Typechecking Higher-Order Security Libraries

Karthik Bhargavan\textsuperscript{3,1}, Cédric Fournet\textsuperscript{2,1}, and Nataliya Guts\textsuperscript{1}

\textsuperscript{1} MSR-INRIA Joint Centre
\textsuperscript{2} Microsoft Research
\textsuperscript{3} INRIA

Abstract. We propose a flexible method for verifying the security of ML programs that use cryptography and recursive data structures. Our main applications are X.509 certificate chains, secure logs for multi-party games, and XML digital signatures. These applications are beyond the reach of automated cryptographic verifiers such as ProVerif, since they require some form of induction. They can be verified using refinement types (that is, types with embedded logical formulas, tracking security events). However, this entails replicating higher-order library functions and annotating each instance with its own logical pre- and post-conditions. Instead, we equip higher-order functions with precise, yet reusable types that can refer to the pre- and post-conditions of their functional arguments, using generic logical predicates. We implement our method by extending the F7 typechecker with automated support for these predicates. We evaluate our approach experimentally by verifying a series of security libraries and protocols.

1 Security Verification by Typing

We intend to verify the security of programs that implement protocols and applications (rather than their abstract models). Operating at the level of source code ensures that both design and implementation flaws will be caught, and also facilitates the adoption of verification tools by programmers. In this work, we rely on F7 [Bengtson et al., 2008, Bhargavan et al., 2010], an SMT-based typechecker developed for the modular verification of security protocols and their cryptographic operations written in ML.

Suppose that Bob hosts a web application and Alice is one of his clients. Alice sends a request to Bob, who must authenticate Alice’s request before delivering a response. Bob programs in ML, so he can use the F7 typechecker to validate that his code enforces his security policy. Depending on the control- and data-flow of the protocol between Alice and Bob, typechecking essentially checks that the program obeys the logical pre- and post-conditions specified in the interfaces of the protocol and the cryptographic and communications libraries. The programmer provides a few protocol-specific type annotations (for instance when accepting a message or allocating a key). The rest of the verification is automated.

In practice, protocol implementations involve various data structures, and thus the need for type annotations extends to various library functions that manipulate this data. Despite support for polymorphism à la ML, it is difficult to give these library functions precise, yet polymorphic refinement types. In particular, recursive data processing

\textsuperscript{1} Note to the reviewers: we will present a preliminary version of this work at FCS-PrivMod workshop in July 2010; this workshop has no formal proceedings.
involves higher-order functions, and the programmer must often provide a refinement
type each time they use these functions. Pragmatically, this involves replicating the code
for these functions (and some of the functions they call); annotating each replica with
its ad hoc type; and letting F7 typecheck the replica for each particular usage.

Suppose that the message format used by Alice and Bob is under development and
changes often. Each change trickles down the protocol data flow, causing many changes
to its logical annotations, and possibly further code replication. This hinders code mod-
ularity. Can we write less code and annotations, and focus on the security properties
of our program? In this work we show how using automatic predicates for pre- and
post-conditions allows us to write more flexible and reusable types.

Example  F7 is based on a typed call-by-value lambda calculus, called RCF, described
in more detail in Section 2. Expressions are written in a subset of F#, a dialect of ML.
Types are F# types refined with first-order formulas on ML values. For instance, the
refinement type \( v : \text{int} \{ v > 5 \} \) is the type of integers greater than 5. More precisely, this
type can be given to any expression such that, whenever it returns, its value is greater
than 5. RCF defines judgments for assigning types to expressions and for checking
whether one type is a subtype of another. For instance, \( v : \text{int} \{ v > 5 \} \) is a subtype of \( \text{int} \rightarrow \text{int} \).
Functions can also be given precise refinement types. For instance, the dependent
function type \( v : \text{int} \rightarrow w : \text{int} \{ w > v \} \), a subtype of \( \text{int} \rightarrow \text{int} \), represents functions that, when
called with an integer \( v \), may return only an integer greater than \( v \).

Consider the type \( \alpha \text{ option} \), which is part of the standard ML library. Its instance
\( \text{int} \text{ option} \) is the type of optional integers: its values range over \( \text{None} \) and \( \text{Some } n \), where
\( n \) is an integer. Using option types, we can, for example, program protocols that have
optional fields in their messages. To manipulate a message field of type \( \text{int} \text{ option} \), it is
convenient to use the higher-order library function \( \text{map} \):

\[
\text{val map: (int \rightarrow \text{int}) \rightarrow \text{int \ option} \rightarrow \text{int \ option}}
\]

\[
\text{let map f x \{ \begin{array}{l}
| \text{None} \rightarrow \text{None} \\
| \text{Some}(v) \rightarrow \text{let } w = f v \text{ in Some}(w)
\end{array}}
\]

This function can be applied to any function whose type is a subtype of \( \text{int} \rightarrow \text{int} \), of the
form \( x : \text{int} \rightarrow y : \text{int} \{ C(x, y) \} \) for some formula \( C \) that can refer to both \( x \) and \( y \). Suppose
we compute a value \( y \) using \( \text{map} \) over a function \( f \) with type \( v : \text{int} \rightarrow w : \text{int} \{ w > v \} \):

\[
\text{let } y = \text{map f (Some}(0))
\]

We would like to give \( y \) a type that records the post-condition of \( f \):

\[
\text{val y: int \ option \{ \exists w. y = \text{Some}(w) \wedge w > 0 \}}
\]

What type must \( \text{map} \) have in order for \( y \) to have this type? Within RCF, the most precise
type we can give is

\[
\text{val map: f: (int \rightarrow int) \rightarrow x:(int \ option) \rightarrow y:(int \ option)}}
\]

\[
\{ (x = \text{None} \wedge y = \text{None}) \vee (\exists w. x = \text{Some}(v) \wedge y = \text{Some}(w)) \}
\]

This type accounts for the various cases (\( \text{None} \) vs \( \text{Some} \)) of the argument, but not
for the post-condition of \( f \). In RCF, terms in formulas range over ML values, such as
\( \text{Some}(w) \), but not expressions, such as \( f x \), since their evaluation may cause and depend
on side-effects. Thus, the only way to check that \( y \) has its desired type is to copy the definition of the \( \text{map} \) function just for \( f \) and to annotate and typecheck it again.

\[
\text{val map}_\text{copy}: f:\text{int} \rightarrow \text{w}:\text{int} \{ w > v \} \rightarrow x:\text{int option} \rightarrow y:\text{int option}
\{ (x = \text{None} \land y = \text{None}) \lor (\exists v, w, x = \text{Some}(v) \land y = \text{Some}(w) \land w > v) \}
\]

Our main idea is to let the F7 typechecker automatically inject and check annotations for pre- and post-conditions. This yields precise, generic types for higher-order functions, thereby preventing the need for manual code duplication and annotation. To this end, we introduce predicates \( \text{Pre} \) and \( \text{Post} \) within the types of higher-order functions to refer to the pre- and post-conditions of their functional arguments. For instance, \( \text{Post}(f,v,w) \) can refer to the post-condition of a function parameter \( f \) applied to \( v \) returning \( w \), and we can give \( \text{map} \) the type:

\[
\text{val map}: f:\text{int} \rightarrow x:\text{int option} \rightarrow y:\text{int option}
\{ (x = \text{None} \land y = \text{None}) \lor (\exists v, w, x = \text{Some}(v) \land y = \text{Some}(w) \land \text{Post}(f,v,w)) \}
\]

Whenever \( \text{map} \) is called (say within the definition of \( y \) above), the actual post-condition of \( f \) is statically known \((w > v)\) and can be used instead of \( \text{Post}(f,v,w) \). Hence, \( y \) can be given its desired type without loss of modularity.

