Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Programme in Computer Science

Bachelor's Thesis

Spi2F# – A Prototype Code Generator for Security Protocols

submitted by

Thorsten Tarrach
<thorstent@live.com>

on 2008-10-28

Supervisor
Prof. Dr. Michael Backes

Advisor
Cătălin Hriţcu

Reviewers
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 2008-10-28

Thorsten Tarrach

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, 2008-10-28

Thorsten Tarrach

Abstract

This thesis describes a new prototype tool that automatically generates a secure F# implementation of any protocol described in the Spi calculus. Type systems were previously proposed for analysing the security of both Spi calculus processes and F# implementations. The thesis investigates a formal translation from the Spi calculus to F# that is proved to preserve typability, and therefore the security properties of the original protocol are preserved.

Acknowledgements

I am profoundly indebted to my advisor, Cătălin Hriţcu, for his continuous support during the last months. He helped me on uncounted occasions understanding the papers, taught me the way to prove the translations and helped me with the typesetting of this document.

I would like to thank Michael Backes and Matteo Maffei for reviewing this thesis.

> "Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest." – Isaac Asimov

Contents

1 Introduction

This thesis connects the verification of protocol models and protocol implementations and aims at automatically translating protocols from an abstract calculus to source code in a programming language that can be compiled and run. The translation is proved to preserve the security properties of the original Spi protocol. I have chosen the Spi calculus [Aba99, AB05] as the source for my translation and F# as the target.

1.1 Previous Work

To date, most protocols are defined informally, which can easily lead to security vulnerabilities. Even standardised protocols, that went through thorough peer review, have been shown to contain security vulnerabilities, e.g. in [Low96, WS96, Ble98, Fis03, BCJ+06]. However, once a model of a protocol is formalised one can use many automated tools, such as ProVerif [Bla05], and type systems such as [Aba99, AB01, GJ04, HJ06, BFM07, BCFM07], to verify security properties like secrecy and authenticity.

Some of the techniques from protocol verification have recently been adopted to implementations in real programming languages. While some research is done on low-level languages like C [GLRV05] I will focus on the higher-level language F# for which a tool extracting ProVerif models [BFGT06] as well as a typechecker [BBF+08] exist.

1.2 Usefulness of the translation

To run a protocol formalised in Spi one would either need an interpreter or need to translate it into another programming language. An interpreter would be of little use as most protocols are used as part of a bigger application and this application would have to be written in an actual programming language.

A manual implementation leaves the risk of bugs in the implementation creating security holes in the otherwise secure protocol. This recently became obvious in [Deb08], where the SSL protocol, that is considered secure, was made vulnerable by a poor implementation of the random number generator. One could avoid certain bugs in the implementation phase by using F# as an implementation language and then using a typechecker [BBF+08] to verify security properties, however as protocols or their implementation change they are harder to keep in sync manually. Writing protocols in Spi rather than directly in F#

also has the advantage that Spi is much more abstract and one can focus on the protocol itself rather than implementation details. Currently this is mostly interesting for protocol designers and rapid prototyping. In a stable version of the code generator it could be part of the build process and save developers time. Just as a parser can be generated using yacc [Joh78] a protocol implementation could be created using a code generator.

The automatic generation of F# code is preferable. Even though the implementation of the code generator itself is hard to verify, the resulting code can be checked using the F7 typechecker for F#. This makes F# a more suitable target language at the moment than Java for example.

1.3 Contributions

This thesis contributes to secure protocol implementations by formally defining a translation from Spi protocols to F# implementations. This translation is proved to guarantee that every protocol that typechecks in Spi will also typecheck in F# after the translation. The fact that the protocol typechecks in F# ensures that secrecy and authentication properties of the model carry over to the generated implementation. The generated code therefore has the same security properties as the original Spi protocol. I also implemented a prototype code generator that performs said translation automatically. This tool produces code that can be verified using the typechecker from [BBF$^+$08] as well as compiled and run with some limitations.

Defining such a translation was challenging, but the task was eased by the fact that the target language is a simple and well-defined core calculus. The fact that the two type systems are similar made proving this translation to preserve typability easier. Proving that typability is preserved allowed me to show that security properties are preserved without having to prove the translation functionally correct. For the implementation I took advantage of the infrastructure provided by F# and F7, especially useful were the libraries that already provide a functional interface to the .net cryptographic functions.

1.4 Overview

Chapter 2 presents an extensible variant of the Spi calculus which constitutes the source language of my translation. Chapter 3 presents the target of my translation, which is named RCF and is a core calculus of F#. Chapter 4 defines a translation from Spi to a normal form which is needed as an intermediate step. Chapter 5 gives the translation from Spi to RCF and Section 5.4 proves this translation to preserve typablitiy. Chapter 6 gives an overview of the implementation of the code generator. Chapter 7 summarises and concludes.

2 The Spi calculus

In this thesis I consider an extensible variant of the Spi calculus in which new types, constructors and destructors can be easily added. It was introduced in [AB05] and the type system was later extended in [FGM07]. I use the calculus and the type system to statically enforce authorisation policies on protocols written in Spi. The presentation of the calculus and type system below closely follows [BHM08]. Some of the typing rules around encryption and decryption have to be weakened to make a translation to RCF possible.

2.1 Calculus

This section introduces the extensible Spi calculus from [AB05]. Table 2.1 introduces the syntax of terms, while Table 2.2 introduces the syntax of processes. Their semantics are given in [BHM08], but it is worth noting that the out$(M, N).P$ process is synchronous in this calculus, which means that P is only executed once the message on the channel is received. The let $x = M$ in P construct was added by me and gives names to arbitrary terms and is needed in Chapter 4.

Tables 2.3 and 2.4 list the constructors and destructors that will be used throughout this thesis. Constructors are function symbols that are used to build terms. Destructors are partial functions that can be applied to terms using the let process. In case the destructor is not defined for the argument the let process has an else case.

Table 2.1 Syntax of terms

K, L, M, N ::=		terms
	a, b, c, m, n, k	names
	x, y, z, v, w	variables
	$\langle M_1, \ldots, M_n \rangle$	tuple
	$f(M_1, \ldots, M_n)$	constructor application (f of arity n)

Table 2.2 Syntax of processes

P, Q, R	::=		processes
		$\mathsf{out}(M, N).P$	output
		$\mathsf{in}(M, x).P$	input
		$!\mathsf{in}(M, x).P$	replicated input
		$\mathsf{new}\ a : T.P$	restriction
		$P \mid Q$	parallel composition
		$\mathbf{0}$	null process
		$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$	destructor evaluation
		$\mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P$	pair splitting
		$\mathsf{let}\ x = M\ \mathsf{in}\ P$	let binding
		$\mathsf{assume}(C)$	assume formula
		$\mathsf{assert}(C)$	expect formula to hold

Table 2.3 Constructors

I use the following constructors:

pk^1	returns the corresponding public key for a private key
enc^2	encrypts a message using a public key
vk^1	returns the corresponding verification key for a signing key
sign^2	signs a message using a signing key

Table 2.4 Destructors

I use the following destructors:

eq^2	equality between two types
dec^2	decrypts a message using a private key
check^2	checks a signed message using a verification key

Their semantic is defined as follows:

$$\mathsf{eq}(M, M) \Downarrow \mathsf{true}$$
$$\mathsf{dec}(\mathsf{enc}(M, \mathsf{pk}(K)), K) \Downarrow M$$
$$\mathsf{check}(\mathsf{sign}(M, K), \mathsf{vk}(K)) \Downarrow M$$

2.2 Type system

This section presents the type system for the above calculus. There are some minor differences compared to [BHM08], which are reviewed in Subsection 2.2.2.

Table 2.5 lists the types of the type system, while Table 2.6 lists the typing judgements. The type system uses a typing environment that contains name and variable bindings as well as formulas. The well-formed environment judgement is defined in Table 2.7. The kinding rules are given in Table 2.8, and the subtyping rules in Table 2.9. Kinding rules specify

whether a type is public, tainted or both. Un is a supertype of every public types and a subtype of every tainted type. A type, that is both public and tainted is equivalent to Un.

Tables 2.10 and 2.11 are devoted to the types of the constructors and destructors. Table 2.12 defines the term typing judgement. Table 2.14 lists the rules for typing processes, which are defined using the environment extraction relation (Table 2.13). The Rule (Proc-Let) was added by me to type the newly added let $x = M$ in P construct.

2.2.1 Robust safety

The main property of the type system is robust safety:

Definition 2.2.1 (Safety). *A closed process P is* safe *if and only if for every C and Q such that $P \rightarrow^* \text{new } \widetilde{a} : \widetilde{T}.(\text{assert } C \mid Q)$, there exists an evaluation context $\mathcal{E} = \text{new } \widetilde{b}\widetilde{U} : .[\,] \mid Q'$ such that $Q \equiv \mathcal{E}[\text{assume } C_1 \mid \ldots \mid \text{assume } C_n]$, $fn(C) \cap \widetilde{b} = \emptyset$, and we have that $\{C_1, \ldots, C_n\} \models C$.*

Definition 2.2.2 (Opponent). *A closed process is an* opponent *if it does not contain any* assert *and if the only type occurring therein is* Un*.*

Definition 2.2.3 (Robust Safety). *A closed process P is* robustly safe *if and only if $P \mid O$ is safe for every opponent O.*

Theorem 2.2.4 (Robust Safety). *For every closed process P, if $\Gamma \vdash_{\mathsf{Un}} P$ then P is robustly safe.*

Proof. The proof is given in [BHM08]. □

2.2.2 Differences with respect to [BHM08]

To make a translation to RCF possible I need to weaken the type system from [BHM08] as follows:

- The types of zero-knowledge proofs have been removed, as well as all related rules, constructors and destructors.
- The types Signed and PubEnc have been removed, along with their rules. In the respective constructors and destructors these types have been replaced with Un.
- The kinding rules for SigKey, VerKey, PubKey and PrivKey have been weakened and need more preconditions.

This was necessary because the RCF calculus has no types for zero-knowledge and encryption and signing are expressed through seals. The typing rules in RCF for seals are weaker than the rules for cryptographic types in [BHM08].

Table 2.5 Syntax of types

T,U ::=	types
Un	
Private	
Ch(T)	
SigKey(T)	
VerKey(T)	
PrivKey(T)	
PubKey(T)	
$\langle \widetilde{x} : \widetilde{T} \rangle \{C\}$	

Table 2.6 Typing Judgements

$\Gamma \vdash \diamond$	well-formed environment
$\Gamma \vdash T :: k$	kinding, $k \in \{\mathsf{pub}, \mathsf{tnt}\}$
$\Gamma \vdash T <: U$	subtyping
$f : (T_1, \ldots, T_n) \mapsto T$	constructor typing
$g : (T_1, \ldots, T_n) \mapsto T$	destructor typing
$\Gamma \vdash M : T$	term typing
$\Gamma \vdash P$	well-typed process

Notation: $\Gamma \vdash \mathcal{J}$ is used to denote a judgement where $\mathcal{J} \in \{\diamond, T :: k, T <: U, M : T, P\}$

Table 2.7 Well-formed environment $\Gamma \vdash \diamond$

ENV-EMPTY
$$\emptyset \vdash \diamond$$

ENV-FORMULA
$$\frac{\Gamma \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(\Gamma)}{\Gamma, C \vdash \diamond}$$

ENV-BINDING
$$\frac{\Gamma \vdash \diamond \qquad u \notin \mathit{dom}(\Gamma) \qquad \mathit{free}(T) \subseteq \mathit{dom}(\Gamma)}{\Gamma, u : T \vdash \diamond}$$

Definition: $\mathit{dom}(\emptyset) = \emptyset$; $\mathit{dom}(\Gamma, C) = \mathit{dom}(\Gamma)$; $\mathit{dom}(\Gamma, u : T) = \mathit{dom}(\Gamma) \cup \{u\}$

Definition: $\mathsf{forms}(\emptyset) = \emptyset$; $\mathsf{forms}(\Gamma, C) = \mathsf{forms}(\Gamma) \cup \{C\}$; $\mathsf{forms}(\Gamma, u : T) = \mathsf{forms}(\Gamma)$

Definition: $\mathit{fn}(\phi)$ denotes the set of free names in any phrase ϕ, $\mathit{fv}(\phi)$ the set of free variables, and $\mathit{free}(\phi)$ the set of free names and variables.

