, appeals to a more general lemma relating arbitrary *P*-fragment terms (including

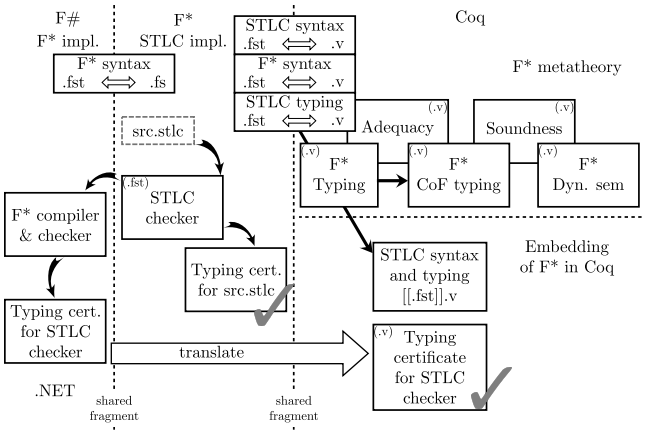

Figure 3. Certifying an STLC checker in F*

those with reducible expressions) to terms in the `Type` fragment of CiC (Swamy et al. 2011). The one case not handled by the general lemma is for embedding the F* type `NEq v1 v2 = _:tag{v1 != v2}`. For this case, we use the premise `coF_val_ty i v (T_Ref _ _ (embed v_1)!= (embed v_2))`, which introduces a ghost refinement. As discussed in §3, the introduction of ghost refinements occurs via a judgment in an external logic, formally represented in the metatheory of F* as a predicate `LogicEntails`. We thus have `LogicEntails i [::] ((embed v_1)!= (embed v_2))`. One of several admissibility restriction on this judgment ensures that it soundly decides primitive (dis-)equalities, from which we conclude `v_1 != v_2`. In effect, this captures our assumption that the F* runtime system properly implements syntactic comparison of values, and that type-safety in F* is modulo the soundness of the external logic.

5. From certifying to certified: STLC in F*

We now return to the problem of certifying an STLC typechecker. Figure 3 illustrates the architecture we use, and §6 shows an instantiation of this architecture to certify a typechecker for F* itself.

The process begins at the top right of the figure, depicting our formalization of F* in Coq (in addition to the formalization of STLC). Next, in contrast to Figure 2, since we program in F*, we share *both* the definitions of the STLC syntax and its reference type system between F* and Coq. We also share the definitions of the F* syntax. We implement the STLC checker in F* and give it a strong specification, stating that it computes (at most) valid STLC typings. We let F* typecheck it and produce its typing certificate then, as in §2 for STLC typings, we translate the certificate to Coq and typecheck it there, against the reference specification of F* and the embedding as an `icompartment` of the STLC syntax and reference typing. We then complete the certification of our checker from its specification and the F* metatheory.

Programming and verifying the STLC checker Our STLC checker in F* is a partial function with the following signature:

```
stlc_checker: g:Lenv → e:Lexpr → t:Ltyp → Partial (LTyping g e t)
```

As discussed in §3, the F* kind system distinguishes partial and total functions. Since we intend `stlc_checker` to be partial (possibly throwing exceptions), we wrap its co-domain with a type constructor `Partial`, defined below, to lift the P-kinded `LTyping g e t` into the $\star$-universe of general computations.

```
type Partial :: P ⇒ ⋆ = P2S : ∀α::P. α → Partial α
```

We can run our checker to compute certificates for sample STLC programs, such as $\lambda x.x$, as follows:

```
let id: Lexpr = (LExpr_Fun "x" LTyp_unit (LExpr_Var "x"))
let t: typ = (LTyp_Fun LTyp_unit LTyp_unit)
let (P2S (pf : LTyping [] id t)) = stlc_checker [] id t
```

Now, instead of translating and checking each such `pf` using Coq, as in §2, we want to certify `stlc_checker` once and then run it repeatedly on STLC programs without further reliance on Coq. The process of certifying `stlc_checker` involves several steps, some of which we have already seen.