We show how to use such \( \text{Pre} \) and \( \text{Post} \) predicates to give precise reusable types to a library of recursive higher-order functions for list processing, and use the library to verify protocol implementations using lists. Verifying such implementations is beyond the reach of typical security verification tools, since their proof requires some form of induction. For example, \( \text{Fs2PV} \) [Bhargavan et al., 2008] compiles F# into the applied pi calculus, for analysis with ProVerif [Blanchet, 2001], a state-of-the-art domain-specific prover. Although \( \text{Fs2PV} \) and ProVerif are able to prove complex XML-based cryptographic protocol code, they do so by limiting the length of lists to some constant value and then inlining and re-verifying the list processing code at each call site.

**Contribution** We present extensions of the RCF type system and the F7 typechecker to automatically support pre- and post-condition predicates. We study three different semantics for these predicates and illustrate their use. We design precise and modular APIs for lists and for several cryptographic protocol implementations using lists, such as X.509 certificates, XML digital signatures, and auditable multi-party protocols.

**Contents** Section 2 recalls the syntax, semantics and main results for F7. Section 3 explains our extension of F7 for pre- and post-conditions, presenting different design choices. Section 4 illustrates the use of pre- and post-conditions to verify a basic authentication protocol. Section 5 illustrates their use to give reusable types to a library for lists. Section 6 describes and evaluates larger verification case studies of cryptographic protocol implementations. Section 7 discusses related work.

This work is part of a long-term effort to develop a refinement type-based security verification framework for F# code. We extend the type system and cryptographic libraries developed in earlier work [Bengtson et al., 2008, Bhargavan et al., 2010]. Additional details, including source code, are available at [http://msr-inria.inria.fr/Projects/sec/infer](http://msr-inria.inria.fr/Projects/sec/infer).
2 Refinement types for ML (review)

We review the syntax and semantics of our core calculus, RCF, and its implementation in the F7 typechecker; we refer to Bengtson et al. [2008] for a detailed description. RCF consists of the standard Fixpoint Calculus [Plotkin, 1985] augmented with local names and message-passing concurrency (as in the pi calculus) and with refinement types. Appendix A shows the full syntax we use. (Our syntax slightly deviate from Bengtson et al.; the main difference is that we have recursive functions, as in F7, instead of a fold constructor; we demand that all function values be given type annotations.)

The source programs described in this paper are written in an extended ML-like syntax that is treated as syntactic sugar for core RCF values and expressions. Values \( M \) include unit, pairs, constructed terms, and (possibly recursive) lambda expressions. Expressions \( e \) are in A-normal form: they include values, function application, pattern matching, let-bindings for sequential composition, fork for parallel composition, and message passing over channels. The concurrency and message passing constructs do not appear in source programs; they are used to symbolically model run-time processes (e.g. the principals running a cryptographic protocol and their adversary) and network-based communications.

For specification purposes, RCF includes constructs for assuming and asserting first-order logic formulas. Formally, as an RCF expression executes, it maintains an abstract log of formulas that have been assumed so far. The expression \( \text{assume} \ C \) adds a formula \( C \) to the log, and the expression \( \text{assert} \ C \) succeeds if \( C \) can be logically derived from the log. We use assumes and asserts to specify correctness and security properties—concretely, these two primitives and all formulas are erased after verification. We say that an expression is safe when all of its asserts succeed in every run.

To statically verify the safety of RCF expressions, we equip it with a refinement type system. Type environments \( E \) keep track of the log of assumed formulas, and typechecking ensures that every asserted formula logically holds in the current environment. (Since asserts may contain quantified formulas that rely on assumes made in concurrent expressions, it is often not possible to dynamically verify the safety of RCF expressions using run-time checks.)

Pretypes \( P \) are ML-like types extended with dependent functions, written \( x : T_1 \rightarrow T_2 \), and dependent pairs. A refinement type \( T \), of the form \( x : P(C) \), is the type of expressions that return values \( M \) of pretype \( P \) such that the formula \( C[M/x] \) can be derived from the log of assumed formulas. Hence, a function type can be fully written out as \( x : P(C) \) \( \rightarrow y : P'(C') \), where its argument has pretype \( P \) and must satisfy the precondition \( C \), and its return value has pretype \( P' \) and is guaranteed to satisfy the postcondition \( C' \).

The type system has the following judgements.

\[
\begin{align*}
E \vdash \cdot & \quad \text{environment } E \text{ is well-formed} \\
E \vdash C & \quad \text{formula } C \text{ holds in environment } E \\
E \vdash T < : T' & \quad T \text{ is a subtype of } T' \text{ in environment } E \\
E \vdash e : T & \quad \text{expression } e \text{ has type } T \text{ in environment } E 
\end{align*}
\]

An environment is well-formed if all the variables in it are well-scoped. A formula holds in an environment if it can be deduced from the formulas in the environment. A
Refinement type $(x:\mathcal{P})\{C\}$ is a subtype of $(x:\mathcal{P}')\{C'\}$ in an environment $E$. $\mathcal{P}$ is a subtype of $\mathcal{P}'$ in $E$ and $E \vdash C \Rightarrow C'$. The rest of the subtyping rules are straightforward.

To illustrate expression typing, we recall four typing rules, those for assumes and asserts, and those for lambda expressions and applications:

$$
\begin{align*}
E \mid \diamond \quad f\mathcal{v}(C) \subseteq \text{dom}(E) & \quad E \mid C

E \mid \text{assume} \quad C : (~: \text{unit})\{C\} & \quad E \mid C

E \mid x : T_1 \rightarrow T_2 <; T & \quad E \mid M : x : T_1 \rightarrow T_2

E, f : T, x : T_1 \vdash e : T_2 & \quad E \mid N : T_1

\quad E \mid \text{rec} f : T, (\text{fun} x \rightarrow e) : x : T_1 \rightarrow T_2 & \quad E \mid (M N) : T_2
\end{align*}
$$

An expression assume $C$ returns a value with postcondition $C$, while assert $C$ requires $C$ to hold in the environment. A recursive function has type $x : T_1 \rightarrow T_2$ if this type is a subtype of its annotation $T$ and its body has type $T_2$ in an environment extended for $f$ and $x$. An application $M N$ has type $T_2$ if $M$ has the function type type $x : T_1 \rightarrow T_2$ and $N$ has a type which is a subtype of $T_1$.