Table 2.8 Kinding ($k \in \{\mathsf{pub}, \mathsf{tnt}\}$) $\Gamma \vdash T :: k$

$$\textsc{Kind-Un}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Un} :: k} \qquad \textsc{Kind-Chan}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{Ch}(T) :: k} \qquad \textsc{Kind-Tuple-Pub}\ \frac{\forall i.\, \Gamma \vdash T_i :: \mathsf{pub} \qquad \Gamma, \widetilde{x} : \widetilde{T}, C \vdash \diamond}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{pub}}$$

$$\textsc{Kind-Tuple-Tnt}\ \frac{\forall i.\, \Gamma \vdash T_i :: \mathsf{tnt} \qquad \Gamma, \widetilde{x} : \widetilde{T}, C \vdash \diamond \qquad \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T}) \models C}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{tnt}}$$

$$\textsc{Kind-SigKey}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{SigKey}(T) :: k} \qquad \textsc{Kind-VerKey}\ \frac{\Gamma \vdash T :: k}{\Gamma \vdash \mathsf{VerKey}(T) :: k} \qquad \textsc{Kind-PubKey}\ \frac{\Gamma \vdash T :: \overline{k}}{\Gamma \vdash \mathsf{PubKey}(T) :: k}$$

$$\textsc{Kind-PrivKey}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{PrivKey}(T) :: k}$$

Table 2.9 Subtyping $\Gamma \vdash T <: U$

$$\textsc{Sub-Pub-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T <: U} \qquad \textsc{Sub-Refl}\ \frac{\Gamma \vdash \diamond \qquad \mathit{free}(T) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash T <: T}$$

$$\textsc{Sub-Tuple}\ \frac{\forall i.\, \Gamma \vdash T_i <: U_i \qquad \Gamma, \widetilde{x} : \widetilde{T}, C \vdash \diamond \qquad \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T}) \cup \{C\} \models C'}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} <: \langle \widetilde{x} : \widetilde{U} \rangle \{C'\}}$$

$$\textsc{Sub-Chan-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{Ch}(T) <: \mathsf{Ch}(U)} \qquad \textsc{Sub-SigKey-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{SigKey}(T) <: \mathsf{SigKey}(U)} \qquad \textsc{Sub-VerKey-Cov}\ \frac{\Gamma \vdash T <: U}{\Gamma \vdash \mathsf{VerKey}(T) <: \mathsf{VerKey}(U)}$$

$$\textsc{Sub-PubKey-Con}\ \frac{\Gamma \vdash U <: T}{\Gamma \vdash \mathsf{PubKey}(T) <: \mathsf{PubKey}(U)} \qquad \textsc{Sub-PrivKey-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{PrivKey}(T) <: \mathsf{PrivKey}(U)}$$

Table 2.10 Typing Constructors $f : (T_1, \ldots, T_n) \mapsto U$

$$\begin{aligned} \mathsf{pk} &: (\mathsf{PrivKey}(T)) \mapsto \mathsf{PubKey}(T) \\ \mathsf{enc} &: (T, \mathsf{PubKey}(T)) \mapsto \mathsf{Un} \\ \mathsf{vk} &: (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T) \\ \mathsf{sign} &: (T, \mathsf{SigKey}(T)) \mapsto \mathsf{Un} \end{aligned}$$

Table 2.11 Typing Destructors $g : (T_1, \ldots, T_n) \mapsto U$

$$\begin{aligned} \mathsf{eq} &: (T, T) \mapsto \mathsf{Un} \\ \mathsf{dec} &: (\mathsf{Un}, \mathsf{PrivKey}(T)) \mapsto T \\ \mathsf{check} &: (\mathsf{Un}, \mathsf{VerKey}(T)) \mapsto T \end{aligned}$$

Table 2.12 Typing Terms $\Gamma \vdash M : T$

ENV
$$\frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T}$$

SUB
$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash M : T'}$$

CONSTR
$$\frac{f : (T_1, \ldots, T_n) \mapsto T \qquad \forall i \in [1, n].\ \Gamma \vdash M_i : T_i}{\Gamma \vdash f(M_1, \ldots, M_n) : T}$$

TUPLE
$$\frac{\forall i \in [1, n].\ \Gamma \vdash M_i : T_i \qquad \Gamma, C\{\widetilde{M}/\widetilde{x}\} \vdash \diamond \qquad \mathsf{forms}(\Gamma) \models C\{\widetilde{M}/\widetilde{x}\}}{\Gamma \vdash \langle M_1, \ldots, M_n \rangle : \langle x_1 : T_1, \ldots, x_n : T_n \rangle\{C\}}$$

Table 2.13 Environment Extraction $P \rightsquigarrow \Gamma$

EXTR-NEW
$$\frac{P \rightsquigarrow \Gamma_P}{\mathsf{new}\ a : T.P \rightsquigarrow a : T, \Gamma_P}$$

EXTR-PAR
$$\frac{P \rightsquigarrow \Gamma_P \qquad Q \rightsquigarrow \Gamma_Q}{P \mid Q \rightsquigarrow \Gamma_P, \Gamma_Q}$$

EXTR-ASSUME
$$\mathsf{assume}(C) \rightsquigarrow C$$

EXTR-EMPTY
$$P \rightsquigarrow \emptyset$$

Table 2.14 Typing Processes $\Gamma \vdash P$

PROC-OUT
$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M, N).P}$$

PROC-(REPL)-IN
$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, x : T \vdash P}{\Gamma \vdash [!]\mathsf{in}(M, x).P}$$

PROC-STOP
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

PROC-NEW
$$\frac{T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{PrivKey}(U), \mathsf{Private}\} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T.P}$$

PROC-PAR
$$\frac{P \rightsquigarrow \Gamma_P \qquad \Gamma, \Gamma_P \vdash Q \qquad Q \rightsquigarrow \Gamma_Q \qquad \Gamma, \Gamma_Q \vdash P}{\Gamma \vdash P \mid Q}$$

PROC-DES
$$\frac{g : (T_1, \ldots, T_n) \mapsto T \qquad \forall i \in [1, n].\ \Gamma \vdash M_i : T_i \qquad \Gamma, x : T \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

PROC-SPLIT
$$\frac{\Gamma \vdash M : \langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\} \qquad \Gamma, x_1 : T_1, \ldots, x_n : T_n, \langle x_1, \ldots, x_n \rangle = M, C\{\widetilde{x}/\widetilde{y}\} \vdash P}{\Gamma \vdash \mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P}$$

PROC-LET
$$\frac{\Gamma \vdash M : T \qquad \Gamma, y : T \vdash P}{\Gamma \vdash \mathsf{let}\ y = M\ \mathsf{in}\ P}$$

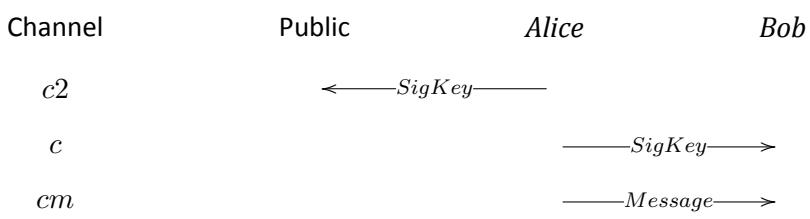
PROC-ASSUME
$$\frac{\Gamma, C \vdash \diamond}{\Gamma \vdash \mathsf{assume}(C)}$$

PROC-ASSERT
$$\frac{\Gamma \vdash \diamond \qquad \mathsf{forms}(\Gamma) \models C}{\Gamma \vdash \mathsf{assert}(C)}$$

2.3 Example

2.3.1 Informal Description

I will use this running example to illustrate the translation from protocol specification into F# code and the corresponding intermediate steps:



In the example protocol Alice sends a signed message to Bob after an initial key distribution phase that is assumed trusted. To do this three channels are used, one message is sent on each channel:

1. On $c2$ and c Alice sends her verification key. While $c2$ is a private channel to Bob, c is public and allows a potential attacker to gain access to the key.
2. On cm Alice sends Bob a message that is signed. This channel is public.

Upon receiving the message from Alice, Bob verifies it is signed with Alice's verification key.

2.3.2 Formalisation

This is the formalised version of the sample according to the calculus presented above:

```
new c : Ch(Un).
new c2 : Ch(VerKey(⟨x1 : Un⟩{Authentic(x1)})).
new cm : Ch(Un).
(
  // Bob
  in(c2, vkA).
  in(cm, m).
  let m1 = check(m, vkA) in
  let ⟨m3⟩ = m1 in
  assert(Authentic(m3))
|
  // Alice
  new sigA : SigKey(⟨x1 : Un⟩{Authentic(x1)}).
  out(c2, vk(sigA)) |
```

```
    out(c, vk(sigA)) |
    new m : Un.
    (assume(Authentic(m))
    |    out(cm, sign(⟨m⟩, sigA)))
)
```

Execution begins by creating the three channels and executing Bob and Alice in parallel, which are inlined in this version. Bob will block immediately until a message is ready on $c2$ and then on cm.

Alice in the mean time will create a signature key and send the corresponding verification key to Bob. She will also send that key out on the public channel c to make it available to everyone else. She will then create a message, sign it and send it to Bob on cm. She also assumes the message to be Authentic as the signature key can only sign authentic messages.

After receiving Alice's message Bob verifies it with her verification key and then assert that it is Authentic. This assertion will always succeed because the verification guarantees that the message was signed with the corresponding signature key, which is never leaked and which only signs Authentic messages. That is verified statically by the typechecker.

In this thesis I show how this protocol is translated into F# code that can be compiled and run.

3 The RCF calculus

RCF is a calculus presented in [BBF+08]. It extends the core of F# with security type annotations. These can be used by the F7 typechecker to verify protocol implementations, F7 also removes these annotations and passes the resulting F# code to the F# compiler.

RCF stands for Refined Concurrent FPC and is basically the FPC argumented with concurrency and refinement types. FPC stands for Fixpoint Calculus [Gun92].

Table 3.1 gives the syntax of expressions and values. Their semantics are given in [BBF+08], but it is worth noting that the $M?$ expression is asynchronous in this calculus, which means that execution does not wait for the message on the channel to be received. This is different from Spi.

The $\mathbf{try}\ A\ \mathbf{catch}\ \_ \rightarrow B$ expression is an addition by me because this construct is needed during the translation of $\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$. The F# implementation supports this construct so that it seems reasonable to add it to the calculus.

Tables 3.2, 3.3 and 3.4 give a list of types available in the RCF calculus. The disjoint sum type and the iso-recursive type have no equivalent is Spi. Judgements are listed in Table 3.5. The syntax of the typing environment is defined in Table 3.6 and the well-formed environment judgement is given in Table 3.7.

Kinding and subtyping rules are listed in Tables 3.8 and 3.9. Tables 3.10 and 3.12 give the typing rules for values and expressions while Table 3.11 lists the extraction rules. Fork and the extraction rules are according to an earlier version of [BBF+08] because they are easier to map to [BHM08].

Definition 3.0.1 (Forms in RCF)**.** *The function* $\mathsf{forms}(E)$ *maps an environment* E *to a set of formulas* $C_1, \ldots, C_n$*.*

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

Definition 3.0.2 (Free in RCF)**.** *$fn(T)$ denotes the set of free names in T, $fv(T)$ the set of free variables, and $free(T)$ the set of free names and variables.*
$free(E) = \bigcup\{free(T) | (u : T) \in E\}$

Definition 3.0.3 (Dom in RCF). *$dom(E)$ is a the set of names and variables defined in E.*

$$\begin{array}{lcl} dom(\emptyset) & = & \emptyset \\ dom(E, u : T) & = & dom(E) \cup \{u\} \\ dom(E, \alpha :: k) & = & dom(E) \cup \{\alpha\} \end{array}$$

Theorem 3.0.4 (Robust Safety). *If $\emptyset \vdash A : \mathsf{Un}$ then A is robustly safe.*

Proof. The proof is given in [BBF+08]. □

Table 3.1 Syntax of values and expressions

a, b, c		name
x, y, z		variable
$M, N ::=$		value
	v	name or variable
	$()$	unit
	$\mathbf{fun}\ x \to A$	function (scope of x is A)
	(M, N)	pair
	$\mathsf{inl}\ M$	left construction of sum type
	$\mathsf{inr}\ M$	right construction of sum type
	$\mathsf{fold}\ M$	construction of recursive type
$A, B ::=$		expression
	M	value
	$M\ N$	application
	$M = N$	syntatic equality
	$\mathbf{let}\ x = A\ \mathbf{in}\ B$	let (scope of x is B)
	$\mathbf{let}\ (x, y) = M\ \mathbf{in}\ A$	pair split (scope of x, y is A)
	$\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B$	constructor match (scope of x is A)
	$\mathbf{try}\ A\ \mathbf{catch}\ \_ \to B$	if A fails execute B
	$(\nu a : T)A$	restriction (scope of a is A)
	$A \Rsh B$	fork
	$M!N$	transmission of M on channel a
	$M?$	receive message off channel
	$\mathbf{assume}\ C$	assumption of formula C
	$\mathbf{assert}\ C$	assertion of formula C

Table 3.2 Syntax of Types

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

Table 3.3 Type Abbreviations

$(T)\mathsf{SK}$	$\triangleq$	$(T \to \mathsf{Un}) * (\mathsf{Un} \to T)$	Signing key
$(T)\mathsf{DK}$	$\triangleq$	$(T \to \mathsf{Un}) * (\mathsf{Un} \to T)$	Decryption key
$(T)\mathsf{EK}$	$\triangleq$	$(T \to \mathsf{Un})$	Encryption key
$(T)\mathsf{VK}$	$\triangleq$	$(\mathsf{Un} \to T)$	Verification key

Table 3.4 Un

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

Table 3.5 Judgments

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

Table 3.6 Syntax of Typing Environments

$\mu ::=$		environment entry
	$\alpha :: k$	kinding
	$\alpha <: \alpha'$	subtype ($\alpha \neq \alpha'$)
	$a : (T)\mathsf{chan}$	name (of channel type)
	$x : T$	variable (of any type)
$E ::= \mu_1, \ldots, \mu_n$		environment

Table 3.7 Well-formed Environment

$$\textsc{Env Empty}\quad \emptyset \vdash \diamond$$

$$\textsc{Type}\quad \frac{E \vdash \diamond \qquad free(T) \subseteq dom(E)}{E \vdash T}$$

$$\textsc{Env Entry}\quad \frac{E \vdash \diamond \qquad free(\mu) \subseteq dom(E) \qquad dom(\mu) \cap dom(E) = \emptyset}{E, \mu \vdash \diamond}$$

$$\textsc{Derive}\quad \frac{E \vdash \diamond \qquad free(C) \subseteq dom(E) \qquad \mathsf{forms}(E) \models C}{E \models C}$$

Table 3.8 Kinding ($k \in \{\mathbf{pub}, \mathbf{tnt}\}$) $E \vdash T :: k$

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

$$\textsc{Kind Var}\quad \frac{E \vdash \diamond \qquad (\alpha :: k) \in E}{E \vdash \alpha :: k}$$

$$\textsc{Kind Unit}\quad \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: k}$$

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

$$\textsc{Kind Pair}\quad \frac{E \vdash T :: k \qquad E, x : T \vdash U :: k}{E \vdash (\Sigma x : T.\, U) :: k}$$

$$\textsc{Kind Sum}\quad \frac{E \vdash T :: k \qquad E \vdash U :: k}{E \vdash (T + U) :: k}$$

$$\textsc{Kind Rec}\quad \frac{E, \alpha :: k \vdash T :: k}{E \vdash (\mu\alpha.T) :: k}$$

$$\textsc{Kind Chan}\quad \frac{E \vdash T :: \mathbf{pub} \qquad E \vdash T :: \mathbf{tnt}}{E \vdash (T)\mathsf{chan} :: k}$$

$$\textsc{Kind Refine Public}\quad \frac{E \vdash \{x : T \mid C\} \qquad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}}$$

$$\textsc{Kind Refine Tainted}\quad \frac{E \vdash T :: \mathbf{tnt} \qquad E, x : T \models C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$$

The following rules for ok-types are derivable.

$$\textsc{Kind Ok Public}\quad \frac{E \vdash \{C\}}{E \vdash \{C\} :: \mathbf{pub}}$$

$$\textsc{Kind Ok Tainted}\quad \frac{E \vdash \{C\} \qquad E \models C}{E \vdash \{C\} :: \mathbf{tnt}}$$

Table 3.9 Subtyping $E \vdash T <: U$

SUB PUBLIC TAINTED
$$\frac{E \vdash T :: \mathbf{pub} \qquad E \vdash T' :: \mathbf{tnt}}{E \vdash T <: T'}$$

SUB VAR
$$\frac{E \vdash \diamond \qquad \alpha \in dom(E)}{E \vdash \alpha <: \alpha}$$

SUB UNIT
$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}}$$

SUB FUN
$$\frac{E \vdash T' <: T \qquad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\, U) <: (\Pi x : T'.\, U')}$$

SUB PAIR
$$\frac{E \vdash T <: T' \qquad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\, U) <: (\Sigma x : T'.\, U')}$$

SUB SUM
$$\frac{E \vdash T <: T' \qquad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

SUB CHAN
$$\frac{E \vdash T <:> T'}{E \vdash (T)\mathsf{chan} <: (T')\mathsf{chan}}$$

SUB REFINE LEFT
$$\frac{E \vdash \{x : T \mid C\} \qquad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$$

SUB REFINE RIGHT
$$\frac{E \vdash T <: T' \qquad E, x : T \models C}{E \vdash T <: \{x : T' \mid C\}}$$

The following rules for ok-types are derivable.

