Refinement Types for Secure Implementations

Jesper Bengtson
Uppsala University

Karthikeyan Bhargavan
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Sergio Maffeis
Imperial College London

Revision of November 2010

Technical Report
MSR–TR–2008–118

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Refinement Types for Secure Implementations

Jesper Bengtson
Uppsala University

Karthikeyan Bhargavan
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Sergio Maffeis
Imperial College London

Abstract

We present the design and implementation of a typechecker for verifying security properties of the source code of cryptographic protocols and access control mechanisms. The underlying type theory is a λ-calculus equipped with refinement types for expressing pre- and post-conditions within first-order logic. We derive formal cryptographic primitives and represent active adversaries within the type theory. Well-typed programs enjoy assertion-based security properties, with respect to a realistic threat model including key compromise. The implementation amounts to an enhanced typechecker for the general purpose functional language F#; typechecking generates verification conditions that are passed to an SMT solver. We describe a series of checked examples. This is the first tool to verify authentication properties of cryptographic protocols by typechecking their source code.

1 Introduction

The goal of this work is to verify the security of implementation code by typing. Here we are concerned particularly with authentication and authorization properties.

We develop an extended typechecker for code written in F# (a variant of ML) [Syme et al., 2007] and annotated with refinement types that embed logical formulas. We use these dependent types to specify access-control and cryptographic properties, as well as desired security goals. Typechecking then ensures that the code is secure.

We evaluate our approach on code implementing authorization decisions and on reference implementations of security protocols. Our typechecker verifies security properties for a realistic threat model that includes a symbolic attacker, in the style of Dolev and Yao [1983], who is able to create arbitrarily many principals, create arbitrarily many instances of each protocol roles, send and receive network traffic, and compromise arbitrarily many principals.

Verifying Cryptographic Implementations In earlier work, Bhargavan et al. [2008b] advocate the cryptographic verification of *reference implementations* of protocols, rather than their handwritten models, in order to minimize the gap between executable and verified code. They automatically extract models from F# code and, after applying various program transformations, pass them to ProVerif, a cryptographic analyzer [Blanchet, 2001, Abadi and Blanchet, 2005]. Their approach yields verified security for very detailed models, but also demands considerable care in programming, in order to control the complexity of global cryptographic analysis for giant protocols. Even if ProVerif scales up remarkably well in practice, beyond a few message exchanges, or a few hundred lines of F#, verification becomes long (up to a few days) and unpredictable (with trivial code changes leading to divergence).

Cryptographic Verification meets Program Verification In parallel with the development of specialist tools for cryptography, verification tools in general are also making rapid progress, and can deal with much larger programs [see for example Flanagan et al., 2002, Filliâtre and Marché, 2004, Barnett et al., 2005, Régis-Gianas and Pottier, 2008]. To verify the security of programs with some cryptography, we would like to combine both kinds of tools. However, this integration is delicate: the underlying assumptions of cryptographic models to account for active adversaries typically differ from those made for general-purpose program verification. On the other hand, modern applications involve a large amount of (non-cryptographic) code and extensive libraries, sometimes already verified; we'd rather benefit from this effort.

Authorization by Typing Logic is now a well established tool for expressing and reasoning about authorization policies. Although many systems rely on dynamic authorization engines that evaluate logical queries against local stores of facts and rules, it is sometimes possible to enforce policies statically. Thus, Fournet et al. [2007a,b] treat policy enforcement as a type discipline; they develop their approach for typed π-calculi, supplemented with cryptographic primitives. Relying on a "says" modality in the logic, they also account for partial trust (in logic specification) in the face of

This is the November 2010 revision of technical report MSR-TR-2008-118, which appeared first in September 2008. An abridged, preliminary version appears in the proceedings of the *21st IEEE Computer Security Foundations Symposium (CSF 2008)*.

partial compromise (in their implementations). The present work is an attempt to develop, apply, and evaluate this approach for a general-purpose programming language.

Outline of the Implementation Our prototype tool, named F7, takes as input module interfaces (similar to F# module interfaces but with extended types) and module implementations (in plain F#). It typechecks implementations against interfaces, and also generates plain F# interfaces by erasure. Using the F# compiler, generated interfaces and verified implementations can then be compiled as usual.

Our tool performs typechecking and partial type inference, relying on an external theorem prover for discharging the logical conditions generated by typing. We currently use plain first-order logic (rather than an authorization-specific logic) and delegate its proofs to Z3 [de Moura and Bjørner, 2008], a solver for Satisfiability Modulo Theories (SMT). Thus, in comparison with previous work, we still rely on an external prover, but this prover is being developed for general program verification, not for cryptography; also, we use this prover locally, to discharge proof obligations at various program locations, rather than rely on a global translation to a cryptographic model.

Reflecting our assumptions on cryptography and other system libraries, some modules have two implementations: a symbolic implementation used for extended typing and symbolic execution, and a concrete implementation used for plain typing and distributed execution. We have access to a collection of F# test programs already analyzed using dual implementations of cryptography [Bhargavan et al., 2008b], so we can compare our new approach to prior work on model extraction to ProVerif. Unlike ProVerif, typechecking requires annotations that include pre- and post-conditions. On the other hand, these annotations can express general authorization policies, and their use makes typechecking more compositional and predictable than the global analysis performed by ProVerif. Moreover, typechecking succeeds even on code involving recursion and complex data structures.

Outline of the Theory We justify our extended typechecker by developing a formal type theory for a core of F#: a concurrent call-by-value λ-calculus named RCF.

To represent pre- and post-conditions, our calculus has standard dependent functions and pairs, and a form of refinement types [Freeman and Pfenning, 1991, Xi and Pfenning, 1999]. A *refinement type* takes the form $\{x : T \mid C\}$; a value M of this type is a value of type T such that the formula $C\{M/x\}$ holds. (Another name for the construction is *predicate subtyping* [Rushby et al., 1998]; $\{x : T \mid C\}$ is the subtype of T characterized by the predicate C.)

To represent security properties, expressions may assume and assert formulas in first-order logic. An expression is *safe* when no assertion can ever fail at run time. By annotating programs with suitable formulas, we formalize security properties, such as authentication and authorization, as expression safety.

Our F# code is written in a functional style, so pre- and post-conditions concern data values and events represented by logical formulas; our type system does not (and need not for our purposes) directly support reasoning about mutable state, such as heap-allocated structures.

Contributions First, we formalize our approach within a typed concurrent λ-calculus. We develop a type system with refinement types that carry logical formulas, building on standard techniques for dependent types, and establish its soundness.

Second, we adapt our type system to account for active (untyped) adversaries, by extending subtyping so that all values manipulated by the adversary can be given a special universal type (Un). Our calculus has no built-in cryptographic primitives. Instead, we show how a wide range of cryptographic primitives can be symbolically coded (and typed) in the calculus, using a seal abstraction [Morris, 1973, Sumii and Pierce, 2007]. The corresponding robust safety properties then follow as a corollary of type safety.

Third, experimentally, we implement our approach as an extension of F#, and develop a new typechecker (with partial type inference) based on Z3 (a fast, incomplete, first-order logic prover).

Fourth, we evaluate our approach on a series of programming examples, involving authentication and authorization properties of protocols and applications; this indicates that our use of refinement types is an interesting alternative to global verification tools for cryptography, especially for the verification of executable reference implementations.

Contents The paper is organized as follows. Section 2 presents our core language with refinement types, and illustrates it by programming access control policies. Section 3 adds typed support for cryptography, using an encoding based on seals, and illustrates it by implementing MAC-based authentication protocols. Section 4 describes our type system and its main properties. Sections 5 and 6 report on the prototype implementation and our experience with programming protocols with our type discipline. Section 7 discusses related work and Section 8 concludes.

Appendixes provide additional details. Appendix A describes the logic and our usage of Z3. Appendix B defines the semantics and safety of expressions. Appendix C establishes properties of the type system. Appendix D details derived forms for types and expressions.Appendix E gives typed encodings for formal cryptography primitives. Appendix F includes the code of an extended example.

2 A Language with Refinement Types

Our calculus is an assembly of standard parts: call-by-value dependent functions, dependent pairs, sums, iso-recursive types, message-passing concurrency, refinement types, subtyping, and a universal type Un to model attacker knowledge. This is essentially the Fixpoint Calculus (FPC) [Gunter, 1992], augmented with concurrency and refinement types. Hence, we adopt the name Refined Concurrent FPC, or RCF for short. This section introduces its syntax, semantics, and type system (apart from Un), together with an example application. Section 3 introduces Un and applications to cryptographic protocols. (Any ambiguities in the informal presentation should be clarified by the semantics in Appendix B and the type system in Section 4.)

2.1 Expressions, Evaluation, and Safety

An *expression* represents a concurrent, message-passing computation, which may return a *value*. A state of the computation consists of (1) a multiset of expressions being evaluated in parallel; (2) a multiset of messages sent on channels but not yet received; and (3) the *log*, a multiset of assumed formulas. The multisets of evaluating expressions and unread messages model a configuration of a concurrent or distributed system; the log is a notional central store of logical formulas, used only for specifying correctness properties.

We write $S \vdash C$ to mean that a formula C logically follows from a set S of formulas. In our implementation, C is some formula in (untyped) first-order logic with equality. In our intended models, terms denote closed values of RCF, and equality $M = N$ is interpreted as syntactic identity between values. (Appendix A gives the details.)

Formulas and Deducibility:

C	logical formula
$\{C_1,\ldots,C_n\} \vdash C$	logical deducibility

We assume collections of *names*, *variables*, and *type variables*. A name is an identifier, generated at run time, for a channel, while a variable is a placeholder for a value. If ϕ is a phrase of syntax, we write $\phi\{M/x\}$ for the outcome of substituting a value M for each free occurrence of the variable x in ϕ. We identify syntax up to the capture-avoiding renaming of bound names and variables. We write $fnfv(\phi)$ for the set of names and variables occurring free in a phrase of syntax ϕ. We say a phrase is *closed* to mean it has no free variables (although it may have free names).

Syntax of Values and Expressions:

a,b,c	name
x,y,z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of recursive type
$M,N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M,N)	pair
$h\ M$	construction
$A,B ::=$	expression
M	value
$M\ N$	application
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)
let $(x,y) = M$ **in** A	pair split (scope of x, y is A)
match M **with**	constructor match
$h\ x \to A$ **else** B	(scope of x is A)
$(\nu a)A$	restriction (scope of a is A)
$A \upharpoonright B$	fork
$a!M$	transmission of M on channel a
$a?$	receive message off channel
assume C	assumption of formula C
assert C	assertion of formula C

To evaluate M, return M at once. To evaluate $M\ N$, if $M = \textbf{fun}\ x \to A$, evaluate $A\{N/x\}$. To evaluate $M = N$, if the two values M and N are the same, return **true** $\triangleq$ inr$()$; otherwise, return **false** $\triangleq$ inl$()$. To evaluate **let** $x = A$ **in** B, first evaluate A; if evaluation returns a value M, evaluate $B\{M/x\}$. To evaluate **let** $(x_1,x_2) = M$ **in** A, if $M = (N_1,N_2)$, evaluate $A\{N_1/x_1\}\{N_2/x_2\}$. To evaluate **match** M **with** $h\ x \to A$ **else** B, if $M = h\ N$ for some N, evaluate $A\{N/x\}$; otherwise, evaluate B.

To evaluate $(\nu a)A$, generate a globally fresh channel name c, and evaluate $A\{c/a\}$. To evaluate $A \upharpoonright B$, start a parallel thread to evaluate A (whose return value will be discarded), and evaluate B. To evaluate $a!M$, emit message M on channel a, and return $()$ at once. To evaluate $a?$, block until some message N is on channel a, remove N from the channel, and return N.

To evaluate **assume** C, add C to the log, and return $()$. To evaluate **assert** C, return $()$. If $S \vdash C$, where S is the set of logged formulas, we say the assertion *succeeds*; otherwise, we say the assertion *fails*. Either way, it always returns $()$.

Expression Safety:

A closed expression A is *safe* if and only if, in all evaluations of A, all assertions succeed. (See Appendix B for formal details.)

The only way for an expression to be unsafe is for an evaluation to lead to an **assert** C, where C does not follow from the current log of assumed formulas. Hence, an expression may fail in other ways while being safe according to this definition. For example, the restriction $(\nu a)a?$ is safe, although it *deadlocks* in the sense no message can be sent on the fresh channel a and so $a?$ blocks forever. The application $()$ $(\mathbf{fun}\, x \to x)$ is safe, but illustrates another sort of failure: it tries to use $()$ as a function, and so is *stuck* in the sense that evaluation cannot proceed.

Assertions and assumptions are annotations for expressing correctness properties. Inasmuch as our notion of safety is relative to the assumptions executed during evaluation, these assumptions must be carefully reviewed.

There is no mechanism in RCF to branch based on whether or not a formula is derivable from the current log. Our intention is to verify safety statically. If we know statically that an expression is safe, there is no reason to implement the log of assumed expressions because every assertion is known to succeed.

Once assumed, a formula remains in the log for the whole run. Thus, if an assert succeeds, then, later in the run, any other assert with the same formula will also succeed. Hence, our notion of safety captures stable safety properties, which is adequate for verifying the security of concurrent protocols.

2.2 Types and Subtyping

We outline the type system; the main purpose for typechecking an expression is to establish its safety. We assume a collection of *type variables*, ranged over by α, β. For any phrase ϕ, the set $\mathit{fnfv}(\phi)$ includes the type variables, as well as the names and (value) variables, that occur free in ϕ. Notice that no types or type variables occur in the syntax of values or expressions. If ϕ is a phrase of syntax, we write $\phi\{T/\alpha\}$ for the outcome of substituting a type T for each free occurrence of the type variable α in ϕ.

Syntax of Types:

$H,T,U,V ::=$	type
unit	unit type
$\Pi x : T.\ U$	dependent function type (scope of x is U)
$\Sigma x : T.\ U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
α	type variable
$\{x : T \mid C\}$	refinement type (scope of x is C)
$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$	ok-type
$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$	Boolean type

(The notation _ denotes an anonymous variable that by convention occurs nowhere else.)

A value of type unit is the unit value $()$. A value of type $\Pi x : T.\ U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$. A value of type $\Sigma x : T.\ U$ is a pair (M,N) such that M has type T and N has type $U\{M/x\}$. A value of type $T+U$ is either $\mathsf{inl}\ M$ where M has type T, or $\mathsf{inr}\ N$ where N has type U. A value of type $\mu\alpha.T$ is a construction $\mathsf{fold}\ M$, where M has the (unfolded) type $T\{\mu\alpha.T/\alpha\}$. A type variable is a placeholder for a type, such as a recursive type. A value of type $\{x : T \mid C\}$ is a value M of type T such that the formula $C\{M/x\}$ follows from the log.

As usual, we can define syntax-directed typing rules for checking that the value of an expression is of type T, written $E \vdash A : T$, where E is a *typing environment*. The environment tracks the types of variables and names in scope.

The core principle of our system is *safety by typing*:

Theorem 1 (Safety) *If $\varnothing \vdash A : T$ then A is safe.*

Proof: See Appendix C. □

Section 4 has all the typing rules; the majority are standard. Here, we explain the intuitions for the rules concerning refinement types, assumptions, and assertions.

The judgment $E \vdash C$ means C is deducible from the formulas mentioned in refinement types in E. For example:

- If E includes $y : \{x : T \mid C\}$ then $E \vdash C\{y/x\}$.

Consider the refinement types $T_1 = \{x_1 : T \mid \mathsf{P}(x_1)\}$ and $T_2 = \{x_2 : \mathsf{unit} \mid \forall z.\mathsf{P}(z) \Rightarrow \mathsf{Q}(z)\}$. If $E = (y_1 : T_1, y_2 : T_2)$ then $E \vdash \mathsf{Q}(y_1)$ via the rule above plus first-order logic.

The introduction rule for refinement types is as follows.

- If $E \vdash M : T$ and $E \vdash C\{M/x\}$ then $E \vdash M : \{x : T \mid C\}$.

A special case of refinement is an *ok-type*, written $\{C\}$, and short for $\{\_ : \mathsf{unit} \mid C\}$: a type of tokens that a formula holds. For example, up to variable renaming, $T_2 = \{\forall z.\mathsf{P}(z) \Rightarrow \mathsf{Q}(z)\}$. The specialized rules for ok-types are:

- If E includes $x : \{C\}$ then $E \vdash C$.
- A value of type $\{C\}$ is $()$, a token that C holds.

The type system includes a subtype relation $E \vdash T <: T'$, and the usual subsumption rule:

- If $E \vdash A : T$ and $E \vdash T <: T'$ then $E \vdash A : T'$.

Refinement relates to subtyping as follows. (To avoid confusion, note that True is a logical formula, which always holds, while $\mathbf{true}$ is a Boolean value, defined as $\mathsf{inr}\ ()$.)

- If $T <: T'$ and $C \vdash C'$ then $\{x : T \mid C\} <: \{x : T' \mid C'\}$.
- $\{x : T \mid \mathsf{True}\} <:> T$.

For example, $\{x : T \mid C\} <: \{x : T \mid \mathsf{True}\} <: T$.

We typecheck $\mathbf{assume}$ and $\mathbf{assert}$ as follows.

- $E \vdash \mathbf{assume}\ C : \{C\}$.

- If $E \vdash C$ then $E \vdash$ **assert** C : unit.

By typing the result of **assume** as $\{C\}$, we track that C can subsequently be assumed to hold. Conversely, for a well-typed **assert** to be guaranteed to succeed, we must check that C holds in E. This is sound because when typechecking any A in E, the formulas deducible from E are a lower bound on the formulas in the log whenever A is evaluated.

For example, we can derive A_{ex} : unit where A_{ex} is the following, where Foo and Bar are nullary predicate symbols.

$$\begin{array}{l}\textbf{let } x = \textbf{assume } \mathsf{Foo}() \Rightarrow \mathsf{Bar}() \textbf{ in}\\ \textbf{let } y = \textbf{assume } \mathsf{Foo}() \textbf{ in assert } \mathsf{Bar}()\end{array}$$

By the rule for assumptions we have:

$$\begin{array}{l}\textbf{assume } \mathsf{Foo}() \Rightarrow \mathsf{Bar}() : \{\mathsf{Foo}() \Rightarrow \mathsf{Bar}()\}\\ \textbf{assume } \mathsf{Foo}() : \{\mathsf{Foo}()\}\end{array}$$

The rule for checking a let-expression is:

- If $E \vdash A : T$ and $E, x{:}T \vdash B : U$ then $E \vdash \textbf{let } x = A \textbf{ in } B : U$.

By this rule, to show A_{ex} : unit it suffices to check

$$E \vdash \textbf{assert } \mathsf{Bar}() : \mathsf{unit}$$

where $E = x : \{\mathsf{Foo}() \Rightarrow \mathsf{Bar}()\}, y : \{\mathsf{Foo}()\}$. The displayed judgment follows by the rule for assertions as we can derive $E \vdash \mathsf{Bar}()$, since we have both $E \vdash \mathsf{Foo}() \Rightarrow \mathsf{Bar}()$ and $E \vdash \mathsf{Foo}()$. Thus A_{ex} is safe.

2.3 Formal Interpretation of our Typechecker

We interpret a large class of F# expressions and modules within our calculus. To enable a compact presentation of the semantics of RCF, there are two significant differences between expressions in these languages. First, the formal syntax of RCF is in an intermediate, reduced form (reminiscent of A-normal form [Sabry and Felleisen, 1993]) where **let** $x = A$ **in** B is the only construct to allow sequential evaluation of expressions. As usual, $A;B$ is short for **let** $\_ = A$ **in** B, and **let** $f\ x = A$ is short for **let** $f =$ **fun** $x \to A$. More notably, if A and B are proper expressions rather than being values, the application $A\ B$ is short for **let** $f = A$ **in** (**let** $x = B$ **in** $f\ x$). In general, the use in F# of arbitrary expressions in place of values can be interpreted by inserting suitable lets.

The second main difference is that the RCF syntax for communication and concurrency ($(\nu a)A$, $A \upharpoonleft B$, $a?$, and $a!M$) is in the style of a process calculus. In F# we express communication and concurrency via a small library of functions, which is interpreted within RCF as follows.

Functions for Communication and Concurrency:

$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$

$\mathsf{chan} \triangleq \textbf{fun}\, x \to (\nu a)(\textbf{fun}\, x \to a!x, \textbf{fun}\, \_ \to a?)$	
$\mathsf{send} \triangleq \textbf{fun}\, c\ x \to \textbf{let}\ (s,r) = c\ \textbf{in}\ s\ x$	send x on c
$\mathsf{recv} \triangleq \textbf{fun}\, c \to \textbf{let}\ (s,r) = c\ \textbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \textbf{fun}\, f \to (f() \upharpoonleft ())$	run f in parallel

We define references in terms of channels.

Functions for References:

$(T)\textbf{ref} \triangleq (T)\mathsf{chan}$

$\textbf{ref}\ M \triangleq \textbf{let}\ r = \mathsf{chan}\ \texttt{"r"}\ \textbf{in}$ $\mathsf{send}\ r\ M; r$	new reference to M
$!M \triangleq \textbf{let}\ x = \mathsf{recv}\ M\ \textbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathsf{recv}\ M; \mathsf{send}\ M\ N$	update M with N

We also assume standard encodings of strings, numeric types, Booleans, tuples, records, algebraic types (including lists) and pattern-matching, and recursive functions. (Appendix D lists the full details.) RCF lacks full-fledged polymorphism, but by duplicating definitions at multiple monomorphic types we can recover the effect of having polymorphic definitions.

We use the following notations for functions with preconditions, and non-empty tuples (instead of directly using the core syntax for dependent function and pair types). We usually omit conditions of the form {True} in examples.

Derived Notation for Functions and Tuples:

$$\{x_1 : T_1 \mid C_1\} \to U \triangleq \Pi x_1 : \{x_1 : T_1 \mid C_1\}.\ U$$

$$(x_1 : T_1 * \cdots * x_n : T_n)\{C\} \triangleq \begin{cases} \Sigma x_1 : T_1.\ \ldots \Sigma x_{n-1} : T_{n-1}.\ \{x_n : T_n \mid C\} & \text{if } n > 0 \\ \{C\} & \text{otherwise}\end{cases}$$

To treat **assume** and **assert** as F# library functions, we follow the convention that constructor applications are interpreted as formulas (as well as values). If h is an algebraic type constructor of arity n, we treat h as a predicate symbol of arity n, so that $h(M_1, \ldots, M_n)$ is a formula.

All of our example code is extracted from two kinds of source files: either extended typed interfaces (.fs7) that declare types, values, and policies; or the corresponding F# implementation modules (.fs) that define them.

We sketch how to interpret interfaces and modules as tuple types and expressions. In essence, an *interface* is a sequence **val** $x_1 : T_1 \ \ldots$ **val** $x_n : T_n$ of *value declarations*, which we interpret by the tuple type $(x_1 : T_1 * \cdots * x_n : T_n)$. A *module* is a sequence **let** $x_1 = A_1 \ \ldots$ **let** $x_n = A_n$ of *value definitions*, which we interpret by the expression **let** $x_1 = A_1$ **in** $\ldots$ **let** $x_n = A_n$ **in** $(x_1, \ldots, x_n)$. If A and T are the interpretations of a module and an interface, our tool checks whether $A : T$. Any type declarations are simply interpreted as abbreviations for types, while a policy statement

assume C is treated as a declaration **val** $x : \{C\}$ plus a definition **let** $x =$ **assume** C for some fresh x.

2.4 Example: Access Control in Partially-Trusted Code

This example illustrates static enforcement of file access control policies in code that is typechecked but not necessarily trusted, such as applets or plug-ins. (See, for example, Dean et al. [1996], Pottier et al. [2001], Abadi and Fournet [2003], and Abadi [2007] for a more general discussion of security mechanisms for partially-trusted code.)

We first declare a type for the logical facts in our policy. We interpret each of its constructors as a predicate symbol: here, we have two basic access rights, for reading and writing a given file, and a property stating that a file is public.

```
type facts =
  | CanRead of string // read access
  | CanWrite of string // write access
  | PublicFile of string // some file attribute
```

For instance, the fact CanRead("C:/README") represents read access to "C:/README". We use these facts to give restrictive types to sensitive primitives. For instance, the declarations

```
val read: file:string{CanRead(file)} → string
val delete: file:string{CanWrite(file)} → unit
```

demand that the function read be called only in contexts that have previously established the fact CanRead(M) for its string argument M (and similarly for write). These demands are enforced at compile time, so in F# the function read just has type string → string and its implementation may be left unchanged.

More operationally, to illustrate our formal definition of expression safety, we may include assertions, and define

```
let read file = assert(CanRead(file)); "data"
let delete file = assert(CanWrite(file))
```

Library writers are trusted to include suitable **assume** statements. They may declare policies, in the form of logical deduction rules, declaring for instance that every file that is writable is also readable:

```
assume ∀x. CanWrite(x) ⇒ CanRead(x)
```

and they may program helper functions that establish new facts. For instance, they may declare

```
val publicfile: file : string → unit{ PublicFile(file) }
assume ∀x. PublicFile(x) ⇒ CanRead(x)
```

and implement publicfile as a partial function that dynamically checks its filename argument.

```
let publicfile f =
  if f = "C:/public/README" then assume (PublicFile(f))
  else failwith "not a public file"
```

The F# library function failwith throws an exception, so it never returns and can safely be given the polymorphic type string → α, where α can be instantiated to any RCF type. (We also coded more realistic dynamic checks, based on dynamic lookups in mutable, refinement-typed, access-control lists. We omit their code for brevity.)

To illustrate our code, consider a few sample files, one of them writable:

```
let pwd = "C:/etc/password"
let readme = "C:/public/README"
let tmp = "C:/temp/tempfile"
let _ = assume (CanWrite(tmp))
```

Typechecking the test code below returns two type errors:

```
let test:unit =
  delete tmp; // ok
// delete pwd; // type error
  let v1 = read tmp in // ok, using 1st logical rule
// let v2 = read readme in // type error
  publicfile readme; let v3 = read readme in () // ok
```

For instance, the second delete yields the error "Cannot establish formula CanWrite(pwd) at acls.fs(39,9)-(39,12)."

In the last line, the call to publicfile dynamically tests its argument, ensuring PublicFile(readme) whenever the final expression read readme is evaluated. This fact is recorded in the environment for typing the final expression.

From the viewpoint of fully-trusted code, our interface can be seen as a self-inflicted discipline—indeed, one may simply **assume** ∀x.CanRead(x). In contrast, partially-trusted code (such as mobile code) would not contain any **assume**. By typing this code against our library interface, possibly with a policy adapted to the origin of the code, the host is guaranteed that this code cannot call read or write without first obtaining the appropriate right.

Although access control for files mostly relies on dynamic checks (ACLs, permissions, and so forth), a static typing discipline has advantages for programming partially-trusted code: as long as the program typechecks, one can safely re-arrange code to more efficiently perform costly dynamic checks. For example, one may hoist a check outside a loop, or move it to the point a function is created, rather than called, or move it to a point where it is convenient to handle dynamic security exceptions.

In the code below, for instance, the function reader can be called to access the content of file readme in any context with no further run time check.

```
let test_higher_order:unit =
  let reader: unit → string =
    (publicfile readme; (fun () → read readme)) in
// let v4 = read readme in // type error
  let v5 = reader () in () // ok
```

Similarly, we programmed (and typed) a function that merges the content of all files included in a list, under the assumption that all these files are readable, declared as

```
val merge: (file:string{ CanRead(file) }) list → string
```

where list is a type constructor for lists, with a standard implementation typed in RCF.

We finally illustrate the use of refinement-typed values within imperative data structures to "store" valid formulas. We may declare an access control list (ACL) database as

```
type entry =
  | Readable of x:string{CanRead(x)}
  | Writable of x:string{CanWrite(x)}
  | Nothing
val acls : (string,entry) Db.t
val safe_read: string → string
val readable: file:string → unit{ CanRead(file) }
```

(where Db.t is a type constructor for our simplified typed database library, parameterized by the types of the keys and entries stored in the database) and implement it as:

```
let acls: (string,entry) Db.t = Db.create()
let safe_read file =
  match Db.select acls file with
  | Readable file → read file
  | Writable file → read file
  | _ → failwith "unreadable"
let readable file =
  match Db.select acls file with
  | Readable f when f = file → ()
  | Writable f when f = file → ()
  | _ → failwith "unreadable"
```

Both safe_read and readable lookup an ACL entry and, by matching, either "retrieve" a fact sufficient for reading the file, or fail. The code below illustrates their usage:

```
let test_acls:unit =
  Db.insert acls tmp (Writable(tmp)); // ok
// Db.insert acls tmp (Readable(pwd)); // type error
  Db.insert acls pwd (Nothing); // ok
  let v6 = safe_read pwd in // ok (but dynamically fails)
  let v7 = readable tmp; read tmp in () // ok
```

3 Modelling Cryptographic Protocols

We introduce our technique for specifying security properties of cryptographic protocols by typing.

3.1 Roles and Opponents as Functions

Following Bhargavan et al. [2008b], we start with plain F# functions that create instances of each role of the protocol (such as client or server). The protocols make use of various libraries (including cryptographic functions, explained below) to communicate messages on channels that represent the public network. We model the whole protocol as an F# module, interpreted as before as an expression that exports the functions representing the protocol roles, as well as the network channel [Sumii and Pierce, 2007]. We express authentication properties (correspondences [Woo and Lam, 1993]) by embedding suitable **assume** and **assert** expressions within the code of the protocol roles.

The goal is to verify that these properties hold in spite of an active opponent able to send, receive, and apply cryptography to messages on network channels [Needham and Schroeder, 1978]. We model the opponent as some arbitrary (untyped) expression O which is given access to the protocol and knows the network channels [Abadi and Gordon, 1999]. The idea is that O may use the communication and concurrency features of RCF to create arbitrary parallel instances of the protocol roles, and to send and receive messages on the network channels, in an attempt to force failure of an **assert** in protocol code. Hence, our formal goal is *robust safety*, that no **assert** fails, despite the best efforts of an arbitrary opponent.

Formal Threat Model: Opponents and Robust Safety

A closed expression O is an *opponent* iff O contains no occurrence of **assert**.
A closed expression A is *robustly safe* iff the application $O\ A$ is safe for all opponents O.

(An opponent must contain no **assert**, or less it could vacuously falsify safety.)