(1) We share the definitions in the F* module `STLC` with Coq to obtain a Coq module `STLC`. As we have seen in §2 and §4.3, this module contains the reference specification of the STLC type system in Coq, as well as various embeddings of STLC into F* abstract syntax within Coq.

```
Module STLC.
Inductive Ltyp : Type := ...
Inductive Lexpr : Type := ...
Definition Lenv := seq (Lbvar * Ltyp).
Inductive LTyping : Lenv → Lexpr → Ltyp → Type := ...
...
Definition iLexpr: inductive := ...
Definition iLTyping : inductive := ...
Definition iSTLC : icompartment := [:: iLexpr; iLTyping...].
...
Definition tembed_Ltyp : typ := (T_Ind "Ltyp").
Definition tembed_Lexpr : typ := (T_Ind "Lexpr").
Definition tembed_LTyping g e t : typ :=
 (T_VApp (T_VApp (T_VApp (T_Ind "LTyping") g) e) t).
...
Fixpoint embed_Ltyp (t: Ltyp) : value := ...
Fixpoint embed_Lexpr (e:Lexpr) : value := ...
Fixpoint embed_Lenv (e:Lenv) : value := ...
...
Definition tembed_LTyping' g e t := ...
```

(2) We print the abstract syntax of `stlc_checker` as an `expr` in Coq. The process of verifying `stlc_checker` involves trusting a parser and pretty printer (implemented in F#), since the main theorem will speak about the reduction to a value of this term. However, this degree of trust is no more that the trust one already places in Coq—to be confident that one has proved the theorem that one set out to prove, one must be careful to have Coq print the statement of the theorem that was proven, with notations and implicit coercions disabled, and to manually inspect that the theorem proven was what was intended—a recipe prescribed by Pollack (1998), among others.

```
Definition stlc_checker : expr := (E_Value (V_Fun ... )) .
```

(3) We verify `stlc_checker` by building a typing derivation for it with respect to the F* reference type system. The means by which this derivation is produced is irrelevant—any valid typing will do. We may, for example, run an (uncertified) F* typechecker implemented in F#, have it construct a derivation, print the resulting derivation, and check it in Coq using the embedded types listed below. (Eventually, using the self-certified F* typechecker of §6, we need not even check this certificate in Coq.)

```
Definition asVal (x:string) := V_Var (GV_Bound x).
Definition wf_iSTLC : wfEnv iSTLC [::] [::] := ... .
Definition stlc_checker_typing
: expr_ty iSTLC stlc_checker
 (T_Prod "g" tembed_Lenv
  (T_Prod "e" tembed_Lexpr
   (T_Prod "t" tembed_Ltyp
    (T_App (T_Ind "Partial")
     (tembed_LTyping (asVal "g") (asVal "e") (asVal "t"))))))
 := (* pretty-printed certificate *)
```

(4) Within Coq, we prove a lemma stating that the F* function `stlc_checker` applied to any F* values `g` of type `tembed_Lenv`, `e` of type `tembed_Lexpr`, and `t` of type `tembed_Ltyp` is an F* expression of type `embed_LTyping g e t`. The proof follows from

three applications of the function-application rule (WFE_App) of the F⋆ reference type system.

```
Definition embed_check g e t :=
  (E_App (E_App (E_App stlc_checker
    (embed_Lenv g)) (embed_Lexpr e)) (embed_Ltyp t)).
Lemma embed_check_ty : forall (g:Lenv) (e:Lexpr) (t:Ltyp),
  expr_ty iSTLC (embed_check g e t)
                (tembed_Partial (tembed_LTyping' g e t)).
```

(5) We admit an instance of Theorem 4 applied to LTyping. Rather than relying on this hand-proved theorem, we could, in principle, formally prove this instance by inversion of the embedded-typing assumption.

```
Axiom LTyping_data_embedding:
  forall (g:Lenv) (e:Lexpr) (t:Ltyp) (r:value),
      coF_val_ty iSTLC r (tembed_LTyping' g e t)
    →LTyping g e t.
```