SUB REFINE
$$\frac{E \vdash T <: T' \qquad E, x : \{x : T \mid C\} \models C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}}$$

SUB OK
$$\frac{E, \{C\} \models C'}{E \vdash \{C\} <: \{C'\}}$$

Table 3.10 Rules for Values $E \vdash M : T$

VAL VAR
$$\frac{E \vdash \diamond \qquad (x : T) \in E}{E \vdash x : T}$$

VAL UNIT
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

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

VAL PAIR
$$\frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\Sigma x : T.\, U)}$$

VAL INL
$$\frac{\mathsf{inl} : (T, T + U) \qquad E \vdash M : T \qquad E \vdash T + U}{E \vdash \mathsf{inl}\ M : T + U}$$

VAL INR
$$\frac{\mathsf{inr} : (U, T + U) \qquad E \vdash M : U \qquad E \vdash T + U}{E \vdash \mathsf{inr}\ M : T + U}$$

VAL FOLD
$$\frac{\mathsf{fold} : (T\{\mu\alpha.T/\alpha\}, \mu\alpha.T) \qquad E \vdash M : T\{\mu\alpha.T/\alpha\} \qquad E \vdash \mu\alpha.T}{E \vdash \mathsf{fold}\ M : \mu\alpha.T}$$

VAL REFINE
$$\frac{E \vdash M : T \qquad E \models C\{M/x\}}{E \vdash X : \{x : T \mid C\}}$$

We can derive an introduction rule for ok-types:

VAL OK
$$\frac{E \models C}{E \vdash () : \{C\}}$$

Table 3.11 Extraction Rules $P \rightsquigarrow E$

EXT EXP
$$A \rightsquigarrow \emptyset$$

EXT RES
$$\frac{A \rightsquigarrow E'}{(\nu a : T)A \rightsquigarrow (a : T, E')}$$

EXT FORK
$$\frac{A_1 \rightsquigarrow E_1 \qquad A_2 \rightsquigarrow E_2}{A_1 \Rsh A_2 \rightsquigarrow (E_1, E_2)}$$

EXT LET
$$\frac{A_1 \rightsquigarrow E_1}{(\mathbf{let}\ x = A_1\ \mathbf{in}\ A_2) \rightsquigarrow E_1}$$

EXTR-ASSUME
$$\mathbf{assume}\ C \rightsquigarrow \{C\}$$

Table 3.12 Rules for Expressions $E \vdash A : T$

EXP SUBSUM
$$\frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'}$$

EXP APPL
$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}$$

EXP SPLIT
$$\frac{E \vdash M : (\Sigma x : T.\, U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \qquad \{x, y\} \cap fv(V) = \emptyset}{E \vdash \textbf{let}\ (x, y) = M\ \textbf{in}\ A : V}$$

EXP MATCH INL INR FOLD
$$\frac{\begin{array}{c} E \vdash M : T \qquad h : (H, T) \\ E, x : H, \_ : \{h\ x = M\} \vdash A : U \qquad x \notin fv(U) \qquad E, \_ : \{\forall x\ .h\ x \neq M\} \vdash B : U \end{array}}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

EXP EQ
$$\frac{E \vdash M : T \qquad E \vdash N : U}{E \vdash M = N : \{x : \mathsf{bool} \mid b = \mathsf{True} \leftrightarrow M = N\}}$$

EXP ASSUME
$$\frac{E \vdash \diamond \qquad free(C) \subseteq dom(E)}{E \vdash \textbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

EXP ASSUME
$$\frac{E \models C}{E \vdash \textbf{assert}\ C : \mathsf{unit}}$$

EXP LET
$$\frac{E \vdash A : T \qquad E, x : T \vdash B : U \qquad x \notin fv(U)}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

EXP RES
$$\frac{E, A : (T)\mathsf{chan} \vdash A : U \qquad a \notin fn(U)}{E \vdash (\nu a : (T)\mathsf{chan})A : U}$$

EXP SEND
$$\frac{E \vdash M : (T)\mathsf{chan} \qquad E \vdash N : T}{E \vdash M!N : \mathsf{unit}}$$

EXP RECV
$$\frac{E \vdash M : (T)\mathsf{chan}}{E \vdash M? : T}$$

EXP FORK
$$\frac{\begin{array}{ccc} free(A_1) \subseteq dom(E) & A_2 \rightsquigarrow E_2 & E, E_2 \vdash A_1 : \mathsf{unit} \\ free(A_2) \subseteq dom(E) & A_1 \rightsquigarrow E_1 & E, E_1 \vdash A_2 : T \end{array}}{E \vdash (A_1 \Rsh A_2) : T}$$

EXP TRY
$$\frac{E \vdash A : T \qquad E \vdash B : T}{E \vdash \textbf{try}\ A\ \textbf{catch}\ \_ \to\ B : T}$$

4 A-Normal From

4.1 Overview

Because in RCF expressions are in an intermediate, reduced form [SF93], as a first step of the code generation we need to eliminate nested constructors and constructors nested inside destructors. Instead constructors will be bound to variables using let statements.

This translation is partial, in that, it does not translate formulas. So if constructors are used in formulas there is no translation to A-NF defined and therefore no translation to RCF possible. The reason there is no translation in formulas is that the variables used by constructors in formulas can be bound in the formula itself, for example by a quantifier. Even if there is no quantifier the let statement would make an assume inactive in a Fork as there is no environment extraction defined for let. Predicates and all the logical connections, however, are supported and need no translation. I assume that the source and the target type systems employ the same authorisation logic. In practice first-order logic is used by both the typecheckers of [BHM08] and [BBF+08].

4.2 Formalisation

4.2.1 Definitions

Definition 4.2.1 (Term Context)**.** *A term context is a term with a hole.*
$\gamma ::= [\,] \mid f(M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n) \mid \langle M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n \rangle$

Definition 4.2.2 (Context application)**.** *Context application, denoted $\gamma(M)$, replaces the hole in γ with the given term M.*
$[\,](M) = M$
$f(M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n)(M) = f(M_1, \ldots, M_{i-1}, \gamma(M), M_{i+1}, \ldots, M_n)$
$\langle M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n \rangle(M) = \langle M_1, \ldots, M_{i-1}, \gamma(M), M_{i+1}, \ldots, M_n \rangle$

Definition 4.2.3 (Y-Function)**.** *The Y-Function extracts the innermost nested term from a term and returns it along with the term context. If such a term does not exist the function is undefined (partial).*
Y : Terms $\rightharpoonup$ Term Context $*$ Term

$Y(f(x_1,\ldots,x_n)) = ([\,], f(x_1,\ldots,x_n))$ *Constructor applied to variables*
$Y(f(x_1,\ldots,x_i,M_{i+1},M_{i+2},\ldots,M_n)) =$
$\quad (f(x_1,\ldots,x_i,\gamma_{i+1},M_{i+2},\ldots,M_n),N_{i+1})$ *if* $Y(M_{i+1}) = (\gamma_{i+1},N_{i+1})$
$Y(\langle x_1,\ldots,x_n\rangle) = ([\,],\langle x_1,\ldots,x_n\rangle)$ *Tuple applied to variables*
$Y(\langle x_1,\ldots,x_i,M_{i+1},M_{i+2},\ldots,M_n\rangle) =$
$\quad (\langle x_1,\ldots,x_i,\gamma_{i+1},M_{i+2},\ldots,M_n\rangle,N_{i+1})$ *if* $Y(M_{i+1}) = (\gamma_{i+1},N_{i+1})$

Lemma 4.2.4. *If* $Y(M) = (\gamma, N)$ *then* $M = \gamma(N)$

Proof. Follows from Definition 4.2.3. □

Definition 4.2.5 (Process Context)**.** *A process context is a process with a top-level term hole.*

$$\begin{aligned} K ::= \;& \mathsf{out}([\,],M).P \mid \mathsf{out}(x,[\,]).P \\ & \mid \mathsf{in}(x,[\,]).P \mid \mathsf{!in}(x,[\,]).P \\ & \mid \mathsf{let}\ x = g([\,])\ \mathsf{then}\ P\ \mathsf{else}\ P' \\ & \mid \mathsf{let}\ \langle x_1,\ldots,x_n\rangle = [\,]\ \mathsf{in}\ P \end{aligned}$$

Definition 4.2.6 (Process Context application)**.** *Context application denoted* $K(M)$ *replaces the hole in* K *with the given term* M*.*
$(\mathsf{out}([\,],N).P)(M) = \mathsf{out}(M,N).P$
$(\mathsf{out}(x,[\,]).P)(M) = \mathsf{out}(x,M).P$
$(\mathsf{in}(x,[\,]).P)(M) = \mathsf{in}(x,M).P$
$(\mathsf{!in}(x,[\,]).P)(M) = \mathsf{!in}(x,M).P$
$(\mathsf{let}\ x = g([\,])\ \mathsf{then}\ P\ \mathsf{else}\ P')(M) = \mathsf{let}\ x = g(M)\ \mathsf{then}\ P\ \mathsf{else}\ P'$
$(\mathsf{let}\ \langle x_1,\ldots,x_n\rangle = [\,]\ \mathsf{in}\ P)(M) = \mathsf{let}\ \langle x_1,\ldots,x_n\rangle = M\ \mathsf{in}\ P$

Definition 4.2.7 (Z-Function)**.** *When the Y-Function is used I assume it is defined, otherwise the Z-Function is not defined either. I omit the cases where the Z-Function is not defined.*
Z : Processes $\nrightarrow$ Process Context $*$ Term Context $*$ Term

$Z(\mathsf{out}(M,M').P) = (\mathsf{out}([\,],M').P,\gamma,N)$
$Z(\mathsf{out}(x,M).P) = (\mathsf{out}(x,[\,]).P,\gamma,N)$
$Z(\mathsf{in}(x,M).P) = (\mathsf{in}(x,[\,]).P,\gamma,N)$
$Z(\mathsf{!in}(x,M).P) = (\mathsf{!in}(x,[\,]).P,\gamma,N)$
$Z(\mathsf{let}\ x = g(M)\ \mathsf{then}\ P\ \mathsf{else}\ P') = (\mathsf{let}\ x = g([\,])\ \mathsf{then}\ P\ \mathsf{else}\ P',\gamma,N)$
$Z(\mathsf{let}\ \langle x_1,\ldots,x_n\rangle = M\ \mathsf{in}\ P) = (\mathsf{let}\ \langle x_1,\ldots,x_n\rangle = [\,]\ \mathsf{in}\ P,\gamma,N)$

where for each of the above cases holds that $Y(M) = (\gamma, N)$

Lemma 4.2.8. *If* $Z(P) = (K,\gamma,M)$ *then* $P = K(\gamma(M))$

Proof. Follows from Definition 4.2.7. □

4.2.2 Translation

$\ll P \gg$	$=$	let $y = N$ in $\ll K(\gamma(y)) \gg$ where $Z(P) = (K, \gamma, N)$ and y is fresh
Other cases (where Y,Z is undefined):		
$\ll$ new $a : T.\ P \gg$	$=$	new $a : T.\ \ll P \gg$, where T is in A-NF
$\ll (P \mid Q) \gg$	$=$	$(\ll P \gg \mid \ll Q \gg)$
$\ll \mathbf{0} \gg$	$=$	$\mathbf{0}$
$\ll$ assume$(C) \gg$	$=$	assume(C), where C is in A-NF
$\ll$ assert$(C) \gg$	$=$	assert(C), where C is in A-NF
$\ll$ out$(x, y).P \gg$	$=$	out$(x, y).\ \ll P \gg$
$\ll$ in$(x, y).P \gg$	$=$	in$(x, y).\ \ll P \gg$
$\ll$!in$(x, y).P \gg$	$=$	!in$(x, y).\ \ll P \gg$
$\ll$ let $x = g(y_1, \ldots, y_n)$ then P else $P' \gg$	$=$	let $x = g(y_1, \ldots, y_n)$ then $\ll P \gg$ else $\ll P' \gg$
$\ll$ let $\langle x_1, \ldots, x_n \rangle = y$ in $P \gg$	$=$	let $\langle x_1, \ldots, x_n \rangle = y$ in $\ll P \gg$

The intuition behind this translation is as follows: It starts at the top-level process, the beginning of the Spi protocol. If the Z-Function yields a result, a new let is inserted which assigns a variable to the extracted term. The term in the process is then replaced with this variable. The process is repeatedly translated until no more terms can be extracted. After that the continuation process is translated (other cases). Some processes do not contain terms by their very nature (e.g. the parallel composition) and the Z-Function is not defined for them. In these cases the translation is also continued with the continuation process(es).