**Type safety** We rely on the main result of Bengtson et al. [2008]: if a program is well-typed, then it is safe. Moreover, if a program is well-typed in an empty environment, then it is robustly safe, that is, it is safe when composed with any expression that has no asserts. Robust safety is useful for protocol security: it states that the properties of the program hold even when composed with an arbitrary active adversary that is given access to the public interface of the program.

**F7 implementation** Our prototype typechecker, F7, is an implementation of the RCF type system that supports a significant subset of F#. In particular, it supports programs that contain type- and value-parametered types, records, polymorphism, mutual recursion, match expressions and mutable references, but it does not, for example, support classes or objects. The typechecker takes two kinds of input files

- F# implementation files (e.g. file.fs) that mention only F# types; and
- F7 interfaces (e.g. file.fs7) with logical assumptions and RCF type annotations.

The typechecker then verifies whether an implementation is well-typed against its interface. To verify the validity of logical formulas (judgement $E \vdash C$), the typechecker can call out to any first-order logic theorem prover. We currently use a leading SMT solver, Z3 to discharge our proof obligations. First-order logic validity is undecidable, so Z3 may fail to prove or disprove some formulas. In these cases, we require additional assumptions (with semi-automated proofs) to verify the program.

3 Reﬁnements for pre- and post-conditions

Classically, for a given function application, a pair of formulas $(C_1, C_2)$ is a valid pair of pre- and post-conditions when, if $C_1$ holds just before calling the function, then $C_2$ holds just after the function completes. Hoare [1969] originally proposed them for arbitrary programs. More recently, for example, Spec# [Barnett et al., 2005] and Code Contracts [Fähndrich et al., 2010] let function definitions be annotated with contracts
(formulas) expressing intended pre- and post-conditions. F7 naturally supports pre- and post-conditions for functions as refinements of their argument and return types. For instance, if an F7 function has type $x_1 : T_1 \{C_1\} \rightarrow x_2 : T_2\{C_2\}$, then asserting $C_1$ before the function call and $C_2$ after the function returns is always safe.

In this section we show how to explicitly refer to pre- and post-conditions of functions using generic predicates indexed by function value. There are at least three ways to define the semantics of these predicates. When speaking of a program with verification annotations, the pre- and post-condition of a function can refer either to the formulas declared with that function, or to the formulas available at the call site, or to events tracking run-time calls and returns. For each semantics, we introduce a pair of generic predicates, informally explain their use, and then give (1) a formal code transformation; and (2) a patch to the F7 typing rules to implement and validate this semantics.

Event-collecting Semantics

Pre- and post-conditions can be seen as events marking the beginning and the end of the execution of a function. We systematically record them by assuming facts for two predicates $Call$ and $Return$: the fact $Call(M,N)$ means that $M$ is a function that has been applied to the argument $N$; $Return(M,N,O)$ means that $M$ is a function that has been applied to $N$ and has returned the value $O$. Formally, this yields a concrete, extensional, finite model, for each partial run of a complete program.

We can use $Call$ and $Return$ to reason about run-time events, instead of introducing ad hoc predicates for that purpose. For instance, if a function $send$ parameterized by $m$ assumes a “begin event” $Send(m)$ before signing a message with payload $m$, we can remove this assume and use instead the generic event $Call(send,m)$ in security specifications. Similarly, suppose that keys are represented as bitstrings, but that the keys in use should be generated only by a designated algorithm $genKey$. We can assign to keys the refinement type $k : bytes \{ Return(genKey(),k) \}$. This pattern frequently applies to cryptographic materials such as nonces, initialization vectors, and tags.

To preserve the consistency of the assumed formulas we rely on a standard notion of positive and negative positions in types and formulas. In the program before the transformation, we forbid positive occurrences of $Call$ and $Return$ in assumed formulas.

Code transformation

We specify this semantics by translating every syntactic function

$$[[ \text{rec } f : T. \text{fun } x \rightarrow e ]]_E = \text{rec } f : T. \text{fun } x \rightarrow \text{assume } Call(f,x); \text{let } r = [[e]]_E \text{ in } \text{assume } Return(f,x,r); r$$

and letting $[[ ]]_E$ be a homomorphism for all other expressions. Thus, we bracket each call with events before and after the call.

Modifying the typechecker

We achieve the same effect as the transformation by directly typechecking functions. We use the rule:

$$x \notin \text{dom}(E) \quad E \vdash x : T_1 \rightarrow T_2 <: T$$
$$E, f : T, x : T_1, Call(f, x) \vdash e : (r : P)\{C\}$$
$$T_2 = (r : P)\{C \land Return(f, x, r)\}$$

$$E \vdash (\text{rec } f : T. \text{fun } x \rightarrow e) : x : T_1 \rightarrow T_2$$
Lemma 1. Suppose that Call and Return do not occur in e.

- Evaluation: for any value M, e \Downarrow M and only if \llbracket e \rrbracket_E \Downarrow \llbracket M \rrbracket_E ;
- Safety: e is safe if and only if \llbracket e \rrbracket_E is safe; and
- Typing: e is well-typed in RCF if and only if \llbracket e \rrbracket_E is well-typed in RCF.

Lemma 2. \llbracket e \rrbracket_E is well-typed in RCF if and only if e is well-typed in RCF_E.

Macro-expansion semantics Pre- and post-conditions may also be seen as pure syntactic sugar, abbreviations that refer to concrete formulas in the types of functions in scope (similar to the definition of pre and post projections of Régis-Gianas and Pottier [2008]). It is useful to refer to the pre- or post-condition of a known and fully annotated function to avoid copying a formula which is big or likely to change during the verification process.

To denote such macro-definitions, we introduce generic predicates \#\text{Pre} and \#\text{Post}. They may occur anywhere in the program or its interface, provided that their first argument is a toplevel variable name that has a declared function type in their scope. Before typechecking, we can always replace each of their occurrence with a concrete formula read off the environment without breaking well-formedness.

Implementation If E(f) = x_1 : P_1\{C_1\} \rightarrow x_2 : P_2\{C_2\}, then we replace any occurrence of \#\text{Pre}(f, M) with C[M/x], and any occurrence of \#\text{Post}(f, M, N) with C'[M/x][N/y]. If the lookup fails, or the returned type is not a function type, preprocessing fails—the macro-definition is ill-formed.

Subtyping-based semantics As opposed to the type annotations of toplevel functions, the declared types of function arguments in higher-order functions are only supertypes of the argument types actually used at their call sites. Thus, as we type the higher-order function, the actual refinements for its argument are unknown, and we cannot just rely on macro-expansion. We refer to these refinements using predicates Pre and Post. Intuitively,

- we use them parametrically while typing higher-order functions, seeing the type of each function argument \( f \) as \( x_1 : P_1\{\text{Pre}(f, x_1)\} \rightarrow x_2 : P_2\{\text{Post}(f, x_1, x_2)\} \);
- we logically relate them to the actual refinements at each call site: as \( f \) is instantiated to some function \( g \) of type \( x_1 : P_1\{C_1\} \rightarrow x_2 : P_2\{C_2\} \), to type the call site, by subtyping of function types, we obtain two proof obligations which will be automatically assumed:

\[
(\phi_{\text{Pre}}) \forall x_1, C_1 \Rightarrow \text{Pre}(f, x_1) \quad (\phi_{\text{Post}}) \forall x_1, x_2, (C_1 \land \text{Post}(f, x_1, x_2)) \Rightarrow C_2
\]

\[
E \vdash P \leftarrow P' \quad E, x : P, C \vdash C'[x/x']
\]

\[
E \vdash (x : P)\{C\} \leftarrow (x' : P')\{C'\}
\]
Relation to the event-based semantics  

Within the body of a higher-order function with function argument \( f \), whenever \( f \) is applied to a value \( M \), the event \( \text{Call}(f, M) \) records this application, and typing requires that the predicate \( \text{Pre}(f, M) \) holds. At runtime, for each instance \( g \) of \( f \), the actual pre-condition of \( g \) holds (by typing) and implies the formal pre-condition of \( f \) (by assumption) so we always have \( \forall x. \text{Call}(f, x) \Rightarrow \text{Pre}(f, x) \).