(6) We also have a specific instance of a canonical forms lemma from the F⋆ metatheory applied to the Partial type constructor. This provides us with a convenient inversion principle to reason about the result of stlc_checker.

```
Lemma Partial_inv: forall r t,
     coF_val_ty iSTLC r (T_App (T_Ind "Partial") t)
  →(exists v, coF_val_ty iSTLC v t).
```

(7) To complete the proof of certification for stlc_checker, we apply Corollary 3 (subject reduction to value) to embed_check_ty, followed by Partial_inv and LTyping_data_embedding.

```
Theorem partial_correctness_of_stlc_checker :
  forall (g:Lenv) (e:Lexpr) (t:Ltyp) (r:value),
       evaluates iSTLC (embed_check g e t) r
    →LTyping g e t.
```

Thus, we reach our stated goal—the F⋆ program stlc_checker (partially) decides the type safety of STLC expressions. Going further, we may rely on the metatheory of STLC to replace LTyping g e t with runtime safety for e.

6. Emancipation through self-certification

The certification method of §5 is clearly not specific to STLC—the formal development generalizes naturally to F⋆ itself. We apply this method to F⋆, as outlined in Figure 1, and obtain a certified F⋆ core typechecker that can be used to check and certify all F⋆ programs independently of Coq. We first explain the steps that lead to the main theorem of the paper. We then discuss some of the subtleties behind the proof, the impact of the *P*-restriction, and some aspects of our adequacy result.

6.1 Self-certifying F⋆: the main development

The first step in our development is to share the inductive types for the F⋆ abstract syntax and reference type system between F⋆ and Coq—this corresponds to step 1 in §5. All the definitions of §4.1 are expressed as F⋆ inductive types in a CoreTyping module, and these definitions are also embedded in Coq within the F⋆ theory. For example, we have mutually-recursive judgments in F⋆:

```
type wfK :: icompartment ⇒ environment ⇒ bindings
      ⇒ kind ⇒ basekind ⇒ P= ...
and wfT :: icompartment ⇒ environment ⇒ bindings
      ⇒ typ ⇒ kind ⇒ P= ...
and value_ty :: icompartment ⇒ environment ⇒ bindings
      ⇒ acontext ⇒ mode ⇒ value ⇒ typ ⇒ P= ...
and expr_ty :: icompartment ⇒ environment ⇒ bindings
      ⇒ acontext ⇒ mode ⇒ expr ⇒ typ ⇒ P= ...
```

embedded, e.g., as

```
Definition i_mutual_ty : inductive :=
  mkIndType [:: (mkIKind "wfK" ...); ... ;(mkIKind "expr_ty" ...)]
            [:: ...].
Definition iCoreTyping : icompartment := [:: ...; i_mutual_ty ].
```

In the rest of this section, we focus on expression typing and omit the details for the other mutually-recursive functions. We implement a core typechecking algorithm, an F⋆ partial function with the following signature:

```
val etyping: i:icompartment →g:environment →b:bindings
  →a:acontext →m:mode →e:expr →t:typ
  →Partial (expr_ty i g b a m e t)
```

The definition of etyping relies on several other top-level let bindings defined in the CoreTyping module. We use an existing F# implementation of the F⋆ typechecker to compile and verify CoreTyping. This F# typechecker is not certified, nor does it actually produce typing derivations that can be certified. Once built, we run the etyping function on (the abstract syntax of) the top-level let bindings in CoreTyping, each a value in the shared inductive type expr, including etyping itself. We print the abstract syntax of each let binding as a Coq value, corresponding to step 2 in §5. In the case of etyping, this is:

```
Definition etyping : expr := (E_Value (V_Fun ...)).
```

