Pre- and Post-conditions for Security Typechecking

Karthik Bhargavan[3,1], Cédric Fournet[2,1], and Nataliya Guts[1]

[1] MSR-INRIA Joint Centre
[2] Microsoft Research
[3] INRIA

Abstract. We verify implementations of cryptographic protocols that manipulate recursive data structures, such as lists. Our main applications are X.509 certificate chains and XML digital signatures. These applications are beyond the reach of automated provers such as ProVerif, since they require some form of induction. They can be verified using the F7 refinement typechecker, at the cost of annotating each data-processing function with logical formulas embedded in its type. However, this entails the duplication of many library functions, so that each instance can be given its own type.

We propose a more flexible method for verifying ML programs that use cryptography and recursion. We annotate standard library functions (such as list processing functions) with precise, yet reusable types that refer to the pre- and post-conditions of their functional arguments, using generic logical predicates. We use these predicates both to specify higher-order functions and to track security events in programs and cryptographic libraries. 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 implementing 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 against first-order logic type interfaces.

Suppose that Alice sends a service request to Bob, who authenticates Alice's request before delivering the service. Bob programs in ML, so the security of his code can be validated using the F7 typechecker to enforce his policy. Depending on the control- and data-flow of the protocol between Alice and Bob, typechecking essentially propagates logical annotations from cryptographic and communications primitives up to the protocol interface. Thus, the programmer

provides a few protocol-specific type annotations (for instance when allocating a key) and 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 involves higher-order functions, and the programmer must repeatedly provide an ad hoc type for each usage of 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. Annotating code is the tedious part of work of an F7 programmer.

Now, suppose that the message format used by Alice and Bob is under development and changes regularly. Each change trickles down the protocol data flow, causing many changes to its logical annotations, and possibly further code replication. This compromises the code modularity. 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 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 Appendix A. Expressions are written in a subset of F#, a dialect of ML. Types are F# types refined with first-order formulas. 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 a value, this value is an integer 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 int.

Using refinements, we can write expressive types for functions. For instance, the dependent function type $v{:}\text{int} \to w{:}\text{int}\ \{w{>}v\}$, a subtype of $\text{int} \to \text{int}$, 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 library in many flavours of ML. The type int option represents an optional integer: it is either of the form $\mathit{Some}\ n$ where n has type int, or it is of the form None. Using option types, we can, for example, easily code up protocols that have optional fields in their messages. To manipulate a message field of type int option, it is convenient to use the higher-order library function $\mathit{Option.map}$:

```
val map: (int → int) → int option → int option
let map f x = match x with
  | None → None
  | Some(v) → let w = f v in Some(w)
```

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

val $f : v : \text{int} \rightarrow w : \text{int}\ \{w{>}v\}$
let $y = map\ f\ (Some(0))$

We would then like to give y a type that reflects the post-condition of f:

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

What type must *map* have in order for y to have this type? The most precise type we can give *map* in RCF accounts for the various cases (*None* vs *Some*) of the argument and the result:

val *map*: f:(int $\rightarrow$ int) $\rightarrow$ x:(int option) $\rightarrow$ y:(int option)
$\{\ (x = None \wedge y = None) \vee (\exists v,w.\ x = Some(v) \wedge y = Some(w))\}$

This type does not capture the possible post-condition of the function f. Indeed, within F7, the only way to check that y has its desired type is to copy the definition of the *map* function just for f and to typecheck it again.

Our main idea is to introduce annotations for pre- and post-conditions that are automatically injected and checked by the F7 typechecker. We introduce predicates *Pre* and *Post* and use them within the type of higher-order functions to refer to the pre- and post-conditions of their functional arguments. For instance, $Post(f,v,w)$ can refer to the post-condition of a function parameter f applied to v returning w, and we can give *map* the type:

val *map*: f:(int $\rightarrow$ int) $\rightarrow$ x:int option $\rightarrow$ y:int option
$\{(x = None \wedge y = None) \vee (\exists v,w.\ x = Some(v) \wedge y = Some(w) \wedge Post(f,v,w))\}$

Whenever *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 $Post(f,v,w)$. Hence, y can be given its desired type without loss of modularity.

