Type inference for correspondence types

Andrew D. Gordon[1]

Microsoft Research
Cambridge, United Kingdom

Hans Hüttel and Rene Rydhof Hansen[2,3]

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Abstract

We present a type and effect system for proving correspondence assertions in a π-calculus with polarized channels, dependent pair types and effect terms. Given a process P and a type environment E, we describe how to generate constraints that are formulae in the Alternating Least Fixed-Point (ALFP) logic. A reasonable model of the generated constraints yields a type and effect assignment such that P becomes well-typed with respect to E if and only if this is possible. The formulae generated satisfy a finite model property; a system of constraints is satisfiable if and only if it has a finite model. As a consequence, we obtain the result that type and effect inference in our system is polynomial-time decidable.

Keywords: π-calculus, correspondence assertions, type inference.

1 Introduction

We develop a type and effect system for guaranteeing authenticity in protocols described in a polarized π-calculus. We develop a *type inference* system for the calculus and show how it can be implemented by generating constraints over the Alternating Least Fixed-Point logic [4,5,15] and using the Succinct Solver [16].

Woo and Lam [19] propose correspondence assertions as a method for checking authenticity properties of cryptographic protocols. Points in a protocol are annotated by labelled assertions $\mathsf{begin}(L)$ and $\mathsf{end}(L)$. A protocol is then safe if, in any run, an $\mathsf{end}(L)$ is always preceded by a corresponding $\mathsf{begin}(L)$.

Gordon and Jeffrey [10,11] show how to base type and effect systems for the spi calculus [2] on this idea. The effect of a spi calculus process is an over-approximation of the set of outstanding end-assertions. If a process has empty effect, then it is

[1] Email: adg@microsoft.com
[2] Email: hans@cs.aau.dk
[3] Email: rrh@cs.aau.dk

safe. The approach proposed uses *type checking*; one must specify types and effects of all processes in the protocol specification and check that it becomes well-typed with respect to the type and effect assignment. Gordon and Jeffrey's type systems are the basis for the Cryptyc tool [7], which has been applied to analyze a range of cryptographic protocols.

Blanchet [3] proves correspondences for processes by abstracting their behaviour using Horn clauses, and then applying a resolution-based algorithm; he implements his technique in the ProVerif tool and has evaluated it successfully against many examples. The correctness proof relies on generating a type system for the underlying process calculus (much as in previous work on secrecy [1]). Hence, although it does not expose the type system to its users, ProVerif proves correspondences by inferring types for processes.

Focardi, Maffei and Placella [8] present an inference algorithm for a system of authentication types. Their algorithm infers tags and types for processes in the ρ-spi calculus [6]. They evaluate their algorithm on a series of security protocols. The ρ-spi type system is based on the idea that explicit tags within ciphertexts record the intended meaning of fields such as nonces; hence, the ρ-spi algorithm does not adapt simply to the calculus considered here, which does not use tags.

A series of papers by Nielson and others [4,5,15] has developed a different approach to static analysis of cryptographic protocols expressed in a process calculus, namely via so-called control flow analysis. Here, the approach is to annotate protocol descriptions with process points and define an over-approximation of the sets of names that may appear at each process point. This is done by generating a set of constraints in the ALFP logic, which is a fragment of first-order logic. There exists a polynomial-time satisfiability algorithm which constructs a finite model if one exists and an ALFP tool, the Succinct Solver [16,17] has been developed.

In this paper we combine these approaches and describe a method for *type inference* for a correspondence type system: Given a polarized π-calculus and a type system with dependent pairs and effect terms, one may, given a process P and an a priori type environment E generate constraints that are ALFP formulae, which can then be solved to provide a type and effect assignment if one exists.

In a recent paper [13], Kobayashi and Kikuchi describe another correspondence type system for a π-calculus (without pairs) and a type inference algorithm. Their approach is fundamentally different from ours; effects are rational numbers and constraints are inequalities over rational numbers.

The remainder of our paper is organized as follows. In section 2 we introduce our polarized π-calculus, and in section 3 our type system. Next, in section 4, we describe how constraints can be captured in the ALFP logic. Then, in section 5 we show that a model of the constraints generated will lead to a type and effect assignment such that P becomes well-typed with respect to E. Finally, we show that type inference now becomes possible since the formulae generated satisfy a finite model property; a system of constraints is satisfiable if and only if it has a finite model.

2 A polarized π-calculus with pairs

We consider an asynchronous π-calculus with polarized channels, dependent pairs and effect terms. The polarized π-calculus, introduced in [18], allows for simple encodings of cryptographic primitives, as exemplified in [12].

2.1 Syntax

In the polarized π-calculus that we consider, we allow composite messages built from names. Names must be equipped with a polarity κ in order to be used as subjects of communication prefixes. Messages that do not contain the projection operations **fst** and **snd** are called *values* and are ranged over by v. The set of messages is ranged over by M and its elements are given by

$$\begin{aligned}\kappa &::= + \mid - \\ v &::= a, b, m, n, x, y \ldots \mid \mathbf{ok} \mid \mathbf{pair}(v_1, v_2) \mid v^{\kappa} \\ M &::= a, b, m, n, x, y... \mid \mathbf{ok} \mid \mathbf{pair}(M_1, M_2) \mid \mathbf{fst}(M_1) \mid \mathbf{snd}(M_1) \mid M^{\kappa}\end{aligned}$$

where **ok** is a special effect term; its only purpose is to populate ok-types (see e.g. [9]), introduced in section 3. The type system assigns effects to these **ok** -terms.