Definition 4.2.9 (Formulas in A-NF)**.** *A formula C is in A-NF if and only if it does not contain any constructors or destructors.*

Definition 4.2.10 (Types in A-NF)**.** *A type T is in A-NF if and only if all formulas in the type are in A-NF.*

4.3 Example

The example protocol from Section 2.3 has the following normal form:

```
new c : Ch(Un).
new c2 : Ch(VerKey(⟨x1 : Un⟩{Authentic(x1)})).
new cm : Ch(Un).
(
  // Bob
  in(c2, vkA).
  in(cm, m).
  let m1 = check(m, vkA) in
  let ⟨m3⟩ = m1 in
  assert(Authentic(m3))
```

```
|
  // Alice
  new sigA : SigKey(⟨x1 : Un⟩{Authentic(x1)}).
  let th1 = vk(sigK) in
  out(c2, th1) |
  out(c, th1) |
  new m : Un.
  (assume(Authentic(m))
  |   let th2 = ⟨m⟩ in
      let th3 = sign(th2, sigA) in
      out(cm, th3))
)
```

The only differences compared to Section 2.3 are for Alice. First of all the verification key is no longer constructed and sent at the same time. Rather the verification key is first assigned to the temporary variable $th1$.

The same is true for the out process inside the parallel process: Rather than nesting the constructors in the out process they are now each assigned to variables $th2$ and $th3$ using a let process.

4.4 Proof

This proof will show that the translation to A-NF preserves typability.

Lemma 4.4.1 (Term Judgement). $\Gamma \vdash \overset{\text{H1}}{\gamma(M)} : U \wedge x \notin dom(\Gamma) \Rightarrow$
$\exists T.\, \Gamma \vdash \overset{\text{C1}}{M} : T \wedge \Gamma, x : T \vdash \overset{\text{C2}}{\gamma(x)} : U$

Proof. We have $\Gamma \vdash \gamma(M) : U$ and need to show (C1): $\exists T.\, \Gamma \vdash M : T$

case	$\gamma(M) = x$: $\quad \Gamma \vdash \gamma(M) : U \Rightarrow T = U$	
case	$\gamma(M) = f(x_1, \ldots, x_{i-1}, M, M_{i+1}, \ldots, M_n)$ $\Gamma \vdash f(x_1, \ldots, M, \ldots, M_n) : U$ $f : (T_1, \ldots, T_n) \to U \; \forall j.\, \Gamma \vdash M_j : T_j$ it follows: $\Gamma \vdash M : T_i$	
case	$\gamma(M) = \langle x_1, \ldots, x_{i-1}, M_i, \ldots, M_n \rangle$	similar

to show C2: $\Gamma, x : T \vdash \gamma(x) : U$

case $\gamma = [\,]$: $[\,](x) = x$ choose $T = U$
case $\gamma = f(M_1, \ldots, M_{i-1}, [\,], M_{i+1}, \ldots, M_n)$
we know $f : (T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n) \mapsto U$ from H1
so $\Gamma, x : T \vdash f(M_1, \ldots, M_{i-1}, x, M_{i+1}, \ldots, M_n) : U$ holds □
case $\gamma = f(M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n)$ induction on γ
case $\langle M_1, \ldots, M_{i-1}, \gamma, M_{i+1}, \ldots, M_n \rangle$ as for constructors

Lemma 4.4.2 (Process Judgement). $\Gamma \vdash^{\text{H1}} K(M) \wedge x \notin dom(\Gamma)$
$\Rightarrow \exists T.\, \Gamma \vdash^{\text{C1}} M : T \wedge \Gamma, x : T \vdash^{\text{C2}} K(x)$

Proof. For case $K = \text{out}([\,], N).P$

By H1 we know to that (1) $\Gamma \vdash \text{out}(M, N).P$ holds. To show: (2)$\exists T.\, \Gamma \vdash^{\text{C1}} M : T$ and (3) $\Gamma, x : T \vdash \text{out}(x, N).P$

to show (2):
By (Proc-Out, given in Table 2.2) and (1) we know that $\Gamma \vdash M : \text{Ch}(T')$. I choose $T = \text{Ch}(T')$, which proves (2).

to show (3) by (Proc-Out):

- $\Gamma \vdash x : \text{Ch}(T')$: We know that M must have type $T = \text{Ch}(T')$ from (2). So x also has type $\text{Ch}(T')$.
- $\Gamma \vdash N : T'$: Holds by applying (Proc-Out) to (1).
- $\Gamma \vdash P$: Holds by applying (Proc-Out) to (1).

The other cases are similar. □

Theorem 4.4.3 (A-NF translation preserves typability). $\forall \Gamma, P.\, \Gamma \vdash P \Rightarrow \Gamma \vdash \ll P \gg$

Proof. By induction on the number of nested constructors and tuples in P.

If the last applied branch of $\ll P \gg$ is $Z(P) = (K, \gamma, N)$ then $\ll P \gg = \text{let } y = N \text{ in } \ll K(\gamma(y)) \gg$

- I use induction on the translation until either Y or Z fails. This means no further reduction is possible.
- By (Proc-Let) I then have to show that $\exists T.\, \Gamma \vdash^{\text{C1}} N : T \wedge \Gamma, y : T \vdash^{\text{C2}} K(\gamma(y))$
- By Lemma 4.2.8 $P = K(\gamma(N))$, so the hyposesis rewrites to $\Gamma \vdash K(\gamma(N))$
- I choose a fresh x and apply Lemma 4.4.2 to C2 which gives me $\exists T'.\, \Gamma \vdash \gamma(^{\text{C3}}N) : T' \wedge \Gamma, x : T' \vdash K(x)$

- I choose a fresh y and apply Lemma 4.4.1 to C3 and obtain $\exists T.\, \Gamma \vdash N : T \wedge \Gamma, y : T \vdash \gamma(y) : T'$
- C1 is thereby shown, remains C2
- By Subsitution Lemma, Weakening and Strengthening: $\Gamma, y : T \vdash \gamma(y) : T' \wedge \Gamma, x : T' \vdash K(x) \Rightarrow \Gamma, x : T', y : T \vdash (K(x))\{\gamma(y)/x\} = \Gamma, y : T \vdash K(\gamma(y))$

If the last applied branch of $\ll P \gg$ is a base case (i.e. $Z(P)$ is undefined) I use the IH. $\square$

5 Code generation

5.1 Overview

This section covers the Spi to RCF automatic code generation. I assume the Spi protocol to be in A-NF as otherwise the translation is not defined.

5.2 Formalisation

5.2.1 Types

$$
\begin{array}{lcl}
[\![\mathsf{Un}]\!] & = & \mathsf{Un} \\
[\![\mathsf{Private}]\!] & = & (\{\mathbf{false}\})\mathsf{chan} \\
[\![\mathsf{Ch}(T)]\!] & = & ([\![T]\!])\mathsf{chan} \\
[\![\langle \widetilde{x} : \widetilde{T} \rangle \{C\}]\!] & = & \sum_{i \in 1,n} x_i : [\![T_i]\!].\{C\} \\
[\![\mathsf{SK}(T)]\!] & = & [\![T]\!]\mathsf{SK} \\
[\![\mathsf{VK}(T)]\!] & = & [\![T]\!]\mathsf{VK} \\
[\![\mathsf{EK}(T)]\!] & = & [\![T]\!]\mathsf{EK} \\
[\![\mathsf{DK}(T)]\!] & = & [\![T]\!]\mathsf{DK}
\end{array}
$$

The translation of most types is straight forward as the same types exist in RCF. Additionally, I require C and T to be in A-NF.

Private has no real equivalent, so I chose a channel that cannot transmit any messages, because the refinement type $\{\mathbf{false}\}$ can never be satisfied.

Tuples are expressed as $\sum_{i \in 1,n} x_i : T_i.\{C\}$, which is a short form for $\Sigma x_1 : T_1.\ (\Sigma x_2 : T_2.\ \ldots (\Sigma x_n : T_n.\ \{C\}))$. These are n nested pairs and the second element of the innermost pair is a refinement type that carries C.

5.2.2 Terms

$$
\begin{array}{lcl}
[\![a]\!] & = & a \\
[\![x]\!] & = & x \\
[\![\langle M_1, \ldots, M_n \rangle]\!] & = & ([\![M_1]\!], \ldots, [\![M_n]\!], ()) \\
[\![f(M_1, \ldots, M_n)]\!] & = & f^* \, ([\![M_1]\!], \ldots, [\![M_n]\!]) \\
 & & \text{where } f : T_1 \ldots T_n \mapsto T \text{ is a Spi constructor and} \\
 & & f^* : (\Pi\_ : (\textstyle\sum_{i \in 1,n} x_i : [\![T_i]\!]). \, [\![T]\!]) \\
 & & \text{is the corresponding function in RCF}
\end{array}
$$

The additional unit in the translation of a tuple is due to the fact that I translate the tuple type with an additional element of type unit conveying the formula.

$(M_1, \ldots, M_n, ())$ is a short form for nested pairs $(M_1, (M_2, \ldots, (M_n, ())))$.

5.2.3 Processes

$$
\begin{array}{lcl}
[\![\mathsf{out}(M, N).\mathbf{0}]\!] & = & [\![M]\!]![\![N]\!] \\
[\![\mathsf{in}(M, x).P]\!] & = & \mathbf{let}\ x = [\![M]\!]?\ \mathbf{in}\ [\![P]\!] \\
[\![!\mathsf{in}(M, x).P]\!] & = & \mathsf{bangIn}\ ([\![M]\!], \mathbf{fun}\ x \to [\![P]\!]) \\
[\![\mathsf{new}\ a : \mathsf{Ch}(T).P]\!] & = & (\nu a : (T)\mathsf{chan})[\![P]\!] \\
[\![\mathsf{new}\ a : T.P]\!] & = & \mathbf{let}\ a = mkT\ ()\ \mathbf{in}\ [\![P]\!] \\
 & & \text{if there is } mkT : \Pi\_ : \mathsf{unit}.\ [\![T]\!] \\
[\![(P \mid Q)]\!] & = & [\![P]\!] \Rsh [\![Q]\!] \\
[\![\mathbf{0}]\!] & = & () \\
[\![\mathsf{let}\ \langle x_{n-1}, x_n \rangle = M\ \mathsf{in}\ P]\!] & = & \mathbf{let}\ (x_{n-1}, z_{n-1}) = M\ \mathbf{in}\ \mathbf{let}\ (x_n, z_n) = z_{n-1}\ \mathbf{in}\ [\![P]\!] \\
[\![\mathsf{let}\ \langle x_i, x_{i+1}, \ldots, x_n \rangle = M\ \mathsf{in}\ P]\!] & = & \mathbf{let}\ (x_i, z_i) = M\ \mathbf{in}\ [\![\mathsf{let}\ \langle x_{i+1}, \ldots, x_n \rangle = z_i\ \mathsf{in}\ P]\!] \\
 & & \text{where } z_i \text{ is a fresh variable} \\
[\![\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] & = & \mathbf{try}\ \mathbf{let}\ x = g^* \, ([\![M_1]\!], \ldots, [\![M_n]\!])\ \mathbf{in}\ [\![P]\!]\ \mathbf{catch}\ \_ \to [\![Q]\!] \\
 & & \text{where } g : T_1 \ldots T_n \mapsto T \text{ is a Spi destructor and} \\
 & & g^* : (\Pi\_ : (\textstyle\sum_{i \in 1,n} x_i : [\![T_i]\!]). \, [\![T]\!]) \\
 & & \text{is the corresponding function in RCF} \\
[\![\mathsf{assume}(C)]\!] & = & \mathbf{assume}\ C \\
[\![\mathsf{assert}(C)]\!] & = & \mathbf{assert}\ C
\end{array}
$$

All types T must be in A-NF.

As Spi uses a synchronous out and RCF uses an asynchronous one the translation only works if the continuation process P is the Null-Process $\mathbf{0}$ as indicated above. Boudol provides a way to encode synchronous messaging in an asynchronous calculus [Bou92], but this involves passing channels as messages, which will not work very well in a practical implementation. I therefore require all Spi protocols to place the out inside a parallel composition.

The RCF calculus only has an expression to generate new channels. New values of other types are generated by calling functions. These mkT functions will use the F# and .net libraries to create an object of the appropriate type.

The replicated input process in Spi has no direct representation in RCF, but can be expressed using a recursive function that is given below.

The tuple split needs a recursive translation as tuples are encoded as nested pairs. In the last step there is an additional z_n split off. This is the unit that was added when the nested pair structure was created.

let is translated using a **try** statement. The idea behind that is that a destructor g can throw an arbitrary exception to indicate that g is not applicable to the arguments $M_1 \ldots M_n$. In this case the result of Q is returned by the **try**.

BangIn

bangIn is a recursive procedure that can be expressed with the help of the fixpoint combinator, which can be implemented in RCF using self-application and then typed using a recursive type. I give the types in ML notation for readability.

$$\begin{array}{lcl} \mathsf{fix} & : & ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \\ \mathsf{bangIn} & : & ((\alpha)\mathsf{chan}, (\alpha \to \mathsf{unit})) \to \beta \end{array}$$

This is the implementation for bangIn:

$$\begin{array}{lcl} \mathsf{bangIn} & = & \mathsf{fix}\ (\lambda b.\lambda(c, f_p).\mathbf{let}\ x = c?\ \mathbf{in} \\ & & ((f_p\ x) \Rsh (b\ (c, f_p)))) \end{array}$$

bangIn waits for input, runs f_p for each message that arrives and then waits for more input. It never returns control to the caller.