3.2 Typing the Opponent

To allow type-based reasoning about the opponent, we introduce a *universal type* Un of data known to the opponent, much as in earlier work [Abadi, 1999, Gordon and Jeffrey, 2003a]. By definition, Un is type equivalent to (both a subtype and a supertype of) all of the following types: unit, $(\Pi x : \mathsf{Un}.\ \mathsf{Un})$, $(\Sigma x : \mathsf{Un}.\ \mathsf{Un})$, $(\mathsf{Un} + \mathsf{Un})$, and $(\mu\alpha.\mathsf{Un})$. Hence, we obtain *opponent typability*, that $O : \mathsf{Un}$ for all opponents O.

It is useful to characterize two *kinds* of type: *public types* (of data that may flow to the opponent) and *tainted types* (of data that may flow from the opponent).

Public and Tainted Types:

Let a type T be *public* if and only if $T <: \mathsf{Un}$.
Let a type T be *tainted* if and only if $\mathsf{Un} <: T$.

We can show that refinement types satisfy the following kinding rules. (Section 4 has kinding rules for the other types, following prior work [Gordon and Jeffrey, 2003b].)

- $E \vdash \{x : T \mid C\} <: \mathsf{Un}$ iff $E \vdash T <: \mathsf{Un}$

- $E \vdash \mathsf{Un} <: \{x : T \mid C\}$ iff $E \vdash \mathsf{Un} <: T$ and $E, x : T \vdash C$

Consider the type $T_1 \triangleq \{x : \mathsf{string} \mid \mathsf{CanRead}(x)\}$. According to the rules above, this type is public, because string is public, but it is only tainted if $\mathsf{CanRead}(x)$ holds for all x (and let's suppose $\mathsf{CanRead}(x)$ does not hold for all x). If we have a value M of this type we can conclude $\mathsf{CanRead}(M)$. The type cannot be tainted, for if it were, we could conclude $\mathsf{CanRead}(M)$ for any M chosen by the opponent.

Dually, for a type that is tainted but not public, consider the function type $T_2 \triangleq T_1 \to \mathsf{Un}$. Specializing the rules to come in Section 4, a function type $T \to U$ is public when T is tainted and U is public, so T_2 is not public. On the other hand, a function type $T \to U$ is tainted when T is public and U is tainted, so T_2 is tainted.

To see why it would be unsafe to consider the type T_2 public, consider the function **fun** $x \to$ **assert** $\mathsf{CanRead}(x)$, which has type T_2. The assertion is safe because x has the refinement type T_1, and so the formula $\mathsf{CanRead}(x)$ holds. Intuitively, it should be safe to give any value of public type to the untyped opponent, but it is unsafe to do so with this function, because the opponent could apply it to some argument x that does not satisfy $\mathsf{CanRead}(x)$.

To summarize, we have the strict inclusions $T_1 <: \mathsf{Un} <: T_2$. This shows that not all types are public (consider T_2), not all types are tainted (consider T_1), hence not all types are equivalent to Un, and Un is not the top type (because T_2 is not its subtype).

Verification of protocols versus an arbitrary opponent is based on a principle of *robust safety by typing*.

Theorem 2 (Robust Safety) *If $\varnothing \vdash A : \mathsf{Un}$ then A is robustly safe.*

Proof: See Appendix C. □

To apply the principle, if expression A and type T are the RCF interpretations of a protocol module and a protocol interface, it suffices by subsumption to check that $A : T$ and T is public. The latter amounts to checking that T_i is public for each declaration **val** $x_i : T_i$ in the protocol interface.

(Some of our example interfaces include declarations of the form **private val** $x_i : T_i$. These declarations are available only within protocol code and are not exported to the attacker, and hence T_i is not necessarily public. We include these declarations for the sake of exposition, and also to inform our typechecker of the enhanced type T_i.)

3.3 A Cryptographic Library

We provide various libraries to support distributed programming. They include polymorphic functions for producing and parsing network representations of values, declared as

```
val pickle: x:α →(p:α pickled)
val unpickle: p:α pickled →(x:α)
```

and for messaging: addr is the type of TCP duplex connections, established by calling connect and listen, and used by calling send and recv. All these functions are public.

The cryptographic library provides a typed interface to a range of primitives, including hash functions, symmetric encryption, asymmetric encryption, and digital signatures. We detail the interface for HMACSHA1, a keyed hash function, used in our examples to build message authentication codes (MACs). This interface declares

```
type α hkey = HK of α pickled Seal
type hmac = HMAC of Un
val mkHKey: unit → α hkey
val hmacsha1: k:α hkey →x:α pickled →h:hmac
val hmacsha1Verify: k:α hkey →xx:Un →h:hmac →x:α
    pickled
```

where hmac is the type of hashes and α hkey is the type of keys used to compute hashes for values of type α.

The function mkHKey generate a fresh key (informally fresh random bytes). The function hmacsha1 computes the joint hash of a key and a pickled value with matching types. The function hmacsha1Verify verifies whether the joint hash of a key and a value (presumed to be the pickled representation of some value of type α) matches some given hash. If verification succeeds, this value is returned, now with the type α indicated in the key. Otherwise, an exception is raised.

Although keyed-hash verification is concretely implemented by recomputing the hash and comparing it to the given hash, this would not meet its typed interface: assume α is the refinement type x : string$\{$CanRead(x)$\}$. In order to hash a string x, one needs to prove $\mathsf{CanRead}(x)$ as a precondition for calling hmacsha1. Conversely, when receiving a keyed hash of x, one would like to obtain $\mathsf{CanRead}(x)$ as a postcondition of the verification—indeed, the result type of hmacsha1Verify guarantees it. At the end of this section, we describe a well-typed symbolic implementation of this interface.

3.4 Example: A Protocol based on MACs

Our first cryptographic example implements a basic one-message protocol with a message authentication code (MAC) computed as a shared-keyed hash; it is a variant of a protocol described and verified in earlier work [Bhargavan et al., 2008b].

We present snippets of the protocol code to illustrate our typechecking method; Appendix F lists the full source code for a similar, but more general protocol. We begin with a typed interface, declaring three types: event for specifying our authentication property; content for authentic payloads; and message for messages exchanged on a public network.

```
type event = Send of string // a type of logical predicate
type content = x:string{Send(x)} // a string refinement
type message = (string ∗ hmac) pickled // a wire format
```

The interface also declares functions, client and server, for invoking the two roles of the protocol.

```
val addr : (string ∗ hmac, unit) addr // a public server address
private val hk: content hkey // a shared secret

private val make: content hkey → content → message
val client: string → unit // start a client

private val check: content hkey → message → content
val server: unit → unit // start a server
```

The client and server functions share two values: a public network address addr where the server listens, and a shared secret key hk. Given a string argument s, client calls the make function to build a protocol message by calling hmacsha1 hk (pickled s). Conversely, on receiving a message at addr, server calls the check function to check the message by calling hmacsha1Verify.

In the interface, values marked as **private** may occur only in typechecked implementations, and hence are not available to the opponent. Conversely, the other values (addr, client, server) must have public types, and may be made available to the opponent.

Authentication is expressed using a single event Send(s) recording that the string s has genuinely been sent by the client—formally, that client(s) has been called. This event is embedded in a refinement type, content, the type of strings s such that Send(s). Thus, following the type declarations for make and check, this event is a pre-condition for building the message, and a post-condition after successfully checking the message.

Consider the following code for client and server:

```
let client text =
  assume (Send(text)); // privileged
  let c = connect addr in
  send c (make hk text)

let server () =
  let c = listen addr in
  let text = check hk (recv c) in
  assert(Send text) // guaranteed by typing
```

The calls to **assume** before building the message and to **assert** after checking the message have no effect at run time (the implementations of these functions simply return ()) but they are used to specify our security policy. In the terminology of cryptographic protocols, **assume** marks a "begin" event, while **assert** marks an "end" event.

Here, the server code expects that the call to check only returns text values previously passed as arguments to client. This guarantee follows from typing, by relying on the types of the shared key and cryptographic functions. On the other hand, this guarantee does not presume any particular cryptographic implementation—indeed, simple variants of our protocol may achieve the same authentication guarantee, for example, by authenticated encryption or digital signature.

Conversely, some implementation mistakes would result in a compile-time type error indicating a possible attack. For instance, removing **private** from the declaration of the authentication key hk, or attempting to leak hk within client, would not be type-correct; indeed, this would introduce an attack on our desired authentication property. Other such mistakes include using the authentication key to hash a plain string, and rebinding text to any other value between the **assume** and the actual MAC computation.

3.5 Example: Logs and Queries

We now relate our present approach to more traditional correspondence properties, stated in terms of run time events. To this end, we explicitly code calls to a secure log function that exclusively records begin- and end-events, and we formulate our security property on the series of calls to this function.

Continuing with our MAC example protocol, we modify the interface as follows:

```
type event = Send of string | Recv of string
private val log : e:event{ ∀x. (e = Recv(x) ⇒ Send(x)) } →
  r:unit{ ∀x. (e = Send(x) ⇒ Send(x)) }
```

The intended correspondence property Recv(x)⇒Send(x) can now be read off the declared type of log. (In this type, Send and Recv are used both as F$^{\#}$ datatype constructors and predicate constructors.)

We also slightly modify the implementation, as follows:

```
let log x = match x with
  | Send text → assume (Send(text))
  | Recv text → assert(Send(text))

let client text =
  log (Send(text)); // we log instead of assuming
  let c = connect addr in
  send c (make hk text)

let server () =
  let c = listen addr in
  let text = check hk (recv c) in
  log (Recv text) // we log instead of asserting
```

The main difference is that **assume** is relegated to the implementation of log; we also omit the redundant **assert** in server code, as the condition follows from the type of both check and log. As a corollary of type soundness, we obtain that, for all runs, every call to log with a Recv event is preceded by a call to log with a matching Send event (by induction on the series of calls to log).

3.6 Example: Principals and Compromise

We now extend our example to multiple principals, with keys shared between each pair of principals. Hence, the keyed hash authenticates not only the message content, but also the sender and the intended receiver. The full implementation is in Appendix F; here we give only the types.

We represent principal names as strings; Send events are now parameterized by the sending and receiving principals, as well as the message text.

```
type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type (;a:prin,b:prin) content = x:string{ Send(a,b,x) }
```

The second event Leak is used in our handling of principal compromise, as described below. The type definition of content has two *value parameters*, a and b; they bind expression variables in the type being defined, much like type parameters bind type variables. (Value parameters appear after type parameters, separated by a semicolon; here, content has no type parameters before the semicolon.)

We store the keys in a (typed, list-based) private database containing entries of the form (a,b,k) where k is a symmetric key of type (;a,b)content hkey shared between a and b.

```
val genKey: prin → prin → unit
private val getKey: a:
  string → b:string → ((;a,b) content) hkey
```

Trusted code can call getKey a b to retrieve a key shared between a and b. Both trusted and opponent code can also call genKey a b to trigger the insertion of a fresh key shared between a and b into the database.

To model the possibility of key leakage, we allow opponent code to obtain a key by calling the function leak:

```
assume ∀a,b,x. ( Leak(a) ) ⇒ Send(a,b,x)
val leak:
  a:prin → b:prin → (unit{ Leak(a) }) ∗ ((;a,b) content) hkey
```

This function first assumes the event Leak(a) as recorded in its result type, then calls getKey a b and returns the key. Since the opponent gets a key shared between a and b, it can generate seemingly authentic messages on a's behalf; accordingly, we declare the policy that Send(a,b,x) holds for any x after the compromise of a, so that leak can be given a public type—without this policy, a subtyping check fails during typing. Hence, whenever a message is accepted, either this message has been sent (with matching sender, receiver, and content), or a key for its apparent sender has been leaked.

3.7 Discussion: Modelling Secrecy

Although this paper focuses on authentication and authorization properties, our type system also guarantees secrecy properties. Without key secrecy, for instance, we would not be able to obtain authenticity by typing for the protocol examples given above.

In a well-typed program, the opponent is given access only to a public interface, so any value passed to the opponent must first be given a public type. On the other hand, the local type of the value does not yield in itself any guarantee of secrecy, since the same value may be given a public type in another environment, under stronger logical assumptions. Informally, the logical formulas embedded in a type indicate the conditions that must hold before values of that type are considered public.

To give a more explicit account of secrecy, we consider a standard "no escape" property that deems a value secret as long as no opponent can gain direct access to the value. (This form of secrecy is adequate for some values; it is weaker than equivalence-based forms of secrecy that further exclude any implicit flow of information from the actual value of a secret to the opponent.)

Robust Secrecy:

Let A be an expression with free variable s. The expression A *preserves the secrecy of* s *unless* C iff the expression **let** $s = ($**fun** $\_ \rightarrow$ **assert** $C)$ **in** A is robustly safe.

This definition does not rely on types; instead, it tests whether the opponent may gain knowledge of s: then, the opponent may also call the function, thereby triggering the guarded assertion **assert** C. By definition of robust safety, the formula C must then follow from the assumptions recorded in the log.

As a simple corollary of Theorem 2 (Robust Safety), we establish a principle of robust secrecy by typing.

Theorem 3 (Robust Secrecy) *If* $s : \{C\} \rightarrow \mathit{unit} \vdash A : \mathit{Un}$, *then* A *preserves the secrecy of* s *unless* C.

Proof: (In this proof, we anticipate the typing rules of Section 4.) By hypothesis, $s : \{C\} \rightarrow \mathsf{unit} \vdash A : \mathsf{Un}$, hence $\varnothing \vdash C$, and thus $\{C\} \vdash$ **assert** $C : \mathsf{unit}$ by (Exp Assert), $\varnothing \vdash ($**fun** $\_ \rightarrow$ **assert** $C) : \{C\} \rightarrow \mathsf{unit}$ by (Val Fun), and $\varnothing \vdash$ **let** $s = ($**fun** $\_ \rightarrow$ **assert** $C)$ **in** $A : \mathsf{Un}$ by (Exp Let). We conclude by Theorem 2 (Robust Safety). □

By inspection of the rules for public kinding, we see that the type $\{C\} \rightarrow \mathsf{unit}$ given to s is public only in environments that entail C, and thus is indeed a type of secrets "unless C holds".

We illustrate secrecy on a two-message protocol example, relying on authenticated, symmetric encryptions instead of MACs. The first message is a session key (k) encrypted under a long-term key; the second message is a secret payload (s) encrypted under the session key. Secrecy is stated unless Leak(a), a fact used below to illustrate the usage of assumptions for modelling key compromise.

We use the following declarations.

```
type empty = u:unit { Leak(a) }
type secret = {nonce:bytes; value:(empty → unit)}
type payload = secret

private val s: payload
private val k0: (payload symkey) symkey

// The protocol uses a fresh session key
// and relies on its authenticated encryption
// client → server : { fresh k }k0
// server → client : { s }k

val addr : (enc, enc) addr
val client: unit → unit
val server: unit → unit
```

Both s (the payload) and k0 (the long-term key) must be declared as private values; otherwise we obtain kinding errors.

We give a definition only for the test secret—the rest of the protocol definitions are similar to those listed above.

```
let s = {nonce = mkNonce();
         value = fun () → assert(Leak(a))} // our test secret
```

We obtain an instance of Theorem 3 (Robust Secrecy) for the expression A that consists of library code plus the protocol code (without the definition of s). As we typecheck the protocol definitions, we would obtain typing errors, for instance, if the client code attempted to leak k0, k, or s on a public channel, or if the server code attempted to encrypt s under a public key instead of k.

We can model the compromise of the client machine by releasing k0 (its only initial secret) to the opponent. The code used to model this situation is typable only with sufficient assumptions: we may for instance define a public function **let** leak()= **assume**(Leak(a)); k0, with an assumption that records the potential loss of secrecy for s.

In a refined example with multiple clients, each with its own long-term key, we may use a more precise secrecy condition, such as $C = \exists$a.(Leak(a)$\wedge$Accept(a)) where Leak(a) records the compromise of a principal named a and Accept(a) records that the server actually accepted to run a session with a as client. Thus, for instance, we may be able to check the secrecy of s despite the compromise of unauthorized clients.

We refer to Gordon and Jeffrey [2005] and Fournet et al. [2007b] for a more general account of secrecy and authorization despite compromise.

3.8 Implementing Formal Cryptography

Morris [1973] describes *sealing*, a programming language mechanism to provide "authentication and limited access." Sumii and Pierce [2007] provide a primitive semantics for sealing within a λ-calculus, and observe the close correspondence between sealing and various formal characterizations of symmetric-key cryptography.

In our notation, a *seal* k for a type T is a pair of functions: the *seal function for* k, of type $T \to$ Un, and the *unseal function for* k, of type Un $\to T$. The seal function, when applied to M, wraps up its argument as a *sealed value*, informally written $\{M\}_k$ in this discussion. This is the only way to construct $\{M\}_k$. The unseal function, when applied to $\{M\}_k$, unwraps its argument and returns M. This is the only way to retrieve M from $\{M\}_k$. Sealed values are opaque; in particular, the seal k cannot be retrieved from $\{M\}_k$.

We declare a type of seals, and a function mkSeal to create a fresh seal, as follows.

```
type α Seal = (α → Un) ∗ (Un → α)
val mkSeal: string → α Seal
```

To implement a seal k, we maintain a list of pairs $[(M_1,a_1);\ldots;(M_n,a_n)]$. The list records all the values M_i that have so far been sealed with k. Each a_i is a fresh name representing the sealed value $\{M_i\}_k$. The list grows as more values are sealed; we associate a reference s with the seal k, and store the current list in s. We maintain the invariant that both the M_i and the a_i are pairwise distinct: the list is a one-to-one correspondence.

The function mkSeal below creates a fresh seal, by generating a fresh reference s that holds an empty list; the seal itself is the pair of functions (seal s, unseal s). The code uses the abbreviations **ref**, !, and := displayed in Section 2.

The code also relies on library functions for list lookups:

```
let rec first f xs = match xs with
  | x::xs → (let r = f x in match r with
                          Some(y) → r
                        | None → first f xs)
  | [] → None
let left z (x,y) = if z = x then Some y else None
let right z (x,y) = if z = y then Some x else None
```

The function first, of type ($\alpha \to \beta$ option)$\to \alpha$ list $\to \beta$ option, takes as parameters a function and a list; it applies the function to the elements of the list, and returns the first non-None result, if any; otherwise it returns None. This function is applied to a pair-filtering function left, defined as **let** left z (x,y)= **if** z = x **then** Some y **else** None, to retrieve the first a_i associated with the value being sealed, if any, and is used symmetrically with a function right to retrieve the first M_i associated with the value being unsealed, if any.

```
type α SealRef = ((α ∗ Un) list) ref
let seal: α SealRef → α → Un = fun s m →
  let state = !s in match List.first (List.left m) state with
  | Some(a) → a
  | None →
    let a: Un = Pi.name "a" in
    s := ((m,a)::state); a
let unseal: α SealRef → Un → α = fun s a →
```

```
  let state = !s in match List.first (List.right a) state with
  | Some(m) → m
  | None → failwith "not a sealed value"
let mkSeal (n:string) : α Seal =
  let s = ref []:α SealRef in
    (seal s, unseal s)
```

Irrespective of the type α for M, sealing returns a public name a, which may be communicated on some unprotected network, and possibly passed to the opponent.

In a variant of seal, we always generate a fresh value a, rather than perform a list lookup; this provides support for non-deterministic encryption and signing (with different, unrelated values for different encryptions of the same value).

Within RCF, we derive formal versions of cryptographic operations, in the spirit of Dolev and Yao [1983], but based on sealing rather than algebra. Our technique depends on being within a calculus with functional values. Thus, in contrast with previous work in cryptographic π-calculi [Gordon and Jeffrey, 2003b, Fournet et al., 2007b] where all cryptographic functions were defined and typed as primitives, we can now implement these functions and retrieve their typing rules by typechecking their implementations.

Appendix E includes listings for the interface and the (typed) symbolic implementation of cryptography. We use seals to derive formal models for MACs (HMACSHA1), symmetric encryption (AES), asymmetric encryption (RSA), and digital signatures (RSASHA1). For example, the functions that model HMACSHA1 are as follows.

```
let mkHKey ():α hkey =
  let s = mkSeal "hkey" in
    HK s
let hmacsha1 (HK(key)) text =
  let (h,_) = key in
  let t = h text in
    HMAC (t)
let hmacsha1Verify (HK key) text (HMAC h) =
  let (_,hv) = key in
  let x:α pickled = hv h in
    if x = text then x else failwith "hmac verify failed"
```

Keys are modelled as seals; computing and verifying MACs then correspond to uses of sealing and unsealing.

Following the same style, we model RSA encryption using the types and functions below.

```
type β deckey = DK of β symkey Seal
type β enckey = EK of (β symkey → Un)
type penc = RSA of Un

let mkRsaDecKey () : β deckey =
  let s = mkSeal "rsakey" in
    DK(s)
let rsaEncKey (DK dk) =
  let (e,d) = dk in EK(e)
let rsaEncrypt (EK (e)) t = RSA(e t)
let rsaDecrypt (DK k) (RSA msg) =
  let (e,d) = k in d msg
```

RSA decryption keys are modelled as seals. RSA encryption keys are public and can be derived from the corresponding decryption key. Encryption and decryption are modelled as sealing and unsealing.

Our abstract functions for defining cryptographic primitives can be seen as symbolic counterparts to the *oracle functions* commonly used in cryptographic definitions of security [see, for instance, Bellare and Rogaway, 1993]. For example, in a random-oracle model for keyed hash functions, an oracle function would take an input to be hashed, perform a table lookup of previously-hashed inputs, and either return the previous hash value, or generate (and record) a fresh hash value. The main difference is that we rely on symbolic name generation, whereas the oracle relies on probabilistic sampling.

4 A Type System for Robust Safety

The type system consists of a set of inductively defined judgments. Each is defined relative to a *typing environment*, E, which defines the variables and names in scope.

Judgments:

$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, type T is syntactically well-formed
$E \vdash C$	formula C is derivable from E
$E \vdash T :: \nu$	in E, type T has kind ν
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Syntax of Kinds:

$\nu ::= \mathbf{pub} \mid \mathbf{tnt}$	kind (public or tainted)

Let $\overline{\nu}$ satisfy $\overline{\mathbf{pub}} = \mathbf{tnt}$ and $\overline{\mathbf{tnt}} = \mathbf{pub}$.

Syntax of Typing Environments:

$\mu ::=$	environment entry
α	type variable
$\alpha :: \nu$	kinding for recursive type α
$\alpha <: \alpha'$	subtyping for recursive types $\alpha \neq \alpha'$
$a \updownarrow T$	channel name
$x : T$	variable
$E ::= \mu_1, \ldots, \mu_n$	environment

$dom(\alpha) = \{\alpha\}$
$dom(\alpha :: \nu) = \{\alpha\}$
$dom(\alpha <: \alpha') = \{\alpha, \alpha'\}$
$dom(a \updownarrow T) = \{a\}$
$dom(x : T) = \{x\}$
$dom(\mu_1, \ldots, \mu_n) = dom(\mu_1) \cup \cdots \cup dom(\mu_n)$

$recvar(E) = \{\alpha, \alpha' \mid (\alpha <: \alpha') \in E\} \cup \{\alpha \mid (\alpha :: \nu) \in E\}$

If $E = \mu_1, \ldots, \mu_n$ we write $\mu \in E$ to mean that $\mu = \mu_i$ for some $i \in 1..n$. We write $T <:> T'$ for $T <: T'$ and $T' <: T$. Let $\mathit{recvar}(E)$ be just the type variables occurring in kinding and subtyping entries of E. Let E be *executable* if and only if $\mathit{recvar}(E) = \varnothing$. Such environments contain names, variables, and type variables (but no entries $\alpha :: \nu$ or $\alpha <: \alpha'$). Let $\mathit{fnfv}(E) = \bigcup\{\mathit{fnfv}(T) \mid (a \updownarrow T) \in E \vee (x : T) \in E\}$.

Rules of Well-Formedness and Deduction:

$$\text{(Env Empty)}\quad \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env Entry)}\quad \frac{E \vdash \diamond \quad \mathit{fnfv}(\mu) \subseteq \mathit{dom}(E) \quad \mathit{dom}(\mu) \cap \mathit{dom}(E) = \varnothing}{E, \mu \vdash \diamond} \qquad \text{(Type)}\quad \frac{E \vdash \diamond \quad \mathit{fnfv}(T) \subseteq \mathit{dom}(E)}{E \vdash T}$$

$$\text{(Derive)}\quad \frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E) \quad \mathsf{forms}(E) \vdash C}{E \vdash C}$$

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup \mathsf{forms}(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

The function $\mathsf{forms}(E)$ maps an environment E to a set of formulas $\{C_1, \ldots, C_n\}$. We occasionally use this set in a context expecting a formula, in which case it should be interpreted as the conjunction $C_1 \wedge \cdots \wedge C_n$, or True in case $n = 0$. For example, $\mathsf{forms}(x : \{C\}) = \{C\}$. To see this, we calculate as follows.

$$\begin{aligned} &\mathsf{forms}(x : \{C\}) \\ &\quad = \mathsf{forms}(x : \{y : \mathsf{unit} \mid C\}) \quad y \notin \mathit{fv}(C) \\ &\quad = \{C\{x/y\}\} \cup \mathsf{forms}(x : \mathsf{unit}) \\ &\quad = \{C\} \end{aligned}$$

Observe also that $\mathsf{forms}(E) = \varnothing$ if E contains only names; formulas are derived only from the types of variables, not from the types of channel names.

The next set of rules axiomatizes the sets of public and tainted types, of data that can flow to or from the opponent.

Kinding Rules: $E \vdash T :: \nu$ for $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$

$$\text{(Kind Var)}\quad \frac{E \vdash \diamond \quad (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu} \qquad \text{(Kind Unit)}\quad \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: \nu}$$

$$\text{(Kind Fun)}\quad \frac{E \vdash T :: \overline{\nu} \quad E, x : T \vdash U :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu}$$

$$\text{(Kind Pair)}\quad \frac{E \vdash T :: \nu \quad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu} \qquad \text{(Kind Sum)}\quad \frac{E \vdash T :: \nu \quad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}$$

$$\text{(Kind Rec)}\quad \frac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu\alpha.T) :: \nu} \qquad \text{(Kind Refine Public)}\quad \frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}}$$

$$\text{(Kind Refine Tainted)}\quad \frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$$

The following rules for ok-types are derivable.

$$\text{(Kind Ok Public)}\quad \frac{E \vdash \{C\}}{E \vdash \{C\} :: \mathbf{pub}} \qquad \text{(Kind Ok Tainted)}\quad \frac{E \vdash \{C\} \quad E \vdash C}{E \vdash \{C\} :: \mathbf{tnt}}$$

The following rules of subtyping are standard [Cardelli, 1986, Pierce and Sangiorgi, 1996, Aspinall and Compagnoni, 2001]. The two rules for subtyping refinement types are the same as in Sage [Gronski et al., 2006].

Subtype: $E \vdash T <: U$

$$\text{(Sub Refl)}\quad \frac{E \vdash T \quad \mathit{recvar}(E) \cap \mathit{fnfv}(T) = \varnothing}{E \vdash T <: T} \qquad \text{(Sub Public Tainted)}\quad \frac{E \vdash T :: \mathbf{pub} \quad E \vdash U :: \mathbf{tnt}}{E \vdash T <: U}$$

$$\text{(Sub Fun)}\quad \frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\, U) <: (\Pi x : T'.\, U')}$$

$$\text{(Sub Pair)}\quad \frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\, U) <: (\Sigma x : T'.\, U')}$$

$$\text{(Sub Sum)}\quad \frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')} \qquad \text{(Sub Var)}\quad \frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

$$\text{(Sub Rec)}\quad \frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathit{fnfv}(T') \quad \alpha' \notin \mathit{fnfv}(T)}{E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')}$$

$$\text{(Sub Refine Left)}\quad \frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \qquad \text{(Sub Refine Right)}\quad \frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

The universal type Un is type equivalent to all types that are both public and tainted; we (arbitrarily) define $\mathsf{Un} \triangleq \mathsf{unit}$. We can show that this definition satisfies the intended meaning: that T is public if and only if T is a subtype of Un, and that T is tainted if and only if T is a supertype of Un. (See Lemma 16 (Public Tainted) in Appendix C.)

The following congruence rule for refinement types is derivable from the two primitive rules for refinement types (Sub Refine Left) and (Sub Refine Right). We also list the special case for ok-types.

(Sub Refine)
$$\frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \vdash C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}}$$

(Sub Ok)
$$\frac{E, \_ : \{C\} \vdash C'}{E \vdash \{C\} <: \{C'\}}$$

Proof: To derive (Sub Refine), we are to show that $E \vdash T <: T'$ and $E, x : \{x : T \mid C\} \vdash C'$ imply $E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}$. By Lemma 2 (Derived Judgments) in Appendix C, $E, x : \{x : T \mid C\} \vdash C'$ implies $E \vdash \{x : T \mid C\}$. By (Sub Refine Left), $E \vdash \{x : T \mid C\}$ and $E \vdash T <: T'$ imply $E \vdash \{x : T \mid C\} <: T'$. By (Sub Refine Right), this and $E, x : \{x : T \mid C\} \vdash C'$ imply $E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}$. □

Next, we present the rules for typing values. The rule for constructions $h\,M$ depends on an auxiliary relation $h : (T, U)$ that delimits the possible argument T and result U of each constructor h.

Rules for Values: $E \vdash M : T$

(Val Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Val Unit)
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

(Val Fun)
$$\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun}\, x \to A : (\Pi x : T.\, U)}$$

(Val Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\Sigma x : T.\, U)}$$

(Val Refine)
$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

(Val Inl Inr Fold)
$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\,M : U}$$

$\mathsf{inl}{:}(T, T{+}U) \quad \mathsf{inr}{:}(U, T{+}U) \quad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

We can derive an introduction rule for ok-types.

(Val Ok)
$$\frac{E \vdash C}{E \vdash () : \{C\}}$$

Proof: From $E \vdash C$ we know that $E \vdash \diamond$ and that $E \vdash C\{()/x\}$. By (Val Unit), $E \vdash () : \mathsf{unit}$. By (Val Refine), $E \vdash () : \{x : \mathsf{unit} \mid C\}$, that is, $E \vdash () : \{C\}$. □

Our final set of rules is for typing arbitrary expressions.