The function etyping applied to each let binding in CoreTyping produces a typing derivation d of type expr_ty, the reference specification of expression typing in the F⋆ type system. Since this derivation is a value in a shared type, we print the derivation as a Coq value (step 3 of §5). For etyping, this is:

```
Definition etyping_cert :
  expr_ty iCoreTyping env [::] [::] Term_level
          etyping (T_Prod ...) := (* large certificate from F*)
```

It is not strictly necessary to run etyping on CoreTyping to produce the certificates—any means of producing the certificate will do. However, by running a typechecked program on itself, we gain confidence that the resulting certificate will also typecheck in Coq. Indeed, once we had set up the infrastructure and typechecked a few certificates on small programs, the remaining 7 GB of certificates were typechecked in Coq slowly, but steadily. Running etyping on itself also provides a useful test of its completeness.

The certificate above provides a typing of etyping in a context env. The context includes assumptions for each top-level let binding that etyping depends on. After checking certificates for these let bindings, we compose them using a substitution lemma from the F⋆ metatheory to obtain the following lemma, where etyping' is a closed term obtained by substituting each of the free variables in etyping with their corresponding value.

```
Lemma etyping_typing :
   coF_expr_ty iCoreTyping etyping' (T_Prod ...).
```

Next, we define an auxiliary function to abbreviate the application of etyping' to its embedded arguments and we show that it is well-typed by successive applications of WFE_App.

```
Definition embed_check_etyping i g b a m e t :=
E_App (E_App etyping' (embed_icompartment i) ...) (embed_typ t).
```

Then, we admit an instance of Theorem 4 for expr_ty, and use Partial_inv and Corollary 3 to conclude the formal proof for the main result of this paper: if the core typechecker applied to (embeddings of) the parameters of the expr_ty judgment returns any certificate r, then the judgment is valid.

THEOREM 5 (Self-certification).

```
Theorem partial_correctness_of_etyping:
  forall i g b a m e t r,
    evaluates iCoreTyping (embed_check_etyping i g b a m e t) r
  →expr_ty i g b a m e t.
```

With this theorem, the F⋆ typechecker is emancipated from Coq. We may use etyping to check any F⋆ source program—for instance

our STLC checker—and, if it succeeds in constructing a derivation, we can be sure that this program is type-correct. We may also use it to bootstrap more sophisticated F* typecheckers. Of course, if the type system were to evolve, we would still have to conduct the metatheory of F*v2 in Coq, but, as long as the changes can still be expressed in F*, we can implement and certify F*v2 using our initial self-certified F* checker, without the need to generate and check large Coq certificates ever again.

6.2 Some technical aspects of the development

We discuss two technicalities in this section, starting with an analysis of the *P*-restriction of §4.3 and then considering the treatment of substitutions, α-conversion, and uniqueness of names.

Analyzing the *P*-restriction Using *P*-restricted types to define the F* type system gives us a convenient way (via Theorem 4) to reason about the adequacy of the specification of our core typechecker. However, the *P*-restriction also imposes some difficulties which require particular care. We have already seen in §4.2 that we cannot use function applications (like `concretekind`) in our reference system, since this falls outside the expressive power of F*'s value-dependent type system. There are also well-formed, potentially useful *P*-types in F* that are not *P*-restricted.

For example, while the basic idea of cofinite quantification can be easily expressed in value-dependent F*, an inductive type of the form shown below is not *P*-restricted, since its values contain function-typed subterms.

```
type value_ty :: _ = ...
  | WFV_Fun : x:bvar → t:typ → e:expr → L:seq pname
              → (y:vlname → NotMem (TermName y) L → expr_ty ... )
              → value_ty ... (V_Fun x t e)
```

Although such types are expressible in F*, we avoid them in our specifications for two reasons. Consider the shape of an F* typing certificate using the value above, say WFV_Fun ... (λy.e) ..... To share this certificate with Coq, we must somehow translate the closure λy.e to a Coq lambda: this involves breaking the closure, inspecting its code, and printing it—in effect a form of higher-order marshalling. Further, the language of F* expressions does not correspond syntactically to the language of Coq terms: typing the body of the closure may require the use of implicit conversions and these have to translated to explicit equality-witnessing coercions in Coq, i.e., even after breaking closures, we need a type-directed translation to marshall them to Coq. The *P*-restriction conveniently sidesteps these problems. As pointed out earlier, implementing a compiler with cofinite quantification is undesirable anyway. For the few cases where we might use function types in our specification (e.g., when using negations), we rely on the correspondence between ghost refinements and reflection provided by Theorem 4.