5.3 Example

This is the automatic translation of the protocol from Section 4.3 using a very early version of the code generator prototype. This prototype did not produce executable F# code, but rather code that followed the RCF syntax.

```
type Predicates =
   Authentic(string)

(vc:(Un)chan)
(vc2:((∑x1:Un.{Authentic(x1)})VK)chan)
(vcm:(Un)chan)
(
   let vkA = c2?;
   let m = cm?;
   let m1 = check m vkA in
   let m3 = fst m1 in
```

```
   let dummy1 = snd m1 in
   let dummy2 = exercise dummy1 in
   assert (Authentic(m3))
↱
   let sigA = mkSK () in
   let th1 = (vk sigA) in
   c2!th1;
   c!th1;
   let m = mkUn () in
   (
      assume (Authentic(m))
   ↱
      let th2 = (m, ()) in
      let th3 = (sign th2 sigA) in
      cm!th3
   )
)
```

The translation is straight forward. The only slightly confusing aspect are the variables called dummy* and the exercise function. This function does nothing and the variables are never used. They are there because of the way the parser parses Spi files and as they have no effect is no harm in leaving them in for the prototype.

5.4 Correctness Proof

I show that the translation from Spi to RCF preserves typability.

The proofs below are all backwards (i.e. goal-oriented). I will start every case by giving the rule from Chapter 2 and then the statement I want to show, which is the translated form of the conclusion of the Spi rule. I will prove each statement by by using appropriate rules from RCF and eventually the hypothesises from the Spi rule that I know to be true.

In each paragraph I apply one rule from Chapter 3, the name of which I give in parenthesis. In that paragraph I list the hypothesises that have to be fulfilled for that conclusion to hold. If a hypothesis obviously holds I state this in parenthesis, otherwise I assign a number to the hypothesis and refer to it in the next paragraph. Some rules may have more than two hypothesis I which case I use a bulleted list.

Theorem 5.4.1 (Trans. Preserves Typing)**.** $\forall P \forall \Gamma.\, \Gamma \vdash P \Rightarrow \exists T.\, [\![\Gamma]\!] \vdash [\![P]\!] : T$

Corollary 5.4.2 (Robust Safety of the Translation)**.** *If P is well-typed then $[\![\ll P \gg]\!]$ is robustly safe.*

Proof. Follows directly from Theorem 5.4.1 and Theorem 4.4.3 using Theorem 3.0.4. □

Before giving the proof for Theorem 5.4.1 some lemmas are required:

Lemma 5.4.3 (Forms-eq)**.** forms($[\![\Gamma]\!]$) = forms(Γ)

Proof. For each case equivalence can be shown trivially. □

Lemma 5.4.4 (Free-Dom)**.** $\Gamma \vdash \mathcal{J} \Rightarrow \mathit{free}(\mathcal{J}) \subseteq \mathit{dom}(\Gamma)$

Proof. Obvious from Tables 2.2 and 2.1. □

Definition 5.4.5 (Environment translation)**.**

$$
\begin{array}{lcl}
[\![\emptyset]\!] & = & \emptyset \\
[\![\Gamma, x : T]\!] & = & [\![\Gamma]\!], x : [\![T]\!] \\
[\![\Gamma, C]\!] & = & [\![\Gamma]\!], \_ : \{C\}
\end{array}
$$

Lemma 5.4.6 (Dom-eq)**.** $\mathit{dom}(\Gamma) = \mathit{dom}([\![\Gamma]\!])$

Proof. Induction on the length of Γ:

- Base case $\Gamma = \emptyset$: $\mathit{dom}(\emptyset) = \mathit{dom}([\![\emptyset]\!])$
- Case $\Gamma = (\Gamma', C)$: $\mathit{dom}(\Gamma', C) = \mathit{dom}(\Gamma')$, apply IH
- Case $\Gamma = (\Gamma', u : T)$: $\mathit{dom}(\Gamma', u : T) = \mathit{dom}(\Gamma') \cup \{u\}$
  $\mathit{dom}([\![\Gamma', u : T]\!]) = \mathit{dom}([\![\Gamma']\!], u : [\![T]\!]) = \mathit{dom}([\![\Gamma']\!]) \cup \{u\}$, apply IH

□

Lemma 5.4.7. $\Gamma, C \vdash \diamond \Rightarrow \mathit{free}(\{C\}) \subseteq \mathit{dom}([\![\Gamma]\!])$

Proof. Follows from (Env-Formula, given in Table 2.7) and Lemma 5.4.6. □

Lemma 5.4.8 (Environments)**.** $\forall \Gamma$ *in A-NF*. $\Gamma \vdash \diamond$ then $[\![\Gamma]\!] \vdash \diamond$

Proof. By induction on $\Gamma \vdash \diamond$

Cases:
ENV-EMPTY
$\emptyset \vdash \diamond$

$$\Gamma = \emptyset \Rightarrow [\![\Gamma]\!] = \emptyset \Rightarrow [\![\Gamma]\!] \vdash \diamond$$

ENV-FORMULA

$$\frac{\Gamma \vdash \diamond \ ^{\text{H1}} \qquad \mathit{free}(C) \subseteq \mathit{dom}(\Gamma) \ ^{\text{H2}}}{\Gamma, C \vdash \diamond}$$

$[\![\Gamma, C]\!] = [\![\Gamma]\!], x : \{y : \mathit{unit} \mid C\}$, where x, y are fresh.

to show (by Env Entry):

- $[\![\Gamma]\!] \vdash \diamond$ (by IH)
- $\mathit{free}\{x\} \subseteq \mathit{dom}([\![\Gamma]\!])$ (by H2 + Lemma 5.4.6)
- $\mathit{dom}([\![\Gamma]\!]) \cap \mathit{dom}(x : \{y : \mathit{unit} \mid C\}) = \emptyset$ (because x is fresh)

ENV-BINDING

$$\frac{\Gamma \vdash \diamond \ ^{\text{H1}} \qquad u \notin \mathit{dom}(\Gamma) \ ^{\text{H2}} \qquad \mathit{free}(T) \subseteq \mathit{dom}(\Gamma) \ ^{\text{H3}}}{\Gamma, u : T \vdash \diamond}$$

to show (by Env Entry):

1. $[\![\Gamma]\!] \vdash \diamond$ (by IH)
2. $\mathit{free}(u : [\![T]\!]) \subseteq \mathit{dom}([\![\Gamma]\!])$
   $\Leftarrow \mathit{free}(u : [\![T]\!]) = \mathit{free}([\![T]\!]) = \mathit{free}(T)$ (by def of $[\![T]\!]$)
   $\wedge \mathit{dom}(\Gamma) = \mathit{dom}([\![\Gamma]\!])$ (by Lemma 5.4.6)
   $\wedge \mathit{free}(T) \subseteq \mathit{dom}(\Gamma)$ (by H3)
3. $\mathit{dom}(u : [\![T]\!]) \cap \mathit{dom}([\![\Gamma]\!]) = \emptyset$
   $\Leftarrow \mathit{dom}(u : [\![T]\!]) = \{u\}$
   $\wedge \mathit{dom}([\![\Gamma]\!]) = \mathit{dom}(\Gamma)$ (by Lemma 5.4.6)
   $\wedge u \notin \mathit{dom}(\Gamma)$ (by H2)

□

Lemma 5.4.9 (Kinding). $\forall \Gamma, T, k.\ \Gamma \vdash T :: k \Rightarrow [\![\Gamma]\!] \vdash [\![T]\!] :: k$
$k \in \{\mathsf{pub}, \mathsf{tnt}\}$
Let $\overline{\mathsf{pub}} = \mathsf{tnt}$ *and* $\overline{\mathsf{tnt}} = \mathsf{pub}$

Proof. By induction on $\Gamma \vdash T :: k$

Cases:

KIND-UN

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Un} :: k}$$

trivial from definition of Un

KIND-CHAN

$$\frac{\Gamma \vdash \overset{\text{H1}}{T} :: \mathsf{pub} \qquad \Gamma \vdash \overset{\text{H2}}{T} :: \mathsf{tnt}}{\Gamma \vdash \mathsf{Ch}(T) :: k}$$

follows from (Kind Chan) in Table 3.8

KIND-TUPLE-PUB

$$\frac{\forall i.\, \Gamma \vdash \overset{\text{H1}}{T_i} :: \mathsf{pub} \qquad \overset{\text{H2}}{\Gamma, \widetilde{x} : \widetilde{T}, C \vdash \diamond}}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{pub}}$$

to show: $[\![\Gamma]\!] \vdash \sum_{i \in 1,n} x_i : [\![T_i]\!].\{C\} :: \mathbf{pub}$

it remains to show

(by Kind Pair) $[\![\Gamma]\!] \vdash [\![T_i]\!] :: \mathbf{pub}$ (by H1 + IH) and (2) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\} :: \mathbf{pub}$

to show (2) (by Kind Refine Public) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \mathsf{unit} :: \mathbf{pub}$ and (3) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\}$

to show (3) (by Type) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \diamond$ (by H2 + Weakening) and $\mathit{free}(\{C\}) \subseteq \mathit{dom}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!])$ (by H2 and Lemma 5.4.7)

KIND-TUPLE-TNT

$$\frac{\forall i.\, \Gamma \vdash \overset{\text{H1}}{T_i} :: \mathsf{tnt} \qquad \overset{\text{H2}}{\Gamma, \widetilde{x} : \widetilde{T}, C \vdash \diamond} \qquad \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T}) \overset{\text{H3}}{\models} C}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{tnt}}$$

to show: $[\![\Gamma]\!] \vdash \sum_{i \in 1,n} x_i : [\![T_i]\!].\{C\} :: \mathbf{tnt}$

it remains to show:

(by Kind Pair) $[\![\Gamma]\!] \vdash [\![T_i]\!] :: \mathbf{tnt}$ (by H1 + IH) and (2) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\} :: \mathbf{tnt}$

to show (2) (by Kind Refine Tainted) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \mathsf{unit} :: \mathbf{tnt}$ and $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \models C$

$\Leftarrow$ it remains to show (Derive)

- $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \diamond$ (by weakening H2)
- $\mathit{free}(C) \subseteq \mathit{dom}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!])$ (by H2 and Lemma 5.4.7)
- $\mathsf{forms}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!]) \models C$ (by H3 and Lemma 5.4.3)

KIND-SIGKEY

$$\frac{\Gamma \vdash \overset{\text{H1}}{T} :: \mathsf{tnt} \qquad \Gamma \vdash \overset{\text{H2}}{T} :: \mathsf{pub}}{\Gamma \vdash \mathsf{SigKey}(T) :: k}$$

to show: $[\![\Gamma]\!] \vdash (T)\mathsf{SK} :: k$

$[\![\mathsf{SigKey}(T)]\!] = ([\![T]\!])\mathsf{SK} = ([\![T]\!] \to \mathsf{Un}) * (\mathsf{Un} \to [\![T]\!])$

Using (Kind Pair) and then (Kind Fun) on $([\![T]\!] \to \mathsf{Un})$ we get that $T :: \mathbf{pub}$; from $(\mathsf{Un} \to [\![T]\!])$ we get that $T :: \mathbf{tnt}$ for $(T)\mathsf{SK} :: \mathbf{pub}$. It is the other way round for $(T)\mathsf{SK} :: \mathbf{tnt}$.
So $[\![T]\!]$ has to be both **pub** and **tnt**. This follows from H1, H2 and IH.

KIND-PRIVKEY

$$\frac{\Gamma \vdash T :: \mathsf{tnt} \qquad \Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \mathsf{PrivKey}(T) :: k}$$

like SigKey

KIND-VERKEY

$$\frac{\Gamma \vdash T :: k}{\Gamma \vdash \mathsf{VerKey}(T) :: k}$$

to show: $[\![\Gamma]\!] \vdash (T)\mathsf{VK} :: k$

$[\![\mathsf{VerKey}(T)]\!] = ([\![T]\!])\mathsf{VK} = (\mathsf{Un} \to [\![T]\!])$

(Kind Fun) is covariant for the result T

KIND-PUBKEY

$$\frac{\Gamma \vdash T :: k}{\Gamma \vdash \mathsf{VerKey}(T) :: \overline{k}}$$

to show: $[\![\Gamma]\!] \vdash (T)\mathsf{DK} :: \overline{k}$

$[\![\mathsf{PubKey}(T)]\!] = ([\![T]\!])\mathsf{DK} = ([\![T]\!] \to \mathsf{Un})$

(Kind Fun) is contravariant for the argument T

□

Lemma 5.4.10 (Subtyping). $\forall \Gamma, T, T'.\ \Gamma \vdash T <: T' \Rightarrow [\![\Gamma]\!] \vdash [\![T]\!] <: [\![T']\!]$

Proof. By induction on $\Gamma \vdash T <: T'$

Cases:
SUB-PUB-TNT

$$\frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T <: U}$$

to show: $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![T']\!]$
(by Lemma 5.4.9) we have that $[\![\Gamma]\!] \vdash [\![T]\!] :: \mathbf{pub} \wedge [\![\Gamma]\!] \vdash [\![U]\!] :: \mathbf{tnt}$
by applying (Sub Public Tainted) from Table 3.9 in F7, $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![T']\!]$ follows.