Rules for Expressions: $E \vdash A : T$

(Exp Subsum)
$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

(Exp Appl)
$$\frac{E \vdash M : (\Pi x : T.\, U) \quad E \vdash N : T}{E \vdash M\,N : U\{N/x\}}$$

(Exp Split)
$$\frac{\begin{array}{l} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \\ \{x, y\} \cap \mathit{fv}(V) = \varnothing \end{array}}{E \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ A : V}$$

(Exp Match Inl Inr Fold)
$$\frac{\begin{array}{l} E \vdash M : T \quad h : (H, T) \\ E, x : H, \_ : \{h\,x = M\} \vdash A : U \\ E, \_ : \{\forall x. h\,x \neq M\} \vdash B : U \end{array}}{E \vdash \mathbf{match}\ M\ \mathbf{with}\ h\,x \to A\ \mathbf{else}\ B : U}$$

(Exp Eq)
$$\frac{E \vdash M : T \quad E \vdash N : U \quad x \notin \mathit{fv}(M, N)}{\begin{array}{r} E \vdash M = N : \{x : \mathsf{bool} \mid (x = \mathbf{true} \wedge M = N) \vee \\ (x = \mathbf{false} \wedge M \neq N)\} \end{array}}$$

(Exp Assume)
$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

(Exp Assert)
$$\frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathsf{unit}}$$

(Exp Let)
$$\frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin \mathit{fv}(U)}{E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U}$$

(Exp Res)
$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin \mathit{fn}(U)}{E \vdash (\nu a)A : U}$$

(Exp Send)
$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}}$$

(Exp Recv)
$$\frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

(Exp Fork)
$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}$$

In rules for pattern-matching pairs and constructors, we use equations and inequations within refinement types to track information about the matched variables: (Exp Split) records that M is the pair (x, y); (Exp Match Inl Inr Fold) records that M is $h\,x$ when A runs and that M is not of that form when B runs. Rule (Exp Eq) similarly tracks the result of equality tests.

The final rule, (Exp Fork) for $A_1 \Rsh A_2$, relies on an auxiliary function to extract the top-level formulas from A_2 for use while typechecking A_1, and to extract the top-level formulas from A_1 for use while typechecking A_2. The function $\overline{A}$ returns a formula representing the conjunction of each C occurring in a top-level **assume** C in an expression A, with restricted names existentially quantified.

Formula Extraction: $\overline{A}$

$\overline{(\nu a)A} = \exists a.\overline{A}$ $\quad\overline{A_1 \upharpoonleft A_2} = \overline{A_1} \wedge \overline{A_2}$
$\overline{\textbf{let } x = A_1 \textbf{ in } A_2} = \overline{A_1}$ $\quad\overline{\textbf{assume } C} = C$ $\quad\overline{A} = \text{True}$ if A matches no other rule.

5 Implementing Refinement Types for F#

We implement a typechecker, known as F7, that takes as input a series of extended RCF interface files and F# implementation files and, for every implementation file, performs the following tasks: (1) typecheck the implementation against its RCF interface, and any other RCF interfaces it may use; (2) kindcheck its RCF interface, ensuring that every public value declaration has a public type; and then (3) generate a plain F# interface by erasure from its RCF interface. The programming of these tasks almost directly follows from our type theory. In the rest of this section, we only highlight some design choices and implementation decisions.

For simplicity, we do not provide syntactic support for extended types or non-atomic formulas in implementation files. To circumvent this limitation, one can always move extended types and complex formulas to the RCF interface by adding auxiliary declarations.

5.1 Handling F# Language Features

Our typechecker processes F# programs with many more features than the calculus of Section 2. Thus, type definitions also feature mutual recursion, algebraic datatypes, type abbreviations, and record types; value definitions also feature mutual recursion, polymorphism, nested patterns in let- and match-expression, records, exceptions, and mutable references. As described in Section 2, these constructs can be expanded out to simpler types and expressions within RCF. Hence, for example, our typechecker eliminates type abbreviations by inlining, and compiles records to tuples. The remaining constructs constitute straightforward generalizations of our core calculus. For example, polymorphic functions represent a family of functions, one for each instance of a type variable; hence, when checking a specific function application, our typechecker uses the argument type and expected result type to first instantiate the function type and then typecheck it. (Appendix D provides additional details on these codings.)

5.2 Annotating Standard Libraries

Any F# program may use the set of *pervasive* types and functions in the standard library; this library includes operations on built-in types such as strings, Booleans, lists, options, and references, and also provides system functions such as reading and writing files and pretty-printing. Hence, to check a program, we must provide the typechecker with declarations for all the standard library functions and types it uses. When the types for these functions are F# types, we can simply use the F# interfaces provided with the library and trust their implementation. However, if the program relies on extended RCF types for some library functions, we must provide our own RCF interface. For example, the following code declares two functions on lists:

```
assume
  (∀x, u. Mem(x,x::u)) ∧
  (∀x, y, u. Mem(x,u) ⇒ Mem(x,y::u)) ∧
  (∀x, u. Mem(x,u) ⇒ (∃y, v. u = y::v ∧ (x = y ∨ Mem(x,v))))
val mem: x:α → u:α list → r:bool{r=true ⇒ Mem(x,u)}
val find: (α → bool) → (u:α list → r:α { Mem(r,u) })
```

We declare an inductive predicate Mem for list membership and use it to annotate the two library functions for list membership (mem) and list lookup (find). Having defined these extended RCF types, we have a choice: we may either trust that the library implementation satisfies these types, or reimplement these functions and typecheck them. For lists, we reimplement (and re-typecheck) these functions; for other library modules such as String and Printf, we trust the F# implementation.

5.3 Implementing Trusted Libraries

In addition to the standard library, our F# programs rely on libraries for cryptography and networking. We write their concrete implementations on top of .NET Framework classes. For instance, we define keyed hash functions as:

```
open System.Security.Cryptography
type α hkey = HK of bytes
type hmac = bytes
let mkHKey () = HK (mkNonce())
let hmacsha1 (HK k) (P x) =
  (new HMACSHA1 (k)).ComputeHash x
let hmacsha1Verify (HK k) (P x) (h:bytes) =
  let hh = (new HMACSHA1 (k)).ComputeHash x in
  if h = hh then P x else failwith "hmac verify failed"
```

Similarly, the network send and recv are implemented using TCP sockets (and not typechecked in RCF).

We also write symbolic implementations for cryptography and networking, coded using seals and channels, and typechecked against their RCF interfaces. These implementations can also be used to compile and execute programs symbolically, sending messages on local channels (instead of TCP sockets) and computing sealed values (instead of bytes); this is convenient for testing and debugging, as one can inspect the symbolic structure of all messages.

	F# Definitions	F# Declarations	RCF Declarations	Analysis Time	Z3 Obligations
Typed Libraries	440 lines	125 lines	146 lines	12.1s	12
Access Control (Section 2.4)	104 lines	16 lines	34 lines	8.3s	16
MAC Protocol (Section 3.4)	40 lines	9 lines	12 lines	2.5s	3
Logs and Queries (Section 3.5)	37 lines	10 lines	16 lines	2.8s	6
Secrecy (Section 3.7)	51 lines	18 lines	41 lines	2.7s	6
Principals & Compromise (Section 3.6)	48 lines	13 lines	26 lines	3.1s	12
Flexible Signatures (Section 6)	167 lines	25 lines	52 lines	14.6s	28

Table 1. Typechecking Example Programs

5.4 Type Annotations and Partial Type Inference

Type inference for dependently-typed calculi, such as RCF, is undecidable in general. For top-level value definitions, we require that all types be explicitly declared. For subexpressions, our typechecker performs type inference using standard unification-based techniques for plain F# types (polymorphic functions, algebraic datatypes) but it may require annotations for types carrying formulas.

5.5 Generating Proof Obligations for Z3

Following our typing rules, our typechecker must often establish that a condition follows from the current typing environment (such as when typing function applications and kinding value declarations). If the formula trivially holds, the typechecker discharges it; for more involved first-order-logic formulas, it generates a proof obligation in the Simplify format [Detlefs et al., 2005] and invokes the Z3 prover. Since Z3 is incomplete, it sometimes fails to prove a valid formula.

The translation from RCF typing environments to Simplify involves logical re-codings. Thus, constructors are coded as injective, uninterpreted, disjoint functions. Hence, for instance, a type definition for lists

type (α) list = Cons **of** $\alpha * \alpha$ list | Nil

generates logical declarations for a constant Nil and a binary function Cons, and the two assumptions

assume $\forall$x,y. Cons(x,y) $\neq$ Nil.
assume $\forall$x,y,x',y'.
  (x = x' $\wedge$ y = y') $\Leftrightarrow$ Cons(x,y) = Cons(x',y').

Each constructor also defines a predicate symbol that may be used in formulas. Not all formulas can be translated to first-order-logic; for example, equalities between functional values cannot be translated and are rejected.

5.6 Evaluation

We have typechecked all the examples of this paper and a few larger programs. Table 1 summarizes our results; for each example, it gives the number of lines of typed F# code, of generated F# interfaces, and of declarations in RCF interfaces, plus typechecking time, and the number of proof obligations passed to Z3. Since F# programmers are expected to write interfaces anyway, the line difference between RCF and F# declarations roughly indicates the additional annotation burden of our approach.

The first row is for typechecking our symbolic implementations of lists, cryptography, and networking libraries. The second row is an extension of the access control example of Section 2; the next three rows are variants of the MAC protocol of Section 3. The second-last row is an example adapted from earlier work [Fournet et al., 2007a]; it illustrates the recursive verification of any chain of certificates. The final row implements the protocol described next in Section 6.

The examples in this paper are small programs designed to exercise the features of our type system; our results indicate that typechecking is fast and that annotations are not too demanding. Recent experiments [Bhargavan et al., 2009] indicate that our typechecker scales well to large examples; it can verify custom cryptographic protocol code with around 2000 lines of F# in less than 3 minutes. In comparison with an earlier tool FS2PV [Bhargavan et al., 2008b] that compiles F# code to ProVerif, our typechecker succeeds on examples with recursive functions, such as the last row in Table 1, where ProVerif fails to terminate. It also scales better, since we can typecheck one module at a time, rather than construct a large ProVerif model. On the other hand, FS2PV requires no type annotations, and ProVerif can also prove injective correspondences and equivalence-based properties [Blanchet et al., 2008].

6 Application: Flexible Signatures

We illustrate the controlled usage of cryptographic signatures with the same key for different intents, or different protocols. Such reuse is commonplace in practice (at least for long-term keys) but it is also a common source of errors (see Abadi and Needham [1996]), and it complicates protocol verification.

The main risk is to issue *ambiguous signatures*. As an informal design principle, one should ensure that, whenever a signature is issued, (1) its content follows from the current protocol step; and (2) its content cannot be interpreted otherwise, by any other protocol that may rely on the same key. To this end, one may for instance sign nonces, identities, session identifiers, and tags as well as the message payloads to make the signature more specific.

Our example is adapted from protocol code for XML digital signatures, as prescribed in web services security standards [Eastlake et al., 2002, Nadalin et al., 2004]. These signatures consist of an XML "signature information", which represents a list of (hashed) elements covered by the signature, together with a binary "signature value", a signed cryptographic hash of the signature information. Web services normally treat received signed-information lists as sets, and only check that these sets cover selected elements of the message—possibly fewer than those signed, to enable partial erasure as part of intermediate message processing. This flexibility induces protocol weaknesses in some configurations of services. For instance, by providing carefully-crafted inputs, an adversary may cause a naive service to sign more than intended, and then use this signature (in another XML context) to gain access to another service.

For simplicity, we only consider a single key and two interpretations of messages. We first declare types for these interpretations (either requests or responses) and their network format (a list of elements plus their joint signature).

```
type id = int // representing message GUIDs
type events =
  | Request of id ∗ string // id and payload
  | Response of id ∗ id ∗ string // id, request id, and payload
type element =
  | IdHdr of id // Unique message identifier
  | InReplyTo of id // Identifier for some related messsage
  | RequestBody of string // Payload for a request message
  | ResponseBody of string // Payload for a response message
  | Whatever of string // Any other elements
type siginfo = element list
type msg = siginfo ∗ dsig
```

Depending on their constructor, signed elements are interpreted for requests (RequestBody), responses, (InReplyTo, ResponseBody), both (IdHdr), or none (Whatever). We formally capture this intent in the type declaration of the information that is signed:

```
type verified = x:siginfo{
  (∀id, b.(Mem(IdHdr(id),x) ∧ Mem(RequestBody(b),x))
                                    ⇒ Request(id,b) )
∧ (∀id, req, b.(Mem(IdHdr(id),x) ∧ Mem(ResponseBody(b),x)
       ∧ Mem(InReplyTo(req),x)) ⇒ Response(id,req,b) ) }
```

Thus, the logical meaning of a signature is a conjunction of message interpretations, each guarded by a series of conditions on the elements included in the signature information.

We only present code for requests. We use the following declarations for the key pair and for message processing.

```
private val k: (verified,unit) privkey
private val sk: verified sigkey
val vk: verified verifkey
private val mkMessage: verified → msg
private val isMessage: msg → verified

type request = (id:id ∗ b:string){ Request(id,b) }
val isRequest: msg → request
private val mkPlainRequest: request → msg
private val mkRequest: request → siginfo → msg
```

To accept messages as a genuine requests, we just verify its signature and find two relevant elements in the list:

```
let isMessage (msg,dsig) =
  let signed: siginfo → siginfo pickled = pickle in
  unpickle (rsasha1Verify vk (signed msg) dsig)
let isRequest msg =
  let si = isMessage msg in
  let i = find_id si in
  let r = find_request si in
  (i,r)
```

For producing messages, we may define (and type):

```
let mkMessage siginfo =
  let signed: verified → verified pickled = pickle in
  (siginfo, rsasha1 sk (signed siginfo))
let mkPlainRequest (id,payload) =
  let l1: element list = [] in
  let ide: element = IdHdr(id) in
  let reqe : element = RequestBody(payload) in
  let ls:element list = ide::reqe::l1 in
  mkMessage ls

let mkRequest (id,payload) extra : msg =
  check_harmless extra;
  let ide: element = IdHdr(id) in
  let reqe : element = RequestBody(payload) in
  let ls:element list = ide::reqe::extra in
  mkMessage ls
```

While mkPlainRequest uses a fixed list of signed elements, mkRequest takes further elements to sign as an extra parameter. In both cases, typing the list with the refinement type verified ensures (1) Request(id,b), from its input refinement type; and (2) that the list does not otherwise match the two clauses within verified. For mkRequest, this requires some dynamic input validation check_harmless extra where check_harmless is declared as

```
val check_harmless: x: siginfo → r: unit {
  ( ∀s. not(Mem(IdHdr(s),x)))
∧ ( ∀s. not(Mem(InReplyTo(s),x)))
∧ ( ∀s. not(Mem(RequestBody(s),x)))
∧ ( ∀s. not(Mem(ResponseBody(s),x))) }
```

and recursively defined as

```
let rec check_harmless m = match m with
  | IdHdr(_)::_ → failwith "bad"
  | InReplyTo(_)::_ → failwith "bad"
  | RequestBody(_)::_ → failwith "bad"
  | ResponseBody(_)::_ → failwith "bad"
  | _::xs → check_harmless xs
  | [] → ()
```

On the other hand, the omission of this check, or an error in its implementation, would be caught as a type error.

To conclude this example, we provide an alternative declaration for type verified. This type specifies a more restrictive interpretation of signatures: it assumes that the relevant elements appear in a fixed order at the head of the list. (This corresponds roughly to our most precise model in earlier work, which relied on an ad hoc specification of list within ProVerif.)

```
type verifiedprefix = x:siginfo{
  ( ∀id, b, extra.( x = IdHdr(id)::RequestBody(b)::extra ⇒
        Request(id,b) ))
∧ ( ∀id, req, b, extra.( x = IdHdr(id)::InReplyTo(req)::
      ResponseBody(b)::extra
  ⇒ Response(id,req,b) )) }
```

Formally, our typechecker confirms that verified is a subtype of prefixverified. For instance, we may use it instead of verified for typing mkRequest (and even remove the call to check_harmless), but not for typing isRequest.

7 Related Work

RCF is intended for verifying security properties of implementation code, and is related to various prior type systems and static analyses. We describe some of the more closely related approaches. (See also Section 1 for a comparison with prior work of the authors.)

Verification tools for cryptographic protocol implementations CSur was the first tool to analyze the source code of cryptographic protocols [Goubault-Larrecq and Parrennes, 2005]; it can verify protocol code in C annotated with logical assertions, by generating proof obligations for an external first-order-logic theorem-prover.

In prior work [Bhargavan et al., 2008b,a] a subset of F# was translated into different variants of the applied π-calculus which could be verified by Blanchet's theorem provers ProVerif [Blanchet, 2001] and CryptoVerif [Blanchet, 2006] respectively. The use of specialized provers enables the verification of complex cryptographic protocols but is problematic with large implementations.

ASPIER [Chaki and Datta, 2009] has been applied to verify code of the central loop of OpenSSL. It performs no interprocedural analysis and relies on unverified user-supplied abstractions of all functions called from the central loop. ASPIER is based on software model-checking techniques, and proves properties of OpenSSL assuming bounded numbers of active sessions.

Program verification using dependent types Like standard forms of constructive type theory [Martin-Löf, 1984, Constable et al., 1986, Coquand and Huet, 1988, Parent, 1995], our system RCF relies on dependent types (that is, types which contain values), and it can establish logical properties by typechecking. There are, however, three significant differences in style between RCF and constructive type theory. Most notably, RCF does not rely on the Curry-Howard correspondence, which identifies types with logical formulas; instead, RCF has a fixed set of type constructors, and is parameterized by the choice of a logic, which may or may not be constructive. Secondly, types in RCF may contain only values, but not arbitrary expressions, such as function applications. Thirdly, properties of functions are stated by refining their argument and result types with preconditions and postconditions, rather than by developing a behavioural equivalence on functions.

Our treatment of refinement types follows Sage [Gronski et al., 2006], a functional programming language with a rich type system including refinement types. Typechecking generates proof obligations that are sent to an automatic theorem prover; those that cannot be proved automatically are compiled down to run time checks.

Our approach of annotating programs with pre- and postconditions has similarities with extended static checkers used for program verification, such as ESC/Java [Flanagan et al., 2002], Spec# [Barnett et al., 2005], and ESC/Haskell [Xu, 2006]. Such checkers have not been used to verify security properties of cryptographic code, but they can find many other kinds of errors. For instance, Poll and Schubert [2007] use ESC/Java2 [Cok and Kiniry, 2004] to verify that an SSH implementation in Java conforms to a state machine specification. Combining approaches can be even more effective, for instance, Hubbers et al. [2003] generate implementation code from a verified protocol model and check conformance using an extended static checker. In recent work, Régis-Gianas and Pottier [2008] enrich a core functional programming language with higher order logic proof obligations. These are then discharged either by an automatic or an interactive theorem prover depending on the complexity of the proof.

In comparison with these approaches, we propose subtyping rules that capture notions of public and tainted data, and we provide functional encodings of cryptography. Hence, we achieve typability for opponents representing active attackers. Also, we use only stable formulas: in any given run, a formula that holds at some point also holds for the rest of the run; this enables a simple treatment of programs with concurrency and side-effects. More precise stateful properties can still be specified and verified within

RCF using a refined state monad [Borgström et al., 2010].

One direction for further research is to avoid the need for refinement type annotations, by inference. A potential starting point is a recent line of work based on Liquid Types [Rondon et al., 2008, Kawaguchi et al., 2009, Rondon et al., 2010], a polymorphic system of refinement types for ML, together with a type inference algorithm based on predicate abstraction.

Type systems for security Type systems for information flow have been developed for code written in many languages, including Java [Myers, 1999], ML [Pottier and Simonet, 2003], and Haskell [Li and Zdancewic, 2006]. Further works extend them with support for cryptographic mechanisms [for example, Askarov and Sabelfeld, 2005, Askarov et al., 2006, Vaughan and Zdancewic, 2007, Fournet and Rezk, 2008].

These systems seek to guarantee non-interference properties for programs annotated with confidentiality and integrity levels. In contrast, our system seeks to guarantee assertion-based security properties, commonly used in authorization policies and cryptographic protocol specifications, and disregards implicit flows of information.

These systems also feature various privileged primitives for declassifying confidential information and endorsing untrusted information, which play a role similar to our **assume** primitive for injecting formulas.

Type systems with logical effects, such as ours, have also been used to reason about the security of models of distributed systems. For instance, type systems for variants of the π-calculus [Fournet et al., 2007b, Cirillo et al., 2007, Maffeis et al., 2008] and the λ-calculus [Jagadeesan et al., 2008] can guarantee that expressions follow their access control policies. Type systems for variants of the π-calculus, such as Cryptyc [Gordon and Jeffrey, 2002], have been used to verify secrecy, authentication, and authorization properties of protocol models. Unlike our tool, none of these typecheckers operates on source code.

The AURA type system [Vaughan et al., 2008, Jia et al., 2008] also enforces authorization by relying on value-dependent types, but it takes advantage of the Curry-Howard isomorphism for a particular intuitionistic logic [Abadi, 2007]; hence, proofs are manipulated at run time, and may be stored for later auditing; in contrast, we erase all formulas and discard proofs after typechecking.

Fable [Swamy et al., 2008] is a core formalism for expressing security policies; its type system does not in itself guarantee security properties, but additional proofs can build on type safety to establish properties including access control, information flow, and provenance. Fine [Swamy et al., 2010, Chen et al., 2010] is another refinement type system for F#, partly inspired by RCF; it extends the F# source language with dependent, refinement, and affine types that can be used to express and statically verify information flow and stateful authorization policies. Moreover, source programs typechecked with Fine can be compiled to proof-carrying code in a low-level intermediate language. To use the Fine typechecker and compiler, the programmer writes in an extended source language with extensive type annotations. In contrast, our typechecker works with pure F# programs with all annotations provided in an external RCF interface. Moreover, our verification case studies focus on the use of cryptography to enforce security policies, while the the use of cryptographic primitives with Fine remains future work.

Beyond typechecking, many verification techniques also rely on checking logical properties of protocols, using for instance pre- and post-conditions in a protocol logic with domain-specific axioms [Durgin et al., 2003, Datta et al., 2007].

Security verification using RCF Our type system and its typechecker have been used to verify implementations of complex cryptographic protocols and security mechanisms.

- Backes et al. [2009] use RCF as the formal basis of a compiler for zero-knowledge protocols; the compiler takes a verified (well-typed) protocol model and generates a well-typed RCF program, hence preserving the desired security properties. In subsequent work, Backes et al. [2010a] extend RCF with union and intersection types, with application to verifying code for zero-knowledge proofs and other security protocols.

- Baltopoulos and Gordon [2009] use F7 to validate an improved compilation strategy for the Links multi-tier programming language [Cooper et al., 2006], where keyed hashes and encryption protect the integrity and secrecy of web application data held in HTML forms.

- Bhargavan et al. [2009] use F7 as a component of a verifying protocol compiler for multi-party sessions; the compiler generates a protocol implementation along with type annotations and the typechecker verifies that the implementation meets its high-level security specification.

- Guts et al. [2009] also use our typechecker to verify the correct use of security audit logs in distributed applications; well-typed programs are guaranteed to log enough information to later convince a judge that a particular sequence of events occurred.

- Bhargavan et al. [2010b] extend the F7 typechecker with support for implicit predicates representing the pre- and post-conditions of functions, in order to verify applications that use higher-order functions to perform cryptographic operations over recursive data structures, such as lists.

- Bhargavan et al. [2010a] develop a revised set of cryptographic libraries for F7, with embedded logical invariants, and use them to verify a web services security protocol stack and an implementation of the widely-deployed Cardspace protocol for federated identity management. Their main motivation is to extend the scope of cryptographic verification by typing.

  Their libraries rely on different, more logic-oriented design principles, and different types for the opponent. Kinds in RCF provide a simple, built-in mechanism for constraining types in opponent interfaces; they suffice to model many cryptographic primitives, but not all of those found in large case studies. Bhargavan et al. also rely on RCF and F7, but do not rely on kinds.

  Instead, their approach involves developing (and assuming) a logical theory of symbolic cryptographic structures (including, in particular, a *Pub* predicate that represents the attacker knowledge). To verify a given protocol, one can then extend and customize the logical theory, rather than just declaring types.make

  Intuitively, instead of types with nested constructors subject to kinding, they mostly use refinement types of the form $\{x : bytes \mid C\}$, carrying logical specifications C subject to subtyping, Instead of the kinding judgments $E \vdash T :: \textbf{pub}$ and $E \vdash T :: \textbf{tnt}$, they use subtyping judgments $E \vdash T <: \{x : bytes \mid Pub(x)\}$ and $E \vdash \{x : bytes \mid Pub(x)\} <: T$, respectively, where *Pub* is an ordinary predicate of their theory.

  Both sets of libraries are distributed with F7. Theirs yield a more general treatment of secrecy, and a flexible model for additional cryptographic patterns. For example, their libraries include weaker, unauthenticated, encryption algorithms that one can use to build composite patterns such as hybrid encryption, which are not easily encodable with our libraries. Conversely, their libraries are more complex, and their usage sometimes requires hand proofs as well as typechecking.

- This article focuses on verifying security protocols written in RCF against a formal model of cryptography expressed with seals. Subsequent work by Backes et al. [2010b] and by Fournet [2009] develops techniques for verifying RCF code against the computational model of cryptography. Additionally, Backes et al. show a formal correspondence between seals and a formal algebra in the style of Dolev and Yao [1983].

Finally, a tutorial article [Gordon and Fournet, 2010] develops the calculus RCF in several stages (but without kinds), and summarizes the various projects building on it.

8 Conclusion

The use of logical formulas as computational effects is a valuable way to integrate program logics and type systems, with application to security.

Acknowledgments Discussions with Bob Harper and Dan Licata were useful. Aslan Askarov, Michael Greenberg, and Aleks Nanevski commented on drafts of this paper. Kenneth Knowles suggested a proof technique. Cătălin Hriţcu and Matteo Maffei reported an error in a proof and suggested the use of Lemma 14 (Replacing Tainted Bounds) in Appendix C.2. Pierre-Yves Strub reported an error in a proof in Appendix C.3. Nikolaj Bjørner and Leonardo de Moura provided help with Z3. Sergio Maffeis was supported by EPSRC grant EP/E044956/1.

A Logics

Formally, RCF is parameterized by the choice of a logic, in the sense that our typed calculus depends only on a series of abstract properties of the logic, rather than on a particular semantics for logic formulas.

Experimentally, our prototype implementation uses ordinary first order logic with equality, with terms that include all the values M, N of Section 2.1 (including functional values). During typechecking, this logic is partially mapped to the Simplify input of Z3, with the implementation restriction that no term should include any functional value. This restriction prevents discrepancies between run time equality in RCF and term equality in F$^{\#}$.

We first give an abstract definition of the logic used for the theorems, and then give a concrete definition of the logic used in the implementation. Other interesting instances of logics for our verification purposes include authorization logics with "says" modalities [Abadi et al., 1993], which may be used to give a logical account of principals and partial trust by typing [Fournet et al., 2007b]. Accordingly, we refer to our parametric logic as an authorization logic.

A.1 Definition of Authorization Logic

We give a generic, partial definition of logic that captures only the logical properties that are used to establish our typing theorems.

An *authorization logic* is given as a set of *formulas* defined by a grammar that includes the one given below and a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas that meets the properties listed below.

Minimal Syntax of Formulas:

p	predicate symbol

$C ::=$	formula
$p(M_1,\dots,M_n)$	atomic formula
$M = M'$	equation
$C \wedge C'$	conjunction
$C \vee C'$	disjunction
$\neg C$	negation
$\forall x.C$	universal quantification
$\exists x.C$	existential quantification

$$\mathsf{True} \triangleq () = () \quad \mathsf{False} \triangleq \neg\mathsf{True} \quad M \neq M' \triangleq \neg(M = M')$$
$$(C \Rightarrow C') \triangleq (\neg C \vee C') \quad (C \Leftrightarrow C') \triangleq (C \Rightarrow C') \wedge (C' \Rightarrow C)$$

Properties of Deducibility: $S \vdash C$

S,C stands for $S \cup \{C\}$; in (Subst), σ ranges over substitutions of values for variables and permutations of names.

(Axiom) $$\frac{}{C \vdash C}$$ (Mon) $$\frac{S \vdash C}{S,C' \vdash C}$$ (Subst) $$\frac{S \vdash C}{S\sigma \vdash C\sigma}$$ (Cut) $$\frac{S \vdash C \quad S,C \vdash C'}{S \vdash C'}$$

(And Intro) $$\frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$$ (And Elim) $$\frac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$$ (Or Intro) $$\frac{S \vdash C_i}{S \vdash C_0 \vee C_1}\; i = 0,1$$

(Exists Intro) $$\frac{S \vdash C\{M/x\}}{S \vdash \exists x.C}$$ (Exists Elim) $$\frac{S \vdash \exists x.C \quad S,C \vdash C' \quad x \notin fv(S,C')}{S \vdash C'}$$

(Eq) $$\frac{}{\varnothing \vdash M = M}$$ (Ineq) $$\frac{M \neq N \quad fv(M,N) = \varnothing}{\varnothing \vdash M \neq N}$$

(Ineq Cons) $$\frac{h\,N = M \text{ for no } N \quad fv(M) = \varnothing}{\varnothing \vdash \forall x.h\,x \neq M}$$

We have a derived property (True) $\varnothing \vdash \mathsf{True}$.

Although these properties are mostly standard in first-order logic, they are not complete; for instance, we do not set any axiom for negation, so our typing results apply both to intuitionistic and classical logics. Also, we do not provide enough properties to discharge the proof obligations when typing our examples.

We use property (Mon) for the soundness of typing sub-expressions, and use property (Subst) for establishing substitution lemmas. We also implicitly use (Subst) for handling the terms of RCF up to α-conversion on bound names and variables.

We use the properties (And Intro), (And Elim), (Exists Intro), (Exists Elim), and (True) in the proof of Lemma 28 ($\Rrightarrow$ Preserves Logic), to show that the formula $\overline{A}$ extracted from an expression A is preserved by structural equivalence.

We use the properties (Eq), (Ineq), and (Or Intro) in the proof of Lemma 30 ($\rightarrow$ Preserves Logic), for the soundness of the typing rule (Exp Eq). Similarly, we use property (Ineq Cons) for the soundness of (Exp Match Inl Inr Fold).

Since functions **fun** $x \rightarrow A$ are values, they may occur in atomic formulas or equations. Still, these functions are essentially inert in the logic; they can be compared for equality but the logic does not allow reasoning about the application of functions. Said otherwise, the equational theory $M = M'$ is only up to α-conversion, but not for instance β-conversion. Recall that we identify the syntax of values up to the consistent renaming of bound variables, so that, for example, **fun** $x \rightarrow x$ and **fun** $y \rightarrow y$ are the same value. Hence, $\varnothing \vdash$ **fun** $x \rightarrow x =$ **fun** $y \rightarrow y$ is an instance of (Eq).