We show how to use such *Pre* and *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 that use these functions. Verifying such implementations is beyond the reach of typical symbolic verification tools, since their proof requires some form of induction. For example, Fs2pv [Bhargavan et al., 2008] is a verification tool that compiles implementation code in F# into the applied π-calculus, for analysis with ProVerif [Blanchet, 2001], a state-of-the-art domain-specific prover. Although 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 F7 typechecker to automatically support pre- and post-condition predicates. We study three different semantics for these predicates and give examples of their use. We design precise and modular APIs for lists and several cryptographic protocol implementations using lists (X509 certificates and XML signatures).

Contents Section 2 explains our extension of F for pre- and post-conditions, presenting different design choices. Section 3 illustrates the use of pre- and post-conditions to specify and verify a simple authentication protocol. Section 4 illustrates the use of pre- and post-conditions to give reusable types to a library for lists. Section 5 describes and evaluates larger verification case studies of cryptographic protocol implementations. Section 6 discusses related work.

2 Refinements for pre- and post-conditions

Classically, a pair of formulas (C_1, C_2) models a valid pair of pre- and post-conditions for a given function application when, if formula C_1 holds just before calling the function, then formula 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 $x{:}(x{:}T\ \{C_1\}) \rightarrow y{:}T'\{C_2\}$, 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. The semantics of these predicates is not as straightforward as it seems. When speaking of a program with verification annotations, the pre- and post-condition of a function can refer either to the formulas associated with that function (irrespective of its call sites) or to the concrete events of function call and termination. 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.

2.1 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 record them by assuming facts for two predicates *Call* and *Return*: *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.

As a direct benefit, 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 using a designated algorithm *genKey*. We can assign to keys the refinement type k : bytes { *Return*(*genKey*,(),k)}. This pattern frequently applies to various cryptographic materials, such as nonces, initialization vectors, and tags.

Code transformation We specify this semantics by replacing every syntactic function using the translation:

$$[\![\mathit{rec}\ f\colon T.\ \mathbf{fun}\ x \to e]\!]_E = \mathit{rec}\ f\colon T.\ \mathbf{fun}\ x \to \\ \mathbf{assume}\ \mathit{Call}(f,x); \\ \mathbf{let}\ r = [\![e]\!]_E\ \mathbf{in} \\ \mathbf{assume}\ \mathit{Return}(f,x,r); \\ r$$

where $[\![]\!]_E$ is 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 injecting formulas during typechecking, modifying the rule (Typ Fun). We call the modified typing system RCF_E

Results Let e be a closed program. We check that our transformation does not affect the operational behaviour, safety and well-typedness of programs that do not use *Call* and *Return*, and that the code transformation and the modified typing rule yield the same typing judgments.

Lemma 1. *Suppose that Call and Return do not occur in e.*

- Evaluation: *For any value M, $e \longrightarrow^* M$, if and only if $[\![e]\!]_E \longrightarrow^* [\![M]\!]_E$;*
- Safety: *e is safe if and only if $[\![e]\!]_E$ is safe; and*
- Typing: *e is well-typed in RCF if and only if $[\![e]\!]_E$ is well-typed in RCF.*

Lemma 2. *$[\![e]\!]_E$ is well-typed in RCF if and only if e is well-typed in RCF_E.*

2.2 Macro-expansion semantics

Pre- and post-condition 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 $\#\mathit{Pre}$ and $\#\mathit{Post}$. They may occur anywhere in the program or its interface, provided that their first argument is a variable name that has a declared function type in the scope of their occurrence. Then *before typechecking* we can always safely replace this occurrence with the concrete formula read off its variable type.

Implementation For any values f, M, N in the environment E, if $E(f) = x\colon T\ \{C\} \to y\colon T'\{C'\}$, then we replace any occurrence of $\#\mathit{Pre}(f,M)$ with $C[M/x]$, and any occurrence of $\#\mathit{Post}(f,M,N)$ with $C'[M/x][N/y]$. If the lookup fails, or the returned type is not a function type, the preprocessing fails—the macro-definition is ill-formed.

2.3 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 $x{:}(x{:}P\{\mathit{Pre}(f,x)\}) \to y{:}P'\{\mathit{Post}(f,x,y)\}$.
- we logically relate them to the actual refinements at each call site: as f is instantiated to some function g of type $x{:}(x{:}P\{C\}) \to y{:}P'\{C'\}$, to type the call site, by subtyping we have two proof obligations:

  $\forall x.\ C \Rightarrow \mathit{Pre}(f,x)$
  $\forall x,y.\ \mathit{Post}(f,x,y) \Rightarrow C'$

  which we will be automatically assumed.