SUB-REFL

$$\frac{\Gamma \vdash \diamond \qquad free(T) \subseteq dom(\Gamma)}{\Gamma \vdash T <: T}$$

Refexivity of the RCF subtyping relation is claimed in Lemma 15 from [BBF+08]

SUB-TUPLE

$$\frac{\forall i.\ \Gamma \vdash T_i \overset{\text{H1}}{<:} U_i \qquad \Gamma, \widetilde{x} : \widetilde{T} \overset{\text{H2}}{,} C \vdash \diamond \qquad \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T} \overset{\text{H3}}{)} \cup \{C\} \models C'}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} <: \langle \widetilde{x} : \widetilde{U} \rangle \{C'\}}$$

to show: $[\![\Gamma]\!] \vdash \sum_{i \in 1,n} x_i : [\![T_i]\!].\{C\} <: \sum_{i \in 1,n} y_i : [\![T'_i]\!].\{C'\}$

(by Sub Pair) $\forall i.\ [\![\Gamma]\!] \vdash [\![T_i]\!] <: [\![U_i]\!]$ (by H1+IH) and (2) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\} <: \{\_ : \mathsf{unit} \mid C'\}$

to show (2) (by Sub Refine Right) (3) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\} <: \mathsf{unit}$ and (4) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!], \{C\} \models C'$

to show (3) (by Sub Refine Left) (5) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \{\_ : \mathsf{unit} \mid C\}$ and $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \mathsf{unit} <: \mathsf{unit}$ (by Sub Unit)

to show (5) (by Type) $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!] \vdash \diamond$ (by H2 + Weakening) and $free(\{C\}) \subseteq dom([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!])$ (by H2 and Lemma 5.4.7)

$\Leftarrow$ it remains to show (4) by (Derive)

- $[\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!], \{C\} \vdash \diamond$ (by H2 + Lemma 5.4.8)
- $free(C') \subseteq dom([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!], \{C\})$ (implied by H3)
- $\mathsf{forms}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!], \{C\}) \models C'$
  $\Leftarrow \mathsf{forms}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!], \{C\}) = \mathsf{forms}([\![\Gamma]\!], \widetilde{x} : [\![\widetilde{T}]\!]) \cup \{C\}$
  (by Lemma 5.4.3) $= \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T}) \cup \{C\}$
  $\wedge \mathsf{forms}(\Gamma, \widetilde{x} : \widetilde{T}) \cup \{C\} \models C'$ (by H3)

SUB-CHAN-INV

$$\frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{Ch}(T) <: \mathsf{Ch}(U)}$$

follows from (Sub Chan) in Table 3.9

SUB-SIGKEY-INV

$$\frac{\Gamma \vdash T \overset{\text{H1}}{<:>} U}{\Gamma \vdash \mathsf{SigKey}(T) <: \mathsf{SigKey}(U)}$$

$[\![\mathsf{SigKey}(T)]\!] = ([\![T]\!])\mathsf{SK} = ([\![T]\!] \to \mathsf{Un}) * (\mathsf{Un} \to [\![T]\!])$

to show: $[\![\Gamma]\!] \vdash ([\![T]\!] \to \mathsf{Un}) * (\mathsf{Un} \to [\![T]\!]) <: ([\![U]\!] \to \mathsf{Un}) * (\mathsf{Un} \to [\![U]\!])$

it remains to show:

(by Sub Pair) $[\![\Gamma]\!] \vdash ([\![T]\!] \to \mathsf{Un}) <: ([\![U]\!] \to \mathsf{Un})$ and $[\![\Gamma]\!] \vdash (\mathsf{Un} \to [\![T]\!]) <: (\mathsf{Un} \to [\![U]\!])$

(by Sub Fun) we still need to show:

- $[\![\Gamma]\!] \vdash [\![U]\!] <: [\![T]\!]$ (by H1)
- $[\![\Gamma]\!] \vdash \mathsf{Un} <: \mathsf{Un}$ (by Refl)
- $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![U]\!]$ (by H1)

Sub-PrivKey-Inv
$$\frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{PrivKey}(T) <: \mathsf{PrivKey}(U)}$$

Same as SigKey

Sub-VerKey-Cov
$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \mathsf{VerKey}(T) <: \mathsf{VerKey}(U)}$$

$[\![\mathsf{VerKey}(T)]\!] = ([\![T]\!])\mathsf{VK} = (\mathsf{Un} \to [\![T]\!])$

(Sub Fun) $\to$ is covariant for the result T

Sub-PubKey-Con
$$\frac{\Gamma \vdash U <: T}{\Gamma \vdash \mathsf{PubKey}(T) <: \mathsf{PubKey}(U)}$$

$[\![\mathsf{PubKey}(T)]\!] = ([\![T]\!])\mathsf{DK} = ([\![T]\!] \to \mathsf{Un})$

(Sub Fun) $\to$ is contravariant for the argument T

□

Lemma 5.4.11 (Term Typing). $\forall M \forall \Gamma \forall T.\, \Gamma \vdash M : T \Rightarrow [\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]$

Proof. By induction on $\Gamma \vdash M : T$

Cases:

Env
$$\frac{\Gamma \vdash \diamond^{\text{H1}} \qquad u : T \in \Gamma^{\text{H2}}}{\Gamma \vdash u : T}$$

to show: $[\![\Gamma]\!] \vdash u : [\![T]\!]$

equivalently we show: $[\![\Gamma]\!] \vdash \diamond$ (by H1 and Lemma 5.4.8) and $u : [\![T]\!] \in [\![\Gamma]\!]$ (by H2 and Definition 5.4.5)

SUB

$$\frac{\Gamma \vdash \overset{\text{H1}}{M} : T \qquad \Gamma \vdash \overset{\text{H2}}{T} <: T'}{\Gamma \vdash M : T'}$$

to show: $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T']\!]$

by (Exp Subsum)

- $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]$ (by H1+IH)
- $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![T']\!]$ (by H2 and Lemma 5.4.10)

CONSTR

$$\frac{f : (T_1, \ldots, T_n) \mapsto T \overset{\text{H1}}{\qquad} \forall i \in [1,n].\ \Gamma \overset{\text{H2}}{\vdash} M_i : T_i}{\Gamma \vdash f(M_1, \ldots, M_n) : T}$$

to show: $[\![\Gamma]\!] \vdash f^*\ [\![M_1]\!] \ldots [\![M_n]\!] : [\![T]\!]$
I assume there is a function $f^* : (\Pi\_ : (\sum_{i \in 1,n} y_i : [\![T_i]\!]).\ [\![T]\!])$ in RCF.

by (Exp Appl)

- $[\![\Gamma]\!] \vdash f^* : (\Pi\_ : (\sum_{i \in 1,n} y_i : [\![T_i]\!]).\ [\![T]\!])$
- $[\![\Gamma]\!] \vdash [\![M_i]\!] : [\![T_i]\!]$ form (H2) by IH

TUPLE

$$\frac{\forall i \in [1,n].\ \Gamma \overset{\text{H1}}{\vdash} M_i : T_i \qquad \Gamma, C\{\widetilde{M}/\widetilde{x}\} \overset{\text{H2}}{\vdash} \diamond \qquad \mathsf{forms}(\Gamma) \overset{\text{H3}}{\models} C\{\widetilde{M}/\widetilde{x}\}}{\Gamma \vdash \langle M_1, \ldots, M_n \rangle : \langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}}$$

to show: $[\![\Gamma]\!] \vdash ([\![M_1]\!], \ldots, [\![M_n]\!], ()) : \sum_{i \in 1,n} x_i : T_i.\{C\}$

it remains to show

- (by Val Pair) $[\![\Gamma]\!] \vdash [\![M_i]\!] : [\![T_i]\!]$ (by H1 + IH) and
  (2) $[\![\Gamma]\!] \vdash () : \{\_ : \mathsf{unit} \mid C\}\{M_i/x_i\}_{i \in 1,n}$
- to show (2) (by Val Refine) $[\![\Gamma]\!] \vdash () : \mathsf{unit}$ and $[\![\Gamma]\!] \models C\{M_i/x_i\}_{i \in 1,n}$

it remains to show (Derive)

- $[\![\Gamma]\!] \vdash \diamond$ (by weakening H2 + Lemma 5.4.8)
- $\mathit{free}(C\{M_i/x_i\}_{i \in 1,n}) \subseteq \mathit{dom}([\![\Gamma]\!])$ (by H2 and Lemma 5.4.7)
- $\mathsf{forms}([\![\Gamma]\!]) \models C\{M_i/x_i\}_{i \in 1,n}$ (by H3 and Lemma 5.4.3)

□

Lemma 5.4.12 (Environment Extraction)**.** $\forall P.\ P \rightsquigarrow \Gamma_P \Rightarrow [\![P]\!] \rightsquigarrow [\![\Gamma_P]\!]$

Proof. Induction over $|P|$

EXTR-NEW

$$\frac{P \overset{\text{H1}}{\rightsquigarrow} \Gamma_P}{\mathsf{new}\ a : T.P \rightsquigarrow a : T, \Gamma_P}$$

to show (by Ext Res): $[\![P]\!] \rightsquigarrow [\![\Gamma_P]\!]$ (by IH+H1)

EXTR-PAR

$$\frac{P \overset{\text{H1}}{\rightsquigarrow} \Gamma_P \qquad Q \overset{\text{H2}}{\rightsquigarrow} \Gamma_Q}{P \mid Q \rightsquigarrow \Gamma_P, \Gamma_Q}$$

to show (by Ext Fork): $[\![P]\!] \rightsquigarrow [\![\Gamma_P]\!]$ and $[\![Q]\!] \rightsquigarrow [\![\Gamma_Q]\!]$ (by IH+H1+H2)

EXTR-ASSUME

$\mathsf{assume}(C) \rightsquigarrow C$

exactly what we want

EXTR-EMPTY

$P \rightsquigarrow \emptyset$

exactly what we want

□

Lemma 5.4.13 (Unit-Return)**.** *Every translated process returns* unit*:* $\forall P, \Gamma.\ [\![\Gamma]\!] \vdash [\![P]\!] : \mathsf{unit}$

Proof. I show this below when typing the translated processes. □

Proof for Theorem 5.4.1

With these lemmas we can now prove Theorem 5.4.1: $\Gamma \vdash P \Rightarrow [\![\Gamma]\!] \vdash [\![P]\!] : \mathsf{unit}$

Proof. By induction on $\Gamma \vdash P$

Cases:

PROC-OUT

$$\frac{\Gamma \vdash M \overset{\text{H1}}{:} \mathsf{Ch}(T) \qquad \Gamma \vdash N \overset{\text{H2}}{:} T \qquad \Gamma \vdash \mathbf{0}}{\Gamma \vdash \mathsf{out}(M, N).\mathbf{0}}$$

to show: $[\![\Gamma]\!] \vdash [\![M]\!]![\![N]\!] : \mathsf{unit}$

to show (by Exp Send):

- $[\![\Gamma]\!] \vdash [\![M]\!] : ([\![T]\!])\mathsf{chan}$ (by H1 and Lemma 5.4.11)
- $[\![\Gamma]\!] \vdash [\![N]\!] : [\![T]\!]$ (by H2 and Lemma 5.4.11)

PROC-IN

$$\frac{\Gamma \vdash M : \mathsf{Ch}(T)^{\,H1} \qquad \Gamma, x : T \vdash P^{\,H2}}{\Gamma \vdash \mathsf{in}(M, x).P}$$

to show: $[\![\Gamma]\!] \vdash \mathbf{let}\ x = [\![M]\!]?\ \mathbf{in}\ [\![P]\!] : \mathsf{unit}$

to show (by Exp Let and Exp Recv)

- $[\![\Gamma]\!] \vdash M : ([\![T]\!])\mathsf{chan}$ (by H1 and Lemma 5.4.11)
- $[\![\Gamma]\!], x : T \vdash [\![P]\!] : \mathsf{unit}$ (by H2+IH and Lemma 5.4.13)
- $x \notin fv(\mathsf{unit})$

PROC-REPL-IN

$$\frac{\Gamma \vdash M : \mathsf{Ch}(T)^{\,H1} \qquad \Gamma, x : T \vdash P^{\,H2}}{\Gamma \vdash !\mathsf{in}(M, x).P}$$

translation: $!\mathsf{in}(M, x).P = \mathsf{bangIn}\ ([\![M]\!], \mathbf{fun}\ x \to [\![P]\!])$

to show (by Exp Appl):

- $[\![\Gamma]\!] \vdash \mathsf{bangIn} : \Pi\_ : (\Sigma\_ : ([\![T]\!])\mathsf{chan}.\ (\Pi\_ : [\![T]\!].\ \mathsf{unit})).\ T'$ (by def of bangIn) I choose T' to be unit.
- (1) $[\![\Gamma]\!] \vdash ([\![M]\!], \mathbf{fun}\ x \to [\![P]\!]) : \Sigma\_ : ([\![T]\!])\mathsf{chan}.\ (\Pi\_ : [\![T]\!].\ \mathsf{unit})$

it remains to show (by Val Pair on (1))

- $[\![\Gamma]\!] \vdash [\![M]\!] : ([\![T]\!])\mathsf{chan}$ (by H1 and Lemma 5.4.11)
- (2) $[\![\Gamma]\!] \vdash \mathbf{fun}\ x \to [\![P]\!] : \Pi\_ : [\![T]\!].\ \mathsf{unit}$

by (Val Fun) on (2) it remains to show: $[\![\Gamma]\!], x : [\![T]\!] \vdash [\![P]\!] : \mathsf{unit}$ (by H2+IH and Lemma 5.4.13)

PROC-STOP

$$\frac{\Gamma \vdash \diamond^{\,H1}}{\Gamma \vdash \mathbf{0}}$$

$[\![\mathbf{0}]\!] = ()$
(by Val Unit) I need to show: $[\![\Gamma]\!] \vdash \diamond$ (by H1+Lemma 5.4.8)

PROC-NEW

$$\frac{T \in \{\mathsf{Ch}(U)\} \qquad \Gamma, a : T \vdash P^{\,H2}}{\Gamma \vdash \mathsf{new}\ a : T.P}$$

to show: $[\![\Gamma]\!] \vdash (\nu a : (T)\mathsf{chan})[\![P]\!] : \mathsf{unit}$

it remains to show (by Exp Res):

- $[\![\Gamma]\!], a : ([\![T]\!])\mathsf{chan} \vdash [\![P]\!] : \mathsf{unit}$ (by H2+IH and Lemma 5.4.13)
- $a \notin fn(\mathsf{unit})$