A.2 An Authorization Logic based on First-Order Logic

For the sake of a self-contained exposition, we review classical first-order logic (predicate calculus) with equality, as supported by the Z3 prover used by our typechecker.

First-Order Logic (Review) The syntax of first-order logic consists of sets of *formulas*, C, and *terms*, t, induced by sets of *predicate symbols*, p, and *function symbols*, f.

Syntax of First-Order Terms and Formulas:

$$
\begin{aligned}
t &::= x \mid f(t_1,\dots,t_n)\\
C &::= p(t_1,\dots,t_n) \mid (t = t') \mid\\
&\qquad \mathsf{False} \mid C \wedge C' \mid C \vee C' \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C
\end{aligned}
$$

$$\neg C \triangleq (C \Rightarrow \mathsf{False}) \qquad t \neq t' \triangleq \neg(t = t')$$

We recall a proof system, FOL, for classical first-order logic with equality in the style of Gentzen's natural-deduction. (More precisely, this is the theory of classical first-order logic with equality as implemented in Isabelle [Paulson, 1991], presented using sequents following, for example, Dummett [1977] and Paulson [1987].

Proof Theory FOL: $S \vdash C$

(FOL Assume)
$$\frac{C \in S}{S \vdash C}$$

(FOL Refl)
$$\frac{}{S \vdash t = t}$$

(FOL Subst)
$$\frac{S \vdash t = t' \quad S \vdash C\{t/x\}}{S \vdash C\{t'/x\}}$$

(FOL And Intro)
$$\frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$$

(FOL And Elim)
$$\frac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$$

(FOL Or Intro)
$$\frac{S \vdash C_i}{S \vdash C_0 \vee C_1} \quad i = 0,1$$

(FOL Or Elim)
$$\frac{S \vdash C_0 \vee C_1 \quad S, C_0 \vdash C' \quad S, C_1 \vdash C'}{S \vdash C'}$$

(FOL False)
$$\frac{S \vdash \mathsf{False}}{S \vdash C}$$

(FOL Classical)
$$\frac{S, \neg C \vdash C}{S \vdash C}$$

(FOL Imply Intro)
$$\frac{S, C \vdash C'}{S \vdash C \Rightarrow C'}$$

(FOL Imply Elim)
$$\frac{S \vdash C \Rightarrow C' \quad S \vdash C}{S \vdash C'}$$

(FOL All Intro)
$$\frac{S \vdash C \quad x \notin fv(S)}{S \vdash \forall x.C}$$

(FOL All Elim)
$$\frac{S \vdash \forall x.C}{S \vdash C\{t/x\}}$$

(FOL Exists Intro)
$$\frac{S \vdash C\{t/x\}}{S \vdash \exists x.C}$$

(FOL Exists Elim)
$$\frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin fv(S, C')}{S \vdash C'}$$

The only rule of FOL that is specific to classical logic is (FOL Classical). The proof theory IFOL [Paulson, 1991] for intuitionistic first-order logic consists of all the rules of FOL apart from (FOL Classical).

An Authorization Logic To construct an authorization logic from FOL, we begin by specifying a particular instance of FOL, and translation from the formulas of authorization logic into this instance.

The syntaxes of formulas in the two logics are essentially the same. The only subtlety in the translation is that the phrases of RCF syntax, including values M and expressions within values, that may occur in authorization logic formulas include binders, while the syntax of first-order terms does not. Our solution is to use the standard first-order *locally-nameless representation* of syntax with binders introduced by de Bruijn [1972]. Each bound name or variable in an RCF phrase is represented as a numeric index, while each free name or variable is represented by itself. We assume that the set of variables of RCF coincides with the variables of FOL, and that each of the (countable) set of names of RCF is included as a nullary function symbol (that is, a constant) in FOL. Moreover, we assume there is a function symbol for each form of RCF phrase, zero and successor symbols to represent indexes, a function symbol to form a bound variable from an index, and one to form a bound name from an index. We refer to these function symbols (including names) as *syntactic*. Hence, any phrase of RCF has a representation as a first-order term; in particular, we write $\underline{M}$ for the term representing the value M. (We omit the standard details of the locally nameless representation; for a discussion see, for example, Gordon [1994] and Aydemir et al. [2008].) Notice that if M is obtained from N by consistent renaming of bound names and variables then $\underline{M}$ and $\underline{N}$ are identical first-order terms.

Hence, we may obtain an FOL formula $\underline{C}$ from an authorization logic formula C via a homomorphic translation with base cases $\underline{p(M_1,\dots,M_n)} = p(\underline{M_1},\dots,\underline{M_n})$ and $\underline{M = N} = \underline{M} = \underline{N}$. We extend the translation to sets of formulas: $\underline{S} = \{\underline{C_1},\dots,\underline{C_n}\}$ when $S = \{C_1,\dots,C_n\}$.

In our intended model, the semantics of a term is an element of a domain defined as the free algebra with constructors corresponding to each of the syntactic function symbols. Hence, the domain is the set of closed phrases of RCF in de Bruijn representation.

We extend the theory FOL with standard axioms valid in the underlying free algebra, that syntactic function symbols yield distinct results, and are injective. (The notation $\vec{x} = \vec{y}$ means $x_1 = y_1 \wedge \dots \wedge x_n = y_n$ where $\vec{x}$ and $\vec{y}$ are the lists $x_1,\dots,x_n$ and $y_1,\dots,y_n$.)

Additional Rules for FOL/F:

(F Disjoint)
$$\frac{f \neq f' \text{ syntactic}}{S \vdash \forall \vec{x}.\forall \vec{y}.f(\vec{x}) \neq f'(\vec{y})}$$

(F Injective)
$$\frac{f \text{ syntactic}}{S \vdash \forall \vec{x}.\forall \vec{y}.f(\vec{x}) = f(\vec{x}) \Rightarrow \vec{x} = \vec{y}}$$

We can use Z3, or some other general SMT solver, to check whether a sequent $S \vdash C$ is derivable in FOL/F by simply declaring an axiom for each instance of (F Disjoint) and (F Injective). (The problem is semi-decidable so the SMT solver may fail to determine whether or not the sequent is derivable.)

Now, we define our authorization logic: we take the set of formulas to be exactly the minimal syntax of Appendix A.1, and we define the deducibility relation $S \vdash C$ to hold if and only if the sequent $\underline{S} \vdash \underline{C}$ is derivable in the theory FOL/F.

Theorem 4 (Logic) *FOL/F is an authorization logic.*

Proof: We derive all the properties required of an authorization logic from the proof system FOL/F.

- We obtain (Mon) by induction on the proof of $S \vdash C$ (possibly using a renaming to meet the side conditions of (FOL All Intro) and (FOL Exists Elim)).
- We obtain (Subst) for name permutations by induction on the proof of $S \vdash C$, as in particular the instances of (F Disjoint) and (F Injective) are preserved by such permutations.
- We obtain (Subst) for value substitutions as follows. Assume $C_1, \dots, C_n \vdash C$. We have
$$\vdash C_1 \Rightarrow \dots \Rightarrow C_n \Rightarrow C$$
by (FOL Imply Intro) for $i \in 1..n$, then (FOL All Intro) for each variable in the domain of σ, then (FOL All Elim) for each variable in the domain of σ, to obtain
$$\vdash C_1\sigma \Rightarrow \dots \Rightarrow C_n\sigma \Rightarrow C\sigma$$
We finally use (Mon) to add $C_i\sigma$ as an hypothesis then (FOL Imply Elim) for $i \in 1..n$, and finally obtain $C_1\sigma, \dots, C_n\sigma \vdash C\sigma$.
- We obtain (Cut) by (FOL Imply Intro) and (FOL Imply Elim).
- We obtain (Ineq) from (F Disjoint) and (F Injective).
- We obtain (Ineq Cons) from (F Disjoint) as follows. Consider some closed value M, such that there is no N with $h\ N = M$. Suppose that f is the outer syntactic function symbol of M considered as a term $M = f(\vec{M})$. Since M is a closed value, it can only be unit, a function, a pair, or a construction $h'\ M'$ where $h \neq h'$; in each case, the symbol f is distinct from h. By (F Disjoint), $\vdash \forall x.\forall \vec{y}.h\ x \neq f(\vec{y})$. By re-ordering the quantifiers, and (FOL All Elim), $\vdash \forall x.h\ x \neq f(\vec{M})$, that is, $\vdash \forall x.h\ x \neq M$.

The other properties follow immediately. $\square$

Since the derivations do not need (FOL Classical), the intuitionistic variation IFOL/F could also serve as an authorization logic.

B Semantics and Safety of Expressions

This appendix formally defines the operational semantics of expressions, and the notion of expression safety, as introduced in Section 2.

An expression can be thought of as denoting a *structure*, given as follows. We define the meaning of **assume** C and **assert** C in terms of a structure being *statically safe*.

Let an *elementary expression*, e, be any expression apart from a let, restriction, fork, message send, or an assumption.

Structures and Static Safety:

$\prod_{i\in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \dots \Rsh A_n$
$\mathscr{L} ::= \{\} \mid (\mathbf{let}\ x = \mathscr{L}\ \mathbf{in}\ B)$

$$\mathbf{S} ::= (\nu a_1)\dots(\nu a_\ell)\ ((\prod_{i\in 1..m} \mathbf{assume}\ C_i) \Rsh (\prod_{j\in 1..n} c_j!M_j) \Rsh (\prod_{k\in 1..o} \mathscr{L}_k\{e_k\}))$$

Let structure $\mathbf{S}$ be *statically safe* if and only if, for all $k \in 1..o$ and C, if $e_k = \mathbf{assert}\ C$ then $\{C_1, \dots, C_m\} \vdash C$.

Structures formalize the idea, explained in Section 2.1, that a state has three components:

(1) a series of elementary expressions e_k being evaluated in parallel contexts;

(2) a series of messages M_j sent on channels but not yet received; and

(3) the *log*, a series of assumed formulas C_i.

Heating: $A \Rrightarrow A'$

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)
$A \Rrightarrow A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \Rrightarrow \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Heat Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)
$() \Rsh A \equiv A$	(Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$\mathbf{assume}\ C \Rrightarrow \mathbf{assume}\ C \Rsh ()$	(Heat Assume ())
$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$	(Heat Res Fork 1)
$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$	(Heat Res Fork 2)
$a \notin fn(B) \Rightarrow \mathbf{let}\ x = (\nu a)A\ \mathbf{in}\ B \Rrightarrow (\nu a)\mathbf{let}\ x = A\ \mathbf{in}\ B$	(Heat Res Let)
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$	(Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$	(Heat Fork Comm)
$\mathbf{let}\ x = (A \Rsh A')\ \mathbf{in}\ B \equiv A \Rsh (\mathbf{let}\ x = A'\ \mathbf{in}\ B)$	(Heat Fork Let)

Lemma 1 (Structure)
For every expression A, there is a structure $\mathbf{S}$ *such that* $A \Rrightarrow \mathbf{S}$.

Proof: The proof is by structural induction on A. □

Reduction: $A \to A'$

$(\mathbf{fun}\, x \to A)\ N \to A\{N/x\}$	(Red Fun)
$(\mathbf{let}\ (x_1,x_2) = (N_1,N_2)\ \mathbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\}$	(Red Split)
$(\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B) \to \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$	(Red Match)
$M = N \to \begin{cases} \mathbf{true} & \text{if } M = N \\ \mathbf{false} & \text{otherwise} \end{cases}$	(Red Eq)
$a!M \upharpoonright a? \to M$	(Red Comm)
$\mathbf{assert}\ C \to ()$	(Red Assert)
$\mathbf{let}\ x = M\ \mathbf{in}\ A \to A\{M/x\}$	(Red Let Val)
$A \to A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \to \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Red Let)
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$	(Red Res)
$A \to A' \Rightarrow (A \upharpoonright B) \to (A' \upharpoonright B)$	(Red Fork 1)
$A \to A' \Rightarrow (B \upharpoonright A) \to (B \upharpoonright A')$	(Red Fork 2)
$A \to A' \quad \text{if } A \Rrightarrow B, B \to B', B' \Rrightarrow A'$	(Red Heat)

Expression Safety:

An expression A is *safe* if and only if, for all A' and $\mathbf{S}$, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is statically safe.

C Properties of the Type System

The structure of this appendix is as follows.

- Appendix C.1 develops basic properties of the type system, such as weakening, strengthening, and exchange lemmas.
- Appendix C.2 contains the proof of Lemma 15 (Public Down/Tainted Up), which characterizes the relationship between the public and tainted kinds and subtyping.
- Appendix C.3 establishes properties of subtyping, principally Lemma 20 (Transitivity), that subtyping is transitive.
- Appendix C.4 presents an alternative characterization of the expression typing relation, avoiding the non-structural rule (Val Refine), by building its effect into each of the structural rules for values; this characterization is useful in various subsequent proofs.
- Appendix C.5 proves various properties of substitution.
- Appendix C.6 establishes Theorem 1 (Safety). The main lemmas in the proof are Proposition 29 ($\Rrightarrow$ Preserves Types) and Proposition 31 ($\to$ Preserves Types).
- Finally, Appendix C.7 establishes Theorem 2 (Robust Safety); the main additional lemma needed is Lemma 34 (Opponent Typability), that any opponent expression can be typed within the system.

C.1 Basic Properties

We begin with some standard properties of our type system. To state them, we let $\mathcal{J}$ range over $\{\diamond, T, C, T :: \nu, T <: T', A : T\}$.

Lemma 2 (Derived Judgments)

(1) *If $E \vdash T$ then $E \vdash \diamond$ and $fnfv(T) \subseteq dom(E)$.*

(2) *If $E \vdash C$ then $E \vdash \diamond$ and $fnfv(C) \subseteq dom(E)$.*

(3) *If $E \vdash T :: \nu$ then $E \vdash T$.*

(4) *If $E \vdash T <: T'$ then $E \vdash T$ and $E \vdash T'$.*

(5) *If $E \vdash A : T$ then $E \vdash T$ and $fnfv(A) \subseteq dom(E)$.*

Proof: The proof is by a simultaneous induction on the depth of derivation of the judgments. □

Lemma 3 (Type Variable Strengthening)
Let entry μ be one of α, $\alpha :: \nu'$, or $\alpha <: \alpha'$, and assume that $dom(\mu) \cap fnfv(E') = \varnothing$.

(1) *If $E, \mu, E' \vdash \diamond$ then $E, E' \vdash \diamond$.*

(2) *If $E, \mu, E' \vdash C$ and $dom(\mu) \cap fnfv(C) = \varnothing$ then $E, E' \vdash C$.*

(3) *If $E, \mu, E' \vdash T :: \nu$ and $dom(\mu) \cap fnfv(T) = \varnothing$ then $E, E' \vdash T :: \nu$.*

(4) *If $E, \mu, E' \vdash T <: T'$ and $dom(\mu) \cap fnfv(T, T') = \varnothing$ then $E, E' \vdash T <: T'$.*

Proof: By an induction on the depth of derivation of $E, \mu, E' \vdash \mathcal{J}$, using property (Cut) of the logic. □

Lemma 4 (Anon Variable Strengthening)
If $E, \_ : \{C\}, E' \vdash \mathcal{J}$ and forms$(E, E') \vdash C$ then $E, E' \vdash \mathcal{J}$.

Proof: Recall that an anonymous variable $\_$ is used nowhere else than its occurrence in the typing environment, so implicitly it cannot occur on the right-hand side of any judgment. The proof is by induction on the depth of derivation of $E, \_ : \{C\}, E' \vdash \mathcal{J}$. □

Lemma 5 (Exchange)
If $E,\mu_1,\mu_2,E' \vdash \mathcal{J}$ and $dom(\mu_1) \cap fnfv(\mu_2) = \varnothing$ then $E,\mu_2,\mu_1,E' \vdash \mathcal{J}$.

Proof: By an induction on the depth of derivation of $E,\mu_1,\mu_2,E' \vdash \mathcal{J}$. □

Lemma 6 (Weakening)
If $E,E' \vdash \mathcal{J}$ and $E,\mu,E' \vdash \diamond$ then $E,\mu,E' \vdash \mathcal{J}$.

Proof: By an induction on the depth of derivation of $E,E' \vdash \mathcal{J}$. The case for (Derive) depends on the monotonicity property (Mon) of the logic. □

The following lemma captures the idea that the formulas in a tainted type cannot be relied upon, because any data produced by the opponent may flow into a tainted type.

Lemma 7 (Kinding)
If $E \vdash T :: \textbf{tnt}$, and $x \notin dom(E)$, then $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)$.

Proof: When $x \notin fv(T)$ we have the following (noting that if T is a refinement type we can put it in the form $\{x : U \mid C\}$ up to alpha-conversion):

$$\mathsf{forms}(x:T) = \begin{cases} \{C\} \cup \mathsf{forms}(x:U) & \text{if } T = \{x:U \mid C\} \\ \varnothing & \text{otherwise} \end{cases}$$

The proof is by induction on the derivation of $E \vdash T :: \textbf{tnt}$. The only case for which $\mathsf{forms}(x : T) \neq \varnothing$ is for (Kind Refine Tainted), when $T = \{x : U \mid C\}$ and we have $E \vdash U :: \textbf{tnt}$ and $E,x : U \vdash C$. We are to show $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)$, that is, $\mathsf{forms}(E) \vdash \mathsf{forms}(x : U) \cup C$. By induction hypothesis, $\mathsf{forms}(E) \vdash \mathsf{forms}(x : U)$. By definition, $E,x : U \vdash C$ implies $\mathsf{forms}(E),\mathsf{forms}(x : U) \vdash C$. By (Cut), we get $\mathsf{forms}(E) \vdash C$. By (And Intro), we get $\mathsf{forms}(E) \vdash \mathsf{forms}(x : U) \cup C$, as required. □

If T is a subtype of T', then the set of formulas $\mathsf{forms}(x : T)$ is logically stronger than the set of formulas $\mathsf{forms}(x : T')$.

Lemma 8 (Logical Subtyping)
If $E \vdash T <: T'$ and $x \notin dom(E)$ then $\mathsf{forms}(E),\mathsf{forms}(x : T) \vdash \mathsf{forms}(x : T')$.

Proof: By induction on the derivation of $E \vdash T <: T'$, using property (Cut) of the logic. In case (Sub Public Tainted), we have $E \vdash T :: \textbf{pub}$ and $E \vdash T' :: \textbf{tnt}$. By Lemma 7 (Kinding), we get that $E,x : \mathsf{Un} \vdash C$ for all $C \in \mathsf{forms}(x : T')$, which implies that $\mathsf{forms}(E) \vdash C$ for all $C \in \mathsf{forms}(x : T')$. By (Mon), we get $\mathsf{forms}(E),\mathsf{forms}(x : T) \vdash \mathsf{forms}(x : T')$. □

Our system enjoys a standard bound weakening property, that an occurrence of T in the environment of a judgment can be replaced by a subtype T'.

Lemma 9 (Bound Weakening)
Suppose that $E \vdash T' <: T$. If $E,x : T,E' \vdash \mathcal{J}$ then $E,x : T',E' \vdash \mathcal{J}$. Moreover the depth of the derivation of the second judgment equals that of the first (except where $\mathcal{J}$ is a typing judgment).

Proof: By induction on the derivations of $E,x : T,E' \vdash \mathcal{J}$, using Lemma 8 (Logical Subtyping) and property (Cut) of the logic. □

Recall that an ok-type $\{C\}$ is a token witnessing that the formula C holds, and is defined to be the refinement type $\{\_ : \mathsf{unit} \mid C\}$. The final lemmas in this section state some simple properties of ok-types.

Lemma 10 (Bound Weakening Ok)
Suppose that $E,C' \vdash C$. If $E,x : \{C\},E' \vdash \mathcal{J}$ then $E,x : \{C'\},E' \vdash \mathcal{J}$.

Proof: Corollary of Lemma 9 (Bound Weakening) and (Sub Ok). □

Lemma 11 (Sub Refine Left Refl)
If $E \vdash \{x : T \mid C\}$ then $E \vdash \{x : T \mid C\} <: T$.

Proof: If $E \vdash \{x : T \mid C\}$ then $E \vdash T$. By (Sub Refl), $E,\_ : \{x : T \mid C\} \vdash T <: T$. By (Sub Refine Left), $E \vdash \{x : T \mid C\} <: T$. □

Lemma 12 (And Sub)
If $E \vdash \{x : T \mid C_1 \wedge C_2\}$ then: $E \vdash \{x : T \mid C_1 \wedge C_2\} <:> \{x : \{x : T \mid C_1\} \mid C_2\}$.

Proof: Suppose $E \vdash \{x : T \mid C_1 \wedge C_2\}$, which is to say $E \vdash \diamond$ and $fnfv(T) \subseteq dom(E)$ and $fnfv(C_1,C_2) \subseteq dom(E) \cup \{x\}$.

By (Sub Refl), $E \vdash T <: T$. By (Derive) and (And Elim), $E,x : \{x : T \mid C_1 \wedge C_2\} \vdash C_1$ because $\mathsf{forms}(x : \{x : T \mid C_1 \wedge C_2\}) = \{C_1 \wedge C_2\} \cup \mathsf{forms}(T)$. Hence, by (Sub Refine), we have:

$$E \vdash \{x : T \mid C_1 \wedge C_2\} <: \{x : T \mid C_1\}$$

By (Derive) and (And Elim), $E,x : \{x : T \mid C_1 \wedge C_2\} \vdash C_2$. Hence, by (Sub Refine Right), we obtain the forwards inclusion:

$$E \vdash \{x : T \mid C_1 \wedge C_2\} <: \{x : \{x : T \mid C_1\} \mid C_2\}$$

By (Sub Refine Left), twice, we have $E \vdash \{x : \{x : T \mid C_1\} \mid C_2\} <: T$. By (Derive) and (And Intro), $E,x : \{x : \{x : T \mid C_1\} \mid C_2\} \vdash C_1 \wedge C_2$ because $\mathsf{forms}(x : \{x : \{x : T \mid C_1\} \mid C_2\}) = \{C_1,C_2\} \cup \mathsf{forms}(x : T)$. Hence, by (Sub Refine Right), we obtain the backwards inclusion:

$$E \vdash \{x : \{x : T \mid C_1\} \mid C_2\} <: \{x : T \mid C_1 \wedge C_2\}$$

□

Lemma 13 (Ok $\wedge$)
We have $E, \_ : \{C_1\}, \_ : \{C_2\}, E' \vdash \mathcal{J}$ if and only if $E, \_ : \{C_1 \wedge C_2\}, E' \vdash \mathcal{J}$.

Proof: By inductions on the derivation of each judgment and properties (And Intro), (And Elim), and (Cut) of the logic. □

C.2 Properties of Kinding

We introduced in Section 3.2 a universal type Un of data known to the opponent. Lemma 16 (Public Tainted) is a standard characterization [Gordon and Jeffrey, 2003b] of the public and tainted kinds: a type T is public if and only if it is a subtype of Un, and a type is tainted if and only if it is a supertype of Un. The next two lemmas are needed in the proof of this main lemma.

The proofs of Lemma 15 (Public Down/Tainted Up) in this section and Lemma 20 (Transitivity) in the next both rely on the following *compartmental notation* $E[E']$ for environments.

Compartmental Notation for Environments: $E[E']$

Let $E[(E'_i)^{i \in 1..n}]$ denote the environment obtained by inserting $E'_1, \ldots, E'_n$ at fixed positions between the entries of E, subject to the constraint that E is executable.

Lemma 14 (Replacing Tainted Bounds)
*If $E, x : T, E' \vdash U :: \nu$ and $E \vdash T ::$ **tnt** and $E \vdash V$ then $E, x : V, E' \vdash U :: \nu$.*

Proof: The proof is by induction on the derivation of $E, x : T, E' \vdash U :: \nu$. In each case we assume that $E \vdash T ::$ **tnt** and $E \vdash V$.

(Kind Var) Here $E, x : T, E' \vdash \alpha :: \nu$ derives from $E, x : T, E' \vdash \diamond$ and $(\alpha :: \nu) \in (E, x : T, E')$. By assumption $E \vdash V$, we have $E, x : V, E' \vdash \diamond$. We also have $(\alpha :: \nu) \in (E, x : V, E')$. By (Kind Var), $E, x : V, E' \vdash \alpha :: \nu$.

(Kind Unit) Here $E, x : T, E' \vdash$ unit $:: \nu$ derives from $E, x : T, E' \vdash \diamond$. By assumption $E \vdash V$, we have $E, x : V, E' \vdash \diamond$. By (Kind Unit), $E, x : V, E' \vdash$ unit $:: \nu$.

(Kind Fun) Here $E, x : T, E' \vdash (\Pi y : T'.\ U') :: \nu$ derives from $E, x : T, E' \vdash T' :: \overline{\nu}$ and $E, x : T, E', y : T' \vdash U' :: \nu$. By induction hypothesis, $E, x : V, E' \vdash T' :: \overline{\nu}$ and $E, x : V, E', y : T' \vdash U' :: \nu$. By (Kind Fun), $E, x : V, E' \vdash (\Pi y : T'.\ U') :: \nu$.

(Kind Pair) Here $E, x : T, E' \vdash (\Sigma y : T'.\ U') :: \nu$ derives from $E, x : T, E' \vdash T' :: \nu$ and $E, x : T, E', y : T' \vdash U' :: \nu$. By induction hypothesis, $E, x : V, E' \vdash T' :: \nu$ and $E, x : V, E', y : T' \vdash U' :: \nu$. By (Kind Pair), $E, x : V, E' \vdash (\Sigma y : T'.\ U') :: \nu$.

(Kind Sum) Here $E, x : T, E' \vdash (T' + U') :: \nu$ derives from $E, x : T, E' \vdash T' :: \nu$ and $E, x : T, E' \vdash U' :: \nu$. By induction hypothesis, $E, x : V, E' \vdash T' :: \nu$ and $E, x : V, E' \vdash U' :: \nu$. By (Kind Sum), $E, x : V, E' \vdash (T' + U') :: \nu$.

(Kind Rec) Here $E, x : T, E' \vdash (\mu\alpha.T') :: \nu$ derives from $E, x : T, E', \alpha :: \nu \vdash T' :: \nu$. By induction hypothesis, $E, x : V, E', \alpha :: \nu \vdash T' :: \nu$. By (Kind Rec), $E, x : V, E' \vdash (\mu\alpha.T') :: \nu$.

(Kind Refine Public) Here $E, x : T, E' \vdash \{y : T' \mid C\} ::$ **pub** derives from $E, x : T, E' \vdash \{y : T' \mid C\}$ and $E, x : T, E' \vdash T' ::$ **pub**. Given $E, x : T, E' \vdash \{y : T' \mid C\}$ we can also show $E, x : V, E' \vdash \{y : T' \mid C\}$, by assumption $E \vdash V$. By induction hypothesis, $E, x : V, E' \vdash T' ::$ **pub**. By (Kind Refine Public),$E, x : V, E' \vdash \{y : T' \mid C\} ::$ **pub**.

(Kind Refine Tainted) Here $E, x : T, E' \vdash \{y : T' \mid C\} ::$ **tnt** derives from $E, x : T, E' \vdash T' ::$ **tnt** and $E, x : T, E', y : T' \vdash C$. By induction hypothesis, $E, x : V, E' \vdash T' ::$ **tnt**. By Lemma 7 (Kinding), $E \vdash T ::$ **tnt** implies that forms$(E) \vdash$ forms$(x : T)$. Hence, $E, x : T, E', y : T' \vdash C$ implies $E, x : V, E', y : T' \vdash C$. By (Kind Refine Tainted), we obtain $E, x : V, E' \vdash \{y : T' \mid C\} ::$ **tnt**. □

Lemma 15 (Public Down/Tainted Up)
Suppose that E is executable.

(1) *If $E \vdash T <: T'$ and $E \vdash T' ::$ **pub** then $E \vdash T ::$ **pub**.*

(2) *If $E \vdash T ::$ **tnt** and $E \vdash T <: T'$ then $E \vdash T' ::$ **tnt**.*

Proof: The lemma is an instance (for $n = 0$) of the following more general statement:

If $E \vdash T <: T'$ where $E = E_0[(\alpha_i <: \alpha'_i)\ ^{i \in 1..n}]$ then for all $\hat{E} = E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)\ ^{i \in 1..n}]$, we have:

(1) If $\hat{E} \vdash T' ::$ **pub** then $\hat{E} \vdash T ::$ **pub**.

(2) If $\hat{E} \vdash T ::$ **tnt** then $\hat{E} \vdash T' ::$ **tnt**.

In the following proof, in the case for (Sub Fun), we appeal to the following adaptation of Lemma 9 (Bound Weakening):

If $E \vdash T' <: T$ and $(\hat{E}, x : T, E') \vdash U :: \nu$ then $(\hat{E}, x : T', E') \vdash U :: \nu$.

The adaptation is proved by induction on the derivation of $(\hat{E}, x : T, E') \vdash U :: \nu$, with appeal to Lemma 8 (Logical Subtyping) in the case for (Kind Refine Tainted).

The proof is by induction on the derivation of $E \vdash T <: T'$ where $E = E_0[(\alpha_i <: \alpha'_i)\ ^{i \in 1..n}]$. Consider any $\hat{E} = E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)\ ^{i \in 1..n}]$.

(Sub Refl) Here $T = T'$ and parts (1) and (2) follow at once.

(Sub Var) We have $E \vdash \alpha <: \alpha'$ derived from $E \vdash \diamond$ and $(\alpha <: \alpha') \in E$.

We must have $\alpha = \alpha_j$ and $\alpha' = \alpha'_j$ for some $j \in 1..n$.

For part (1), assume that $\hat{E} \vdash \alpha'_j :: \mathbf{pub}$. This can only be derived by (Kind Var), and so $\nu_j = \mathbf{pub}$ (since the α_i, α'_i are distinct). Hence, we get $\hat{E} \vdash \alpha_j :: \mathbf{pub}$ by (Kind Var).

For part (2), assume that $\hat{E} \vdash \alpha_j :: \mathbf{tnt}$. This can only be derived by (Kind Var), and so $\nu_j = \mathbf{tnt}$ (since the α_i, α'_i are distinct). Hence, we get $\hat{E} \vdash \alpha_j :: \mathbf{pub}$ by (Kind Var).