Substitution, α-conversion, and adequacy In order to show its adequacy with regards to the cofinite system, we carefully restrict the (re-)use of names in the reference system: in particular, in any term, we require that any sequence of nested binders always bind distinct names. For instance, the value $\lambda x.\lambda x.x$ is not well-formed in our reference system.

Our core typechecker rejects such programs outright, since it relies on the F* front-end to introduce unique names. However, it must also maintain the invariant, notably as it applies substitutions. Consider typing the expression f λy:t.y in a context that binds f to the type x:(a:t → t) → y:t → t' x. All these terms meet our unique-binding requirement, but after the substitution (y:t → t' x) [(y:t → t / x)] the resulting type y:t → t' (y:t → t) binds *y* twice and would be rejected by the reference type system. (Such a type and substitution are usually considered well-formed, inasmuch as there is no name capture.) To prevent such instances, substitutions in our reference system are, essentially, partial functions that are undefined whenever they would produce a term that breaks our invariant. For this reason also, it is convenient to specify substitutions using inductive types (relations) rather than functions. To type the expression above, our typechecker first explicitly α-converts the type of f, renaming its bound variable y, then applies the substitution.

7. Programming and verifying the F* checker

The new self-certified typechecker, CoreTyping, is only a small part of the F* compiler. The concrete syntax of F* is parsed by modules implemented in F#, the parse tree is desugared into a minimal AST which is then typed by another, *ad hoc* source typechecker that implements various heuristics, including a form of bidirectional local type inference. This main typechecker makes calls to Z3 to prove refinement properties, and then builds a fully annotated core AST for the program, that is, a value in the shared inductive type expr. This AST is then passed to CoreTyping to build a reference typing. This section discusses the design of CoreTyping and presents several experimental aspects of the self-certification process.

7.1 The design of CoreTyping

The first difficulty in implementing CoreTyping is to make F* typing syntax-directed. We use the source typechecking pass for this: every use of subtyping and subkinding in the AST is annotated with an explicit type- or kind-ascription form, serving as an indicator to the core typechecker to apply the subtyping judgment. This simplifies core typing, but does not make it fully syntax-directed, for two reasons. First, the signature of etyping shown in the previous section is for a pure typechecking algorithm—it requires the context to provide the expected type of every subterm. Instead, we implement a function etyping_aux that computes and returns the type of an expression (relying on ascriptions etc.) with its certificate. The function etyping is simply a top-level wrapper of etyping_aux that checks that the computed type matches the type provided by the caller. A second difficulty in implementing etyping_aux is the splitting of affine contexts. The reference type system features nondeterministic context splits (as it should), but this is not directly implementable. Instead, etyping_aux implements a bottom-up computation of used affine names to provide a precise split of the affine context. The signature of etyping_aux follows:

```
val etyping_aux: i:icompartment → g:environment → b:bindings
  → a:seq bvar → m:mode → e:expr
  → (t:typ * used:seq bvar * unused:seq bvar *
     Split a used unused * expr_ty i g b used m e t)
```

This function returns a type t, a partition of the affine context a into two fragments used and unused, and a derivation for the typing of the expression e in the affine context used at the type t. Note that the default tuple constructor in F* (encoded using an inductive type of the kind α::⋆ ⇒ (α ⇒ ⋆) ⇒ ⋆) constructs types in the ⋆-universe. As such, the F* kinding rules treat etyping_aux as a partial function, without the need for an additional Partial wrapper.