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 $\mathit{Call}(f,M)$ records this application, and typing requires that the predicate $\mathit{Pre}(f,M)$ holds. At runtime, for each instance g of f, the actual pre-condition of g holds (by typing) and implies the formal precondition of f (by assumption) so we always have

$\forall x.\ \mathit{Call}(f,x) \Rightarrow \mathit{Pre}(f,x)$

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

$\forall x,y.\ \mathit{Return}(f,x,y) \Rightarrow \mathit{Post}(f,x,y)$

Code transformation To support Pre and 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:

$$
\begin{array}{l}
[\![\ \textbf{let}\ f = e : (f : (x_1{:}P_1\{C_1\} \to x_2{:}P_2\{C_2\})\ \{C_f\})\ \textbf{in}\ e'\ ]\!]_S = \\
\quad \textbf{let}\ f = [\![e]\!]_S\ \textbf{in} \\
\quad \textbf{assume}\ \forall x_1.\ C_1 \Rightarrow \mathit{Pre}(f,x_1); \\
\quad \textbf{assume}\ \forall x_1,x_2.\ \mathit{Post}(f,x_1,x_2) \Rightarrow C_2; \\
\quad \textbf{assume}\ \forall x_1,x_2.\ \mathit{Return}(f,x_1,x_2) \Rightarrow \mathit{Post}(f,x_1,x_2); \\
\quad [\![e']\!]_S \\
[\![\ \textbf{rec}\ f{:}\ x{:}T1 \to T2.\ \textbf{fun}\ x \to\ e\ ]\!]_S = \\
\quad \textbf{rec}\ f{:}\ x{:}T1 \to T2.\ \textbf{fun}\ x \to [\![\textbf{let}\ x = (x : T1)\ \textbf{in}\ e]\!]_S
\end{array}
$$

where $[\![]\!]_S$ 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 synctatic function definition, ensuring that all functional arguments are annotated in higher-order functions.

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

$$E \oplus x : T \triangleq E, x : T, \{\forall x_1.\ C_1 \Rightarrow \mathit{Pre}(x,x_1)\}, \{\forall x_1,x_2.\ \mathit{Post}(x,x_1,x_2) \Rightarrow C_2\}, \{\forall x_1,x_2.\ \mathit{Return}(x,x_1,x_2) \Rightarrow \mathit{Post}(x,x_1,x_2)\}$$

Otherwise $E \oplus x : T$ is just $E, x : T$. We call the modified type system RCF_S.

Restrictions on the use of Pre/Post To take advantage of Pre and Post while preserving their consistency, we rely on a standard notion of positive and negative positions in types and formulas. Hence, for instance, assumed formulas are positive, while asserted formulas are negative. We require that programmers use Pre only in negative positions, and use Post only in positive positions.

Results We obtain a variant of Lemma 1 for the subtyping semantics: we have a similar Evaluation property, but for Safety and Typability we only have the direct implication (since the injected assumes and asserts depend on the type annotations of the functions).

We also prove two flavours of Correctness: we have a variant of Lemma 2 that relates typing with RCF_S and the specification $[\![]\!]_S$. Besides, using another program transformation, the lemma below confirms that *Pre* and *Post* can be eliminated by inlining the code of higher-order functions at each call site and annotating them with ad hoc types. We omit the full definition of the corresponding code transformation, $\langle\rangle$ (details can be found in the online companion paper).

Lemma 3 (Correctness). *Let e be an expression, T_R a type that contains neither Pre nor Post, and E a typing environment where Pre occurs only negatively and Post occurs only positively.*

If $E \vdash e : T_R$ using RCF_S, then $E \vdash \langle e \rangle : T_R$ using RCF_E.