(Sub Public Tainted) We have $E \vdash T <: U$ derived from $E \vdash T :: \mathbf{pub}$ and $E \vdash U :: \mathbf{tnt}$.

For part (1), we obtain $\hat{E} \vdash T :: \mathbf{pub}$ because the derivation of $E \vdash T :: \mathbf{pub}$ makes no use of the entries $(\alpha_i <: \alpha'_i)^{\ i \in 1..n}$.

For part (2), we have $\hat{E} \vdash U :: \mathbf{tnt}$ because the derivation of $E \vdash U :: \mathbf{tnt}$ makes no use of the entries $(\alpha_i <: \alpha'_i)^{\ i \in 1..n}$.

(Sub Fun) We have $E \vdash (\Pi x : T.\ U) <: (\Pi x : T'.\ U')$ derived from $E \vdash T' <: T$ and $(E, x : T') \vdash U <: U'$.

For part (1), assume that $\hat{E} \vdash (\Pi x : T'.\ U') :: \mathbf{pub}$.

This can only be derived by (Kind Fun), from $\hat{E} \vdash T' :: \mathbf{tnt}$ and $(\hat{E}, x : T') \vdash U' :: \mathbf{pub}$.

By induction hypothesis (2), $\hat{E} \vdash T' :: \mathbf{tnt}$ and $E \vdash T' <: T$ imply $\hat{E} \vdash T :: \mathbf{tnt}$.

By induction hypothesis (1), $(E, x : T') \vdash U <: U'$ and $(\hat{E}, x : T') \vdash U' :: \mathbf{pub}$ imply $(\hat{E}, x : T') \vdash U :: \mathbf{pub}$.

By Lemma 2 (Derived Judgments), $\hat{E} \vdash T :: \mathbf{tnt}$ implies $\hat{E} \vdash T$.

By Lemma 14 (Replacing Tainted Bounds), $(\hat{E}, x : T') \vdash U' :: \mathbf{pub}$ and $\hat{E} \vdash T' :: \mathbf{tnt}$ and $\hat{E} \vdash T$ imply $(\hat{E}, x : T) \vdash U :: \mathbf{pub}$.

By (Kind Fun), $\hat{E} \vdash (\Pi x : T.\ U) :: \mathbf{pub}$.

For part (2), assume that $\hat{E} \vdash (\Pi x : T.\ U) :: \mathbf{tnt}$.

This can only be derived by (Kind Fun), from $\hat{E} \vdash T :: \mathbf{pub}$ and $(\hat{E}, x : T) \vdash U :: \mathbf{tnt}$.

By induction hypothesis (1), $E \vdash T' <: T$ and $\hat{E} \vdash T :: \mathbf{pub}$ imply $\hat{E} \vdash T' :: \mathbf{pub}$.

By the adaptation of Lemma 9 (Bound Weakening) introduced at the start of this proof, $E \vdash T' <: T$ and $(\hat{E}, x : T) \vdash U :: \mathbf{tnt}$ imply $(\hat{E}, x : T') \vdash U :: \mathbf{tnt}$.

By induction hypothesis (2), $(\hat{E}, x : T') \vdash U :: \mathbf{tnt}$ and $(E, x : T') \vdash U <: U'$ imply $(\hat{E}, x : T') \vdash U' :: \mathbf{tnt}$.

By (Kind Fun), from $\hat{E} \vdash T' :: \mathbf{pub}$ and $(\hat{E}, x : T') \vdash U' :: \mathbf{tnt}$ we obtain $\hat{E} \vdash (\Pi x : T'.\ U') :: \mathbf{tnt}$, as desired.

(Sub Pair), (Sub Sum) These cases follow similarly to the case for (Sub Fun).

(Sub Rec) We have $E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')$ derived from $E_0[(\alpha_i <: \alpha'_i)^{\ i \in 1..n}, \alpha <: \alpha'] \vdash T <: T'$ and $\alpha \notin \mathit{fnfv}(T')$ and $\alpha' \notin \mathit{fnfv}(T)$ and $\{\alpha, \alpha'\} \cap \mathit{fnfv}(E_0) = \varnothing$

For part (1), assume that $\hat{E} \vdash (\mu\alpha'.T') :: \mathbf{pub}$. This can only be derived by (Kind Rec), from $E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)^{\ i \in 1..n}, \alpha' :: \mathbf{pub}] \vdash T' :: \mathbf{pub}$. By Lemma 6 (Weakening), $E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)^{\ i \in 1..n}, \alpha :: \mathbf{pub}, \alpha' :: \mathbf{pub}] \vdash T' :: \mathbf{pub}$. By induction hypothesis, $E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)^{\ i \in 1..n}, \alpha :: \mathbf{pub}, \alpha' :: \mathbf{pub}] \vdash T :: \mathbf{pub}$. By Lemma 3 (Type Variable Strengthening), $\alpha' \notin \mathit{fnfv}(T)$ and $\alpha' \notin \mathit{fnfv}(E_0)$ implies $E_0[(\alpha_i :: \nu_i, \alpha'_i :: \nu_i)^{\ i \in 1..n}, \alpha :: \mathbf{pub}] \vdash T :: \mathbf{pub}$. By (Kind Rec), $\hat{E} \vdash (\mu\alpha.T) :: \mathbf{pub}$.

Part (2) follows by a symmetric argument.

(Sub Refine Left) We have $E \vdash \{x : T \mid C\} <: T'$ derived from $E \vdash \{x : T \mid C\}$ and $E \vdash T <: T'$.

For part (1), assume that $\hat{E} \vdash T' :: \mathbf{pub}$. By induction hypothesis, $\hat{E} \vdash T :: \mathbf{pub}$. We have $\hat{E} \vdash \{x : T \mid C\}$. By (Kind Refine Public), $\hat{E} \vdash \{x : T \mid C\} :: \mathbf{pub}$.

For part (2), assume that $\hat{E} \vdash \{x : T \mid C\} :: \mathbf{tnt}$. This can only be derived by (Kind Refine Tainted), from $\hat{E} \vdash T :: \mathbf{tnt}$ and $(\hat{E}, x : T) \vdash C$. By induction hypothesis, $\hat{E} \vdash T' :: \mathbf{tnt}$.

(Sub Refine Right) We have $E \vdash T <: \{x : T' \mid C\}$ derived from $E \vdash T <: T'$ and $(E, x : T) \vdash C$.

For part (1), assume that $\hat{E} \vdash \{x : T' \mid C\} :: \mathbf{pub}$. This can only be derived by (Kind Refine Public), from $\hat{E} \vdash \{x : T' \mid C\}$ and $\hat{E} \vdash T' :: \mathbf{pub}$. By induction hypothesis, $\hat{E} \vdash T :: \mathbf{pub}$.

For part (2), assume that $\hat{E} \vdash T :: \mathbf{tnt}$. By induction hypothesis, $\hat{E} \vdash T' :: \mathbf{tnt}$. We have $(E, x : T) \vdash C$. By (Kind Refine Tainted), $\hat{E} \vdash \{x : T' \mid C\} :: \mathbf{tnt}$. □

Lemma 16 (Public Tainted)
For all T and executable E:

(1) $E \vdash T :: \mathbf{pub}$ *if and only if* $E \vdash T <: \mathsf{Un}$.

(2) $E \vdash T :: \mathbf{tnt}$ *if and only if* $E \vdash \mathsf{Un} <: T$.

Proof: By definition, $\mathsf{Un} \triangleq \mathsf{unit}$. By (Kind Unit), $E \vdash \mathsf{Un} :: \mathbf{pub}$ and $E \vdash \mathsf{Un} :: \mathbf{tnt}$. The forward implications follow by (Sub Public Tainted), the reverse implications by Lemma 15 (Public Down/Tainted Up). □

C.3 Properties of Subtyping

The main result in this section is transitivity of subtyping, perhaps the most difficult proof in the development, because it needs a relatively complex inductive argument.

The proof of transitivity depends on the following lemma, the first two of which concern the use of recursive type variables declared by entries $\alpha <: \alpha'$ in the typing environment.

Lemma 17 (Rec Kinding)
If $E \vdash T :: \nu$ and $(\alpha <: \alpha') \in E$ then $\alpha \notin fnfv(T)$ and $\alpha' \notin fnfv(T)$.

Proof: By induction on the derivation of $E \vdash T :: \nu$. □

Lemma 18 (Rec Subtyping)
If $E \vdash T <: T'$ and $(\alpha <: \alpha') \in E$ we have that: $\{\alpha, \alpha'\} \cap fnfv(T) = \varnothing$ if and only if $\{\alpha, \alpha'\} \cap fnfv(T') = \varnothing$.

Proof: By induction on the derivation of $E \vdash T <: T'$. □

The following lemma formalizes the intuition that the formulas decorating the type in the environment are all that matter as far as the kinding and subtyping judgments are concerned. In particular, we can replace an environment entry $x : T$ with $x : (T)^\sharp$, where $(T)^\sharp$ is the refinement of the unit type given as follows.

Formulizing a Type:

$$(T)^\sharp \triangleq \{x : \mathsf{unit} \mid \mathsf{forms}(x : T)\}$$

Lemma 19 (Formulize Type) *Assume $E, x : T, E' \vdash \diamond$.*

(1) $E, x : (T)^\sharp, E' \vdash \diamond$.

(2) $E, x : T, E' \vdash C$ *iff* $E, x : (T)^\sharp, E' \vdash C$.

(3) $E, x : T, E' \vdash U :: \nu$ *iff* $E, x : (T)^\sharp, E' \vdash U :: \nu$.

(4) $E, x : T, E' \vdash U <: U'$ *iff* $E, x : (T)^\sharp, E' \vdash U <: U'$.

Moreover, the depth of the derivations of each pair of judgments is the same.

Proof: Each direction follows by induction on the derivation of the assumed judgment. □

Lemma 20 (Transitivity)
If E is executable and $E \vdash T <: T'$ and $E \vdash T' <: T''$ then $E \vdash T <: T''$.

Proof: The lemma is an instance of the following more general statement, which we prove by a simultaneous induction on the sum of the depth of derivations of the antecedent judgments:

(1) $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: T''$ imply $E_{02} \vdash T <: T''$

(2) $E_{12} \vdash T'' <: T'$ and $E_{01} \vdash T' <: T$ imply $E_{02} \vdash T'' <: T$

where E_{01}, E_{12}, and E_{02} take the form

$$\begin{aligned} E_{01} &= E[(\alpha_i \, R_i \, \alpha_i')^{\; i \in 1..n}] \\ E_{12} &= E[(\alpha_i' \, R_i \, \alpha_i'')^{\; i \in 1..n}] \\ E_{02} &= E[(\alpha_i \, R_i \, \alpha_i'')^{\; i \in 1..n}] \end{aligned}$$

for some number n, distinct type variables α_i, α_i', α_i'', relations $R_i \in \{<:, <:^{-1}\}$ for $i \in 1..n$, and executable environment E with $E \vdash \diamond$. (We write $R \in \{<:, <:^{-1}\}$ to mean that relation R is either the subtype relation (in which case $\alpha \, R \, \alpha'$ stands for $\alpha <: \alpha'$) or its inverse (in which case $\alpha \, R \, \alpha'$ stands for $\alpha' <: \alpha$).)

Since E is executable, none of the type variables α_i, α_i', α_i'' occurs in types in E.

We prove part (1) in detail. We first assume (*) that the last rule in the derivation of $E_{12} \vdash T' <: T''$ is neither (Sub Refl), nor (Sub Public Tainted), nor (Sub Refine Right); we prove that $E_{02} \vdash T <: T''$ by a case analysis of the last rule in the derivation of $E_{01} \vdash T <: T'$.

(Sub Refl) In this case $T = T'$ and $E_{01} \vdash T <: T$ follows from $E_{01} \vdash T$ with $fnfv(T) \cap recvar(E_{01}) = \varnothing$ and we have $E_{12} \vdash T <: T''$. We have $fnfv(T) \subseteq dom(E_{01}) \cap dom(E_{12}) = dom(E) \cup \{\alpha_i'^{\; i \in 1..n}\}$ and so we get that $fnfv(T) \subseteq dom(E)$.

We have $E_{12} \vdash T <: T''$ and none of the type variables α_i', α_i'' occurs in T; so, by Lemma 18 (Rec Subtyping), none of these variables occurs in T''. Hence, $fnfv(T'') \subseteq dom(E)$. We may therefore obtain $E_{02} \vdash T <: T''$ from $E_{12} \vdash T <: T''$ by removing the subtype declarations of E_{12} with Lemma 3 (Type Variable Strengthening) and introducing the subtype declarations of E_{02} with Lemma 6 (Weakening).

(Sub Public Tainted) In this case, $E_{01} \vdash T <: T'$ follows from $E_{01} \vdash T :: \textbf{pub}$ and $E_{01} \vdash T' :: \textbf{tnt}$.

By Lemma 17 (Rec Kinding), none of the type variables α_i, α_i', α_i'' occurs in T or T'.

We may therefore obtain $E \vdash T :: \textbf{pub}$ from $E_{01} \vdash T :: \textbf{pub}$ by removing the subtype declarations of E_{01} with Lemma 3 (Type Variable Strengthening).

Similarly, we may obtain $E \vdash T' :: \textbf{tnt}$ from $E_{01} \vdash T' :: \textbf{tnt}$ by removing the subtype declarations of E_{01} with Lemma 3 (Type Variable Strengthening).

We have $E_{12} \vdash T' <: T''$ and none of the type variables α_i', α_i'' occurs in T'; so, by Lemma 18 (Rec Subtyping), none of the type variables α_i, α_i', α_i'' occurs in T'' either.

We may therefore obtain $E \vdash T' <: T''$ from $E_{12} \vdash T' <: T''$ by removing the subtype declarations of E_{12} with Lemma 3 (Type Variable Strengthening).

By Lemma 15 (Public Down/Tainted Up), given that E is executable, $E \vdash T' :: \textbf{tnt}$ and $E \vdash T' <: T''$ imply $E \vdash T'' :: \textbf{tnt}$.

By (Sub Public Tainted), this and $E \vdash T :: \textbf{pub}$ imply $E \vdash T <: T''$.

We obtain $E_{02} \vdash T <: T''$ from $E \vdash T <: T''$. by introducing the subtype declarations of E_{02} with Lemma 6 (Weakening).

(Sub Fun) If $E_{01} \vdash T <: T'$ follows by (Sub Fun), then $T = \Pi x : T_1.\, T_2$ and $T' = \Pi x : T_1'.\, T_2'$ with $E_{01} \vdash T_1' <: T_1$ and $E_{01}, x : T_1' \vdash T_2 <: T_2'$, and $E_{12} \vdash (\Pi x : T_1'.\, T_2') <: T''$.

By assumption (*), the latter must be obtained via (Sub Fun), so that $T'' = \Pi x : T_1''.\, T_2''$ with $E_{12} \vdash T_1'' <: T_1'$ and $E_{12}, x : T_1'' \vdash T_2' <: T_2''$.

By Lemma 8 (Logical Subtyping), $E_{12} \vdash T_1'' <: T_1'$ implies $\mathsf{forms}(E_{12}), \mathsf{forms}(x : T_1'') \vdash \mathsf{forms}(x : T_1')$, which is to say $\mathsf{forms}(E_{01}), \mathsf{forms}(x : T_1'') \vdash \mathsf{forms}(x : T_1')$, since, by construction, $\mathsf{forms}(E_{12}) = \mathsf{forms}(E_{01})$.

By definition, $(T_1'')^\sharp = \{x : \mathsf{unit} \mid \mathsf{forms}(x : T_1'')\}$ and $(T_1')^\sharp = \{x : \mathsf{unit} \mid \mathsf{forms}(x : T_1')\}$.

By (Sub Refine), $E_{01} \vdash (T_1'')^\sharp <: (T_1')^\sharp$.

By Lemma 19 (Formulize Type), $E_{01}, x : T_1' \vdash T_2 <: T_2'$ implies $E_{01}, x : (T_1')^\sharp \vdash T_2 <: T_2'$.

By Lemma 9 (Bound Weakening), we obtain $E_{01}, x : (T_1'')^\sharp \vdash T_2 <: T_2'$.

By Lemma 19 (Formulize Type), $E_{12}, x : T_1'' \vdash T_2' <: T_2''$ implies $E_{12}, x : (T_1'')^\sharp \vdash T_2' <: T_2''$.

By induction hypothesis (1), from $E_{01}, x : (T_1'')^\sharp \vdash T_2 <: T_2'$ and $E_{12}, x : (T_1'')^\sharp \vdash T_2' <: T_2''$ we obtain $E_{02}, x : (T_1'')^\sharp \vdash T_2 <: T_2''$.

(We can apply the induction hypothesis because our applications of Lemma 19 (Formulize Type) and Lemma 9 (Bound Weakening) preserve the depths of derivation.)

By Lemma 19 (Formulize Type), we obtain $E_{02}, x : T_1'' \vdash T_2 <: T_2''$.

By induction hypothesis (2), from $E_{12} \vdash T_1'' <: T_1'$ and $E_{01} \vdash T_1' <: T_1$ we obtain $E_{02} \vdash T_1'' <: T_1$.

By (Sub Fun), we obtain $E_{02} \vdash (\Pi x : T_1.\, T_2) <: (\Pi x : T_1''.\, T_2'')$.

(Sub Pair), (Sub Sum) These cases follow similarly to the case for (Sub Fun).

(Sub Var) If $E_{01} \vdash T <: T'$ follows by (Sub Var), then T' must be a type variable, and so, given assumption (*), the judgment $E_{12} \vdash T' <: T''$ can only follow from (Sub Var). It must be, then, that $T = \alpha_j$ and $T' = \alpha_j'$ and $T'' = \alpha_j''$ and $R_j = <:$ for some $j \in 1..n$. We have $(\alpha_j <: \alpha_j'') \in E_{02}$, and so, by (Sub Var), we obtain: $E_{02} \vdash T <: T''$, as required.

(Sub Rec) If $E_{01} \vdash T <: T'$ follows by (Sub Rec), it must be that $T = \mu\alpha.U$ and $T' = \mu\alpha'.U'$, and we have $E[(\alpha_i\, R_i\, \alpha_i')^{\,i \in 1..n}, \alpha <: \alpha'] \vdash U <: U'$ and $E_{12} \vdash (\mu\alpha'.U') <: T''$. By assumption (*), the latter can only follow from (Sub Rec), and so $T'' = \mu\alpha''.U''$, with $E[(\alpha_i'\, R_i\, \alpha_i'')^{\,i \in 1..n}, \alpha' <: \alpha''] \vdash U' <: U''$. By induction hypothesis (1), we obtain $E[(\alpha_i\, R_i\, \alpha_i'')^{\,i \in 1..n}, \alpha <: \alpha''] \vdash U <: U''$. By (Sub Rec), we obtain $E_{02} \vdash U <: U''$.

(Sub Refine Left) If $E_{01} \vdash T <: T'$ follows by (Sub Refine Left), then $T = \{x : U \mid C\}$ and we have $E_{01} \vdash \{x : U \mid C\}$ and $E_{01} \vdash U <: T'$. By induction hypothesis (1), this and $E_{12} \vdash T' <: T''$ imply $E_{02} \vdash U <: T''$. By (Sub Refine Left), we obtain $E_{02} \vdash T <: T''$.

(We can apply the induction hypothesis because the sum of the depth of derivations of $E_{01} \vdash U <: T'$ and $E_{12} \vdash T' <: T''$ is less than the sum of the depth of derivations of $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: T''$.)

(Sub Refine Right) If $E_{01} \vdash T <: T'$ follows by (Sub Refine Right), then $T' = \{x : W \mid C\}$ and we have $E_{01} \vdash T <: W$ and $E_{01}, x : T \vdash C$.

By assumption (*), and since T' is a refinement type, the derivation of $E_{12} \vdash T' <: T''$ must use rule (Sub Refine Left). It must be that $E_{12} \vdash W <: T''$. By induction hypothesis (1), $E_{02} \vdash T <: T''$. (We can apply the induction hypothesis because the sum of the depth of derivations of $E_{01} \vdash T <: W$ and $E_{12} \vdash W <: T''$ is less than the sum of the depth of derivations of $E_{01} \vdash T <: \{x : W \mid C\}$ and $E_{12} \vdash \{x : W \mid C\} <: T''$.)

On the other hand, our assumption (*) may not hold, that is, we may have $E_{01} \vdash T <: T'$ and a derivation of $E_{12} \vdash T' <: T''$ using (Sub Refl), (Sub Public Tainted), or (Sub Refine Right). We consider these three possibilities below; the arguments are similar to the cases above for when $E_{01} \vdash T <: T'$ follows by (Sub Refl), (Sub Public Tainted), or (Sub Refine Right).

(Sub Refl) In this case $T' = T''$ and $E_{12} \vdash T <: T$ follows from $E_{12} \vdash T''$ with $\mathit{fnfv}(T'') \cap \mathit{recvar}(E_{12}) = \varnothing$ and we have $E_{01} \vdash T <: T''$. We have $\mathit{fnfv}(T'') \subseteq \mathit{dom}(E_{01}) \cap \mathit{dom}(E_{12}) = \mathit{dom}(E) \cup \{\alpha_i'^{\;i \in 1..n}\}$ and so we get that $\mathit{fnfv}(T'') \subseteq \mathit{dom}(E)$. We have $E_{01} \vdash T <: T''$ and none of the type variables α_i', α_i'' occurs in T''; by Lemma 18 (Rec Subtyping), none of these variables occurs in T, so that $\mathit{fnfv}(T) \subseteq \mathit{dom}(E)$. We may therefore obtain $E_{02} \vdash T <: T''$ from $E_{01} \vdash T <: T''$ by removing the subtype declarations of E_{01} with Lemma 3 (Type Variable Strengthening) and introducing the subtype declarations of E_{12} with Lemma 6 (Weakening).

(Sub Public Tainted) In this case, $E_{12} \vdash T' :: \mathbf{pub}$ and $E_{12} \vdash T'' :: \mathbf{tnt}$.

By Lemma 17 (Rec Kinding), none of the type variables α_i, α_i', α_i'' occurs in T' or T''.

We have $E_{01} \vdash T <: T'$ and none of the type variables α_i, α_i' occurs in T'; so, by Lemma 18 (Rec Subtyping), none of the type variables α_i, α_i', α_i'' occurs in T either.

We may therefore obtain $E \vdash T <: T'$ from $E_{01} \vdash T <: T'$ by removing the subtype declarations of E_{01} with Lemma 3 (Type Variable Strengthening).

We may also obtain $E \vdash T' :: \mathbf{pub}$ from $E_{12} \vdash T' :: \mathbf{pub}$ by removing the subtype declarations of E_{12} with Lemma 3 (Type Variable Strengthening).

We may also obtain $E \vdash T'' :: \mathbf{tnt}$ from $E_{12} \vdash T'' :: \mathbf{tnt}$ by removing the subtype declarations of E_{12} with Lemma 3 (Type Variable Strengthening).

By Lemma 15 (Public Down/Tainted Up), given that E is executable, $E \vdash T <: T'$ and $E \vdash T' :: \mathbf{pub}$ imply $E \vdash T :: \mathbf{pub}$.

By (Sub Public Tainted), $E \vdash T :: \mathbf{pub}$ and $E \vdash T'' :: \mathbf{tnt}$ imply $E \vdash T <: T''$.

We obtain $E_{02} \vdash T <: T''$ from $E \vdash T <: T''$ by introducing the subtype declarations of E_{02} with Lemma 6 (Weakening).

(Sub Refine Right) It must be that $T'' = \{x : U \mid C'\}$ and $E_{12} \vdash T' <: U$ and $E_{12}, x : T' \vdash C'$.

By induction hypothesis (1), $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: U$ imply $E_{02} \vdash T <: U$.

(We can apply the induction hypothesis because the sum of the depth of derivations of $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: U$ is less than the sum of the depth of derivations of $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: \{x : U \mid C'\}$.)

Since $\mathsf{forms}(E_{12}, x : T') = \mathsf{forms}(E_{02}, x : T')$, we have $E_{02}, x : T' \vdash C'$.

By Lemma 8 (Logical Subtyping), $E_{01} \vdash T <: T'$ implies that $\mathsf{forms}(E), \mathsf{forms}(x : T) \vdash \mathsf{forms}(x : T')$.

We obtain $E_{02}, x : T \vdash C'$ because $\mathsf{forms}(E_{02}, x : T) = \mathsf{forms}(E, x : T)$ and $\mathsf{forms}(E), \mathsf{forms}(x : T) \vdash \mathsf{forms}(x : T')$ and $\mathsf{forms}(E, x : T') \vdash C'$.

By (Sub Refine Right), $E_{02} \vdash T <: \{x : U \mid C'\}$.

The proof of part (2) is symmetric to the proof of part (1); we detail only those parts of the case analysis that examine the relations R_i within the environments E_{12} and E_{01}. Assume first that the last rule in the derivation of $E_{01} \vdash T' <: T$ is neither (Sub Refl), nor (Sub Public Tainted), nor (Sub Refine Left); we prove that $E_{02} \vdash T'' <: T$ by a case analysis of the last rule in the derivation of $E_{12} \vdash T'' <: T'$.

(Sub Var) If $E_{12} \vdash T'' <: T'$ follows by (Sub Var), then T' must be a type variable, and so the judgment $E_{01} \vdash T' <: T$ can only follow from (Sub Var). It must be, then, that $T'' = \alpha_j''$ and $T' = \alpha_j'$ and $T = \alpha_j$ and $R_j = <:^{-1}$ for some $j \in 1..n$. We have $(\alpha_j'' <: \alpha_j) \in E_{02}$, and so, by (Sub Var), we obtain: $E_{02} \vdash T'' <: T$, as required.

(Sub Rec) If $E_{12} \vdash T'' <: T'$ follows by (Sub Rec), it must be that $T'' = \mu\alpha''.U''$ and $T' = \mu\alpha'.U'$, and we have $E[(\alpha_i' \; R_i \; \alpha_i'')^{\, i \in 1..n}, \alpha'' <: \alpha'] \vdash U'' <: U'$ and $E_{01} \vdash (\mu\alpha'.U') <: T$. The latter can only follow from (Sub Rec), and so $T = \mu\alpha.U$, with $E[(\alpha_i \; R_i \; \alpha_i')^{\, i \in 1..n}, \alpha' <: \alpha] \vdash U' <: U$. By induction hypothesis (2), we obtain $E[(\alpha_i \; R_i \; \alpha_i'')^{\, i \in 1..n}, \alpha'' <: \alpha] \vdash U'' <: U$. By (Sub Rec), we obtain $E_{02} \vdash U'' <: U$.

The rest of part (2) is exactly symmetric to part (1). □

C.4 Alternative Formulation of Typing

We present an alternative definition of expression typing, which avoids the non-structural rule (Val Refine), and hence is useful in the proofs of Lemma 23 (Substitution), Proposition 29 ($\Rrightarrow$ Preserves Types) and Proposition 31 ($\rightarrow$ Preserves Types).

Alternative Rules for Typing Values: $E \vdash A : T$

(Val Var Refine)
$$\frac{E \vdash C\{x/y\} \quad (x : T) \in E}{E \vdash x : \{y : T \mid C\}}$$

(Val Unit Refine)
$$\frac{E \vdash C\{()/y\}}{E \vdash () : \{y : \mathsf{unit} \mid C\}}$$

(Val Fun Refine)
$$\frac{E, x : T \vdash A : U \quad E \vdash C\{\mathbf{fun}\, x \to A/y\}}{E \vdash \mathbf{fun}\, x \to A : \{y : (\Pi x : T.\, U) \mid C\}}$$

(Val Pair Refine)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\} \quad E \vdash C\{(M,N)/y\}}{E \vdash (M,N) : \{y : (\Sigma x : T.\, U) \mid C\}}$$

(Val Inl Inr Fold Refine)
$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U \quad E \vdash C\{h\, M/y\}}{E \vdash h\, M : \{y : U \mid C\}}$$

Lemma 21 (Alternative Typing)
Assuming that E is executable, the expression typing relation $E \vdash A : T$ is the least relation closed under the alternative rules for values displayed above together with the original rules for expressions.

Proof: For the duration of this proof we write $E \vdash_{alt} A : T$ to mean that the judgment follows from the alternative rules for values together with the original rules for expressions.

Each of the alternative rules (Val Var Refine), (Val Unit Refine), (Val Fun Refine), (Val Pair Refine), and (Val Inl Inr

Fold Refine) is derivable in the original system, by appeal to the original rule and (Val Refine) in each case. Hence, we can show that $E \vdash_{alt} A : T$ implies $E \vdash A : T$, by induction on the derivation of $E \vdash_{alt} A : T$. We omit the details.

To complete the proof, we prove that $E \vdash M : T$ implies $E \vdash_{alt} M : T$ by induction on the derivation of $E \vdash M : T$.

(Val Refine) We have $E \vdash M : \{x : T \mid C\}$ derived from $E \vdash M : T$ and $E \vdash C\{M/x\}$. By induction hypothesis, $E \vdash_{alt} M : T$. Therefore there must be an instance of one of the alternative rules of the form

$$\frac{(\dots) \qquad E \vdash C'\{M/x\}}{E \vdash_{alt} M : \{x : T' \mid C'\}}$$

followed by some instances of (Exp Subsum) such that, by Lemma 20 (Transitivity), $E \vdash \{x : T' \mid C'\} <: T$. Since $E \vdash C'\{M/x\}$ and $E \vdash C\{M/x\}$, by (And Intro) we have $E \vdash (C' \wedge C)\{M/x\}$, so, by the same alternative rule, we have $E \vdash_{alt} M : \{x : T' \mid C' \wedge C\}$. By Lemma 12 (And Sub), $E \vdash \{x : T' \mid C' \wedge C\} <: \{x : \{x : T' \mid C'\} \mid C\}$. By (Sub Refine), $E \vdash \{x : T' \mid C'\} <: T$ implies $E \vdash \{x : \{x : T' \mid C'\} \mid C\} <: \{x : T \mid C\}$. By Lemma 20 (Transitivity), $E \vdash \{x : T' \mid C' \wedge C\} <: \{x : T \mid C\}$. By (Exp Subsum), $E \vdash_{alt} M : \{x : T \mid C\}$.

(Exp Subsum) We have $E \vdash M : T'$ derived from $E \vdash M : T$ and $E \vdash T <: T'$. By induction hypothesis, $E \vdash_{alt} M : T$. By (Exp Subsum), $E \vdash_{alt} M : T'$.