PROC-NEW

$$\frac{T \in \{\mathsf{Un}, \mathsf{SigKey}(U), \mathsf{PrivKey}(U), \mathsf{Private}\} \qquad \Gamma, a : \overset{\mathsf{H1}}{T} \vdash P}{\Gamma \vdash \mathsf{new}\ a : T P}$$

I take $\mathsf{SigKey}(U)$ as an example, the other cases are similar.
Translation: $\mathbf{let}\ a = mkSK\ ()\ \mathbf{in}\ [\![P]\!]$

To show (by Exp Let):

- (1) $[\![\Gamma]\!] \vdash mkSK\ () : ([\![U]\!])\mathsf{SK}$
- $[\![\Gamma]\!], a : ([\![U]\!])\mathsf{SK} \vdash P : \mathsf{unit}$ (by H1+IH and Lemma 5.4.13)
- $a \notin fv(\mathsf{unit})$

It remains to show (Exp Appl) on (1):

- $[\![\Gamma]\!] \vdash mkSK : (\Pi\_ : \mathsf{unit}.\ ([\![U]\!])\mathsf{SK})$ (by def of $mkSK$)
- $[\![\Gamma]\!] \vdash () : \mathsf{unit}$

PROC-PAR

$$\frac{\overset{\mathsf{H0}}{P \rightsquigarrow \Gamma_P} \qquad \Gamma, \overset{\mathsf{H1}}{\Gamma_P} \vdash Q \qquad \overset{\mathsf{H1.5}}{Q \rightsquigarrow \Gamma_Q} \qquad \Gamma, \overset{\mathsf{H2}}{\Gamma_Q} \vdash P}{\Gamma \vdash P \mid Q}$$

to show: $[\![\Gamma]\!] \vdash [\![P]\!] \Rsh [\![Q]\!] : \mathsf{unit}$

we need to show by (Exp Fork)

- $free([\![P]\!]) \subseteq dom([\![\Gamma]\!])$ (by H2 and Lemma 5.4.4, Γ_Q only contains formulas)
- $[\![P]\!] \rightsquigarrow [\![\Gamma_P]\!]$ (by Lemma 5.4.12 and H0)
- $[\![\Gamma]\!], [\![\Gamma_Q]\!] \vdash [\![P]\!] : \mathsf{unit}$ (by H2 + IH and Lemma 5.4.13)
- $free([\![Q]\!]) \subseteq dom([\![\Gamma]\!])$ (by H1 and Lemma 5.4.4, Γ_P only contains formulas)
- $[\![Q]\!] \rightsquigarrow [\![\Gamma_Q]\!]$ (by Lemma 5.4.12 and H1.5)
- $[\![\Gamma]\!], [\![\Gamma_P]\!] \vdash [\![Q]\!] : \mathsf{unit}$ (by H1 + IH and Lemma 5.4.13)

PROC-DES

$$\frac{\overset{\mathsf{H1}}{g : (T_1, \ldots, T_n) \mapsto T} \qquad \forall i \in [1, n].\ \overset{\mathsf{H2}}{\Gamma \vdash M_i : T_i} \qquad \overset{\mathsf{H3}}{\Gamma, x : T \vdash P} \qquad \overset{\mathsf{H4}}{\Gamma \vdash Q}}{\Gamma \vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

to show: $[\![\Gamma]\!] \vdash \textbf{try let } x = g^* \ ([\![M_1]\!], \ldots, [\![M_n]\!]) \textbf{ in } [\![P]\!] \textbf{ catch } \_ \to [\![Q]\!] : \mathsf{unit}$
I assume g^* exists and $g^* : (\Pi\_ : (\sum_{i \in 1,n} y_i : [\![T_i]\!]).\ [\![T]\!])$

to show (by Exp Try):

- (1) $[\![\Gamma]\!] \vdash \textbf{let } x = g^* \ \widetilde{M} \textbf{ in } [\![P]\!] : \mathsf{unit}$
- $[\![\Gamma]\!] \vdash [\![Q]\!] : \mathsf{unit}$ (by H4+IH and Lemma 5.4.13)

remains to show (by Exp Let on (1)):

- (2) $[\![\Gamma]\!] \vdash g^* \ \widetilde{M} : [\![T]\!]$
- $[\![\Gamma]\!], x : [\![T]\!] \vdash [\![P]\!] : \mathsf{unit}$ (by H3+IH and Lemma 5.4.13)
- $a \notin fv(\mathsf{unit})$

remains to show (by Exp Appl on (2)):

- $[\![\Gamma]\!] \vdash g^* : (\Pi\_ : (\sum_{i \in 1,n} y_i : [\![T_i]\!]).\ [\![T]\!])$ (by def of g^*)
- $[\![\Gamma]\!] \vdash [\![\widetilde{M}]\!] : [\![\widetilde{T}]\!]$ (by H2)

PROC-SPLIT

$$\frac{\Gamma \vdash M : \langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\} \quad \text{H1} \qquad \Gamma, x_1 : T_1, \ldots, x_n : T_n, \langle x_1, \ldots, x_n \rangle = M, C\{\widetilde{x}/\widetilde{y}\} \vdash P \quad \text{H2}}{\Gamma \vdash \mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P}$$

As this translation is defined recursively I prove this case by induction on the length of the tuple.

Base case: where $M : \langle y_{n-1} : T_{n-1}, y_n : T_n \rangle \{C\}$

to show: $[\![\Gamma]\!] \vdash \textbf{let } (x_{n-1}, z_{n-1}) = M \textbf{ in let } (x_n, z_n) = z_{n-1} \textbf{ in } [\![P]\!] : \mathsf{unit}$

to show (by Exp Split):

- $[\![\Gamma]\!] \vdash M : \Sigma y_{n-1} : T_{n-1}.\ (\Sigma y_n : T_n.\ \mathsf{unit})$ (by H1 + the way I translate tuples in Subsection 5.2.1)
- (2) $[\![\Gamma]\!], x_{n-1} : T_{n-1}, z_{n-1} : (\Sigma y_n : T_n.\ \mathsf{unit}), \_ : \{(x_{n-1}, z_{n-1}) = M\} \vdash \textbf{let } (x_n, z_n) = z_{n-1} \textbf{ in } [\![P]\!] : \mathsf{unit}$
- $\{x_{n-1}, z_{n-1}\} \cap fv(\mathsf{unit}) = \emptyset$ (obvious)

to show (2) (by Exp Split):

- $[\![\Gamma]\!], x_{n-1} : T_{n-1}, z_{n-1} : (\Sigma y_n : T_n.\ \mathsf{unit}), \_ : \{(x_{n-1}, z_{n-1}) = M\} \vdash z_{n-1} : \Sigma y_n : T_n.\ \mathsf{unit}$ (obvious)
- $[\![\Gamma]\!], x_{n-1} : T_{n-1}, z_{n-1} : (\Sigma y_n : T_n.\ \mathsf{unit}), \_ : \{(x_{n-1}, z_{n-1}) = M\}, x_n : T_n, z_n : \mathsf{unit}, \_ : \{(x_n, z_n) = z_{n-1}\} \vdash [\![P]\!] : \mathsf{unit}$ (by H2+IH and Lemma 5.4.13)

- $\{x_n, z_n\} \cap fv(\mathsf{unit}) = \emptyset$ (obvious)

Recursive case: where $M : \langle y_i : T_i, y_{i+1} : T_{i+1}, \ldots, y_n : T_n \rangle \{C\}$

to show: $[\![\Gamma]\!] \vdash \mathbf{let}\ (x_i, z_i) = M\ \mathbf{in}\ [\![\mathsf{let}\ \langle x_{i+1}, \ldots, x_n \rangle = z_i\ \mathsf{in}\ P]\!] : \mathsf{unit}$

to show by (Exp Split):

- $[\![\Gamma]\!] \vdash M : \Sigma y_i : T_i.\ (\sum_{j \in i+1,n} y_j : T_j.\{C\})$ (by H1 + the way I translate tuples in Subsection 5.2.1)
- $[\![\Gamma]\!], x_i : T_i, z_i : (\sum_{j \in i+1,n} y_j : T_j.\{C\}), \_ : \{(x_i, z_i) = M\} [\![\mathsf{let}\ \langle x_{i+1}, \ldots, x_n \rangle = z_i\ \mathsf{in}\ P]\!] : \mathsf{unit}$ (by H2 + Induction on the recursive case)
- $\{x_i, z_i\} \cap fv(\mathsf{unit}) = \emptyset$ (obvious)

PROC-ASSUME

$$\frac{\overset{\text{H1}}{\Gamma, C \vdash \diamond}}{\Gamma \vdash \mathsf{assume}(C)}$$

to show (by Exp Assume):

- $[\![\Gamma]\!] \vdash \diamond$ (by H1 and Lemma 5.4.8)
- $free(C) \subseteq dom([\![\Gamma]\!])$ (by H1 and Lemma 5.4.7)

The return type of $\mathbf{assume}\ C$ is $\{C\}$. To fulfil Lemma 5.4.13 we need a return type unit. By (Sub Refine Left) $\{C\} <: \mathsf{unit}$.

PROC-ASSERT

$$\frac{\overset{\text{H1}}{\Gamma \vdash \diamond} \qquad \overset{\text{H2}}{\mathsf{forms}(\Gamma) \models C}}{\Gamma \vdash \mathsf{assert}(C)}$$

to show (by Exp Assert): $[\![\Gamma]\!] \models C$

it remains to show (Derive)

- $[\![\Gamma]\!] \vdash \diamond$ (by H1 + Lemma 5.4.8)
- $free(C) \subseteq dom([\![\Gamma]\!])$ (follows implicitly from H2)
- $\mathsf{forms}([\![\Gamma]\!]) \models C$ (by H2 and Lemma 5.4.3)

□

6 Implementation

In the previous chapter I formalised a translation from Spi to F# and showed that this translation preserves security.

To achieve this in practice I wrote a code generator that generates F# code from Spi. I also made the typechecker from [BHM08] adjustable so that new types, constructors and destructors can easily be added.

6.1 Extensible typechecker

The original version of the typechecker [BHM08] had all types, constructors, destructors, kinding rules and so on hard coded. But since I needed a modified typechecker a more general approach was needed.

The rules are now in a config file that specifies the types, their kinding and subtyping rules as well as constructors and destructors and their typing. Instead of feeding the config file to the typechecker at the same time as the protocol I took a two step approach.

In the first step the code of the typechecker is generated using a config file. This code generator, which has nothing to do with the code generator outlined in the next chapter, uses template files and replaces certain sections with generated code. In a second step the generated typechecker is compiled using the OCaml compiler and yields a typechecker executable. This typechecker can then be run on Spi protocols that adhere to the variant of the calculus specified in the config file.

This approach has the advantage that the generated typechecker is checked by the OCaml compiler and no refactoring of the typechecker was necessary.

A separate manual [Tar08c] for the code generation process of the typechecker is available together with a development design document [Tar08b].

6.2 F# code generator

The F# code generator takes a Spi protocol and outputs two files: a .fs file and a .fs7 file. The .fs file contains the implementation of the protocol and can be directly compiled by the F# compiler. The .fs7 file contains an interface definition for the .fs file. Unlike a normal F#

interface the .fs7 file may contain RCF specific types, especially refinement types. The F7 typechecker uses both the .fs and the .fs7 file to typecheck an F# module[1].

This separation between the typed interface and the implementation is a major difference from the theoretical RCF that does not make such a distinction. The .fs file cannot contain RCF specific types as it has to be fully compatible with the F# compiler. The .fs7 file on the other hand can only talk about functions and not about local variables within functions. This means that in order to enforce a certain type for a local variable I need to generate an auxiliary function that returns this type.

6.2.1 Symbolic vs. Concrete

During compile time the code can be linked against two different libraries, both having the same interface. The symbolic library does not do any actual cryptographic operations, but rather hides objects in abstract types. However, it is useful for prototyping and debugging and it can be verified by F7.

The concrete library on the other hand does perform the actual cryptographic operations required by the protocol. It was not formally verified and one has to trust the .net/Windows implementation of those functions. Additionally real cryptographic operations can only be performed on byte streams, which means that objects need to be serialised and deserialised. Even though the symbolic and concrete library have the same interface the concrete one can cause run time exceptions where the symbolic one works just fine because of the deserialisation and subsequent type casting.

6.2.2 Shortcomings of F7

In order for the .fs files to be fully compatible with the F# compiler they must not contain any other F7 specific syntactic constructs. The current version of F7 therefore has some shortcomings, which influence my code generation:

1. ↱ does not exist in F7. The closest construct is Fork, which is not completely equivalent to ↱. Fork requires a functions as an argument and environment extraction for functions is not defined. In practice this means assumes in one branch of Fork do not apply to the other branch. I work around this by extracting assumes out of the Fork and put them in front of the Fork.
2. Tuples constructed in F# code do not carry the formulas that were attached to the types of the tuple elements. To work around this I have to create an auxiliary function that specifies the tuple elements' types as arguments and the desired return type.
3. Auxiliary functions are needed to specify refinement types for local variables because refinement types are not allowed in .fs files.

[1]Every F# programme consists of one or more modules. Typically every .fs file represents one module.

6.2.3 Details of the code generation

In this section I will give a high level overview on how the generation works. For more details please refer to the Spi2RCF development design document [Tar08a].

The code generation works in a number of steps:

1. The protocol is first translated into A-NF. To achieve this I search for nested terms, extract them into a new let, where I assign a fresh variable to the term, and replace any occurrence of the term with the new variable.

2. Each predicate is translated into a data type with one constructor and as many arguments as specified by the predicate arity.

3. The main process, which is denoted in the Spi protocol by the keyword "process" (see Subsection 6.3.1), is translated. This translation does not generate any code but returns the types of global variables.

4. A function is generated for each named process. The global variables obtained in step 3 are passed as arguments if needed. The new functions are placed in the .fs file, while the types of the arguments and return types are placed in the .fs7 file. Also in this process auxiliary functions are generated to work around the shortcomings of F7 above.

5. In a last step the main process is translated again, this time the arguments of the named processes are known and the code is finally output.