3 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*.

$$A \longrightarrow B : m \mid (\mathtt{mac}\ k_{AB}\ m)$$

(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:

```
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;
     m
```

The *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 *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 *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 *send*, *recv* on any TCP connection to intercept and interject a message of his choice. He may construct and verify MACs by calling *hmac* and *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 *server* function returns a message m when called with a, b, and a key k generated by the *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:

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

```
val server:
    a:str → b:str →
    k:key{Return(mkKey,[a;b],k)} →
    m:bytespub
    {Call(client,[a;b;k;m]) ∨ Pub(k)}
```

To verify that the code actually meets these types, we rely on the unforgeability of MACs, expressed as types for the cryptographic library:

```
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:
    a:str → b:str →
    k:key{MKey(k)} →
    m:bytes →
    h:bytes →
    unit{MACSays(k,m) ∨ Pub(k)}
```

Note that *hmac* has as precondition a predicate *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 type of *hmac* then checks that *k* is allowed to MAC *m*, and the type of *hmac_verify* says 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*:

assume $\forall a,b,k.\ \mathit{Return}(\mathit{mkKey},[a;b],k) \Rightarrow (\mathit{MACSays}(k,m) \Leftrightarrow \mathit{Call}(\mathit{client},[a;b;k;m]))$

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 Any number of symbolic verification tools can verify the simple protocol code written here. Tools such as ProVerif [Blanchet, 2001] can even automatically infer the logical assumption on *MACSays*, thus requiring fewer annotations than our method. However, as we shall see in Section 5, verifying complex protocols that manipulate flexible message formats requires a form of induction that is beyond the reach of tools such as ProVerif.

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.

4 Example: A Reusable Typed Interface for *List*

Lists are perhaps the most commonly-used data structures in functional programs. The F# *List* library provides efficient implementations of several recursive list processing functions, and for maximum reusability, 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.

In this section, we detail our approach on the function*List.fold*, the general iterator on lists (also called *fold_left*). Its ML type is **val** *fold*: $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathit{list} \to \alpha$. 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, *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:

```
let rec                          assume ∀f,acc,l.                 assume ∀f,acc,l,r.
  fold f acc l =                 PreFold(f,acc,l)                 PostFold(f,acc,l,r)
  match l with                   ⇔                                ⇔
   | [] → acc                     (l=[]                            ((l=[] ∧ r=acc)
                                  ∨                                ∨
   | hd :: tl →                   (∃hd,tl. l=hd::tl ∧              (∃hd,tl. l=hd::tl ∧
     let acc' = f acc hd in       Pre(f,[acc;hd]) ∧
                                  (∀acc'. Post(f,[acc;hd],acc')    (∃acc'. Post(f,[acc;hd],acc')
     fold f acc' tl                     ⇒ PreFold(f,acc',tl))))    ∧ PostFold(f,acc',tl,r))))
```

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, *PreFold* must hold for the tail of the list and this new accumulator. *PostFold* is defined similarly.

The resulting type for *fold*:

val *fold*: $f{:}(\alpha \to \beta \to \alpha) \to acc{:}\alpha \to l{:}\beta\ \ list\{PreFold(f,acc,l)\} \to r{:}\alpha\ \{PostFold(f,acc,l,r)\}$

is precise and easy to typecheck against the code of *fold*, yet difficult to use at call sites. Indeed, even for a function with no pre-condition ($\forall x.\ Pre(f,x)$), proving $PreFold(f,acc,l)$ requires the use of induction, which is generally beyond the reach of the SMT solver Z3 that underlies F7. For example, even the following simple functions cannot be typechecked with the above type for *fold*:

```
let idfold: int → int→ int = fun a → fun b → a
let testidfold: int list → int = fun l → fold idfold 0 l
```

Only by adding (and proving by hand) an assumption that functions with no pre-conditions can always be used with *fold* can this code be typechecked.

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 *fold*. We introduce a generic predicate *Inv* that is used to define logical *invariants* for functions that may be used as an argument to *fold*. The formula $Inv(f,aux,acc,l)$ is an invariant that holds when the function f is being applied to a list of elements: l is the remainder of the list, acc is the intermediate result of the computation, and aux contains function-specific auxiliary information about the initial arguments to the *fold*.