Similarly, when \( f \) returns, we have \( \text{Return}(f, M, N) \), and its formal post-condition \( \text{Post}(f, M, N) \) implies the actual post-condition for any instance \( g \) of \( f \) (by assumption) so we always have \( \forall x, y. \text{Return}(f, x, y) \Rightarrow \text{Post}(f, x, y) \).

Code transformation  

To support \( \text{Pre} \) and \( \text{Post} \), we rely on the event-based semantics, so we first apply the event-based code transformation, then we transform every let binding whose expression has a function type annotation:

\[
\begin{align*}
\llbracket \text{let } f = e : (f : (x_1 : P_1 \{ C_1 \} \rightarrow x_2 : P_2 \{ C_2 \} \{ C_f \}) \{ C_f \}) \text{ in } e' \rrbracket_S &= \llbracket e = [e]_S \text{ in assume } \phi_{\text{Pre}} ; \text{assume } \phi_{\text{Post}} ; \rrbracket_S \\
\llbracket \text{rec } f : x : T_1 \rightarrow T_2, \text{fun } x \rightarrow e \rrbracket_S &= \llbracket \text{rec } f : x : T_1 \rightarrow T_2, \text{fun } x \rightarrow (\llbracket \text{let } x = (x : T_1) \text{ in } e \rrbracket_S) \rrbracket_S
\end{align*}
\]

where \( \llbracket \rrbracket_S \) is a homomorphism for all other expressions. The first clause applies to every function binding (since they are always annotated). The second clause applies to every syntactic function definition, ensuring that all functional arguments are annotated in higher-order functions.

Modifying the typechecker  

We modify F7 to support \( \text{Pre} \) and \( \text{Post} \) by modifying insertions of variables entries with function types into the typing environment. Hence, \( E \) extended with \( f : T \) is now written \( E \oplus f : T \), and defined by pattern matching on \( T \). If \( T \) is a function type, it is of the form \( f(x_1 : P_1 \{ C_1 \} \rightarrow x_2 : P_2 \{ C_2 \}) \{ C_f \} \) and we let

\[
E \oplus f : T \triangleq E, f : T, \phi_{\text{Pre}}, \phi_{\text{Post}}
\]

Otherwise \( E \oplus f : T \) is just \( E, f : T \). We call the modified type system RCF\(_S\).

To maintain logical consistency we require that programmers use \( \text{Pre} \) only negatively and \( \text{Post} \) only positively in their assumptions.

Results  

We obtain a variant of Lemma 1 for the subtyping semantics: we have a similar Evaluation property. The proof of Safety involves showing the logical consistency of the injected assumptions.

We also prove two flavours of Correctness: we have a variant of Lemma 2 that relates typing with RCF\(_S\) and the specification \( \llbracket \rrbracket_S \). Besides, we show that \( \text{Pre} \) and \( \text{Post} \) can be eliminated by duplicating the code of higher-order functions at each call site and annotating them with ad hoc types. Let \( T \) be a higher-order function type \( f(x : T_1 \rightarrow T_2) \{ C_f \} \rightarrow T_3 \) such that the only \( \text{Pre} \) and \( \text{Post} \) within \( C_f \) and \( T_3 \) are of the form \( \text{Pre}(f, M) \) and \( \text{Post}(f, M, N) \) for values \( M \) and \( N \). These \( \text{Pre} \) and \( \text{Post} \) can be eliminated by duplicating the code of higher-order functions at each call site and annotating them with ad hoc types.

Lemma 3 (Inlining). Let the expression \( \text{let } f = \text{rec } f : T, e \text{ in } e' \) use no \( \text{Pre} \) or \( \text{Post} \) and be well-typed in RCF\(_S\). If the typing judgment of \( e \) does not use the formulas injected by RCF\(_S\), then any function application \( fg \) in \( e' \), where \( g \) is a well-scoped toplevel function, can be rewritten as \( eg \) and given type \( T_3 \) where all occurrences of \( \text{Pre}(f, M) \) and \( \text{Post}(f, M, N) \) are replaced with \( \#\text{Pre}(g, M) \) and \( \#\text{Post}(g, M, N) \).
4 Example: A MAC-based Authentication Protocol

As a preliminary example, we consider a simple client-server authentication protocol. We shall see how to specify and verify an implementation for this protocol using only the event-collecting semantics of the events Call and Return.

\[ A \rightarrow B : m | (\text{mac } k_{AB} m) \]

(The symbol | represents an invertible concatenation of bytestrings.) When a principal \( a \) (playing role \( A \)) wants to send a message \( m \) to principal \( b \) (playing role \( B \)), it also sends a MAC over \( m \) computed with a key \( k_{ab} \) known only to \( a \) and \( b \). This MAC authenticates the sender (only \( a \) or \( b \) could have sent this message) and protects the integrity of the message (the sender must have intended to send message \( m \)).

This simple protocol can be implemented in ML as three functions:

```ml
let mkKey a b = hmac_keygen()
let client a b k m = let c = Net.connect p in let h = hmac k m in let w = concat m h in Net.send c w
let server a b k = let c = Net.listen p in let w = Net.recv c in let (m, h) = iconcat w in hmac_verify k m h;
```

The \textit{mkKey} function generates a fresh MAC key for use with messages sent from \( a \) to \( b \). (We assume that messages in the reverse direction will use a separate key.) The \textit{client} function takes such a shared key \( k \) and uses it to protect a message \( m \) that \( a \) wishes to send to \( b \) over the public network. The \textit{server} function receives a message over the public network and uses a shared MAC key to verify the MAC on the message.

This protocol code runs in a hostile environment where an attacker may use the public interfaces of the protocol and the libraries to interfere with the protocol. The attacker may call the networking functions \textit{send}, \textit{recv} on any TCP connection to intercept and interject messages of his choice. He may construct and verify MACs by calling \textit{hmac} and \textit{hmac\_verify} with keys that he already knows. He may also start any number of copies of the client and server and get them to communicate with each other.

The authentication goal for the protocol is that if the \textit{server} function returns a message \( m \) when called with \( a \), \( b \), and a key \( k \) generated by the \textit{mkKey} function, then the server knows that some client for \( a \) sent this message \( m \) to \( b \). In particular, an adversary who does not know a key generated for \( a \) and \( b \) cannot fool \( b \) into accepting a message that was not sent by \( a \).