The set of process terms contains the standard process constructs of the π-calculus [14] together with local declarations and conditional expressions. Moreover, there are three constructs that do not affect behaviour, namely the *correspondence assertions* **begin** $\ell(M)$ and **end** $\ell(M)$ and the construct **exercise**$(M); P$ which allows us to introduce the effects associated with M. The set of process terms is defined by

$$\begin{aligned}P, Q, R ::= {} & \mathbf{in}(M, a); P \mid \mathbf{!in}(M, a); P \mid \mathbf{out}(M, N) \mid \mathbf{new}\ a : T; P \mid (P \mid Q) \mid \mathbf{0} \\ \mid {} & \mathbf{let}\ x = M\ \mathbf{in}\ P \mid \mathbf{if}\ v_1 = v_2\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ \mid {} & \mathbf{exercise}(M); P \mid \mathbf{begin}\ \ell(M) \mid \mathbf{end}\ \ell(M)\end{aligned}$$

2.2 Semantics

The semantics of our calculus is a standard reduction semantics in the spirit of [14], defined using structural equivalence and the reduction relation $\rightarrow$.

Message Reduction: $M > N$

Let $M > M'$ be the least relation on messages closed with respect to:

$\mathbf{fst}(\mathbf{pair}(M_1, M_2)) > M_1$ (Red Fst)

$\mathbf{snd}(\mathbf{pair}(M_1, M_2)) > M_2$ (Red Snd)

and closed under message constructors.

We write $M \downarrow v$ if $M >^* v$ where v is a value.

The definition of the structural congruence relation $\equiv$ over process terms is that of [14].

The reduction relation for processes is defined below; we write $\kappa_1 \asymp \kappa_2$ if $\kappa_1 = +$ and $\kappa_2 = -$. Also note that **begin** $\ell(M)$ and **end** $\ell(M)$ have no reductions.

Reduction Semantics for Processes: $P \to P'$

$P \equiv Q, Q \to Q', Q' \equiv P' \Rightarrow P \to P'$	(Red Struct)
$P \to P' \Rightarrow P \mid Q \to P' \mid Q$	(Red Par)
$P \to P' \Rightarrow \mathbf{new}\ a; P \to \mathbf{new}\ a; P'$	(Red Res)
$M \downarrow m, \kappa_1 \asymp \kappa_2 \Rightarrow$ $\mathbf{out}(M^{\kappa_1}, N) \mid \mathbf{in}(m^{\kappa_2}, a); P \to P\{^N/_a\}$	(Red Comm)
$M \downarrow m, \kappa_1 \asymp \kappa_2 \Rightarrow$ $\mathbf{out}(M^{\kappa_1}, N) \mid !\mathbf{in}(m^{\kappa_2}, a); P \to P\{^N/_a\} \mid !\mathbf{in}(m^{\kappa_2}, a); P$	(Red Repl)
$M \downarrow v \Rightarrow \mathbf{let}\ x = M\ \mathbf{in}\ P \to P\{^v/_x\}$	(Red True)
$\mathbf{if}\ v = v\ \mathbf{then}\ P\ \mathbf{else}\ Q \to P$	(Red True)
$v \neq v' \Rightarrow \mathbf{if}\ v = v'\ \mathbf{then}\ P\ \mathbf{else}\ Q \to Q$	(Red False)
$\mathbf{exercise}(M); P \to P$	(Red Exc)

We write $P \to^*_{\equiv} Q$ if $P \to^* Q$ or $P \equiv Q$.

2.3 Safety

In this paper, we shall consider *safety*. A process is safe with respect to the correspondence assertions of the annotated protocol if every end-assertion encountered has been preceded by a begin-assertion with the same label.

Definition 2.1 A process P is *safe* if whenever $P \to^*_{\equiv} \mathbf{new}\ \boldsymbol{a}; (\mathbf{end}\ \ell(M) \mid P')$ we have M',P'' with $P' \equiv \mathbf{begin}\ \ell(M') \mid P''$ and $M \equiv M'$.

The extension of our results to *robust safety* where an opponent process can interfere with P, is a topic of ongoing work.

3 A Type and Effect System for Safety

We now define a correspondence type and effect system for statically verifying that a given process is safe.

3.1 Effects and types

Definition 3.1 [Effects] A *simple effect* is a labelled message $\ell(M)$. A *ground effect* is a set of simple effects and is ranged over by S. We let Effects denote the set of ground effects and denote inclusion of effects by $S_1 \leq S_2$.

In general, effects may contain effect variables.

Effects

$S ::=$	effects
$\ell(M)$	effect with label ℓ
S_1, S_2	composite effect
R	effect variable

Our type system contains the following: channel types, for names that are used as communication channels; dependent pair types and ok-types, used to carry an effect.