As an example, consider the function *fmem* that can be used with *fold* to search for an element in a list; *fmem* takes an element v to search for, an accumulator acc and an integer n, and returns true if either acc is true or if $v = n$. The invariant for the partial application *fmem v* is shown below; its auxiliary argument 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 *linit*. The invariant says that the remaining list l contains a subset of the elements in *linit*, and that the accumulator is true only if v is a member of *linit*.

```
let fmem v                val fmem: v:α →          (∀v,f. Post(fmem,v,f) ⇒
    acc                     acc:bool →               (∀iv,acc,l. Inv(f,iv,acc,l) ⇔
    n                       n:α →                    (∃x,linit. iv=(x,linit) ∧ x = v
    =                       found:bool{              ∧ (∀y. Mem(y,l) ⇒ Mem(y,linit))
  if v = n                  (v = n                   ∧ (( Mem(x,linit)
  then true                   ∧ found = true)                  ∧ acc=true)
  else acc                  ∨ (found = acc)}               ∨ acc = false ))))
```

The next step, following Régis-Gianas and Pottier [2008], is to prove that the invariant is *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 *Hereditary* that captures this notion and use it to type *fold* as follows; to use this style we need to add an additional argument to *fold* that holds the auxiliary values needed to maintain its invariants.

```
let rec fold v f acc l =          assume (∀f. Hereditary(f)
  match l with                      ⇔
    | [] → acc
    | hd :: tl →                    ( ∀v,acc,h,t. Inv(f,v,acc,hd::tl) ⇒
      let acc' = f acc hd in        (Pre(f,[acc;hd])
      fold v f acc' tl              ∧ (∀r. Post(f,[acc;hd],r) ⇒ Inv(f,v,r,tl)))))
```

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

The type of *List.fold* then 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 *mem*:

```
val mem: v:α → l: α list → b:bool { b=true ⇒ Mem(v,l)}
let mem v l = let f = fmem v in fold (v,l) f false l
```

we must prove that $\forall v,f.\ Post(fmem,v,f) \Rightarrow Hereditary(f)$, and that the invariant of *fmem v* holds for the initial values **false**,*l*. For a simple function like *fmem* this can be proved automatically, but for more complex functions *Hereditary* may have to be proved by hand. The rest of the typechecking is fully automatic.

5 Case Studies: Cryptographic Protocol Implementations

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

5.1 XML Digital Signatures

The XML digital signature standard specified cryptographic mechanisms that are designed 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 involved tree and list processing. For example, consider the following single message protocol, where the principal a uses an XML signature to protect $n \geq 1$ XML elements $m_1, \ldots, m_n$ located at URIs #1, ..., #n within the message, using the MAC key k_{ab}. A slightly simplified version of this protocol is as follows:

$$
\begin{array}{l}
A \longrightarrow B : \langle \texttt{Message} \rangle \\
\quad \langle \texttt{Signature} \rangle \ \texttt{base64}\ (m_1 \mid (\texttt{mac}\ k_{AB}\ m_1))\ \langle / \texttt{Signature} \rangle \\
\quad \ldots \\
\quad \langle \texttt{Signature} \rangle \ \texttt{base64}\ (m_n \mid (\texttt{mac}\ k_{AB}\ m_n))\ \langle / \texttt{Signature} \rangle \\
\langle / \texttt{Message} \rangle
\end{array}
$$

The security goal for this protocol is simply that each m_i is individually authenticated. Since, the different messages in the list are not correlated with each other, the server cannot authenticate the list as a whole, but this protocol may be used as a component within a larger protocol that enforces a more demanding security property.

Using *List.map* we can program and verify the server processing code:

```
val xml_mac_verify:
  a:str → b:str →
  k:key{Return(mkKey,[a;b],k)} →
  mh:item →
  m:item{(∃ml'. Mem(m,ml') ∧
   Call(client,[a;b;k;ml'])) ∨ Pub(k)}

let server a b k =
  let c = Net.listen p in
  let w = Net.recv c in
  let mhl = fromXml w in
  let mvf = xml_mac_verify a b k in
  let ml = List.map mvf mhl in
   ml
```

```
val server:
  a:str → b:str →
  k:key{Return(mkKey,[a;b],k)} →
  ml:itemlist{∀m. Mem(m,ml) ⇒
 (∃ml'. Mem(m,ml') ∧
   Call(client,[a;b;k;ml'])) ∨ Pub(k)}

assume ∀a,b,k.
  Return(mkKey,[a;b],k) ⇒
    (MACSays(k,m) ⇔
    (∃ml'. Call(client,[a;b;k;ml']) ∧
           Mem(m,ml'))
```

The function *xml_mac_verify* parses an XML signature, extracts the message and its MAC, and verifies the MAC. Its post-condition guarantees that the message must have been sent by a valid client (as part of an authenticated message). The type of the server function says that every message accepted by the server must have been sent by the client as part of some authenticated message sent to the server.