We express this security goal within the refinement types for these functions:

```ml
val mkKey: a:str -> b:str -> k:key
val client: a:str -> b:str -> key{Return(mkKey,[a;b],k)} -> m:bytespub -> unit
val server: a:str -> b:str -> key{Return(mkKey,[a;b],k)} -> m:bytespub
```

To verify that the code actually meets these types, we rely on the unforgeability of MACs, expressed as types for the cryptographic library, as shown in Appendix C.

In particular, the function \textit{hmac} has a precondition \textit{MACSays}(k,m), representing the conditions under which the key \( k \) may be used to MAC \( m \). Every protocol that uses
MACs must specify MACSays for its keys that it uses. The function hmac_verify has a post-condition that it returns a value \( m \) only if either MACSays\((k,m)\) or if the key \( k \) is public, that is, known to the attacker. For the keys in our authentication protocol, we use MACSays to specify that a key \( k \) generated for \( a \) and \( b \) using mkKey will only be used to MAC a message \( m \) after client has been called with \( a, b, k, \) and \( m \):

\[
\text{assume } \forall a,b,k. \text{Return}(\text{mkKey},[a:b:k],k) \Rightarrow (\text{MACSays}(k,m) \iff \text{Call}([a:b:k:m]))
\]

We can then verify by typing that our code meets the security goal, and by the type safety theorem of RCF we have that our protocol implementation is secure against our attacker model.

**Comparison with other methods** Many symbolic verification tools can handle the simple protocol above. Tools such as ProVerif [Blanchet, 2001] can even automatically infer the logical assumption on MACSays, thus requiring almost no annotations. In comparison to earlier work on F7, our type specification above uses the events Call and Return. In their absence, the programmer would have to define his own predicates corresponding to these events and enforce their relationship to the function calls by assuming them within protocol code. Here, these events are declared and managed automatically.

### 5 Example: A Reusable Typed Interface for Lists

Lists are perhaps the most commonly-used data structures in functional programs. The F# List library provides efficient implementations of recursive list processing functions; for generality, these functions are typically higher-order and polymorphic. Our goal is to give this library a reusable refinement typed interface, using our Pre and Post predicates and their subtyping-based semantics. The full interface is listed in Appendix B.

We detail our approach on the function List.fold, the general iterator on lists (also called fold_left). Its ML type is \( \text{val fold: } (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list } \rightarrow \alpha \). It takes as argument a function \( f \), an initial accumulator \( a \), a list \( l \) and traverses the list \( l \), applying \( f \) to the current accumulator and the next value in the list to obtain a new accumulator; when it reaches the end of the list, it returns the accumulator. For example, \( \text{fold} (+) 0 \) \([1;2;3;4]\) computes the sum of the elements in the list.

**First attempt: Using Recursive Predicates** Let us define two predicates PreFold and PostFold to represent the pre- and post-condition of fold. By inspecting the code for fold (on the left below) we can define these predicates as shown:

\[
\begin{align*}
\text{let rec } & \quad \text{fold } f \text{ acc } l = \\
\text{match } l & \quad \text{with} \\
| [] & \quad \Rightarrow \text{acc} \\
| \text{hd :: tl} & \quad \Rightarrow \langle \exists \text{hd.',tl}. \text{hd::tl} \land \text{Pre}(f,\text{acc',hd'}),\rangle \land \\
& \quad \text{let acc'} = f \text{ acc' hd' in} \\
\text{fold } f \text{ acc' tl} & \quad \Rightarrow \text{Post}(f,\text{acc'.tl',r})))) \\
\end{align*}
\]

The definition for PreFold can be read as follows. If the list is empty, there is no pre-condition. Otherwise, the pre-condition of the argument \( f \) must hold for the head of
the list and the current accumulator, and if \( f \) terminates and returns a new accumulator, 
\textit{PreFold} must hold for the tail of the list and this new accumulator. \textit{PostFold} is defined similarly. The resulting type 
\[ \text{val } \text{fold}: \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \text{list}\{\text{PreFold}(f, \text{acc}, l)\} \rightarrow \beta \rightarrow \text{PostFold}(f, \text{acc}, l, r) \]
is precise and easy to typecheck against the code of \( \text{fold} \), yet difficult to use at call sites. Indeed, even for a function with no pre-condition (\( \forall x. \text{Pre}(f, x) \)), proving \( \text{PreFold}(f, \text{acc}, l) \) requires the use of induction, which is generally beyond the reach of the SMT solver Z3 that underlies F7. Can we use a non-recursive predicate to specify \( \text{fold} \)?

\textbf{Second attempt: Using Invariants} In our second approach, we adopt the style of Régis-Gianas and Pottier [2008] for specifying higher-order iterators, such as \( \text{fold} \). We introduce a generic predicate \( \text{In} \) that is used to define logical invariants for functions that may be used as an argument to \( \text{fold} \). The formula \( \text{In}(f, \text{aux}, \text{acc}, l) \) is an invariant that holds when the function \( f \) is being applied to a list of elements: \( l \) is the remainder of the list, \( \text{acc} \) is the intermediate result of the computation, and \( \text{aux} \) contains function-specific auxiliary information about the initial arguments to the \( \text{fold} \).

As an example, consider the function \( \text{fmem} \) that can be used with \( \text{fold} \) to search for an element in a list; its code, type, and invariant are as follows:

\[
\begin{align*}
\text{let } \text{fmem } v & \quad \text{val } \text{fmem}: v: \alpha \rightarrow (\forall x.f. \text{Post}(\text{fmem}, v, f)) \Rightarrow \\
& \quad \text{acc} \rightarrow (\forall x.f. \text{Inv}(f, \text{iv}, \text{acc}, l) \Leftrightarrow \\
& \quad n: \alpha \rightarrow a: \rightarrow (\exists \text{linit} \cdot \text{iv} = (x, \text{linit}) \land x = v) \\
& \quad = \quad \text{found}: \text{bool}\{ \\
& \quad \text{if } v = n \quad (v = n) \\
& \quad \text{then true} \quad \land \text{found} = \text{true} \\
& \quad \text{else acc} \quad \lor (\text{found} = \text{acc}) (\lor \text{acc} = \text{false}) )
\end{align*}
\]

The function \( \text{fmem} \) takes an element \( v \) to search for, an accumulator \( \text{acc} \) and an integer \( n \), and returns true if either \( \text{acc} \) is true or if \( v = n \). The invariant for the partial application \( \text{fmem } v \) is shown below; its auxiliary argument \( \text{aux} \) is a pair consisting of the integer we are searching for and the initial list. Its auxiliary argument is a pair consisting of the integer \( v \) to search for, and the initial list \( \text{linit} \). The invariant says that the remaining list \( l \) contains a subset of the elements in \( \text{linit} \), and that the accumulator is true only if \( v \) is a member of \( \text{linit} \).