6.2.4 Restrictions

My code generator is currently a prototype and has some restrictions, that should be resolved in future versions:

- Currently only linking against the symbolic library works; the concrete library causes run time errors. The reason for this is, that on the F# side I rely solely on type inference by the compiler. In case of serialisation and deserialisation the resulting objects have to be cast to the required type, which is done implicitly in F#. This cast, however, fails because the inferred type is not the same as the actual type of the object. This can be fixed by making type annotations in the .fs file which may require type inference at code generation time.

- Only global assumes may contain formulas. Assumes in functions may only talk about predicates. This is because the .fs files cannot contain formulas. For simplicity I require that global assumes have to be in a process called "policy" to be recognised.

- Channels can only transport messages of one type, even if the channel is of type (Un)chan. This is because the F# compiler does not understand subtyping of Un and basically treats it as α. One could work around this by serialising all data and have all channels transport bytes.
- All variables in the Spi protocol have to have a unique name because the code generator ignores the scope of the variables. This can be overcome by making the code generator aware of variable scopes.

6.3 Example

In Section 4.3 I presented the A-NF form of the example protocol. Then, in Section 5.3 I presented the translation into RCF syntax.

6.3.1 The .spi file

The presentation of the example protocol in Section 4.3 uses the Spi syntax outlined in Chapter 2. The input for the typechecker and the code generator looks different:

```
1  (** A very simple, one message protocol *)
2
3  predicate Authentic(*1*).
4
5  let bob =
6      in(c2, vkA);
7      in(cm, m);
8      let m1 = check(m,vkA) in
9      let <m3> = m1 in
10     assert (*#Authentic(m3)*).
11
12 let alice =
13     new sigA (*: SigKey(<x1:Un>[*#Authentic(x1)*]) *);
14     let th1 = vk(sigA) in
15     out(c2, th1)|
16     out(c, th1)|
17     new m (*: Un *);
18     (assume (*#Authentic(m)*)
19     |    let th2 = <m(*=x*)>(*#Authentic(x)*) in
20          let th3 = sign(th2,sigA) in
21          out(cm,th3)).
22
23 process
24     new c (*: Ch Un *);
25     new c2 (*: Ch VerKey(<x1:Un>[*#Authentic(x1)*]) *);
26     new cm (*: Ch Un *);
27     (bob | alice)
```

The main difference to notice here is that Alice and Bob are no longer inlined, but they are given a name and later refered to by this name. The actual protocol starts at `process`. At line 27 the protocol then executes Bob and Alice in parallel as if they were inlined.

Below are the two files that are output by the Spi2F# code generator when it translates the above protocol:

6.3.2 The .fs file

```
1  #light "off"
2
3  // open some standard namespaces
4  open Spi
5  open Crypto
6  open Pi
7
8  type ('a) pAuthentic = Authentic of 'a
9
10
11 let bob  cm c2 () =
12     let vkA = recv c2 in
13     let m = recv cm in
14     let m1 = verif vkA m in
15     let (m3,_) = m1 in
16     let (_,dummy1) = m1 in
17     let dummy2 = exercise dummy1 in
18     let _ = expect (Authentic(m3)) in
19      ()
20
21 let mkSigKey4  () = mkSigKey ()
22
23 let mkPair5  x = (x,())
24
25 let alice  cm c c2 () =
26     let sigA = mkSigKey4  () in
27     let th1 = (vk sigA) in
28     let _ = send c2 th1 in
29     let _ = send c th1 in
30     let m = mkUn () in
31     let _ = assume (Authentic(m)) in
32     let th2 = (mkPair5  m ) in
33     let th3 = (sign sigA th2) in
34     let _ = send cm th3 in
35      ()
36
37 let mkCh6  () = mkCh ()
38
39 let mkCh7  () = mkCh ()
```

```
40
41 let mkCh8  () = mkCh ()
42
43 let main  () =
44     let c = mkCh6  () in
45     let c2 = mkCh7  () in
46     let cm = mkCh8  () in
47     let _ = Fork(
48         (fun () -> bob  cm c2 ()),
49         (fun () -> alice  cm c c2 ())
50     ) in
51      ()
52
53
54 ;;
55
56 let _ = main ()
```

The compiler instruction `#light "off"` is used to define the syntax the file uses. The `open` statement opens the modules `Crypto` and `Pi` that are part of the F7 library, while `Spi` is a module written by me that offers some additional functions. `Spi` has been verified with the F7 typechecker.

The predicate `Authentic` has been translated into a data type with one argument of type `'a`. This `'a` can be a different type every time the data type is instantiated.

We can see here that Bob and Alice have been turned into functions and the main process is called `main` and runs Bob and Alice in parallel. Line 56 ensures that the execution of the compiled programme will start at `main`.

In line 15 Bob splits `m1`, which is a tuple of arity 1 in the Spi protocol. The reason is the translation of tuples (Subsection 5.2.1) that always adds one more element of type `unit` at the end.

The Assume in line 31 was previously inside a parallel process. It has been moved out due to shortcoming 1 in Subsection 6.2.2. The Fork was then unnecessary and has been removed.

The `mkCh{6,7,8}` functions are the auxiliary functions mentioned above. For the F# compiler they are useless as one could call `mkCh` directly, but they are needed in the F7 interface file below to enforce F7 specific refinement types for the return values. The same is true for `mkSigKey4`. `mkPair5` returns a pair with a refinement type for the second element

6.3.3 The .fs7 file

```
1 // open some standard namespaces
2 open Spi
3 open Crypto
4 open Pi
```

```
5
6  type ('a) pAuthentic = Authentic of 'a
7
8  val bob : cm:(Un) chan
9      -> c2:(((x1:Un*(foo:unit{Authentic(x1)}))) verifkey) chan
10     -> unit -> unit
11
12 val mkSigKey4 :  unit -> ((x1:Un*(foo:unit{Authentic(x1)}))) SigKey
13
14 val mkPair5 : x:'a{Authentic(x)} -> (x:'a * (unit{Authentic(x)}))
15
16 val alice : cm:(Un) chan -> c:(Un) chan
17     -> c2:(((x1:Un*(foo:unit{Authentic(x1)}))) verifkey) chan
18     -> unit -> unit
19
20 val mkCh6 :  unit -> (Un) chan
21
22 val mkCh7 :  unit
23     -> (((x1:Un*(foo:unit{Authentic(x1)}))) verifkey) chan
24
25 val mkCh8 :  unit -> (Un) chan
26
27 val main : unit -> unit
```

The opened modules and the predicate need to be repeated in the interface file.

Bob and Alice take the channels as arguments plus one additional unit and return unit. The additional unit argument is not needed, but is added during the genration in case a function has no other arguments. All arguments are given names in case a formula in a refinement type refers to them. This is not the case in this example, however. These names have nothing to do with the argument names in the .fs file. The code generator just reuses those names here.

The function `mkPair5` takes only one argument and creates a pair out of that. This is again due to the way tuples are translated in Subsection 5.2.1; the last element always being a refinement type.

Some of the auxiliary functions have a refinement type as a return type. This type could not be given to a local variable in the .fs file so these functions are needed. As only the F7 typechecker sees this file all the `mkCh` functions are considered to return `'a chan` for the F# compiler. Auxiliary functions that do not have a refinement type are not needed, but are generated anyway by the prototype.

7 Conclusion

7.1 Summary

This thesis contributes to secure protocol implementations by allowing protocols expressed in the Spi calculus to be translated into F# code.

I presented a formal definition of this code generator and a proof that a protocol that typechecks in Spi will also typecheck in F# using F7. This ensures that secrecy and authentication properties carry over. As an intermediate step of the translation to F# I defined a normal form for Spi protocols and gave a translation from an ordinary Spi protocol to this normal form. I also outlined the implementation of this translation that caused a number of problems because RCF and the input required by F7 are different. The implementation allows real code to be translated, typechecked and executed against the symbolic library.

7.2 Related work

This is not the first attempt to build a code generator for Spi. For instance, Pironti et al. created a code generator from Spi to Java [PSD04, PS07]. Their spi2java code generator is much more mature than my prototype and it can actually produce real-world protocol implementations, like a working SSL client for example. However, they cannot guarantee the generated code has the same security properties as the Spi protocol. They aim to be as close to the protocol as possible with their code generation, but Java currently has no security typechecker available. This work was recently extended in [Bus08] with a new code generator named expi2java. Busenius adds a new type system featuring nested types, and is more flexible, extensible, customisable and interoperable than spi2java.

During the code generation phase spi2java takes not just the Spi protocol as input, but an XML config file as well. This config file can specify the classes to use for representation of data types, as well as encryption/decryption parameters and serialisation/deserialisation parameters. The generated code is then embedded in a template Java application.

A similar generator called ACG-C# was written by Jeon et al. in [JKC05]. ACG-C# requires an additional step in which Casper translates an abstract protocol into a CSP script [RS01] that is verified. The C# code is then generated from this CSP script. They have chosen C# as a target because of the integrated .net code access security features.

7.3 Future Work

A main drawback of the translation defined in this thesis is that it is partial: formulas must not contain any constructors and destructors at all, this makes the automated translation of a number of existing Spi protocols impossible. This is not trivial to resolve and will require further investigation.

The main aim of a code generator is to generate protocol implementations compatible to existing standards, just as spi2java and expi2java do today. There is still a lot missing from my prototype to achieve this:

For one, messages cannot cross process boundaries, meaning server and client have to run in the same process. To change this client and server need more information, for instance a URI to connect to and a port to listen to.

In order to be interoperable with existing protocol implementations the encoding to use for data must be specified. At the same time the cryptographic functions need information on which algorithm to use and how long keys are supposed to be. Some protocols may even have a negotiation phase, which would need to be encoded in Spi.

The above could be solved using some config files and code templates that can be used during the translation phase, as currently done by spi2java and expi2java.

Some information is, however, not available until runtime. Messages for example are currently just abstract units, but to be of any use they need to have content that is not known at code generation time. Also a client usually needs a UI and a server needs some control mechanism to shut it down. To provide this code, which is probably several times longer than the generated code, at code generation time seems unpractical. I suggest compiling the generated code into a DLL that can be used by any application that implements the protocol. The host application is not limited to F#, but can also be written in C# and many other languages that support the CLI [ECM06].

Other aspects of the code generator not addressed in the prototype are efficiency and extensibility. To achieve the latter a plug-in architecture would be one possibility. Also some script code could be fed to the code generator along with the Spi protocol.

Another area that was not addressed in my current work is zero-knowledge. In [BHM08] zero-knowledge is part of the type system and the typechecker can deal with zero-knowledge. A translation of these types and processes would be an interesting addition to this work.

Bibliography

[AB01] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25--41. Springer-Verlag, 2001.

[AB05] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM (JACM)*, 52(1):102--146, 2005.

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749--786, 1999.

[BBF+08] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. *21st IEEE Computer Security Foundations Symposium (CSF 2008)*, 2008. http://research.microsoft.com/F7/.

[BCFM07] M. Backes, A. Cortesi, R. Focardi, and M. Maffei. A calculus of challenges and responses. In *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 101--116. ACM Press, 2007.

[BCJ+06] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57--87, 2006.

[BFGT06] K. Bhargavan, C. Fournet, A.D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139--152, 2006.

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563--617, 2007.

[BHM08] M. Backes, C. Hriţcu, and M. Maffei. Type-checking Zero-knowledge. *21st IEEE Computer Security Foundations Symposium (CSF 2008)*, 2008. Implementation available at http://www.infsec.cs.uni-sb.de/projects/zk-typechecker.

[Bla05] B. Blanchet. *ProVerif Automatic Cryptographic Protocol Verifier User Manual*, 2005.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1--12. Springer-Verlag, 1998.

[Bou92] Gérard Boudol. Asynchrony and the Pi-calculus. *Research Report*, 1702, 1992.

[Bus08] Alex Busenius. Expi2Java -- An Extensible Code Generator for Security Protocols, 2008.

[Deb08] Debian. DSA-1571-1 openssl -- predictable random number generator, 2008. http://www.debian.org/security/2008/dsa-1571.

[ECM06] ECMA. Common Language Infrastructure (CLI). Standard ECMA-335, June 2006.

[FGM07] C. Fournet, A. Gordon, and S. Maffeis. A Type Discipline for Authorization in Distributed Systems. *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 31--48, 2007.

[Fis03] Dennis Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).

[GJ04] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435--484, 2004.

[GLRV05] J. Goubault-Larrecq, M. Roger, and K.N. Verma. Abstraction and resolution modulo AC: How to verify Diffie--Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219--251, 2005.

[Gun92] C.A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[HJ06] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195--1263, 2006.

[JKC05] C.W. Jeon, I.G. Kim, and J.Y. Choi. Automatic Generation of the C# Code for Security Protocols Verified with Casper/FDR. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications-Volume 2*, pages 507--510. IEEE Computer Society Washington, DC, USA, 2005.

[Joh78] S.C. Johnson. *YACC-Yet Another Compiler-Compiler*. Bell Laboratories, 1978.

[Low96] G. Lowe. Some new attacks upon security protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 162--169, 1996.

[PS07] A. Pironti and R. Sisto. An Experiment in Interoperable Cryptographic Protocol Implementation Using Automatic Code Generation. In *Computers and Communications, 2007. ISCC 2007. IEEE Symposium on*, pages 839--844, 2007.

[PSD04] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004*, volume 1, pages 400--405, 2004.

[RS01] P. Ryan and S. Schneider. *The Modelling and Analysis of Security Protocols: The Csp Approach*. Addison-Wesley Professional, 2001.

[SF93] AMR Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Higher-Order and Symbolic Computation*, 6(3):289--360, 1993.

[Tar08a] Thorsten Tarrach. *Spi2RCF development design document*, 2008. Will be made available with the online version of this thesis.

[Tar08b] Thorsten Tarrach. *Typechecker development design document*, 2008. Will be made available with the online version of this thesis.

[Tar08c] Thorsten Tarrach. *Typechecker manual*, 2008. Will be made available with the online version of this thesis.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29--40, 1996.