The use of *List.map* here avoids the need to inline the recursive code for *map* in the code for *server*. In an earlier verification of XML digital signatures using F7, there were four instances where we needed to inline list-processing functions and define new type annotations. These are no longer necessary reducing the annotation burden by roughly one-fourth.

5.2 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 *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:

$$\begin{aligned} A \longrightarrow B : \; & \texttt{Certificate}(A_1 \mid pk_{A_1} \mid \texttt{rsa\_sign}\ sk_{CA}\ (A_1 \mid pk_{A_1})) \\ & \texttt{Certificate}(A_2 \mid pk_{A_2} \mid \texttt{rsa\_sign}\ sk_{A_1}\ (A_2 \mid pk_{A_2})) \\ & \ldots \\ & \texttt{Certificate}(A \mid pk_A \mid \texttt{rsa\_sign}\ sk_{A_{n-1}}\ (A \mid pk_A)) \end{aligned}$$

The code for verifying such certification chains uses *List.fold* to chain together the results of verifying each step in the path.

```
val verify:
    x:cert{Certificate(x)} →
    b:bytes →
    r:cert {Certifies(x,r)
        ∧ Certificate(r)}
let verify_all c path =
    fold2 c verify c path
```

```
assume ∀ca,x,h,l.
    Inv(verify,ca,x,l) ⇔
    (Certificate(x) ∧
     Certifies(ca,x))
val verify_all:
    x:cert{Certificate(x)} →
    l: bytes list →
    r:cert {Certifies(x,r)}
```

The predicate *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 *verify_all* takes as an argument a certificate *ca* for a trusted certification authority and it accepts only those certification paths that begin

with certificates issued with *ca*'s public-key. To typecheck *verify_all* we define the *fold* invariant for *verify* as the property that the accumulator x always has a valid certificate (*Certificate*(x)) and that there is a valid path from the initial certificate *ca* to x (*Certifies*(*ca*,x)).

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

6 Related work

Introduced by the seminal work of Hoare [1969], the pre- and post-conditions have been implemented by several extended static checking 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, and even otherwise, their whole-program analysis seldom scales very well.

The RCF type system is the first to use refinement typing for security analysis. It has already being successfully used to verify complex cryptographic applications [Backes et al., 2009, Bhargavan et al., 2009, Guts et al., 2009]. 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 the formulas for security typing are typically more complex. As future work, we hope to reuse Liquid Types to automatically infer as many annotations as possible for our examples as well.

References

M. Backes, C. Hriţcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. In *Workshop on Foundations of Computer Security*, 2009.

M. Barnett, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'05*, volume 3362, pages 49–69, January 2005.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, pages 17–32, 2008.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31:5:1–5:61, December 2008.

K. Bhargavan, R. Corin, P. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*, 2009.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, pages 445–456, 2010.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF'09*, 2009.

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/.

M. Fähndrich, M. Barnett, and F. Logozzo. Embedded Contract Languages. In *SAC OOPS*, 2010.

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.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, pages 363–379, 2005.

N. Guts, C. Fournet, and F. Z. Nardelli. Reliable evidence: Auditability by typing. In *ESORICS*, 2009.

C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. ITU-T, June 1997.

Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, July 2008. URL http://gallium.inria.fr/ fpottier/publis/regis-gianas-pottier-hoarefp.ps.gz.

P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*. ACM, 2008. To appear.

D. N. Xu. Extended static checking for Haskell. In *ACM SIGPLAN workshop on Haskell (Haskell'06)*, pages 48–59. ACM, 2006.

A Refinement types for ML (review)

We recall the syntax and semantics of F7; we refer to Bengtson et al. [2008] for a full description of the F7 typechecker and its underlying calculus RCF.