The next step, following Régis-Gianas and Pottier, is to prove that the invariant is \textit{hereditary}, namely that the invariant of each function \( f \) is at least as strong as its pre-condition, and that the invariant is preserved by function application. We define a predicate \textit{Hereditary} that captures this notion and use it to type \( \text{fold} \) as follows; to use this style we need to add an additional argument to \( \text{fold} \) that holds the auxiliary values needed to maintain its invariants.

\[
\begin{align*}
\text{let rec } \text{fold } v f \text{ acc } l = \\
\text{match } l \text{ with } \quad & \text{assume}(\forall y. \text{Hereditary}(f)) \\
| [] & \rightarrow \text{acc} \leftrightarrow \\
| \text{hd} :: \text{tl} & \rightarrow (\forall v. \text{acc}, \text{h}, \text{t}. \text{Inv}(f, v, \text{acc}, \text{hd} :: \text{tl}) \Rightarrow \\
& (\text{Pre}(f, \text{acc}, \text{hd})) \\
& \text{let acc'} = f \text{ acc } \text{ hd} \text{ in} \\
& \text{fold } v f \text{ acc'} \text{ tl} \\
& (\forall r. \text{Post}(f, [\text{acc}; \text{hd}], r) \Rightarrow \text{Inv}(f, v, r, \text{tl})))
\end{align*}
\]
val \textit{fold} : \gamma \rightarrow f : (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \text{acc} : \alpha \\
\rightarrow \text{xs} : \beta \text{ list} \rightarrow \text{Inv(f,x,acc,xs)} \\
\rightarrow r : \alpha \rightarrow \alpha \{ (\text{xs} = [] \land r = \text{acc}) \lor \text{Inv(f,x,r,[])} \}

The type of \textit{List.fold} requires that (1) the invariant of the iterated function is hereditary; and (2) the invariant holds for the initial accumulator. The post-condition states that the invariant holds for the final accumulator.

Hence, for example, to typecheck that \textit{fold fmem v (\gamma,l)} has the type \text{bool} \{ b = \text{true} \Rightarrow \text{Mem(v,l)} \}, we must prove that \forall f. \text{Post(fmem,v,f)} \Rightarrow \text{Hereditary}(f), and that the invariant of \textit{fmem v} holds for the initial values \text{false}. For a simple function like \textit{fmem} this can be proved automatically, but for more complex functions \textit{Hereditary} may have to be proved by hand. The rest of the typechecking is fully automatic.

6 Case Studies: Cryptographic Protocol Implementations

We use our new types for lists to verify more realistic cryptographic applications. We present three case studies of programs previously verified using F7 and show how our extensions reduce the annotation effort.

XML Digital Signatures The XML digital signature standard specifies cryptographic mechanisms to provide integrity, message authentication, and signer authentication for arbitrary XML data [Eastlake et al., 2002]. These mechanisms are used within web services security protocols to protect messages, and processing each message involve tree and list processing. For example, consider a single-message protocol, where the principal \textit{a} uses an XML signature to protect \( n \geq 1 \) XML elements \( m_1, \ldots, m_n \) located at URIs \#1, \ldots, \#n within the message, using the MAC key \( k_{ab} \). The main security goal for this protocol is that the list \([m_1; \cdots; m_n]\) be authenticated; the protocol is often used as a component within a larger protocol that enforces a more demanding security property. A slightly simplified version of this protocol is as follows:

\[
A \rightarrow B : \langle \text{Message} \rangle \\
\hspace{1cm} m_1 \ m_2 \ldots \ m_n \\
\hspace{1cm} \langle \text{Signature} \rangle \\
\hspace{2cm} \langle \text{SignatureInfo} \rangle \\
\hspace{3cm} \langle \text{Reference} \rangle \text{base64 (sha1 }m_1\rangle \langle /\text{Reference} \rangle \\
\hspace{3cm} \langle \text{Reference} \rangle \text{base64 (sha1 }m_2\rangle \langle /\text{Reference} \rangle \\
\hspace{3cm} \langle /\text{SignatureInfo} \rangle \\
\hspace{2cm} \langle \text{SignatureValue} \rangle \\
\hspace{1cm} \text{base64} \ (\text{mac }k_{AB} \langle \text{SignatureInfo} \rangle \ldots \langle /\text{SignatureInfo} \rangle) \\
\hspace{1cm} \langle /\text{SignatureValue} \rangle \\
\hspace{1cm} \langle /\text{Signature} \rangle \\
\hspace{1cm} \langle /\text{Message} \rangle
\]

In previous work, Bhargavan et al. [2010] used F7 to program and verify a library for manipulating XML signatures. Now we can \textit{List.map} to improve this library and ease its verification. Consider the following excerpt of the library interface and implementation.
The type \texttt{item} represents XML elements; its constructor \texttt{Xn(q,al,il)} corresponds to an XML element of the form \texttt{<q a1>i1</q>}, where \texttt{q} is a qualified name, such as \\texttt{Signature}, \texttt{al} is a list of XML attributes, and \texttt{il} is a list of XML items.

The function \texttt{mkRef} generates a \texttt{sha1} cryptographic hash of its argument and returns it within a \texttt{<Reference>} element. The function \texttt{xml_sign} generates an XML signature over a list of XML elements; it uses \texttt{map} over \texttt{mkRef} to generate a list of references, encapsulates them within a \texttt{<SignatureInfo>} element and MACs it with the given key \texttt{k}. The function \texttt{xml_verify} parses and verifies XML signature. Its post-condition guarantees that the signature must have been generated using \texttt{xml_sign} by a valid client (as part of an authenticated message).

The use of \texttt{List.map} avoids the need to inline the recursive code for \texttt{map} in the code for \texttt{xml_sign} and \texttt{xml_verify}. In our previous verification of the full library, there were four instances where we needed to inline list-processing functions and define new type annotations for each instance. These are no longer necessary, reducing the annotation burden significantly.

\textbf{X.509 Certification Paths} The X.509 recommendation [ITU, 1997] defines a standard format and processing procedure for public-key certificates. Each certificate contains at least a principal name, a public-key belonging to that principal, an issuer, and a signature of the certificate using the private key of the issuer.

On receiving a certificate, the recipient first checks that the issuer is a trusted certification authority and then verifies the signature on the certificate before accepting that the given principal has the given public key. To account for situations where the certification authority may not be known to the recipient, the certificate may itself contain a \textit{certification path}: an ordered sequence of public-key certificates that begins with a certificate issued by a trusted certification authority and ends with a certificate for the desired principal. The X.509 sub-protocol between principals for certification paths can be written as follows:

\[
A \rightarrow B : \text{Certificate}(A_1 \mid pk_{A_1} \mid rsa\_sign \ sk_{CA} \ (A_1 \mid pk_{A_1})) \\
\quad \text{Certificate}(A_2 \mid pk_{A_2} \mid rsa\_sign \ sk_{A_1} \ (A_2 \mid pk_{A_2})) \\
\quad \quad \ldots \\
\quad \text{Certificate}(A \mid pk_{A} \mid rsa\_sign \ sk_{A_{n-1}} \ (A \mid pk_{A}))
\]