Pattern matching and implicit conversions The most complex part of the typechecker deals with pattern matching. One distinctive feature of F* (as compared, say, to Coq) is its support of implicit type conversions and type refinements using equalities introduced in the context due to pattern matching. These conversions are particular convenient for programming in F*, and our core typechecker makes heavy use of them, with 196 conversions at various places. In contrast, Coq programmers must explictly apply conversions, although recent work alleviates some of this burden (Sozeau 2010).

Unification, state, and exceptions The implementation of pattern matching relies on an algorithm to find a substitution that unifies the type of the scrutinee with the type of the pattern. We implement this algorithm first in an *ad hoc* manner, by making use of recursion

(without providing any termination proof), state, exceptions and the like, and in doing so compute a candidate substitution. We then implement a certification pass that builds a proof of validity for the candidate substitution. Programming in this style is convenient and is enabled by F*'s liberal support of recursion in its $\star$-fragment, but it is not particularly efficient. In the future, we hope to directly certify the substitution built by an efficient, stateful unification algorithm. We also use state and exceptions elsewhere in CoreTyping, for generating names, for performing I/O, and flagging errors; F*'s value-dependency makes it easy to cope with these features.

7.2 Engineering the certification process

As a first attempt, we simply printed the F* certificates into text files in the Coq syntax. Although the internal representation of certificates in F* is compact, their printing is rather verbose. For instance, fully printing a certificate for 6 lines of code generates a 480 MB Coq source file, which is too large to verify. Instead, we implement a fairly sophisticated, compact format that enables Coq to check all the certificates that we produce. Thankfully, we need not trust the code that prints the bulk of our certificates. As discussed in §5, we depend only on the statement of etyping_cert in Coq, but this part of the certificate is relatively small, so it can be printed verbatim and inspected manually.

Source-level splitting Our method lends itself naturally to incremental, modular verification: we generate separate certificates for each top-level let binding in the typechecker and compose these to obtain our certification result. This allows us to split our certification into 215 typing judgments. Formally, this approach is justified by the mechanized F* metatheory. Given a sequence of top-level bindings **let** $x_1 = e_1 \ldots$ **let** $x_n = e_n$, we typecheck each e_i in an environment that includes locbindings for each of $x_1 \ldots x_{i-1}$ at their certified types. Our adequacy and substitutivity theorems ensure that the closure of the final function etyping obtained by substituting each x_i with e_i is well-typed in the cofinite system.

Hash consing We share common sub-terms in certificates by hoisting them into sequences of Coq `Definitions`. (Coq also implements hash consing but does not do any memoization while type-checking, so hoisting still helps.) Aggressive hash consing may cause terms to be lifted out of their context, thereby breaking Coq inference for their implicit type arguments, so we implement a partial hash consing that keeps enough context around those terms.

Reflection A large portion of our certificates are devoted to trivial proofs of properties such as list membership and disjointness of lists. We print proofs of these properties by reflection, relying on Coq functions to show the existence of the corresponding proof terms. We rely on six such Coq functions. For example, an F* proof of NotMem x xs for some closed values substituted for x and xs may involve many nested constructors, but its Coq proof is just `notmemP x xs (refl_equal true)`, relying on a simple reflection lemma:

```
Lemma notmemP :
  forall (A : eqType) (x : A) (xs : seq A),
    notmem x xs = true → NotMem x xs.
```

where `notmem x xs` is a Coq boolean function deciding if `x` is not a member of `xs`. When `x` is indeed not a member of `xs`, `notmem x xs` reduces to `true`, leading to `notmem x xs = true` being convertible to `true = true`. Hence, providing a trivial relexivity proof (`refl_equal true`) is sufficient to have Coq check `NotMem x xs`.

Compact identifiers The default Coq representation of strings where each 8-bit character is coded as a sequence of eight booleans is inadequate for large certificates, especially as all our identifiers are coded as strings. Instead, we check certificates using a custom-built Coq version with native support for a compact representation of arrays (Armand et al. 2010).