Syntax for F7

$M, N ::=$	Values	$C ::=$	Formulas
$()$		$P(M_1, \ldots, M_n)$	
x		$M_1 = M_2 \mid M_1 \neq M_2$	
(M, N)		$True \mid False$	
$h(M)$		$not\ C \mid C \Rightarrow C \mid C \Leftrightarrow C$	
$\mathbf{rec}\ f : T.\mathbf{fun}\ x \to e$		$C \wedge C \mid C \vee C$	
$e ::=$	Expressions	$\forall x.C \mid \exists x.C$	
M		$P ::=$	Pretypes
$M\ N$		unit	unit type
$\mathbf{let}\ p = e_1\ \mathbf{in}\ e_2\ \mathbf{else}\ e_3$		$x : T_1 \to T_2$	function type
$\mathbf{assume}\ C$		α	type constructor
$\mathbf{assert}\ C$		$\Sigma_i(h_i : T_i \to \alpha)$	ADT
$e : T$		$\mu\alpha.P$	iso-recursive type
$p ::=$	Patterns	$T, U, V ::=$	Refined Types
$() \mid x \mid (p_1, p_2) \mid h(p)$		$(x : P)\{C\}$	(x is bound in C)

Values include unit, variables, constructed values, dependent pairs, and (recursive) dependent functions. Recursive functions require type annotations; in standard RCF they can be encoded using iso-recursive types. Constructors h range over algebraic type constructors with fixed arities.

Formulas include predicates over values (re-using constructors as predicate symbols), value equalities, and standard first-order logic constructs.

Expressions e are in A-normal form: they can be values, function applications of values, assuming or asserting a formula, pattern-matching let binding of expressions, or type-annotated expressions. An assert *succeeds* if the asserted formula can be logically derived from the conjunction of all previously-assumed formulas. An expression is *safe* when all of its **assert**s succeed in every run.

Patterns (p) include constructor applications, pairs, variables and unit.

Refined types are ML-like pretypes refined with value-dependent formulas. Compared to RCF, our types also include algebraic sum types: $\Sigma_i(h_i : T_i \to \alpha)$ which relates possible constructors h_i and their argument types T_i to the type of the constructed value α. In iso-recursive types $\mu\alpha.P$, the type variable α is bound within the pretype P.

A type $x{:}P\{C\}$ is the type of expressions that return values M of type P such that the formula $C[M/x]$ follows from the assumed formulas. Function types have refinements on their arguments and on their return value: the argument of a function with type $x{:}(x{:}P\{C\}) \to y{:}P'\{C'\}$ have type P and satisfy formula C, while its return value has type P' and satisfies the formula C'. Refined types can be erased to plain ML (pre)types.

We denote by E typing environments which include bindings of variables to types $x : T$, assumed formulas F, and type definitions. We use the following judgments:

$E \vdash C$	formula C holds in the environment E
$E \vdash \diamond$	the environment E is well-formed
$E \vdash x : T$	x has type T in the environment E
$E \vdash T <: T'$	T is a subtype of T' in the environment E

Preparing the ground for our extension of the typechecker in Section 2, we recall F7 typing rules for functions and function applications.

$$\frac{E \vdash x : T_1 \to T_2 <: T \qquad E, f : T, x : T_1 \vdash e : T_2}{E \vdash \mathbf{rec}\ f : T.(\mathbf{fun}\ x \to e)\ :\ x : T_1 \to T_2} \qquad \frac{E \vdash M : x : T \to T' \qquad E \vdash N : T_a \qquad E \vdash T_a <: T}{E \vdash (M\ N) : T'}$$

A recursive function has type $x : T_1 \to T_2$ 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' if the value M is a function with type $x : T \to T'$ and the value N has a type which is a subtype of T.

Typechecking by F7 relies on the main result of Bengtson et al. [2008]: if a program is well-typed, then it is safe. Also, if a program is well-typed in an empty environment, then the program is safe when applied to any expression with no occurrence of **assert** (such expressions model malicious active attackers).

F7 implementation Our prototype typechecker F7 is an extension of RCF that supports a subset of F#. In particular, it supports programs that contain type- and value- parametered types, records, polymorphism, mutual recursion, match expressions and references.

The typechecker takes two sorts of input files

- pure F# implementation files that contain program, possibly annotated with F# types (file.fs)
- RCF interfaces containing full refinement types, definitions and formulas(file.fs7)

All extended types and formulas needed in F# can be moved to the interface by defining additional definitions.

The typechecker can verify if an implementation is well-typed against an interface or erase an interface into a plain F# type. To check validity of typing constraints, the typechecker either establishes it internally for trivial cases, or marshals it into Simplify format and calls the Z3 theorem prover (which can fail, either because the query is unsatisfiable, or because Z3 is incomplete).