We write and verify a new library for manipulating X.509 certificates. The code for verifying certificates uses \texttt{List.fold} to process certification paths:
val verify: 
\[ x : \text{cert}\{\text{Certificate}(x)\} \rightarrow b : \text{bytes} \rightarrow \\
\quad r : \text{cert}\{\text{Certifies}(x, r) \wedge \text{Certificate}(r)\} \]

let verify_all c path = 
\[ \text{fold } c \text{ verify } c \text{ path} \]

val verify_all: 
\[ x : \text{cert}\{\text{Certificate}(x)\} \rightarrow l : \text{bytes list} \rightarrow \\
\quad r : \text{cert}\{\text{Certifies}(x, r)\} \]

\[ \text{assume } \forall \text{ca}, x, h, l. \\
\quad \text{Inv}(\text{verify}, \text{ca}, x, l) \leftrightarrow \\
\quad (\text{Certificate}(x) \wedge \text{Certifies}(\text{ca}, x)) \]

The predicate \(\text{Certifies}(x, y)\) specifies that there is some sequence of certificates \(x = x_0, x_1, \ldots, x_n = y\) such that the principal mentioned in each \(x_i\) has issued the certificate \(x_{i+1}\); hence if every principal mentioned in this sequence is honest, then we can trust that the public-key in the final certificate \(y\) indeed belongs to the principal mentioned in \(y\).

The function \(\text{verify_all}\) takes as an argument a certificate \(\text{ca}\) for a trusted certification authority and it accepts only those certification paths that begin with certificates issued with \(\text{ca}\)'s public-key. To typecheck \(\text{verify_all}\) we define the \text{fold} invariant for \(\text{verify}\) as the property that the accumulator \(x\) always has a valid certificate (\(\text{Certificate}(x)\)) and that there is a valid path from the initial certificate \(\text{ca}\) to \(x\) (\(\text{Certifies}(\text{ca}, x)\)).

The use of \(\text{List.fold}\) in \(\text{verify_all}\) is the most natural way of writing this code. We could inline the code for \(\text{List.fold}\) and redo the work of typechecking it for this instance, but reusing the types and formulas in \(\text{List}\) is more modular, and we believe, the right way of developing proofs for such cryptographic applications.

Compact types for Audit: F7 can be used to verify more complex security properties than just authentication. Guts et al. [2009] show how to use refinement types to specify and verify the auditability of a protocol implementation. Informally, a program collecting evidence has an auditable property if this property can be checked by the program (immediately) and by a third-party judge (a posteriori) using the evidence.

For example, consider the authentication protocol in Section 4. We may require that the server should be able to convince a judge that a valid client sent him a given message. As evidence for this property, the server can present the message and its MAC. However, this evidence is not enough to convince the judge, since the MAC is based on a shared key known to both parties, so the judge cannot decide whether the client or the server created the MAC. Hence, we say that the property is not auditable in this program.

To achieve auditability, the parties can instead use public-key signatures: a client signs the message using its private key so that the server—or any third party who has access to the public key—can check the signature to authenticate the message.

To verify that the new program is indeed auditable, we first define a judge function that checks the evidence for an instance of the audited property. All verification functions used by the judge must be total, terminating and deterministic, and may only use data (such as public keys) agreed on beforehand by all protocol participants. Then, the audited program is annotated with \(\text{audit}\) requests where the property is expected to be auditable: that is, if the judge was called at this program point with the same arguments, it would succeed.

In the example of Section 4, suppose we add an audit request for client authentication to the server function. The judge function for this property would be:
The judge simply calls the signature verification function `rsa_verify`, which never throws exceptions and returns `true` if and only if the message has been signed using the public key. (The full interface for public-key signature functions is in Appendix 7.) Hence, the type of the judge is:

```plaintext
val judge: a: str → b: str → k: key{∃ sk. PubPrivKeyPair(k, sk)} → m: bytes → s: bytes → t: bool { t=true ⇔ IsSignature(s, sk, m)}
```

To formally check the auditability of this protocol, first we need to show that the judge is correct: whenever it returns `true`, the audited property holds:

```plaintext
∀a,b,k,m,s. #Post(judge, [a; b; k; m; s]) ⇒ (∃ sk. PubPrivKeyPair(k, sk) ∧ Call(client, [a; b; k; m]))
```

Then, to check that at every audit request the judge would have returned `true`, we set the precondition of the `audit` primitive to `#Post` of the judge:

```plaintext
val audit: a: str → b: str → k: key{∃ sk. PubPrivKeyPair(k, sk)} → m: bytes → s: bytes { #Post(judge, [a; b; k; m])} → unit
```

The use of the macro-expansion predicate `#Post` here is convenient when changing the type of audit, as the protocol or the judge change. Indeed, we cannot use the `Return` event or `Post` predicate here because both rely on a function having been called; here the call to `audit` must be typable without the judge function having been called.

**An auditable multi-party protocol** We have also used the three semantics for pre- and postconditions to verify an auditable multi-party protocol. In this game between `n` players and a server, the participants only make minimal trust assumptions on other players’ behaviour, and shield themselves from various attacks.

Because the number of the players is not known in advance, participants have to manipulate lists of cryptographic evidence. In particular, the `List.forall` is used for three different checks, so in the original F7 implementation the function code had to be replicated and given an ad hoc type for each check. Using the subtyping-based semantics, the same library function `List.forall` can be used at all call sites, which makes the code both more compact and more readable.

### 7 Related work

Pre- and post-condition checking has been implemented by several program verification tools [Barnett et al., 2005, Flanagan et al., 2002, Xu, 2006]. Our approach is the closest to the work by Régis-Gianas and Pottier [2008] who show how to use Hoare-style annotations to check correctness of programs written in a call-by-value language with recursive higher-order functions and polymorphic types. They extract proof obligations out of programs, and prove them using automated provers. However, their system only uses declared types, and disregards subtyping and events.

Symbolic security verification techniques for programs have used a variety of techniques from model-checking to cryptographic theorem-proving [Goubault-Larrecq and Parrennes, 2005, Chaki and Datta, 2009, Bhargavan et al., 2008]. Such methods are often fully automated and require little program annotation. However, they generally do
not apply to programs with recursive data structures. Moreover, whole program analysis seldom scales as well as modular techniques such as typechecking.

The RCF type system is the first to use refinement typing for security analysis. Its implementation in the F7 typechecker has already been successfully used to verify complex cryptographic applications [Backes et al., 2009, Bhargavan et al., 2009].

Fine [Swamy et al., 2010, Chen et al., 2010] is another refinement typechecker for F#. It extends the F# source language with refinement types and focuses on its secure compilation. Fine has a notion of predicate polymorphism that can capture some of the benefits of our pre- and post-condition predicates. To use them, the programmer declares predicate parameters for higher-order functions and explicitly instantiates these predicates at each call site. In contrast, our approach is able to verify legacy programs written purely in F# by automatically injecting pre- and post-condition predicates.

By using a standard program verification technique, we hope to benefit from recent advances in verification technology. For example, Liquid Types [Rondon et al., 2008] have been proposed as a technique for automatically inferring refinement types for ML programs. The types inferred by Liquid Types are quite adequate for verifying simple safety properties of a program, but not for the security types in this paper. As future work, we plan to adapt such inference techniques to reduce F7 annotations even further.