Coq-level splitting and opacity Even after splitting, hash-consing, and reflection, some certificates remain too large for Coq to handle. Our largest certificate is 214 MB. Thus, we perform another level of certificate splitting into sequences of definitions at most 10,000 lines a piece; this keeps the size of each resulting piece below 1 MB, faster for Coq to process. Coq checks one piece at a time, with all the previous pieces for the same certificate loaded as libraries. Such loading is very time- and memory-consuming, especially for large certificates. To optimize loading, subgoals of the previous parts are generated as opaque lemmas, allowing Coq to load just their types—and not their proofs—for checking the following pieces. However, all our definitions and lemmas are at the top-level. Thus, due to the transitive nature of the loading mechanism of Coq, it is not possible to load one specific subgoal of a previously checked part without loading the statements of the definitions and subgoals of all previous parts. Hence, checking a piece may spend several minutes just on loading.

7.3 Performance

CoreTyping consists of 5,139 lines of F* source code. We organize the checker into 11 modules. (Although F* has a module system that resembles F#'s, we make no essential use of it in CoreTyping, since this module system is outside the scope of our formalization.) For each module, the table below gives the number of lines of F* source code; the time taken to certify the module within F*; the number of Coq certificates generated; the time taken to print them; their total size; and, finally, the time taken to check them in Coq.

Module	LOC	tc(s)	# certs	pp(s)	size (MB)	Coq
Prims	57	4.7	16	8	5.4	3m
Util	131	0.7	9	10	9.8	6m
FiniteSet	344	3.7	30	90	71.8	1h
Terms	248	0.6	7	12	15.0	9m
Env	254	2.4	12	43	42.4	32m
FreeNames	417	14.3	12	204	197.0	3h
Binders	333	9.3	7	144	156.0	2h
Subst	482	21.1	11	232	246.0	3h
Unify	118	4.3	7	22	70.3	19m
Z3Interface	25	1.9	3	1	53.6	2m
Typing	2730	860.2	101	7641	6426.6	23d
Total	5139	15m23s	215	2h 20m 7s	7.3GB	24d

The table begins with several smaller modules. Prims is the standard prelude for F*; Util provides auxiliary functions such as zip and map, typed with their precise specifications. FiniteSet provides sets and sequences with predicates for set membership, partition, and permutation. We use these to represent typechecking environments. Terms defines the core F* syntax, and corresponds to the inductive types for F* abstract syntax shared with Coq, i.e., kind, typ, value, expr etc. Env defines various types also shared with Coq, such as icompartment and environment. FreeNames provides functions to collect free term-level and type-level variables. Binders defines bindings of term-level and type-level variables. Subst provides both an axiomatization of type and value substitutions as an inductive predicate and functions that do the substitutions and build proofs of their predicates. Unify is used to compute the set of equalities when typing pattern matching.

Typing, the main module, contains the shared definitions of the reference type system, and implements functions that compute derivations for expression typing (etyping), value typing, subtyping, type conversion, and well-formedness checks on the environment. CoreTyping also includes a small module Z3Interface, which, for the moment, simply admits Z3 queries, since these have already been performed on the source AST. This last point is a significant limitation of our current implementation, although not a fundamental one. We plan to integrate CoreTyping with prior work on Fine and DCIL, a predecessor of F*, that extracts and type checks SMT proofs from Z3 (Chen et al. 2010), thus removing it from the TCB.

Overall, it takes the F⋆ compiler over 15 minutes to internally produce a reference typing for the CoreTyping module. This includes the time taken by the source typechecker, the calls to Z3 that it makes, followed by the translation into the core AST, and finally the time taken for CoreTyping to typecheck itself. Compressing and printing the 7 GB of certificates takes longer—roughly 2 hours and 20 minutes. We ran this experiment on a 6 core 3.2GHz Intel Xeon workstation, with 16 GB of RAM, running Windows 7.