The remaining possibilities are that $E \vdash M : T$ is derived by one of the rules (Val Var), (Val Unit), (Val Fun), (Val Pair), or (Val Inl Inr Fold). By appeal to the corresponding alternative rules and property (True), in each case we can derive $E \vdash_{alt} M : \{x : T \mid \mathsf{True}\}$. By Lemma 11 (Sub Refine Left Refl), $E \vdash \{x : T \mid \mathsf{True}\} <: T$. By (Exp Subsum), $E \vdash_{alt} M : T$. □

Lemma 22 (Formulas)
If $E \vdash M : T$ and $x \notin dom(E)$ then $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)\{M/x\}$.

Proof: By appeal to Lemma 21 (Alternative Typing), the proof is by induction on the derivation of $E \vdash M : T$.

In case that $E \vdash M : T$ follows by one of the alternative rules for typing values, by inspection, it follows that we can derive $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)\{M/x\}$.

Otherwise, $E \vdash M : T$ follows by (Exp Subsum) from $E \vdash M : T'$ and $E \vdash T' <: T$ for some T'. By induction hypothesis, $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T')\{M/x\}$. By Lemma 8 (Logical Subtyping), since $x \notin dom(E)$, we have $\mathsf{forms}(E), \mathsf{forms}(x : T') \vdash \mathsf{forms}(x : T)$. By property (Subst), $\mathsf{forms}(E), (\mathsf{forms}(x : T')\{M/x\}) \vdash \mathsf{forms}(x : T)\{M/x\}$. By property (Cut), $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)\{M/x\}$. □

C.5 Properties of Substitution

To state the two substitution lemmas in this section, we need a notation for applying a substitution to the entries in environments. If $x \notin dom(E)$, let $E\{M/x\}$ be the outcome of applying $\{M/x\}$ to each type occurring in E. Similarly, if $\alpha \notin dom(E)$, let $E\{T/\alpha\}$ be the outcome of applying $\{T/\alpha\}$ to each type occurring in E. We define these notations as follows.

Substitution into Typing Environments:

$$E\{M/x\} = (\mu_1\{M/x\}, \dots, \mu_n\{M/x\}) \quad \text{where } x \notin dom(E) \text{ and } E = \mu_1, \dots, \mu_n$$

$$\mu\{M/x\} = \begin{cases} y : (U\{M/x\}) & \text{if } \mu = (y : U) \text{ and } x \neq y \\ a \updownarrow (U\{M/x\}) & \text{if } \mu = a \updownarrow U \\ \mu & \text{otherwise} \end{cases}$$

$$E\{T/\alpha\} = (\mu_1\{T/\alpha\}, \dots, \mu_n\{T/\alpha\}) \quad \text{where } \alpha \notin dom(E) \text{ and } E = \mu_1, \dots, \mu_n$$

$$\mu\{T/\alpha\} = \begin{cases} y : (U\{T/\alpha\}) & \text{if } \mu = (y : U) \\ a \updownarrow (U\{T/\alpha\}) & \text{if } \mu = a \updownarrow U \\ \mu & \text{otherwise} \end{cases}$$

Our first substitution lemma shows how substitution of a value M for a variable x affects various judgments.

Lemma 23 (Substitution)

(1) *If $h : (T, U)$ then $h : (T\{M/x\}, U\{M/x\})$.*

(2) *If $x \notin dom(E)$ then $\mathsf{forms}(E)\{M/x\} = \mathsf{forms}(E\{M/x\})$.*

(3) *If $E, x : U, E' \vdash \diamond$ and $E \vdash M : U$ then $E, (E'\{M/x\}) \vdash \diamond$.*

(4) *If $E, x : U, E' \vdash C$ and $E \vdash M : U$ then $E, (E'\{M/x\}) \vdash C\{M/x\}$.*

(5) *Suppose that $E \vdash M : U$.*

- *If $E, x : U, E' \vdash T$ then $E, (E'\{M/x\}) \vdash T\{M/x\}$.*
- *If $E, x : U, E' \vdash T :: \nu$ then $E, (E'\{M/x\}) \vdash T\{M/x\} :: \nu$.*
- *If $E, x : U, E' \vdash T <: T'$ then $E, (E'\{M/x\}) \vdash T\{M/x\} <: T'\{M/x\}$.*
- *If $E, x : U, E' \vdash A : T$ then $E, (E'\{M/x\}) \vdash A\{M/x\} : T\{M/x\}$.*

Proof:

(1) By cases on the definition of $h : (T, U)$.

(2) By definition of $\mathsf{forms}(E)$.

(3) By induction on the derivation of $E,x:U,E' \vdash \diamond$, Lemma 2 (Derived Judgments), and standard properties of substitution.

(4) The judgment $E,x:U,E' \vdash C$ can only be an instance of (Derive) such as the following:

$$\frac{\begin{array}{l} E,x:U,E' \vdash \diamond \\ \mathit{fnfv}(C) \subseteq \mathit{dom}(E,x:U,E') \\ \mathsf{forms}(E,x:U,E') \vdash C \end{array}}{E,x:U,E' \vdash C}$$

We can rewrite this instance as follows, by expanding definitions, where $S = \mathsf{forms}(x:U)$, we have $\mathsf{forms}(E,E'),S \vdash C$.

By property (Subst), $\mathsf{forms}(E,(E'\{M/x\})),S\{M/x\} \vdash C\{M/x\}$, and by Lemma 22 (Formulas), $E \vdash M : U$ and $x \notin \mathit{dom}(E)$ imply $\mathsf{forms}(E) \vdash S\{M/x\}$.

These two facts, by properties (Mon) and (Cut), entail $\mathsf{forms}(E,(E'\{M/x\})) \vdash C\{M/x\}$.

Hence, we obtain the following instance of (Derive):

$$\frac{\begin{array}{l} E,(E'\{M/x\}) \vdash \diamond \\ \mathit{fnfv}(C\{M/x\}) \subseteq \mathit{dom}(E,(E'\{M/x\})) \\ \mathsf{forms}(E,(E'\{M/x\})) \vdash C\{M/x\} \end{array}}{E,(E'\{M/x\}) \vdash C\{M/x\}}$$

(5) By simultaneous induction on the derivation of each judgment, using the previous points, Lemma 6 (Weakening), and standard properties of substitution. □

The following auxiliary lemma expresses that kinding judgments do not depend on type declarations of the form $\alpha <: \alpha'$.

Lemma 24
If $E,\alpha <: \alpha',E' \vdash T :: \nu$ then $E,\alpha :: \mathbf{pub},\alpha' :: \mathbf{tnt},E' \vdash T :: \nu$.

Proof: By induction on the derivation of $E,\alpha <: \alpha',E' \vdash T :: \nu$. □

Our second substitution lemma shows how substitution of a type T for a variable α affects various judgments.

Lemma 25 (Type Substitution)

(1) *If $E,\alpha,E' \vdash \mathcal{J}$ and $E \vdash T$ and $\mathit{recvar}(E) \cap \mathit{fnfv}(T) = \varnothing$ then $E,(E'\{T/\alpha\}) \vdash \mathcal{J}\{T/\alpha\}$.*

(2) *If $E,\alpha :: \nu,E' \vdash \diamond$ and $E \vdash T :: \nu$ then $E,(E'\{T/\alpha\}) \vdash \diamond$.*

(3) *If $E,\alpha :: \nu,E' \vdash U$ and $E \vdash T :: \nu$ then $E,(E'\{T/\alpha\}) \vdash U\{T/\alpha\}$.*

(4) *If $E,\alpha :: \nu,E' \vdash T' :: \nu'$ and $E \vdash T :: \nu$ then $E,(E'\{T/\alpha\}) \vdash T'\{T/\alpha\} :: \nu'$.*

(5) *If $E,\alpha <: \alpha',E' \vdash T <: T'$ and $E \vdash U <: U'$ then $E,(E'\sigma) \vdash T\sigma <: T'\sigma$ where $\sigma = \{U/\alpha\}\{U'/\alpha'\}$.*

Proof:

(1) For each judgment $\mathcal{J}$, by induction on the derivation of $E,\alpha,E' \vdash \mathcal{J}$.

(2) By induction on the derivation of $E,\alpha :: \nu,E' \vdash \diamond$, noting that, by Lemma 2 (Derived Judgments), $E \vdash T :: \nu$ implies $\mathit{fnfv}(T) \subseteq \mathit{dom}(E)$.

(3) By definition (Type), point (1), and Lemma 2 (Derived Judgments).

(4) By induction on the derivation of $E,\alpha :: \nu,E' \vdash T' :: \nu'$.

(5) By induction on the derivation of $E,\alpha <: \alpha',E' \vdash T <: T'$.

In case (Sub Public Tainted), we have $E,\alpha <: \alpha',E' \vdash T <: T'$ derived from $E,\alpha <: \alpha',E' \vdash T :: \mathbf{pub}$ and $E,\alpha <: \alpha',E' \vdash T' :: \mathbf{tnt}$.

By Lemma 24, we have $E,\alpha :: \mathbf{pub},\alpha' :: \mathbf{tnt},E' \vdash T :: \mathbf{pub}$ and $E,\alpha :: \mathbf{pub},\alpha' :: \mathbf{tnt},E' \vdash T' :: \mathbf{tnt}$.

By point (4), we have $E,(E'\sigma) \vdash T\sigma :: \mathbf{pub}$ and $E,(E'\sigma) \vdash T'\sigma :: \mathbf{tnt}$.

By (Sub Public Tainted), we get: $E,(E'\sigma) \vdash T\sigma <: T'\sigma$. □

C.6 Proof of Theorem 1 (Safety)

We need the following inversion lemma, for analyzing instances of subtyping.

Lemma 26 (Inversion)

(1) *Let T be $\{y : (\Pi x : T''.\, U'') \mid C\}$ or $(\Pi x : T''.\, U'')$.*
If $E \vdash T <: \Pi x : T'.\, U'$
then $E \vdash T' <: T''$ and $E,x:T' \vdash U'' <: U'$.

(2) *Let T be $\{y : (\Sigma x : T''.\, U'') \mid C\}$ or $(\Sigma x : T''.\, U'')$.*
If $E \vdash T <: \Sigma x : T'.\, U'$
then $E \vdash T'' <: T'$ and $E,x:T'' \vdash U'' <: U'$.

(3) *Let T be $\{y : (\mu\alpha.U) \mid C\}$ or $(\mu\alpha.U)$.*
If $E \vdash T <: \mu\alpha'.U'$
then $E \vdash U\{\mu\alpha.U/\alpha\} <: U'\{\mu\alpha'.U'/\alpha'\}$.

(4) *Let T be $\{y : T_1 + T_2) \mid C\}$ or $T_1 + T_2$.*
If $E \vdash T <: U_1 + U_2$
then $E \vdash T_1 <: U_1$ and $E \vdash T_2 <: U_2$.

(5) *Let h be* inl*,* inr*, or* fold*. Let T be $\{y : U \mid C\}$ or U for any U such that $h : (H, U)$. For any H' and U' such that $h : (H', U')$, if $E \vdash T <: U'$ then $E \vdash H <: H'$.*

Proof:

(1) By induction on the derivations of $E \vdash T <: \Pi x : T'.\ U'$, proceeding by a case analysis of the final rule, which can only be (Sub Refl), (Sub Fun), (Sub Refine Left), or (Sub Public Tainted).

The cases for (Sub Refl) or (Sub Fun) are trivial.

The case for (Sub Refine Left) is a straightforward application of the induction hypothesis.

We analyze in detail the case for (Sub Public Tainted).

It must be the case that $E \vdash T$::**pub** and $E \vdash \Pi x : T'.\ U'$::**tnt**.

Since $E \vdash T$::**pub** can only follow by (Kind Fun) and possibly (Kind Refine Public), it must be the case that $E \vdash T''$::**tnt** and $E, x : T'' \vdash U''$::**pub**.

Since $E \vdash \Pi x : T'.\ U'$::**tnt** can only follow by (Kind Fun), it must be the case that $E \vdash T'$::**pub** and $E, x : T' \vdash U'$::**tnt**.

By (Sub Public Tainted), $E \vdash T' <: T''$.

By Lemma 9 (Bound Weakening), $E, x : T' \vdash U''$::**pub**.

By (Sub Public Tainted), $E, x : T' \vdash U'' <: U'$.

(2) By induction on the derivations of $E \vdash T <: \Sigma x : T'.\ U'$, proceeding by a case analysis of the final rule, which can only be (Sub Refl), (Sub Pair), (Sub Refine Left), or (Sub Public Tainted).

The cases for (Sub Refl) and (Sub Pair) are trivial.

The case for (Sub Refine Left) is a straightforward application of the induction hypothesis.

We analyze in detail the case for (Sub Public Tainted).

It must be the case that $E \vdash T$::**pub** and $E \vdash \Sigma x : T'.\ U'$::**tnt**.

Since $E \vdash T$::**pub** can only follow by (Kind Pair) and possibly (Kind Refine Public), it must be the case that $E \vdash T''$::**pub** and $E, x : T'' \vdash U''$::**pub**.

Since $E \vdash \Sigma x : T'.\ U'$::**tnt** can only follow by (Kind Pair), it must be the case that $E \vdash T'$::**tnt** and $E, x : T' \vdash U'$::**tnt**.

By (Sub Public Tainted), $E \vdash T'' <: T'$.

By Lemma 9 (Bound Weakening), $E, x : T'' \vdash U'$::**tnt**.

By (Sub Public Tainted), $E, x : T'' \vdash U'' <: U'$.

(3) By induction on the derivations of $E \vdash T <: \mu\alpha'.U'$, proceeding by a case analysis of the final rule, which can only be (Sub Refl), (Sub Rec), (Sub Refine Left), or (Sub Public Tainted).

The case for (Sub Refl) is trivial.

In case (Sub Rec), we have $E \vdash T <: \mu\alpha'.U'$ derived from $E, \alpha <: \alpha' \vdash U <: U'$ where $T = \mu\alpha.U$.

By point (5) of Lemma 25 (Type Substitution), $E \vdash U\{\mu\alpha.U/\alpha\} <: U'\{\mu\alpha'.U'/\alpha'\}$.

The case for (Sub Refine Left) is a straightforward application of the induction hypothesis.

In case (Sub Public Tainted), we have $E \vdash T$::**pub** and $E \vdash \mu\alpha'.U'$::**tnt**.

Since $E \vdash T$::**pub** can only follow from (Kind Rec) and possibly (Kind Refine Public), it must be the case that $E \vdash \mu\alpha.U$ :: **pub** and E, α :: **pub** $\vdash U$ :: **pub**.

Since $E \vdash \mu\alpha'.U'$::**tnt** can only follow from (Kind Rec), it must be the case that E, α' :: **tnt** $\vdash U'$ :: **tnt**.

By point (4) of Lemma 25 (Type Substitution), $E \vdash U\{\mu\alpha.U/\alpha\}$ :: **pub**.

By point (4) of Lemma 25 (Type Substitution), $E \vdash U'\{\mu\alpha'.U'/\alpha'\}$ :: **tnt**.

By (Sub Public Tainted), $E \vdash U\{\mu\alpha.U/\alpha\} <: U'\{\mu\alpha'.U'/\alpha'\}$.

(4) By induction on the derivations of $E \vdash T <: U_1 + U_2$, proceeding by a case analysis of the final rule, which can only be (Sub Refl), (Sub Sum), (Sub Refine Left), or (Sub Public Tainted).

The cases for (Sub Refl) and (Sub Sum) are trivial.

The case for (Sub Refine Left) is a straightforward application of the induction hypothesis.

We analyze in detail the case for (Sub Public Tainted).

It must be the case that $E \vdash T$::**pub** and $E \vdash U_1 + U_2$::**tnt**.

Since $E \vdash T$::**pub** can only follow by (Kind Sum) and possibly (Kind Refine Public), it must be the case that $E \vdash T_1$::**pub** and $E \vdash T_2$::**pub**.

Since $E \vdash U_1 + U_2$::**tnt** can only follow by (Kind Sum), it must be the case that $E \vdash U_1$::**tnt** and $E \vdash U_2$::**tnt**.

By (Sub Public Tainted), $E \vdash T_1 <: U_1$ and $E \vdash T_2 <: U_2$.

(5) Let T be $\{y : U \mid C\}$ or U for any U such that $h : (H, U)$. There are three ways in which $h : (H, U)$ may be obtained, depending on h.

In case $h =$ inl we have $U = H + S$ for some S. For any H' and S', assume $E \vdash T <: H' + S'$ and we are to show $E \vdash H <: H'$. This follows from point (4).

In case $h = \mathsf{inr}$ we have $U = S + H$ for some S. For any H' and S', assume $E \vdash T <: S' + H'$ and we are to show $E \vdash H <: H'$. This follows from point (4).

In case $h = \mathsf{fold}$ we have $U = \mu\alpha.S$ and $H = S\{U/\alpha\}$ for some S. For any S', assume $E \vdash T <: \mu\alpha.S'$ and we are to show $E \vdash H <: S'\{\mu\alpha.S'/\alpha\}$. This follows from point (3). □

Recall from Section 4 that $\overline{A}$ is the set of formulas extracted from the expression A. For example, $\overline{(\nu a)(\mathbf{assume}\ \mathsf{Foo}(a,x) \upharpoonright \mathbf{assume}\ \mathsf{Bar}(z))} = \exists a.(\mathsf{Foo}(a,x) \wedge \mathsf{Bar}(z))$. The following states that if A is well-typed in environment E, then the formulas extracted from A are well-formed in E, that is, all their free variables are declared in E.

Lemma 27
If $E \vdash A : T$ then $E \vdash \{\overline{A}\}$.

Proof: By Lemma 2 (Derived Judgments), $E \vdash A : T$ implies $E \vdash \diamond$ and $\mathit{fnfv}(A) \subseteq \mathit{dom}(E)$. By induction on the structure of A, $\mathit{fnfv}(\overline{A}) \subseteq \mathit{dom}(E)$. By (Type), $E \vdash \diamond$ and $\mathit{fnfv}(A) \subseteq \mathit{dom}(E)$ imply $E \vdash \{\overline{A}\}$. □

The next two lemmas assert that heating $A \Rrightarrow A'$ preserves the extracted formulas of an expression (that is, the formulas extracted from A' follow from those extracted from A) and also that heating preserves types.

Lemma 28 ($\Rrightarrow$ Preserves Logic)
If $A \Rrightarrow A'$ then $\overline{A'} \vdash \overline{A}$.

Proof: By induction on the derivation of $A \Rrightarrow A'$.

(Heat Refl), (Heat Res Let), and (Heat Fork Let) follow from $\overline{A'} = \overline{A}$ and Property (Axiom).

(Heat Trans) follows from Property (Cut).

(Heat Let) is by induction.

(Heat Res) is by induction plus Property (Exists Intro).

(Heat Fork 1) and (Heat Fork 2) are by induction, Property (Axiom) for $\overline{B}$, and Properties (And Elim) and (And Intro).

(Heat Fork ()), (Heat Msg ()), (Heat Assume ()) follow from $\mathsf{True} \wedge \overline{A} \vdash \overline{A}$ and its converse, derived from Properties (True), (And Elim), and (And Intro).

(Heat Res Fork 1), (Heat Res Fork 2) follow from

$$\exists x.(C' \wedge C) \vdash (C' \wedge \exists x.C) \text{ if } x \notin \mathit{fv}(C')$$

derived from Properties (Exists Elim), (And Elim), (Exists Intro), and (And Intro).

(Heat Fork Assoc), (Heat Fork Comm) follow from $(C \wedge (C' \wedge C'')) \vdash ((C \wedge C') \wedge C'')$ and $(C \wedge C') \vdash (C' \wedge C)$, derived from Properties (And Elim) and (And Intro). □

Proposition 29 ($\Rrightarrow$ Preserves Types)
If E is executable, $E \vdash A : T$, and $A \Rrightarrow A'$, then $E \vdash A' : T$.

Proof: By an induction on the derivation of $A \Rrightarrow A'$.

(Heat Refl) We have $A \Rrightarrow A$ and $E \vdash A : T$, so we are done.

(Heat Trans) We have $A \Rrightarrow A''$ derived from $A \Rrightarrow A'$ and $A' \Rrightarrow A''$.

Assume $E \vdash A : T$. By induction hypothesis, $A \Rrightarrow A'$ implies $E \vdash A' : T$. By induction hypothesis, $A' \Rrightarrow A''$ implies $E \vdash A'' : T$.

(Heat Fork ()) This rule $() \upharpoonright A \equiv A$ means both (1) $() \upharpoonright A \Rrightarrow A$ and (2) $A \Rrightarrow () \upharpoonright A$.

For (1), assume $E \vdash () \upharpoonright A : T$. This must follow from an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{A}\} \vdash () : T_1 \quad E, \_ : \{\overline{()}\} \vdash A : T_2$$

plus some instances of (Exp Subsum) such that $E \vdash T_2 <: T$. By Lemma 4 (Anon Variable Strengthening), we have $E \vdash A : T_2$ because $\{\overline{()}\} = \{\mathsf{True}\}$. By (Exp Subsum), we obtain $E \vdash A : T$ as required.

For (2), assume $E \vdash A : T$. By Lemma 27, (Val Unit), and Lemma 6 (Weakening), we obtain the following.

$$E, \_ : \{\overline{A}\} \vdash () : \mathsf{unit} \quad E, \_ : \{\overline{()}\} \vdash A : T$$

By (Exp Fork), then, we obtain $E \vdash () \upharpoonright A : T$.

(Heat Msg ()) We have $a!M \Rrightarrow a!M \upharpoonright ()$.

Assume $E \vdash a!M : T$. This must follow from an instance of (Exp Send) with conclusion $E \vdash a!M : \mathsf{unit}$ and premises $E \vdash M : T$ and $(a \updownarrow T) \in E$, and some instances of (Exp Subsum) such that $E \vdash \mathsf{unit} <: T$. We can check the following, using (Val Ok).

$$E, \_ : \{\overline{()}\} \vdash a!M : T \quad E, \_ : \{\overline{a!M}\} \vdash () : \mathsf{unit}$$

By (Exp Fork), then, we obtain $E \vdash a!M \upharpoonright () : \mathsf{unit}$. By (Exp Subsum), we obtain $E \vdash a!M \upharpoonright () : T$ as required.

(Heat Assume ()) We have $\mathbf{assume}\ C \Rrightarrow \mathbf{assume}\ C \upharpoonright ()$.

Assume $E \vdash \mathbf{assume}\ C : T$. This must follow from an instance of (Exp Assume) with premise $E, \_ : \{C\} \vdash () : T'$ and conclusion $E \vdash \mathbf{assume}\ C : T'$ plus some instances of (Exp Subsum) such that $E \vdash T' <: T$. We can check the following, since $\{\overline{()}\} = \{\mathsf{True}\}$ and $\{\overline{\mathbf{assume}\ C}\} = \{C\}$.

$$E, \_ : \{\overline{()}\} \vdash \mathbf{assume}\ C : T'$$
$$E, \_ : \{\overline{\mathbf{assume}\ C}\} \vdash () : T'$$

By (Exp Fork), then, we obtain $E \vdash \textbf{assume}\ C \Rsh () : T'$. By (Exp Subsum), we obtain $E \vdash \textbf{assume}\ C \Rsh () : T$.

(Heat Let) We have $\textbf{let}\ x = A\ \textbf{in}\ B \Rrightarrow \textbf{let}\ x = A'\ \textbf{in}\ B$ derived from $A \Rrightarrow A'$.

Assume $E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : T$. This must follow from an instance of (Exp Let) with premises

$$E \vdash A : T' \quad E, x : T' \vdash B : U \quad x \notin fv(U)$$

plus some instances of (Exp Subsum) such that $E \vdash U <: T$. By induction hypothesis, $A \Rrightarrow A'$ implies $E \vdash A' : T'$. Hence, by (Exp Let) and (Exp Subsum) we can conclude that $E \vdash \textbf{let}\ x = A'\ \textbf{in}\ B : T$.

(Heat Res) We have $(\nu a)A \Rrightarrow (\nu a)A'$ derived from $A \Rrightarrow A'$.

Assume $E \vdash (\nu a)A : T$. This must follow from an instance of (Exp Res) with premises

$$E, a : (T_c)\mathsf{chan} \vdash A : U \quad a \notin fn(U)$$

plus some instances of (Exp Subsum) such that $E \vdash U <: T$. By induction hypothesis, $A \Rrightarrow A'$ implies $E, a : (T_c)\mathsf{chan} \vdash A' : U$. Hence, by (Exp Let) and (Exp Subsum) we can conclude that $E \vdash \textbf{let}\ x = A'\ \textbf{in}\ B : T$.

(Heat Fork 1) We have $(A \Rsh B) \Rrightarrow (A' \Rsh B)$ derived from $A \Rrightarrow A'$.

Assume $E \vdash (A \Rsh B) : T$. This must follow from an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{B}\} \vdash A : T_A \quad E, \_ : \{\overline{A}\} \vdash B : T_B$$

plus some instances of (Exp Subsum) such that $E \vdash T_B <: T$. By induction hypothesis, $A \Rrightarrow A'$ implies $E, \_ : \{\overline{B}\} \vdash A' : T_A$. By Lemma 28 ($\Rrightarrow$ Preserves Logic), $A \Rrightarrow A'$ implies $\overline{A'} \vdash \overline{A}$. By Property (Mon), $E, \{\overline{A'}\} \vdash \overline{A}$. By Rule (Sub Ok), $E \vdash \{\overline{A'}\} <: \{\overline{A}\}$. By Lemma 9 (Bound Weakening), $E, \_ : \{\overline{A}\} \vdash B : T_B$ and $E \vdash \{\overline{A'}\} <: \{\overline{A}\}$ imply $E, \_ : \{\overline{A'}\} \vdash B : T_B$. Hence, we can establish:

$$E, \_ : \{\overline{B}\} \vdash A' : T_A \quad E, \_ : \{\overline{A'}\} \vdash B : T_B$$

By (Exp Fork) and (Exp Subsum), then, we obtain $E \vdash (A' \Rsh B) : T$.

(Heat Fork 2) We have $(B \Rsh A) \Rrightarrow (B \Rsh A')$ derived from $A \Rrightarrow A'$.

This is similar to the case for (Heat Fork 1); we omit the details.

(Heat Res Fork 1) We have $A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$ given that $a \notin fn(A')$.

Assume $E \vdash A' \Rsh (\nu a)A : T$. This must obtain from an instance of (Exp Fork) with premises

$$E, \_ : \{\exists a.\overline{A}\} \vdash A' : T_1'$$
$$E, \_ : \{\overline{A'}\} \vdash (\nu a)A : T_1$$

and conclusion $E \vdash A' \Rsh (\nu a)A : T_1$, and some instances of (Exp Subsum) such that $E \vdash T_1 <: T$.

Moreover, there must be an instance of (Exp Res) with premises

$$E, \_ : \{\overline{A'}\}, a : T_a \vdash A : T_2 \quad a \notin fn(T_2)$$

and conclusion $E, \_ : \{\overline{A'}\} \vdash (\nu a)A : T_2$, and some instances of (Exp Subsum) such that $E, \_ : \{\overline{A'}\} \vdash T_2 <: T_1$.

By Lemma 6 (Weakening) and Lemma 20 (Transitivity), $E, \_ : \{\overline{A'}\} \vdash T_2 <: T_1$ and $E \vdash T_1 <: T$ imply $E, \_ : \{\overline{A'}\}, a : T_a \vdash T_2 <: T$. By (Exp Subsum), $E, \_ : \{\overline{A'}\}, a : T_a \vdash A : T$. By Lemma 5 (Exchange), $a \notin fn(A')$ implies:

$$E, a : T_a, \_ : \{\overline{A'}\} \vdash A : T$$

By Lemma 6 (Weakening), we obtain:

$$E, a : T_a, \_ : \{\exists a.\overline{A}\} \vdash A' : T_1'$$

By Lemma 10 (Bound Weakening Ok) and $E, a : T_a, \overline{A} \vdash \exists a.\overline{A}$ we get:

$$E, a : T_a, \_ : \{\overline{A}\} \vdash A' : T_1'$$

Hence we have:

$$E, a : T_a, \_ : \{\overline{A}\} \vdash A' : T_1'$$
$$E, a : T_a, \_ : \{\overline{A'}\} \vdash A : T$$

By (Exp Fork) we obtain:

$$E, a : T_a \vdash (A' \Rsh A) : T$$

By (Exp Res) we obtain, as desired:

$$E \vdash (\nu a)(A' \Rsh A) : T$$

(Heat Res Fork 2) We have $((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$ given that $a \notin fn(A')$.

This case is similar to (Heat Res Fork 1); in both parts we know that $a \notin fn(T)$ because of the use of the rule (Exp Res).

(Heat Res Let) We have $\textbf{let}\ x = (\nu a)A\ \textbf{in}\ B \Rrightarrow (\nu a)\textbf{let}\ x = A\ \textbf{in}\ B$ given that $a \notin fn(B)$.

Assume $E \vdash \textbf{let } x = (\nu a)A \textbf{ in } B : T$. This must follow from an instance of (Exp Let) with premises $E \vdash (\nu a)A : T_2$ and

$$E, x : T_2 \vdash B : T_1$$

(hence $a \notin fn(T_1)$) and $x \notin fv(T_1)$ and conclusion $E \vdash \textbf{let } x = (\nu a)A \textbf{ in } B : T_1$, and some instances of (Exp Subsum) such that $E \vdash T_1 <: T$. Moreover, there must be an instance of (Exp Res) with premises

$$E, a : T_a \vdash A : T_3$$

and $a \notin fv(T_3)$ and conclusion $E \vdash (\nu a)A : T_3$ and some instances of (Exp Subsum) such that $E \vdash T_3 <: T_2$.

By (Exp Subsum),

$$E, a : T_a \vdash A : T_2$$

By Lemma 6 (Weakening),

$$E, a : T_a, x : T_2 \vdash B : T_1$$

By (Exp Let),

$$E, a : T_a \vdash \textbf{let } x = A \textbf{ in } B : T_1$$

By (Exp Res), since we know $a \notin fn(T_1)$,

$$E \vdash (\nu a : T_a)\textbf{let } x = A \textbf{ in } B : T_1$$

By (Exp Subsum),

$$E \vdash (\nu a : T_a)\textbf{let } x = A \textbf{ in } B : T$$

(Heat Fork Assoc) We have $A \Rsh (A' \Rsh A'') \equiv (A \Rsh A') \Rsh A''$, which amounts to (1) $A \Rsh (A' \Rsh A'') \Rrightarrow (A \Rsh A') \Rsh A''$ and (2) $(A \Rsh A') \Rsh A'' \Rrightarrow A \Rsh (A' \Rsh A'')$.