References


A Syntax of RCF

\[
\begin{align*}
M, N & ::= \\
( ) & \text{unit} \\
x & \text{variable} \\
(M, N) & \text{pair} \\
h M & \text{constructor} \\
\text{rec } f : T. \text{fun } x \rightarrow e & \text{recursive function} \\
\end{align*}
\]

\[
\begin{align*}
e & ::= \\
M & \text{value} \\
M N & \text{function application} \\
\text{let } x = e_1 \text{ in } e_2 & \text{sequential composition} \\
\text{let } (x, y) = M \text{ in } e & \text{pair projection} \\
\text{match } M \text{ with } h x \rightarrow e_1 \text{ else } e_2 & \text{pattern matching (else optional)} \\
\text{assume } C & \text{assume formula} \\
\text{assert } C & \text{expect formula} \\
\text{assert } C & \text{annotated expression} \\
\text{assume } C & \text{parallel composition} \\
p(M_1, \ldots, M_n) & \text{restriction, create new channel } a \\
\alpha M & \text{send } M \text{ on channel } a \\
\alpha ? & \text{receive message off channel } a \\
\end{align*}
\]

\[
\begin{align*}
C & ::= \\
\text{True } | \text{False} & \text{constants} \\
M_1 = M_2 \mid M_1 \neq M_2 & \text{comparison} \\
P(M_1, \ldots, M_n) & \text{predicate application} \\
\text{not } C \mid C \land C \mid C \lor C & \text{boolean operators} \\
\forall x. C \mid \exists x. C & \text{first-order quantification} \\
P & ::= \\
\text{unit} & \text{dependent function type (scope of } x \text{ is } T_2) \\
x : T_1 \rightarrow T_2 & \\
\end{align*}
\]
dependence pair type (scope of \( x \) is \( T_2 \))

\( \alpha \)

type constructor

\( \Sigma_i(t_i : T_i \rightarrow \alpha) \)

algebraic datatype (sum type)

\( \mu \alpha \cdot P \)

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

\( T, U, V ::= \)

Refinement Types

\( (x : P)\{C\} \)

\( x \) of pretype \( P \) such that \( C \) (scope of \( x \) is \( C \))

\( E ::= \)

Type Environment

**B List Library Interface**

**assume**

\((\forall x, u. \text{Mem}(x,x;u)) \land \)

\((\forall x, y, u. \text{Mem}(x,u) \Rightarrow \text{Mem}(x,y;u)) \land \)

\((\forall x, u. \text{Mem}(x,u) \Rightarrow (\exists y, v. u = y;v \land (x = y \lor \text{Mem}(x,v)))) \)\)

**val**\( \text{mem}: x : \alpha \rightarrow u : \alpha \text{ list} \rightarrow r : \text{bool}\{r = \text{true} \Rightarrow \text{Mem}(x,u)\}\)

**val**\( \text{find}: f(\alpha \rightarrow \text{bool}) \rightarrow \)

\( u : \alpha \text{ list}\{\{\forall x. \text{Mem}(x,u) \Rightarrow \text{Pre}(f(x))\} \rightarrow \)

\( r : \alpha \{ \text{Mem}(r,u) \}\}

**val**\( \text{forall}: t(\alpha \rightarrow \text{bool}) \rightarrow \)

\( x : \alpha \text{ list}\{\{\forall x. \text{Mem}(x,xs) \Rightarrow \text{Pre}(t,x))\} \rightarrow \)

\( b : \text{bool}\{b = \text{true} \Rightarrow (\forall x. \text{Mem}(x,xs) \Rightarrow \text{Post}(r,x],\text{true}))\}\}

**val**\( \text{exists}: t(\alpha \rightarrow \text{bool}) \rightarrow \)

\( x : \alpha \text{ list}\{\{\forall x. \text{Mem}(x,xs) \Rightarrow \text{Pre}(t,x))\} \rightarrow \)

\( b : \text{bool}\{b = \text{true} \Rightarrow (\exists x. \text{Mem}(x,xs) \land \text{Post}(r,x],\text{true}))\}\}

**val**\( \text{iter}: f(\alpha \rightarrow \text{unit}) \rightarrow \)

\( l : \alpha \text{ list}\{\{\forall x. \text{Mem}(x,l) \Rightarrow \text{Pre}(f,x))\} \rightarrow \)

\( r : \text{unit}\{\forall x. \text{Mem}(x,l) \Rightarrow \text{Post}(f,x],[l])\}\)

**assume**

\((\forall x, y, u, v. \text{Mem}2((x,y),(x':u,x':v))) \land \)

\((\forall x, y, u, v, x', y'. \text{Mem}2((x,y),(u,b)) \Rightarrow \text{Mem}2((x',y'),(u,b))) \land \)

\((\forall x, y, u, v. \text{Mem}((x,y),(u,v)) \Rightarrow (\exists y, l, i, t. \text{Post}(r,[l],y;v) \land i = y;v \land i = \text{true}) \land (l1 = \text{true} \land l2 = \text{true}) \land \text{Pre}(f,[x]))\)\)

**val**\( \text{map}: f(\alpha \rightarrow \beta) \rightarrow \)

\( l : \alpha \text{ list}\{\{\forall x. \text{Mem}(x,l) \Rightarrow \text{Pre}(f,x))\} \rightarrow \)

\( r : \beta \text{ list}\{\forall x, y. \text{Mem}2((x,y),(l,r)) \Rightarrow \text{Post}(f,x),y)\}\)

**assume**\( (\forall x, acc,\text{ht}. \text{Inv}(f,\text{acc},\text{ht};[l])) \Rightarrow \)

\((\text{Pre}(f,[\text{acc},\text{ht}]))\)
∀ r. Post(f, [acc; hd], r => Inv(f, v, r, [])))

val fold : v: γ → f: (α → β → α) {Hereditary(f)} → acc: α → xs: β list → r: α { (xs = [] ∧ r=acc) ∨ Inv(f, v, r, []) }

C Cryptographic Library Interface (Excerpt)

val hmac_keygen: unit → k: key {MKey(k)}

val hmac: k: key {MKey(k)} → m: bytes {MACSays(k, m)} → h: bytes {Pub(m) ⇒ Pub(h)}

val hmac_verify: k: key {MKey(k)} → m: bytes → h: bytes → unit {MACSays(k, m)}

val rsa_keygen: unit → k: key {PrivKey(k)}

val rsa_pub: sk: key {PrivKey(sk) → k: key {PubPrivKeyPair(k, sk)}}

val rsa_sign: k: key {PrivKey(k)} → m: bytes {SignSays(k, m)} → h: bytes {Pub(m) ⇒ Pub(h)}

val rsa_verify: k: key {PubKey(k)} → m: bytes → h: bytes → b: bool {b = true ⇔ (∃ sk. PubPrivKeyPair(k, sk) ∧ IsSignature(h, sk, m))}

assume ∀ h, sk, m. IsSignature(h, sk, m) ∧ PrivKey(sk) ⇒ SignSays(sk, m)