Checking the certificates in Coq takes much longer. To give an idea of the impact of our optimizations, we turn them off one at a time and report the checking time for a sample module (Env) on the workstation. Reflection greatly helps as we heavily rely on list properties; it also reduces certificate size by 65%. To our surprise, Coq is much faster on certificates split into smaller pieces. Opacity reduces the memory footprint but has less impact on time.

base time	−reflection	−strings	−splitting	−opacity
13m27s	35m4s	37m54s	25m23s	14m14s

Overall, checking all certificates for CoreTyping took 24 machine-days, spread across 6 machines. Beside the workstation, we used five dual core 2.8 GHz Intel Xeon machines with 24GB of RAM running Windows 7. On all the machines we used a 64-bit build of Coq for Linux running under Oracle Virtual Box. Fortunately, with self-certification complete, we never have to export a certificate again. F⋆ has been emancipated.

8. Conclusions

In summary, we have presented *self-certification*, a general method by which a certifying typechecker in a language of suitable expressive power can be bootstrapped into a certified typechecker. We have implemented this technique for F⋆, a dependently typed programming language for .NET, and relied on a formalization of the theory of F⋆ in Coq to prove our methodology sound.

With an efficient certified typechecker in place, we aim to use it for a number of applications. Reflecting the security-oriented focus of prior work on F⋆, we hope to write applications that dynamically receive, load, core-typecheck, and run mobile code, possibly along the lines of code-carrying authorization (Maffeis et al. 2008). Also, in work currently underway, we use F⋆ to build certified execution engines for various authorization logics.

We conclude with some perspective: mechanized metatheories for programming languages are gaining popularity, and rightly so. However, until now, these mechanization efforts have only served to increase the reliability of formal results, without much impact on the status of language implementations. Self-certification provides a path to build a certified core checker with an effort that compares favorably with mechanizing the metatheory in the first place. We hope other language designers will give it a try.

References

T. Acar, C. Fournet, and D. Shumow. Design and verification of a crypto-agile distributed key manager. Technical report, MSR, 2010.

A. W. Appel. Axiomatic bootstrapping: a guide for compiler hackers. *ACM TOPLAS*, 16, November 1994.

M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In *ITP*, 2010.

L. Augustsson. Cayenne: A language with dependent types. In *ICFP*, 1998.

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

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL*, 2008.

B. Barras. Sets in coq, coq in sets. *J. Formalized Reasoning*, 2010.

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

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

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

C. Casinghino, H. D. Eades, G. Kimmell, V. Sjöberg, T. Sheard, A. Stump, and S. Weirich. The preliminary design of the Trellys core language. In *PLPV*, 2011.

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

A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010a.

A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. PLDI, 2010b.

C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *PLDI*, 2000.

J. Davis. *A self-verifying theorem prover*. PhD thesis, U.T. Austin, 2009.

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

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

T. Hart and M. Levin. The new compiler. AI Memo 39, MIT, 1962.

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

C. Keller and B. Werner. Importing HOL Light into Coq. In *ITP*, 2010.

X. Leroy. A locally nameless solution to the POPLmark challenge. Research report 6098, INRIA, Jan. 2007.

X. Leroy. The CompCert verified compiler, software and commented proof, Mar. 2011.

P. Letouzey. Coq extraction, an overview. In *LTA '08*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon. Code-carrying authorization. In *ESORICS '08*, 2008.

C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming School*, 2004.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3), 1999.

M. Moskal. Rocket-fast proof checking for SMT solvers. In *TACAS*, 2008.

M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving*, Aug. 2011.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, 2008.

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

R. Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. 1998.

M. Sozeau. Equations: A dependent pattern-matching compiler. *LNCS*, 6172, 2010.

A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in Guru. In *PLPV*, 2008.

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

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

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, Sept. 2011. See also the full paper at MSR-TR-2011-37.

The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.3*. INRIA, 2011. At URL `http://coq.inria.fr/`.

H. Xi. Applied type system: Extended abstract. In *Types for Proofs and Programs*, pages 394–408, 2003.