For (1), assume $E \vdash A \Rsh (A' \Rsh A'') : T$. There must be an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{A'} \wedge \overline{A''}\} \vdash A : T_1$$
$$E, \_ : \{\overline{A}\} \vdash (A' \Rsh A'') : T_2$$

and some instances of (Exp Subsum) such that $E \vdash T_2 <: T$. There must be an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{A}\}, \_ : \{\overline{A''}\} \vdash A' : T_3$$
$$E, \_ : \{\overline{A}\}, \_ : \{\overline{A'}\} \vdash A'' : T_4$$

and some instances of (Exp Subsum) such that $E \vdash T_4 <: T_2$.

By Lemma 13 (Ok $\wedge$), we have:

$$E, \_ : \{\overline{A'}\}, \_ : \{\overline{A''}\} \vdash A : T_1$$

By Lemma 5 (Exchange), we obtain:

$$E, \_ : \{\overline{A''}\}, \_ : \{\overline{A'}\} \vdash A : T_1$$
$$E, \_ : \{\overline{A''}\}, \_ : \{\overline{A}\} \vdash A' : T_3$$

Hence, by (Exp Fork), we obtain:

$$E, \_ : \{\overline{A''}\} \vdash (A \Rsh A') : T_3$$

By Lemma 13 (Ok $\wedge$), we have:

$$E, \_ : \{\overline{A \Rsh A'}\} \vdash A'' : T_4$$

Hence, by (Exp Fork), we obtain:

$$E \vdash (A \Rsh A') \Rsh A'' : T_4$$

By (Exp Subsum), and $E \vdash T_4 <: T$, we obtain:

$$E \vdash (A \Rsh A') \Rsh A'' : T$$

Part (2) follows by a symmetric argument.

(Heat Fork Comm) We have $(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$.

This case is similar to (Heat Fork Assoc); we omit the details.

(Heat Fork Let) We have $\textbf{let } x = (A \Rsh A') \textbf{ in } B \equiv A \Rsh (\textbf{let } x = A' \textbf{ in } B)$, which amounts to (1) $\textbf{let } x = (A \Rsh A') \textbf{ in } B \Rrightarrow A \Rsh (\textbf{let } x = A' \textbf{ in } B)$ and (2) $A \Rsh (\textbf{let } x = A' \textbf{ in } B) \Rrightarrow \textbf{let } x = (A \Rsh A') \textbf{ in } B$.

For (1), assume $E \vdash \textbf{let } x = (A \Rsh A') \textbf{ in } B : T$. This must follow from an instance of (Exp Let) with premises

$$E \vdash (A \Rsh A') : T_1 \quad E, x : T_1 \vdash B : T_2 \quad x \notin fv(T_2)$$

and some instances of (Exp Subsum) such that $E \vdash T_2 <: T$. There must be an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{A'}\} \vdash A : T_3 \quad E, \_ : \{\overline{A}\} \vdash A' : T_4$$

and some instances of (Exp Subsum) such that $E \vdash T_4 <: T_1$.

By (Exp Subsum), $E, \_ : \{\overline{A}\} \vdash A' : T_1$.

By Lemma 6 (Weakening), $E, \_ : \{\overline{A}\}, x : T_1 \vdash B : T_2$.

By (Exp Let), $E, \_ : \{\overline{A}\} \vdash \textbf{let } x = A' \textbf{ in } B : T_2$.

We have $\overline{\textbf{let } x = A' \textbf{ in } B} = \overline{A'}$.

By (Exp Fork), $E \vdash A \Rsh \textbf{let } x = A' \textbf{ in } B : T_2$.

By (Exp Subsum), $E \vdash T_2 <: T$ implies $E \vdash A \Rsh \textbf{let } x = A' \textbf{ in } B : T$.

For (2), assume $E \vdash A \upharpoonleft (\textbf{let}\, x = A' \textbf{ in}\, B) : T$. This must follow from an instance of (Exp Fork) with premises

$$E, \_ : \{\overline{A'}\} \vdash A : T_1$$
$$E, \_ : \{\overline{A}\} \vdash (\textbf{let}\, x = A' \textbf{ in}\, B) : T_2$$

(since $\overline{\textbf{let}\, x = A' \textbf{ in}\, B} = \overline{A'}$) and some instances of (Exp Subsum) such that $E \vdash T_2 <: T$. There must be an instance of (Exp Let) with premises

$$E, \_ : \{\overline{A}\} \vdash A' : T_4$$
$$E, \_ : \{\overline{A}\}, x : T_4 \vdash B : T_3$$

and some instances of (Exp Subsum) such that $E \vdash T_3 <: T_2$.

By (Sub Refine Right):

$$E, \_ : \{\overline{A}\} \vdash T_4 <: \{\_ : T_4 \mid \overline{A}\}$$

Given the following

$$E, \_ : \{\overline{A'}\} \vdash A : T_1$$
$$E, \_ : \{\overline{A}\} \vdash A' : \{\_ : T_4 \mid \overline{A}\}$$

we conclude by (Exp Fork):

$$E \vdash A \upharpoonleft A' : \{\_ : T_4 \mid \overline{A}\}$$

By (Sub Refine Left):

$$E, \_ : \{\overline{A}\} \vdash \{\_ : T_4 \mid \overline{A}\} <: T_4$$

By Lemma 9 (Bound Weakening):

$$E, \_ : \{\overline{A}\}, x : \{\_ : T_4 \mid \overline{A}\} \vdash B : T_3$$

By Lemma 4 (Anon Variable Strengthening):

$$E, x : \{\_ : T_4 \mid \overline{A}\} \vdash B : T_3$$

By (Exp Let):

$$E \vdash \textbf{let}\, x = (A \upharpoonleft A') \textbf{ in}\, B : T_3$$

By (Exp Subsum): $E \vdash T_3 <: T$ implies

$$E \vdash \textbf{let}\, x = (A \upharpoonleft A') \textbf{ in}\, B : T$$

□

Similarly, the next two lemmas assert that reduction $A \rightarrow A'$ preserves the extracted formulas of an expression and also that reduction preserves types.

Lemma 30 ($\rightarrow$ Preserves Logic)
If $A \rightarrow A'$ then $\overline{A'} \vdash \overline{A}$.

Proof: By induction on the derivation of $A \rightarrow A'$.

(Red Comm) follows from $\mathsf{True} \vdash \mathsf{True} \wedge \mathsf{True}$, derived from Properties (True) and (And Intro).

All other base case follow from $\overline{A'} \vdash \mathsf{True}$, derived from Properties (True) and (Mon).

The context cases are handled by induction hypothesis, as in the proof of Lemma 28 ($\Rrightarrow$ Preserves Logic).

(Red Heat) follows from Lemma 28 ($\Rrightarrow$ Preserves Logic) (twice), the induction hypothesis, and Properties (Mon) and (Cut). □

Proposition 31 ($\rightarrow$ Preserves Types)
If E is executable, $fv(A) = \varnothing$, $E \vdash A : T$, and $A \rightarrow A'$, then $E \vdash A' : T$.

Proof: By induction on the derivation of $A \rightarrow A'$, using Lemma 21 (Alternative Typing). Below we implicitly appeal to Lemma 20 (Transitivity) several times (which we may do because of the assumption that E is executable).

(Red Fun) It must be the case that $E \vdash (\textbf{fun}\, x \rightarrow A)\ N : T$ follows by an instance of (Exp Appl) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash (\textbf{fun}\, x \rightarrow A)\ N : U'\{N/x\}$, $E \vdash U'\{N/x\} <: T$ and $E \vdash \textbf{fun}\, x \rightarrow A : \Pi x : T'.\, U'$ and $E \vdash N : T'$ (by Lemma 20 (Transitivity)).

Moreover, it must be the case that $E \vdash \textbf{fun}\, x \rightarrow A : \Pi x : T'.\, U'$ follows by an instance of (Val Fun Refine) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash \textbf{fun}\, x \rightarrow A : \{y : \Pi x : T''.\, U'' \mid C\}$ and $E, x : T'' \vdash A : U''$ and $E \vdash \{y : \Pi x : T''.\, U'' \mid C\} <: \Pi x : T'.\, U'$.

By Lemma 26 (Inversion)(1), $E \vdash T' <: T''$ and $E, x : T' \vdash U'' <: U'$.

By Lemma 23 (Substitution), $E \vdash U''\{N/x\} <: U'\{N/x\}$.

By (Exp Subsum), $E \vdash N : T''$.

By Lemma 23 (Substitution), $E \vdash A\{N/x\} : U''\{N/x\}$

By (Exp Subsum), $E \vdash A\{N/x\} : T$.

(Red Split) It must be the case that $E \vdash \textbf{let}\ (x,y) = (N_1,N_2)\ \textbf{in}\ A : T$ follows by an instance of (Exp Split) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash (N_1,N_2) : (\Sigma x : T'.\, U')$, $E, x : T', y : U', \_ : \{(x,y) = (N_1,N_2)\} \vdash A : V$ and $\{x,y\} \cap fv(V) = \varnothing$, where $E \vdash V <: T$.

Moreover, it must be the case that $E \vdash (N_1,N_2) : (\Sigma x : T'.\, U')$ follows by an instance of (Val Pair Refine) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash (N_1, N_2) : \{y : \Sigma x : T''.\, U'' \mid C\}$ and $E \vdash N_1 : T''$, $E, x : T'' \vdash N_2 : U''$ and $E \vdash \{y : \Sigma x : T''.\, U'' \mid C\} <: \Sigma x : T'.\, U'$.

By Lemma 26 (Inversion)(2), $E \vdash T'' <: T'$ and $E, x : T'' \vdash U'' <: U'$.

By Lemma 9 (Bound Weakening), $E, x : T'', y : U'', \_ : \{(x,y) = (N_1, N_2)\} \vdash A : V$

By Lemma 23 (Substitution), $E, y : U''\{N_1/x\}, \_ : \{(N_1, y) = (N_1, N_2)\} \vdash A\{N_1/x\} : V$.

By Lemma 23 (Substitution), $E, \_ : \{(N_1, N_2) = (N_1, N_2)\} \vdash A\{N_1/x\}\{N_2/x\} : V$.

(In these uses of Lemma 23 (Substitution), the substitutions both apply to V but leave it unchanged because $\{x, y\} \cap fv(V) = \varnothing$.)

By Properties (Eq) and (Mon), $\mathsf{forms}(E) \vdash (N_1, N_2) = (N_1, N_2)$, thus by Lemma 4 (Anon Variable Strengthening), $E \vdash A\{N_1/x\}\{N_2/x\} : V$.

By (Exp Subsum), $E \vdash A\{N_1/x\}\{N_2/x\} : T$.

(Red Match) It must be the case that $E \vdash$ **match** M **with** $h\ x \to A$ **else** $B : T$ follows by an instance of (Exp Match Inl Inr Fold) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash$ **match** M **with** $h\ x \to A$ **else** $B : U$, $E \vdash M : T'$, $h : (H, T')$, $E, x : H, \_ : \{h\ x = M\} \vdash A : U$ and $E, \_ : \{\forall x. h\, x \neq M\} \vdash B : U$ and $E \vdash U <: T$, and hence $x \notin fv(U)$.

Suppose there is no h, N such that $M = h\, N$. we obtain $E, \_ : \{\forall x. h\, x \neq M\} \vdash B : T$ by (Exp Subsum).

Since $fv(A) = \varnothing$, we have $fv(M) = \varnothing$. Hence, by property (Ineq Cons), we have $\varnothing \vdash \forall x. h\, x \neq M$.

Hence, by property (Mon) and Lemma 4 (Anon Variable Strengthening), we obtain $E \vdash B : T$.

Otherwise, suppose there is N such that $M = h\ N$. It must be the case that $E \vdash M : T'$ follows by an instance of (Val Inl Inr Fold Refine) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash h\ N : \{y : U' \mid C\}$, $h : (H', U')$, $E \vdash N : H'$.

By Lemma 26 (Inversion), $E \vdash H' <: H$.

By Lemma 9 (Bound Weakening), $E, x : H', \_ : \{h\, x = M\} \vdash A : U$.

By Lemma 23 (Substitution), $E, \_ : \{h\ N = M\} \vdash A\{M/x\} : U$.

By Properties (Eq) and (Mon), $E \vdash h\ N = M$, so by Lemma 4 (Anon Variable Strengthening), $E \vdash A\{M/x\} : U$.

By (Exp Subsum), $E \vdash A\{M/x\} : T$.

(Red Eq) It must be the case that $E \vdash M = N : T$ follows by an instance of (Exp Eq) after a certain number of instances of (Exp Subsum).

Hence, it must be the case that $E \vdash M = N : \{b : \mathsf{bool} \mid b = \mathbf{true} \Leftrightarrow M = N\}$ and $E \vdash M : T$ and $E \vdash N : U$.

Moreover, by Lemma 20 (Transitivity), $E \vdash \{b : \mathsf{bool} \mid b = \mathbf{true} \Leftrightarrow M = N\} <: T$.

We split the proof in two cases.

- If $M = N$ then $A' = \mathbf{true}$.
  By definition, $\mathbf{true} = \mathsf{inr}()$.
  By (Eq) in the logics and (Val Inl Inr Fold) for inr, $E \vdash \mathbf{true} : \{y : \mathsf{bool} \mid y = \mathbf{true} \Leftrightarrow M = M\}$.
  By (Exp Subsum), $E \vdash \mathbf{true} : T$.
- If $M \neq N$ then $A' = \mathbf{false}$. By definition, $\mathbf{false} = \mathsf{inl}()$.
  By (Val Inl Inr Fold), $E \vdash \mathbf{false} : \{y : \mathsf{bool} \mid y = \mathbf{true} \Leftrightarrow M = N\}$.
  By (Exp Subsum), $E \vdash \mathbf{false} : T$.

(Red Comm) By (Exp Fork), (Exp Send), (Exp Recv), (Exp Subsum) and Lemma 4 (Anon Variable Strengthening), noticing that $\overline{c?} = \mathsf{True}$.

(Red Assert) By (Exp Assert), (Val Unit Refine), (Exp Subsum) and Lemma 11 (Sub Refine Left Refl).

(Red Let Val) By (Exp Let), (Exp Subsum) and Lemma 23 (Substitution).

(Red Let) By (Exp Let), (Exp Subsum) and the induction hypothesis.

(Red Res) By (Exp Res), (Exp Subsum) and the induction hypothesis.

(Red Fork 1) By (Exp Fork), (Exp Subsum) and the induction hypothesis.

(Red Fork 2) By (Exp Fork), (Exp Subsum) and the induction hypothesis.

(Red Heat) By Proposition 29 ($\Rrightarrow$ Preserves Types) and the induction hypothesis. □

Our next results are that typing implies static safety and indeed safety.

Lemma 32 (Static Safety)
If $\varnothing \vdash \mathbf{S} : T$ *then* $\mathbf{S}$ *is statically safe.*

Proof: Consider an arbitrary structure $\mathbf{S}$:

$$(\nu a_1) \ldots (\nu a_\ell)((A_1 \upharpoonleft A_2) \upharpoonleft A_3)$$

where $A_1 = (\prod_{i\in 1..m} \textbf{assume } C_i)$ and $A_2 = (\prod_{j\in 1..n} c_j!M_j)$ and $A_3 = (\prod_{k\in 1..o} \mathscr{L}_k\{e_k\})$. Suppose that $e_p = \textbf{assert } C$ for some $p \in 1..o$. We are to show that $\{C_1,\ldots,C_m\} \vdash C$.

Our hypothesis $\varnothing \vdash \mathbf{S} : T$ must be obtained from ℓ instances of (Exp Res), interleaved with some instances of (Exp Subsum), from hypothesis

$$E_\ell \vdash (A_1 \upharpoonright A_2) \upharpoonright A_3 : U_1$$

with $E_\ell = a_1 \updownarrow T_1, \ldots, a_\ell \updownarrow T_\ell$, for some type U_1 and types $T_1, \ldots, T_l$.

There must be an instance of (Exp Fork) and some instances of (Exp Subsum) such that

$$\begin{array}{l} E_\ell, \_ : \{\overline{A_3}\} \vdash (A_1 \upharpoonright A_2) : U_{12} \\ E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\} \vdash A_3 : U_3 \end{array}$$

for some U_{12} and U_3.

There must be an instance of (Exp Fork) and some instances of (Exp Subsum) such that

$$\begin{array}{l} E_\ell, \_ : \{\overline{A_3}\}, \_ : \{\overline{A_2}\} \vdash A_1 : U_1 \\ E_\ell, \_ : \{\overline{A_3}\}, \_ : \{\overline{A_1}\} \vdash A_2 : U_2 \end{array}$$

for some U_1 and U_2.

Since $\mathscr{L}_k\{e_k\}$ is an elementary expression e_k surrounded by a stack of let-expressions, we have that $\overline{\mathscr{L}_k\{e_k\}} = \mathsf{True}$ for each $k \in 1..o$. Since $E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\} \vdash A_3 : U_3$, it follows that $E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\} \vdash \mathscr{L}_p\{e_p\} : U_p$ for some type U_p, and therefore there is an instance of (Exp Assert) such that $E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\} \vdash \textbf{assert } C : \mathsf{unit}$ follows from $E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\} \vdash C$, and therefore there is an instance of (Derive) with $\mathsf{forms}(E_\ell, \_ : \{\overline{A_1} \wedge \overline{A_2}\}) \vdash C$.

Since E_ℓ contains only names, $\mathsf{forms}(E_\ell) = \varnothing$. By definition of A_1 and A_2, it must be that $\mathsf{forms}(\_ : \{\overline{A_1} \wedge \overline{A_2}\}) = \{C_1,\ldots,C_n\}$. Hence, we have $\{C_1,\ldots,C_n\} \vdash C$, as desired. □

Restatement of Theorem 1 (Safety)
If $\varnothing \vdash A : T$ then A is safe.

Proof: Consider any A' and $\mathbf{S}$ such that $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$; it suffices to show that $\mathbf{S}$ is statically safe. By Proposition 31 ($\to$ Preserves Types), $\varnothing \vdash A : T$ and $A \to^* A'$ imply $\varnothing \vdash A' : T$. By Proposition 29 ($\Rrightarrow$ Preserves Types), this and $A' \Rrightarrow \mathbf{S}$ imply $\varnothing \vdash \mathbf{S} : T$. By Lemma 32 (Static Safety), this implies $\mathbf{S}$ is statically safe. □

C.7 Proof of Theorem 2 (Robust Safety)

First, we note that Un is type equivalent to a range of types.

Lemma 33 (Universal Type)
Given $E \vdash \diamond$ we have $E \vdash \mathsf{Un} <:> T$ for each T below:

$$\{\mathsf{unit}, (\Pi x : \mathsf{Un}.\ \mathsf{Un}), (\Sigma x : \mathsf{Un}.\ \mathsf{Un}), (\mathsf{Un} + \mathsf{Un}), (\mu\alpha.\mathsf{Un})\}$$

Proof: By appeal to Lemma 16 (Public Tainted), it suffices to show that $E \vdash T :: \textbf{pub}$ and $E \vdash T :: \textbf{tnt}$ for each type T in the statement of this lemma. All of these kinding judgments directly follow from the kinding rules. □

The next lemma establishes that any opponent can be well-typed using Un to type its free names. The lemma is a little more general—it applies to any expression containing no Assert; an opponent is any such expression with no free variables.

Lemma 34 (Opponent Typability)
*Suppose $E \vdash \diamond$ and that E is executable. If O is an expression containing no **assert** such that $(a \updownarrow \mathsf{Un}) \in E$ for each name $a \in fn(O)$, and $(x : \mathsf{Un}) \in E$ for each variable $x \in fv(O)$, then $E \vdash O : \mathsf{Un}$.*

Proof: The proof is by induction on the structure of O; in each case we obtain $E \vdash O : \mathsf{Un}$ using the expression typing rule corresponding to the structure of O, the rule of subsumption (Exp Subsum), and the properties of the type Un stated in Lemma 33 (Universal Type).

In the case for an opponent $M = N$ we additionally appeal to Lemma 11 (Sub Refine Left Refl).

In the cases for an opponent that is a split, a match, or a fork, we additionally appeal to Lemma 6 (Weakening).

In the cases mentioning a constructor $h \in \{\mathsf{inl}, \mathsf{inr}, \mathsf{fold}\}$ we appeal to the following instances of the constructor judgment: $E \vdash \mathsf{inl} : (\mathsf{Un}, \mathsf{Un}+\mathsf{Un})$ and $E \vdash \mathsf{inr} : (\mathsf{Un}, \mathsf{Un}+\mathsf{Un})$ and $E \vdash \mathsf{fold} : (\mu\alpha.\mathsf{Un}, \mathsf{Un})$. □

Finally, we prove that robust safety follows by typing.

Restatement of Theorem 2 (Robust Safety)
If $\varnothing \vdash A : \mathsf{Un}$ then A is robustly safe.

Proof: Consider any opponent O with $fn(O) = \{a_1,\ldots,a_n\}$. We are to show the application $O\ A$ is safe. Let $E = a_1 \updownarrow \mathsf{Un}, \ldots, a_1 \updownarrow \mathsf{Un}$. By Lemma 34 (Opponent Typability), $E \vdash O : \mathsf{Un}$. By (Exp Subsum) and Lemma 33 (Universal Type), $E \vdash O : (\Pi x : \mathsf{Un}.\ \mathsf{Un})$. By Lemma 6 (Weakening), $E \vdash A : \mathsf{Un}$. We can easily derive $E \vdash \textbf{let } f = O \textbf{ in } (\textbf{let } x = A \textbf{ in } f\ x) : \mathsf{Un}$, that is, $E \vdash O\,A : \mathsf{Un}$. Hence, we can derive $\varnothing \vdash (\nu a_1)\ldots(\nu a_n)(O\ A) : \mathsf{Un}$. By Theorem 1 (Safety), $(\nu a_1)\ldots(\nu a_n)(O\ A)$ is safe. Restriction does not affect safety, so it follows that $O\ A$ is itself safe, as required. □

D Derived Forms

In our code examples, we use F# syntax for expressions and a convenient F#-like syntax for types. Elaborating on Section 2.3, we describe how these syntactic forms are derived in RCF, our core language.

RCF has a reduced syntax for expressions; the more general expression syntax of F# is derived by inserting **let**-expressions. (We assume that the inserted bound variables are fresh.)

Implicit Lets:

$A; B \triangleq$ **let** $\_ = A$ **in** B
$(A, B) \triangleq$ **let** $x = A$ **in let** $y = B$ **in** (x, y)
$h\ A \triangleq$ **let** $x = A$ **in** $h\ x$
$A\ B \triangleq$ **let** $x = A$ **in let** $y = B$ **in** $x\ y$
let $(x, y) = A$ **in** $B \triangleq$ **let** $z = A$ **in let** $(x, y) = z$ **in** B
match A **with** $h\ x \to B$ **else** $B' \triangleq$
  let $z = A$ **in match** z **with** $h\ x \to B$ **else** B'
$A = B \triangleq$ **let** $x = A$ **in let** $y = B$ **in** $x = y$

We use the following derived syntax for function and tuple types:

Function and Tuple Types:

$T_1 \to T_2 \triangleq \Pi\_ : T_1.\ T_2$
$x_1 : T_1 \to T_2 \triangleq \Pi x_1 : T_1.\ T_2$
$T_1 * T_2 \triangleq \Sigma\_ : T_1.\ T_2$
$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$
$(x_1 : T_1 * \cdots * x_n : T_n)\{C\} \triangleq$
  $\Sigma x_1 : T_1. \ldots \Sigma x_{n-1} : T_{n-1}.\ \{x_n : T_n \mid C\}$

We also support *type abbreviations* in module interfaces:

$$\textbf{type}\ (\alpha_1, \ldots, \alpha_n; x_1{:}T_1, \ldots, x_m{:}T_m)F = \Sigma$$

where Σ is either a type or a type expression defining an algebraic sum type

$$\begin{aligned}\Sigma ::=\ & T \\ & (|h_i\ \textbf{of}\ T_i)_{i \in 1..k} (k \geq 1)\end{aligned}$$

The first form is not recursive: the type T does not contain F. The latter form defines a recursive sum type; we require that all appearances of F in $T_1, \ldots, T_k$ be of the form $(\beta_1, \ldots, \beta_n; y_1, \ldots, y_m)F$, that is, it may only have type and term variables as parameters. It also defines constructors $h_1, \ldots, h_k$ derived from inl, inr, and fold. We translate type abbreviations and their constructors as follows:

Algebraic Types:

type $(\alpha_1, \ldots, \alpha_n; x_1 : T_1, \ldots, x_m : T_m)F = \Sigma$ defines:

$(T_1, \ldots, T_n; M_1, \ldots, M_m)F \triangleq$
  $T\{T_1/\alpha_1; \ldots; T_n/\alpha_n; M_1/x_1; \ldots; M_m/x_m\}$
  when $\Sigma = T$.
$(T_1, \ldots, T_n; M_1, \ldots, M_m)F \triangleq$ (non-recursive case)
  $Sum(U_1, \ldots, U_k)$
  when $\Sigma = (|h_i\ \textbf{of}\ U_i)_{i \in 1..k}$
  and F does not occur in $U_1, \ldots, U_k$.
$(T_1, \ldots, T_n; M_1, \ldots, M_m)F \triangleq$ (recursive case)
  $\mu\beta.(Sum(U_1, \ldots, U_k)\{\beta/(\alpha_1, \ldots, \alpha_n; x_1, \ldots, x_m)F\})$
  when $\Sigma = (|h_i\ \textbf{of}\ U_i)_{i \in 1..k}$
  and F occurs in at least one of $U_1, \ldots, U_k$.

$Sum(U_1) \triangleq U_1$
$Sum(U_1, U_2, \ldots, U_k) \triangleq U_1 + Sum(U_2, \ldots, U_k)$

Constructors for non-recursive sum types are defined as:
$h_i\ M \triangleq [(\mathsf{inr}]^{i-1}\ (\mathsf{inl}\ M)) \ldots)$ $i = 1..k-1$
$h_k\ M \triangleq [(\mathsf{inr}]^{k-1}\ M) \ldots)$

Constructors for recursive sum types are defined as:
$h_i\ M \triangleq \mathsf{fold}([(\mathsf{inr}]^{i-1}\ (\mathsf{inl}\ M)) \ldots)$ $i = 1..k-1$
$h_k\ M \triangleq \mathsf{fold}([(\mathsf{inr}]^{k-1}\ M) \ldots)$

We may also write h_i instead of both h_i **of** unit and $h_i()$ when $U_i = \mathsf{unit}$. With this encoding, we can define primitive types such as Booleans, integers, strings, lists, and options:

type bool = **false** | **true**
type int = Zero | Succ **of** int
type string = Str **of** int list
type α list = op_Nil | op_ColonColon **of** $\alpha * \alpha$ list
type α option = None | Some **of** α

We treat constants, such as strings and integers, as syntactic sugar for applications of these constructors. The F# operators op_Nil and op_ColonColon stand for the list constructors [] and ::.

As an example, the type bool above defines a disjoint sum type; we then derive conditional branching in terms of constructor matching:

Booleans and Conditional Branching:

$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$
false $\triangleq$ inl ()
true $\triangleq$ inr ()
if A **then** B **else** $B' \triangleq$
  match A **with true** $\to B$ **else match** A **with false** $\to B'$

General pattern matching is derived using nested constructor matching and **let**-expressions:

Pattern Matching:

match A **with** $(h_i\ x_i \to A_i)_{i \in 1..k} \triangleq$
    match A **with** $h_1\ x_1 \to A_1$
    else match A **with** $(h_j\ x_j \to A_j)_{j \in 2..k}$
match A **with** $h\ x \to B \triangleq$ **match** A **with** $h\ x \to B$
    else failwith "match failed"
match A **with** $h\ N \to B$ **else** $B' \triangleq$ **match** A **with**
    $h\ x \to$ **match** x **with**
      $N \to B$
      else B'
    else B'
match A **with** $x \to B$ **else** $B' \triangleq$ **let** $x = A$ **in** B
match A **with** $(x, y) \to B$ **else** $B' \triangleq$ **let** $(x, y) = A$ **in** B

Via a standard encoding [Gunter, 1992, p241], we obtain a fixpoint operator using iso-recursive types as follows.

```
type (α,β)Fix = Fold of ((α,β)Fix → (α → β))
let unfold (Fold (f)) = f
let fix : ((α → β) → (α → β)) → (α → β) =
  begin
  (fun f →
    (fun x → fun y → f (unfold x x) y)
    (Fold (fun x → fun y → f (unfold x x) y)))
  end
let add: int→int→int = fix (fun add → fun x → fun y → if x=0
     then y else add (x−1) (y+1))
```

Using this fixpoint operator, we can encode recursive functions at any function type:

Functions and Recursion:

$$\textbf{let } f\, x_1 \ldots x_n = A \textbf{ in } B \triangleq \textbf{let } f = (\textbf{fun } x_1 \to \ldots \textbf{fun } x_n \to A) \textbf{ in } B$$
$$\textbf{let rec } f = A \triangleq \textbf{let } f = \mathsf{fix}\ (\textbf{fun } f \to A)$$

Finally, we define our primitive functions for communication and concurrency in terms of expressions in our core language. (We introduced some of these functions in Section 2.)