RCF$_E$ Below we show the rule that replaces the rule (Typ Fun) in RCF$_E$.

$$\frac{x \notin dom(E) \qquad E \vdash x : T_1 \to T_2 <: T \qquad E, f : T, x : T_1, Call(f, x) \vdash e : (z : P)\{C\} \qquad T_2 = (z : P)\{C \wedge Return(f, x, z)\}}{E \vdash (recf : T.\mathbf{fun}\ x \to e) : x : T_1 \to T_2}$$

B List Library Interface

assume
$(\forall x,\ u.\ Mem(x,x::u)) \wedge$
$(\forall x,\ y,\ u.\ Mem(x,u) \Rightarrow Mem(x,y::u)) \wedge$
$(\forall x,\ u.\ Mem(x,u) \Rightarrow (\exists y,\ v.\ u = y::v \wedge (x = y \vee Mem(x,v))))$

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

val $find$: $f{:}(\alpha \rightarrow bool) \rightarrow$
$u{:}\alpha\ list\{(\forall x.\ Mem(x,u) \Rightarrow Pre(f,x))\} \rightarrow$
$r{:}\alpha\{\ Mem(r,u)\ \}$

val $forall$: $t{:}(\alpha \rightarrow bool) \rightarrow$
$xs{:}\alpha\ list\ \{(\forall x.\ Mem(x,xs) \Rightarrow Pre(t,x))\} \rightarrow$
$b{:}bool\ \{(b = \mathbf{true} \Rightarrow (\forall x.\ Mem(x,xs) \Rightarrow \mathrm{Post}(t,[x],\mathbf{true})))\ \}$

val $exists$: $t{:}(\alpha \rightarrow bool) \rightarrow$
$xs{:}\alpha\ list\ \{(\forall x.\ Mem(x,xs) \Rightarrow Pre(t,x))\} \rightarrow$
$b{:}bool\ \{(b = \mathbf{true} \Rightarrow (\exists x.\ Mem(x,xs) \wedge \mathrm{Post}(t,[x],\mathbf{true})))\ \}$

val $iter$: $f{:}(\alpha \rightarrow \mathrm{unit}) \rightarrow$
$l{:}\alpha\ list\ \{(\forall x.\ Mem(x,l) \Rightarrow \mathrm{Pre}(f,[x]))\} \rightarrow$
$r{:}\mathrm{unit}\ \{\forall x.\ Mem(x,l) \Rightarrow \mathrm{Post}(f,[x],())\}$

assume
$(\forall x,\ y,\ u,\ v.\ Mem2((x,y),(x::u,x::v)) \wedge$
$(\forall x,\ y,\ u,\ v,\ x',\ y'.\ Mem2((x,y),(u,b)) \Rightarrow Mem2((x,y),(x'::u,y'::v))) \wedge$
$(\forall x,\ y,\ u,\ v.\ Mem((x,y),(u,v)) \Rightarrow (\exists y1,y2,t1,t2.\ l1{=}y1::t1 \wedge l2{=}y2::t2$
$\wedge ((y1{=}x \wedge y2{=}y) \vee Mem2((x,y),(t1,t2))))))$

val map: $f{:}(\alpha \rightarrow \beta) \rightarrow$
$l{:}\alpha\ list\ \{(\forall x.\ Mem(x,l) \Rightarrow \mathrm{Pre}(f,[x]))\} \rightarrow$
$r{:}\beta\ list\ \{\forall x,y.\ Mem2((x,y),(l,r)) \Rightarrow \mathrm{Post}(f,[x],y)\}$

assume $(\forall f.\ Hereditary(f)$
$\Leftrightarrow$
$(\ \forall v,acc,h,t.\ Inv(f,v,acc,hd::tl) \Rightarrow$
$(\mathrm{Pre}(f,[acc;hd])$
$\wedge (\forall r.\ \mathrm{Post}(f,[acc;hd],r) \Rightarrow Inv(f,v,r,tl)))))$

val $fold$: $v{:}\ \gamma \rightarrow f{:}(\alpha \rightarrow \beta \rightarrow \alpha)\ \{Hereditary(f)\} \rightarrow$
$acc{:}\alpha \rightarrow$
$xs{:}\beta\ list\ \{Inv(f,v,acc,xs)\} \rightarrow$
$r{:}\alpha\ \{\ (xs = [] \wedge r{=}acc) \vee Inv(f,v,r,[])\ \}$