Functions for Communication and Concurrency:

$\mathsf{failwith} \triangleq \textbf{fun}\, x \to (\nu a)a?$	block on failure
$\mathsf{op\_Equals} \triangleq \textbf{fun}\, x\ y \to x = y$	equality function
$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$	
$(T)\textbf{ref} \triangleq (T)\mathsf{chan}$	
$\mathsf{chan} \triangleq \textbf{fun}\, x \to (\nu a)(\textbf{fun}\, x \to a!x, \textbf{fun}\ \_ \to a?)$	
$\mathsf{send} \triangleq \textbf{fun}\, c\ x \to \textbf{let}\ (s,r) = c \textbf{ in } s\ x$	send x on c
$\mathsf{recv} \triangleq \textbf{fun}\, c \to \textbf{let}\ (s,r) = c \textbf{ in } r\ ()$	block for x on c
$\mathsf{fork} \triangleq \textbf{fun}\, f \to (f() \upharpoonright ())$	run f in parallel
$\textbf{ref} \triangleq \textbf{fun}\, x \to \textbf{let}\ r = \mathsf{chan}\ \texttt{"r"} \textbf{ in } \mathsf{send}\ r\ x; r$	new reference
$! \triangleq \textbf{fun}\, r \to \textbf{let}\ x = \mathsf{recv}\ r \textbf{ in } \mathsf{send}\ r\ x; x$	dereference r
$:= \triangleq \textbf{fun}\, r\ y \to \textbf{let}\ x = \mathsf{recv}\ r \textbf{ in } \mathsf{send}\ r\ y$	update r with y

We derive the following types for these expressions:

- failwith can be given type $T \to U$ for any T, U (using (Val Fun), (Exp Res), and (Exp Recv));
- the F# equality operator op_Equals (also written =) can be given type $T \to U \to$ bool for any T, U (using (Val Fun) and (Exp Eq));
- chan can be given type $T \to (U)$chan for any T, U (using (Val Fun) and (Exp Res));
- send can be given type (T)chan $\to T \to$ unit for any T (using (Val Fun), (Exp Fork), (Exp Send), and (Val Unit));
- recv can be given type (T)chan $\to T$ for any T (using (Val Fun), (Exp Let), (Exp Recv), and (Val Var));
- fork can be given type (unit $\to T$) $\to$ unit for any T (using (Val Fun), (Exp Fork), (Exp Appl), and (Val Unit)).
- **ref** can be given type $T \to (U)$**ref** for any T, U (using (Val Fun), (Exp Res),(Exp Fork), (Exp Send), and (Val Unit));
- ! can be given type (T)**ref** $\to T$ for any T (using (Val Fun), (Exp Let), (Exp Recv), and (Val Var));
- := can be given type (T)**ref** $\to T \to$ unit for any T (using (Val Fun), (Exp Let), (Exp Recv), (Val Var), (Exp Fork), (Exp Send), and (Val Unit)).

Hence, we have the following polymorphic types for these functions.

```
val failwith : string → (α){false}
val op_Equals : x:α → y:β → (z:bool){z = True ⇒ x = y}
val fork : unit → unit → unit
val chan : string → α chan
val send : α chan → α → unit
val recv : α chan → α
val ref : α → α ref
val !: α ref → α
val := : α ref → α → unit
```

E Typed Encoding of Formal Cryptography

We provide the complete interface and implementation for formal cryptography within RCF.

An RCF Interface for Formal Cryptography

```
module Crypto
open PrimCrypto
open Pi

type str =
    Literal of string
  | Guid of Pi.name
type bytes =
    Concat of bytes ∗ bytes
  | Nonce of Pi.name
val str: s:string → (x:str){x = Literal(s)}
val istr: x:str → (s:string){x = Literal(s)}
val concat: x1:bytes → x2:bytes → c:bytes{c = Concat(x1,x2
    )}
val iconcat: c:bytes → (x1:bytes ∗ x2:bytes){c = Concat(x1,
    x2)}
val mkGuid: unit → str
val mkPassword: unit → str
val mkPasswordPrin: string → str
val mkNonce: unit → bytes
```

```
type α pickled = P of α
val pickle: x:α → (p:α pickled)
val unpickle: p:α pickled → (x:α)

val tup31: α * β * γ → α
val tup32: α * β * γ → β
val tup33: α * β * γ → γ

(* The following design uses Seals rather
    than Tables *)
type α hkey = HK of α pickled Seal
type hmac = HMAC of Un
val mkHKey: unit → α hkey
val hmacsha1: k:α hkey → x:α pickled → h:hmac
val hmacsha1Verify: k:α hkey → xx:Un → h:hmac → x:α
    pickled

type α symkey = Sym of α pickled Seal
type enc = AES of Un
val mkEncKey: unit → α symkey
val aesEncrypt: k:α symkey → x:α pickled → e:enc
val aesDecrypt: k:α symkey → e:enc → x:α pickled

type α sigkey = SK of α pickled Seal
type α verifkey = VK of (Un → α pickled)
type dsig = RSASHA1 of Un
val rsasha1: k:α sigkey → x:α pickled → d:dsig
val rsasha1Verify: k:α verifkey → xx:Un → d:dsig → x:α
    pickled

type β deckey = DK of β symkey Seal
type β enckey = EK of (β symkey → x:Un)
type penc = RSA of Un
val rsaEncrypt: β enckey → β symkey → penc
val rsaDecrypt: β deckey → penc → β symkey

type (α,β) privkey = Priv of α sigkey * β deckey (* One
    for signing, one for encryption *)
type (α,β) pubkey = Pub of α verifkey * β enckey
val rsaKeyGen: unit → (α,β) privkey
val rsaPub: (α,β) privkey → (α,β) pubkey
val sigkey: (α,β) privkey → α sigkey
val deckey: (α,β) privkey → β deckey
val verifkey: (α,β) pubkey → α verifkey
val enckey: (α,β) pubkey → β enckey

type hash = SHA1 of Un
type α hasher = Sha1 of α pickled Seal

val mkSha1: unit → α hasher
val sha1: α hasher → α pickled → hash
```

Generated $F^{\#}$ Interface

```
module Crypto

open PrimCrypto
open Pi
type str
type bytes
val str : (string → str)
val istr : (str → string)
val concat : (bytes → (bytes → bytes))
val iconcat : (bytes → (bytes * bytes))
val mkGuid : (unit → str)
val mkPassword : (unit → str)
val mkPasswordPrin : (string → str)
val mkNonce : (unit → bytes)
type α pickled
val pickle : (α → α pickled)
val unpickle : (α pickled → α)
val tup31 : ((α * β * γ) → α)
val tup32 : ((α * β * γ) → β)
val tup33 : ((α * β * γ) → γ)
val namegen : (string → (unit → Un))
type α hkey
type hmac
type α hmacpred
val mkHKey : (unit → α hkey)
val hmacsha1 : (α hkey → (α pickled → hmac))
val hmacsha1Verify : (α hkey → (Un → (hmac → α pickled)))
type α symkey
type enc
type α encpred
val mkEncKey : (unit → α symkey)
val aesEncrypt : (α symkey → (α pickled → enc))
val aesDecrypt : (α symkey → (enc → α pickled))
type α sigkey
type α verifkey
type dsig
type α dsigpred
val rsasha1 : (α sigkey → (α pickled → dsig))
val rsasha1Verify : (α verifkey → (Un → (dsig → α pickled)))
type β deckey
type β enckey
type penc
type α pencpred
val rsaEncrypt : (β enckey → (β symkey → penc))
val rsaDecrypt : (β deckey → (penc → β symkey))
type (α,β) privkey
type (α,β) pubkey
val rsaKeyGen : (unit → (α, β) privkey)
val rsaPub : ((α, β) privkey → (α, β) pubkey)
val sigkey : ((α, β) privkey → α sigkey)
val deckey : ((α, β) privkey → β deckey)
val verifkey : ((α, β) pubkey → α verifkey)
val enckey : ((α, β) pubkey → β enckey)
type hash
type α hasher
val mkSha1 : (unit → α hasher)
val sha1 : (α hasher → (α pickled → hash))
type α deriver
val mkPSHA1 : (α symkey → α deriver)
val psha1 : (α deriver → (bytes → α symkey))
```

An Implementation of Formal Cryptography

```
#light "off"
module Crypto
open Pi
open PrimCrypto

type str =
      Literal of string
    | Guid of Pi.name
type bytes =
      Concat of bytes * bytes
    | Nonce of Pi.name
let str s = Literal s
let istr s = match s with
| Literal v → v
| _ → failwith "iS failed"
let concat x y = Concat(x,y)
let iconcat s = match s with
    | Concat(x,y) → (x,y)
    | _ → failwith "iconcat failed"
let mkGuid () = Guid (Pi.name "id")
let mkNonce () = Nonce (Pi.name "nonce")
let mkPassword () = Guid (Pi.name "pwd")
let mkPasswordPrin (p:string) = Guid (Pi.name p)

type pickled<α> = P of α
let pickle (x:α) = P x
let unpickle (P x) = x

let tup31 (a,b,c) = a
let tup32 (a,b,c) = b
let tup33 (a,b,c) = c

let fail: unit → α = fun () → failwith "Not Found"

type α hkey = HK of α pickled Seal
type hmac = HMAC of Un

let mkHKey ():α hkey =
   let s = mkSeal "hkey" in
      HK s
let hmacsha1 (HK(key)) text =
   let (h,_) = key in
   let t = h text in
      HMAC (t)
let hmacsha1Verify (HK key) text (HMAC h) =
   let (_,hv) = key in
   let x:α pickled = hv h in
      if x = text then x else failwith "hmac verify failed"

type α symkey = Sym of α pickled Seal

type enc = AES of Un
let mkEncKey () :α symkey =
   let t = mkSeal "symkey" in
      Sym (t)

let aesEncrypt (Sym key) (text:α pickled) : enc =
   let (e,d) = key in
   let x = e text in
      AES(x)
let aesDecrypt (Sym key) (AES msg) : α pickled =
   let (e,d) = key in
      d msg

type α sigkey = SK of α pickled Seal
type α verifkey = VK of (Un → α pickled)
type dsig = RSASHA1 of Un
let rsasha1 (SK(sk)) t =
   let (s,v) = sk in
   let ss = s t in
      RSASHA1(ss)
let rsasha1Verify (VK(v)) t (RSASHA1 sg) =
   let x:α pickled = v sg in
   if x = t then x else failwith "rsasha1 verify failed"

type β deckey = DK of β symkey Seal
type β enckey = EK of (β symkey → Un)
type penc = RSA of Un

let mkRsaDecKey () : β deckey =
   let s = mkSeal "rsakey" in
      DK(s)
let rsaEncKey (DK dk) =
   let (e,d) = dk in EK(e)
let rsaEncrypt (EK (e)) t = RSA(e t)
let rsaDecrypt (DK k) (RSA msg) =
   let (e,d) = k in d msg

(* Private/Public keypairs, for signing and
     encryption *)
type (α,β) privkey = Priv of α sigkey * β deckey
type (α,β) pubkey = Pub of α verifkey * β enckey
let rsaKeyGen () :(α,β) privkey =
   let t1 = mkSeal "sigkey" in
   let t2 = mkSeal "deckey" in
      Priv(SK(t1),DK(t2))
let rsaPub (p:(α,β) privkey) :(α,β) pubkey =
   let (Priv (s,d)) = p in
   let (SK(sk)) = s in
   let (DK(dk)) = d in
   let (s,v) = sk in
   let (e,d) = dk in
   let vk = VK(v) in
   let ek = EK(e) in
   let p = Pub(vk,ek) in
      p
let sigkey (Priv (s,d)) = s
let deckey (Priv (s,d)) = d
let verifkey (Pub (v,e)) = v
let enckey (Pub (v,e)) = e

type hash = SHA1 of Un
type α hasher = Sha1 of α pickled Seal

let mkSha1 () =
  let t = mkSeal "sha1" in
    Sha1 t

let sha1 (Sha1(t)) x =
  let (h,_) = t in
    SHA1(h x)
```

F Example Code

We provide the complete interface and implementation code for the final MAC-based authentication protocol of Section 3.

Refinement-Typed Interface

```
module M
open Pi
open Crypto
open Net

type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type (;a:prin,b:prin) content = x:string{ Send(a,b,x) }

type message = (prin ∗ prin ∗ string ∗ hmac) pickled

private val mkContentKey:
  a:prin → b:prin → ((;a,b)content) hkey
private val hkDb:
  (prin∗prin, a:prin ∗ b:prin ∗ k:(;a,b) content hkey) Db.t
val genKey: prin → prin → unit
private val getKey: a:
  string → b:string → ((;a,b) content) hkey

assume ∀a,b,x. ( Leak(a) ) ⇒ Send(a,b,x)
val leak:
  a:prin → b:prin → (unit{ Leak(a) }) ∗ ((;a,b) content) hkey

val addr : (prin ∗ prin ∗ string ∗ hmac, unit) addr
private val check:
  b:prin → message → (a:prin ∗ (;a,b) content)
val server: string → unit

private val make:
  a:prin → b:prin → (;a,b) content → message
val client: prin → prin → string → unit
```

F# Implementation Code

```
module M
open Pi
open Crypto // Crypto Library
open Net // Networking Library

// Simple F# types for principals, events, payloads, and messages:
type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type content = string
type message = (prin ∗ prin ∗ string ∗ hmac) pickled

// Key database:
let hkDb : ((prin∗prin),(prin∗prin∗(content hkey))) Db.t =
  Db.create ()
let mkContentKey (a:prin) (b:prin) : content hkey =
  mkHKey()
let genKey a b =
  let k = mkContentKey a b in
  Db.insert hkDb (a,b) (a,b,k)
let getKey a b =
  let a',b',sk = Db.select hkDb (a,b) in
  if (a',b') = (a,b) then sk else failwith "select didn't
      find appropriate item"

// Key compromise:
let leak a b =
  assume (Leak(a)); ((),getKey a b)

// Server code:
let addr : (prin ∗ prin ∗ string ∗ hmac, unit) addr =
  http "http://localhost:7000/pwdmac" ""
let check b m =
  let a,b',text,h = unpickle m in
  if b = b' then
    let k = getKey a b in
    (a,unpickle (hmacsha1Verify k (pickle text) h))
  else failwith "Not the intended recipient"
let server b =
  let c = listen addr in
  let (a,text) = check b (recv c) in
    assert(Send(a,b,text))

// Client code:
let make a b s =
  pickle (a,b,s,hmacsha1 (getKey a b) (pickle s))
let client a b text =
  assume (Send(a,b,text));
  let c = connect addr in
  send c (make a b text)

// Execute one instance of the protocol:
let _ = genKey "A" "B"
let _ = fork (fun (u:unit) → client "A" "B" "Hello")
let _ = server "B"
```

Generated F# Interface

```
module M

open Pi
open Crypto
open Net
type prin = string
type event
type content = string
type message = ((prin ∗ prin ∗ string ∗ hmac)) pickled
```

```
val genKey : (prin → (prin → unit))
val leak : (prin → (prin → (unit ∗ content hkey)))
val addr : ((prin ∗ prin ∗ string ∗ hmac), unit) addr
val server : (string → unit)
val client : (prin → (prin → (string → unit)))
```

Typechecking We invoke our typechecker on the example above, along with the interfaces and implementations it depends upon, including our encoding of formal cryptography and symbolic implementations of the trusted libraries. More precisely, the typechecker is given a *program* consisting of:

- Pi: a typed interface to functions for communication and concurrency;
- PrimCrypto: the interface and encoding of seals;
- Crypto, Net: interfaces and symbolic implementations of trusted libraries (see Appendix E; Section 5);
- M: the interface and implementation of the example above.

The type definitions in implementations and interfaces define abbreviations that are eliminated by inlining. Then the interfaces of a program are interpreted as a type T and a set of formulas S_C:

$$T = (e_1 : T_1 * \cdots * e_l : T_l)$$
$$S_C = \{C_1, \ldots, C_m\}$$

where $e_1, \ldots, e_l$ are all the values exported by the interfaces PrimCrypto, Crypto, Net, and M, and $C_1, \ldots, C_n$ are all the formulas assumed in the interfaces.

The program is interpreted as an expression A that assumes the formulas S_C and defines the values $e_1, \ldots, e_l$. By typechecking, we establish that A has public type T ($\varnothing \vdash A : T$ and $C_1, \ldots, C_m \vdash T <:$ Un); hence, by Theorem 2 (Robust Safety), A is robustly safe.

More precisely, the program is then interpreted as an expression A of the form:

```
A =
  let failwith = fun x → (νa)a? in
  ...
  let recv = fun c → let x = c? in x in
   assume C_1;
   ...
   assume C_m;
   let y_1 = B_1 in
   ...
   let y_k = B_k in
    (e_1,...,e_l)
```

where failwith, ..., recv are all the functions defined in Pi; $y_1, \ldots, y_k$ are the values defined in PrimCrypto, Crypto, Net, and M as expressions $B_1, \ldots, B_k$ (we expect that $\{e_1, \ldots, e_l\} \subseteq \{y_1, \ldots, y_m\}$).

To prove that the program is robustly safe, we apply Theorem 2 (Robust Safety), by proving $\varnothing \vdash A :$ Un.

We first use our typechecker to establish:

```
failwith : string→ unit,
...,
recv: α chan → α,
_:{C_1},...,_:{C_m} ⊢
  let y_1 = B_1 in
  ...
  let y_k = B_k in
   (e_1,...,e_l)
  :
  T
```

and to also check that $T <:$ Un.

We then establish, by hand, that each function in Pi has the types stated above; we then obtain $\varnothing \vdash A : T$ by several applications of (Exp Assume) followed by (Exp Let). From $\varnothing \vdash A : T$ and $T <:$ Un, we apply (Exp Subsum) to obtain $\varnothing \vdash A :$ Un.

The full result printed by the typechecker is as follows:

Given Type Declarations:

```
type int = Zero of unit | Succ of int
type α list = op_Nil of unit | op_ColonColon of (α ∗ α list)
type string = Str of int list
type α ref = Ref of α
type tup0 = Tup0 of unit
type α tup1 = Tup1 of α
type (α,β) tup2 = Tup2 of (α ∗ β)
type (α,β,γ) tup3 = Tup3 of (α ∗ β ∗ γ)
type (α,β,γ,δ) tup4 = Tup4 of (α ∗ β ∗ γ∗ δ)
type (α,β,γ,δ,ε) tup5 = Tup5 of (α ∗ β ∗ γ∗ δ ∗ ε)
type (α,β,γ,δ,ε,φ) tup6 = Tup6 of (α ∗ β ∗ γ∗ δ ∗ ε ∗ φ)
type bool = True of unit | False of unit
type α option = None of unit | Some of α
type name
type α chan
type Un = name
type α PrimCrypto.Seal = (α → Un ∗ Un → α)
type α PrimCrypto.SealChan = (α ∗ Un) list chan
type α PrimCrypto.Key = α PrimCrypto.Seal
type α PrimCrypto.HK = α PrimCrypto.Seal
type α PrimCrypto.SK = α PrimCrypto.Seal
type α PrimCrypto.VK = Un → α
type α PrimCrypto.DK = α PrimCrypto.Seal
type α PrimCrypto.EK = α → Un
type ('k,'v) Db.t = Db.Db of ('k ∗ 'v) chan
type α List.m = List.Mem of (α ∗ α list)
type Crypto.str = Crypto.Literal of string | Crypto.Guid of
      name
type Crypto.bytes = Crypto.Concat of (Crypto.bytes ∗
      Crypto.bytes) | Crypto.Nonce of name
type Crypto.nonce = Crypto.bytes
type α Crypto.pickled = Crypto.P of α
type α Crypto.symkey = Crypto.Sym of α Crypto.pickled
      PrimCrypto.Key
type α Crypto.privkey = Crypto.Priv of α Crypto.pickled
      PrimCrypto.SK
type α Crypto.pubkey = Crypto.Pub of α Crypto.pickled
      PrimCrypto.VK
type Crypto.dsig = Crypto.RSASHA1 of Un
type Crypto.enc = Crypto.AES of Un
type α Crypto.hkey = α Crypto.pickled PrimCrypto.HK
type Crypto.hmac = Crypto.HMAC of Un
type (α,β) Net.addr = Net.Ch of (string ∗ (α Crypto.pickled
      chan ∗ β Crypto.pickled chan) chan)
type (α,β) Net.conn = Net.Conn of (α Crypto.pickled chan ∗
      β Crypto.pickled chan)
type M.prin = string
type M.event = M.Send of (M.prin ∗ M.prin ∗ string) | M.Leak
      of M.prin
type (a:M.prin,b:M.prin) M.content = (x:string){M.Send(a, b,
      x)}
type M.message = (M.prin ∗ M.prin ∗ string ∗ Crypto.hmac)
      Crypto.pickled
```

Assuming Value Declarations:

```
val failwith : string → (φ f){false}
val op_Equals : x:α → y:β → (z:bool){z = True ⇒ x = y}
val fork : unit → unit → unit
val chan : string → α chan
val send : α chan → α → unit
val recv : α chan → α
```

Assuming Formulae:

```
assume (∀x. (∀u. List.Mem(x, op_ColonColon (x, u)))) ∧ (∀x. (
      ∀y. (∀u. List.Mem(x, u) ⇒ List.Mem(x, op_ColonColon (
      y, u))))) ∧ (∀x. (∀u. List.Mem(x, u) ⇒ (∃y. (∃v. u =
      op_ColonColon (y, v) ∧ x = y ∨ List.Mem(x, v)))))
assume (∀a. (∀b. (∀x. M.Leak(a) ⇒ M.Send(a, b, x))))
```

Typechecking succeeds for:

```
val PrimCrypto.mkSeal : unit → Un PrimCrypto.Seal
val PrimCrypto.mkKey : unit → Un PrimCrypto.Key
val PrimCrypto.senc : Un PrimCrypto.Key → Un → Un
val PrimCrypto.sdec : Un PrimCrypto.Key → Un → Un
val PrimCrypto.mkHK : unit → Un PrimCrypto.HK
val PrimCrypto.khash : Un PrimCrypto.HK → Un → Un
val PrimCrypto.khashVerify : Un PrimCrypto.HK → Un → Un
val PrimCrypto.mkSK : unit → Un PrimCrypto.SK
val PrimCrypto.vk : Un PrimCrypto.SK → Un PrimCrypto.VK
val PrimCrypto.sign : Un PrimCrypto.SK → Un → Un
val PrimCrypto.verify : Un PrimCrypto.VK → Un → Un
val PrimCrypto.mkDK : unit → Un PrimCrypto.DK
val PrimCrypto.ek : Un PrimCrypto.DK → Un PrimCrypto.
      EK
val PrimCrypto.penc : Un PrimCrypto.EK → Un → Un
val PrimCrypto.pdec : Un PrimCrypto.DK → Un → Un
val Db.create : unit → (Un, Un) Db.t
val Db.select : (Un, Un) Db.t → Un → Un
val Db.find : (Un, Un) Db.t → Un → Un option
val Db.insert : (Un, Un) Db.t → Un → Un → unit
val List.mem : x:Un → u:Un list → (r:bool){r = True ⇒ List.
      Mem(x, u)}
val List.find : Un → bool → u:Un list → (r:Un){List.Mem(r, u)}
val List.first : Un → Un option → Un list → Un option
val List.left : Un → (Un ∗ Un) → Un option
val List.right : Un → (Un ∗ Un) → Un option
val Crypto.str : s:string → (x:Crypto.str){x = Crypto.Literal (s
      )}
val Crypto.istr : x:Crypto.str → (s:string){x = Crypto.Literal (s
      )}
val Crypto.concat : x1:Crypto.bytes → x2:Crypto.bytes → (c:
      Crypto.bytes){c = Crypto.Concat (x1, x2)}
val Crypto.iconcat : c:Crypto.bytes → (x1:Crypto.bytes ∗ x2:
      Crypto.bytes){c = Crypto.Concat (x1, x2)}
val Crypto.mkGuid : unit → Crypto.str
val Crypto.mkNonce : unit → Crypto.bytes
val Crypto.mkPassword : unit → Crypto.str
val Crypto.mkPasswordPrin : string → Crypto.str
val Crypto.mkEncKey : unit → Un Crypto.symkey
val Crypto.check : x:Un → y:Un → (unit){x = y}
val Crypto.mkHKey : unit → Un Crypto.hkey
val Crypto.hmacsha1 : Un Crypto.hkey → Un Crypto.pickled
       → Crypto.hmac
val Crypto.hmacsha1Verify : Un Crypto.hkey → Un Crypto.
      pickled → Crypto.hmac → Un Crypto.pickled
val Crypto.aesEncrypt : Un Crypto.symkey → Un Crypto.
      pickled → Crypto.enc
val Crypto.aesDecrypt : Un Crypto.symkey → Crypto.enc →
      Un Crypto.pickled
val Crypto.rsaKeyGen : unit → Un Crypto.privkey
val Crypto.rsaPub : Un Crypto.privkey → Un Crypto.pubkey
val Crypto.pickle : Un → Un Crypto.pickled
val Crypto.unpickle : Un Crypto.pickled → Un
val Crypto.rsasha1 : Un Crypto.privkey → Un Crypto.pickled
       → Crypto.dsig
val Crypto.rsasha1Verify : Un Crypto.pubkey → Un Crypto.
      pickled → Crypto.dsig → Un Crypto.pickled
val Net.http : string → (Un, Un) Net.addr
val Net.connect : (Un, Un) Net.addr → (Un, Un) Net.conn
val Net.listen : (Un, Un) Net.addr → (Un, Un) Net.conn
val Net.close : (Un, Un) Net.conn → unit
val Net.send : (Un, Un) Net.conn → Un Crypto.pickled →
      unit
val Net.recv : (Un, Un) Net.conn → Un Crypto.pickled
val M.genKey : M.prin → M.prin → unit
val M.leak : a:M.prin → b:M.prin → (unit{M.Leak(a)} ∗ (;a, b)
       M.content Crypto.hkey)
val M.addr : ((M.prin ∗ M.prin ∗ string ∗ Crypto.hmac), unit)
      Net.addr
val M.server : string → unit
val M.client : M.prin → M.prin → string → unit
```

References

M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.

M. Abadi. Access control in a core calculus of dependency. In *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, pages 5–31. Elsevier, 2007. Volume 172 of ENTCS.

M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1): 102–146, 2005.

M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Symposium (NDSS'03)*. Internet Society, February 2003.

M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *LNCS*, pages 197–221. Springer, 2005.

A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 353–369. Springer, 2006.

D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1–2):273–309, 2001.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 3–17. ACM, 2008.

M. Backes, M. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 308–323. IEEE Computer Society, 2009.

M. Backes, C. Hritcu, and M. Maffei. Union and intersection types for secure protocol implementations. Unpublished draft, 2010a.

M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 387–398. ACM Press, 2010b.

I. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2009)*, pages 27–38, 2009.

M. Barnett, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'05*, volume 3362 of *LNCS*, pages 49–69. Springer, January 2005.

M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM Conference on Computer and Communications Security*, pages 459–468, 2008a.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31:5:1–5:61, December 2008b.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 124–140, July 2009.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456. ACM, 2010a.

K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *Asian Symposium on Programming Languages and Systems (APLAS'10)*, pages 47–62, 2010b.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society, 2006.

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. In *Journal of Functional Programming*. Cambridge University Press, September 2010.

L. Cardelli. Typechecking dependent types and subtypes. In *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 45–57. Springer, 1986.

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium*, pages 172–185, 2009.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation for end-to-end verification of security enforcement. In *Programming Language Design and Implementation (PLDI'10)*, pages 412–423. ACM, June 2010.

A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Do As I SaY! Programmatic access control with explicit identities. In *IEEE Computer Security Foundations Symposium (CSF'07)*, pages 16–30, 2007.

D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS'05*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.

R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS. Springer-Verlag, 2006.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). In *Electronic Notes in Theoretical Computer Science (Gordon D. Plotkin Festschrift)*, volume 172, pages 311–358, 2007.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy*, 1996.

D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29 (2):198–208, 1983.

M. A. E. Dummett. *Elements of intuitionism*. Clarendon Press, 1977.

N. Durgin, J. C. Mitchell, and D. Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security (Special Issue of Selected Papers from CSFW-14)*, 11(4):677–721, 2003.

D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation, at `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`.

J. Filliâtre and C. Marché. Multi-prover Verification of C Programs. In *International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002.

C. Fournet. On the computational soundness of cryptographic verification by typing. In *Workshop on Formal and Computational Cryptography (FCC 2009)*, 2009.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335, Jan. 2008.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007a. Article 25.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies in distributed systems. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 31–45, 2007b.

T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM, 1991.

A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications. Proceedings, 1993*, number 780 in LNCS, pages 414–426. Springer, 1994.

A. D. Gordon and C. Fournet. Principles and applications of refinement types. In J. Esparza, B. Spanfelner, and O. Grumberg, editors, *Logics and Languages for Reliability and Security: Proceedings of the NATO Summer School Marktoberdorf 2009*, pages 73–104. IOS Press, 2010. A preliminary version appears as Technical Report MSR-TR-2009-147, Microsoft Research, October 2009.

A. D. Gordon and A. S. A. Jeffrey. Cryptyc: Cryptographic protocol type checker. Software available at `http://cryptyc.cs.depaul.edu/`, 2002.

A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11 (4):451–521, 2003a.

A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2003b.

A. D. Gordon and A. S. A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR 2005—Concurrency Theory*, volume 3653 of *LNCS*, pages 186–201. Springer, 2005.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, volume 3385 of *LNCS*, pages 363–379, 2005.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In R. Findler, editor, *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

C. Gunter. *Semantics of programming languages*. MIT Press, 1992.

N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability by typing. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 168–183. Springer, 2009.

E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing*, pages 213–226, 2003.

R. Jagadeesan, A. S. A. Jeffrey, C. Pitcher, and J. Riely. Lambda-RBAC: Programming with role-based access control. *Logical Methods In Computer Science*, 4(1), 2008.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *International Conference on Functional Programming (ICFP'08)*, pages 27–38. ACM, 2008.

M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *Programming Language Design and Implementation (PLDI'09)*, pages 304–315. ACM, 2009.

P. Li and S. Zdancewic. Encoding information flow in Haskell. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 16–27, 2006.

S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon. Code-carrying authorization. In *13th European Symposium on Research in Computer Security (ESORICS'08)*, volume 5283 of *LNCS*, pages 563–579. Springer, Oct. 2008.

P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999.

A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, Mar. 2004. At `http://www.oasis-open.org/committees/download.php/5941/oasis-200401-wss-soap-message-security-1.0.pdf`.

R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

C. Parent. Synthesizing Proofs from Programs in the Calculus of Inductive Constructions. *Mathematics of Program Construction (MPC'95)*, 947:351–379, 1995.

L. C. Paulson. *Logic and computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1991.

B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS'07*, pages 164–177, 2007.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

F. Pottier, C. Skalka, and S. Smith. A systematic approach to access control. In *Programming Languages and Systems (ESOP 2001)*, volume 2028 of *LNCS*, pages 30–45. Springer, 2001.

Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction*, volume 5133 of *LNCS*, pages 305–335. Springer, 2008.

P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*, pages 159–169. ACM, 2008.

P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 131–144. ACM, 2010.

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.

E. Sumii and B. Pierce. A bisimulation for dynamic sealing. *Theor. Comput. Sci.*, 375(1–3):169–192, 2007.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 96–110, 2008.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *19th European Symposium on Programming (ESOP'10)*, pages 529–549, 2010.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

J. A. Vaughan and S. Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206, Washington, DC, USA, 2007.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-Based Audit. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 177–191, 2008.

T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM, 1999.

D. N. Xu. Extended static checking for Haskell. In *ACM SIGPLAN workshop on Haskell (Haskell'06)*, pages 48–59. ACM, 2006.