Dependent pairs are needed when modelling protocol messages in order to formalise dependencies between message parts. Ok-types are convenient in that they provide a clear separation between types and effects in our system and allow for flexibility; using ok-terms it is straightforward to encode effect type systems with latent effects such as that of [13] by using ok-types and **exercise** (see section 6 for more detail).

The set Types denotes ground types (the set of type expressions that have no occurrences of type variables).

Types and Type Variables

$T ::=$	Type
U, V, W	type variable
$\mathbf{Ch}^{\kappa}(T)$	polarized channel type
$\mathbf{Ch}(T)$	channel type
$\mathbf{Ok}(S)$	ok-type
$\mathbf{Pair}(x : T_1, T_2)$	dependent pair type
$T(x)$	parameterized type

3.2 Type and effect assignments

In our type system, all typings assume that the types are defined by a *type and effect assignment*:

Definition 3.2 A *type and effect assignment* Δ is a pair of functions (Δ_T, Δ_S) such that $\Delta_T : TVar \to$ Types and $\Delta_S : EVar \to$ Effects

Here $TVar$ and $EVar$ denote the set of type variables and effect variables respectively.

We write a type and effect assignment as a set of equations

$$\Delta = \{X_i = T_i \mid 1 \leq i \leq m\} \cup \{S_j = s_j \mid 1 \leq j \leq n\}$$

Conversely, any such set of equations defines a type and effect assignment if every type and effect variable occurring is defined exactly once.

In what follows we always tacitly assume type judgments are relative to some type and effect assignment.

3.3 The type system

Type judgements are relative to a type environment, which may contain name typings, effect assumptions and term assignments. Term assignments are required, since local declarations may appear and since locally declared variables may occur in types.

$$E ::= \varnothing \mid E, a : T \mid E, S \mid E, x = M$$

Environments:

$dom(\varnothing) \triangleq \varnothing \qquad dom(E, a : T) \triangleq dom(E) \cup \{a\}$

$dom(E, S) \triangleq dom(E) \quad dom(E, x = M) \triangleq dom(E)$

$effect(\varnothing) \triangleq \varnothing \quad effect(E, a : T) \triangleq effect(E)$
$effect(E, S) \triangleq effect(E) \cup S$
$effect(E, x = M) \triangleq effect(E)$

Environment induced by a process: *begins*(P)

$begins(\textbf{begin}\ \ell(M)) \triangleq \ell(M)$
$begins(\textbf{end}\ \ell(M)) \triangleq \varnothing$
$begins(\textbf{new}\ a; P) \triangleq S_1, \ldots, S_n$
where $\{S_1, \ldots, S_n\} = \{S \in begins(P) \mid a \notin fv(S)\}$
$begins(P_1 \mid P_2) = begins(P_1), begins(P_2)$
$begins(P) = \varnothing$ for any other P

Definition 3.3 [Substitution] A *substitution* β is a finite sequence of assignments $x_1 = M_1, \ldots, x_k = M_k$, The domain of β is the set of variables $dom(\beta) = \{x_1, ..., x_k\}$. ; We write $\beta x = M$ if $x = M \in \beta$. We assume $fv(M) \subseteq dom(\beta)$.

Substitution induced by an environment: subs(E)

$\text{subs}(\varnothing) \triangleq \varnothing \quad \text{subs}(E, a : T) \triangleq \text{subs}(E)$
$\text{subs}(E, S) \triangleq \text{subs}(E) \quad \text{subs}(E, x = M) \triangleq \text{subs}(E), x = M$

We only consider good substitutions. A good substitution β can be seen as a function, introduces no undefined variables and has no circular dependencies.

Good Substitution: $\vdash \beta$

(Subs Empty) $\dfrac{}{\vdash \varnothing}$ (Subs Let) $\dfrac{\vdash \beta \quad x \notin dom(\beta) \quad x \notin fv(M)}{\vdash \beta, x = M}$

In a nested local declaration $\textbf{let}\ x_1 = M_1\ \textbf{in let}\ x_2 = M_2\ \textbf{in}\ P$, M_2 may contain occurrences of x_1. In our definition of a substitution acting on a term, we therefore need to iterate the substitution. This is expressed in the first clause of the following.

Substitution acting on term: $M\beta$

$x\beta \triangleq v$ where $\quad M\beta \downarrow v \quad$ if $\beta x = M$
$x\beta \triangleq x \quad$ if $x \notin dom(\beta)$
$\textbf{fst}(M)\beta \triangleq v$ where $\textbf{fst}(M\beta) \downarrow v$
$\textbf{snd}(M)\beta \triangleq v$ where $\textbf{snd}(M\beta) \downarrow v$
$\textbf{pair}(M_1, M_2)\beta \triangleq v$ where $\textbf{pair}(M_1\beta, M_2\beta) \downarrow v$

Substitution acting on effect: $S\beta$

$\{\ell_1(M_1), \ldots, \ell_k(M_k)\}\beta \triangleq \{\ell_1(M_1\beta), \ldots, \ell_k(M_k\beta)\}$

An environment is *good* if every occurring name is also defined in it:

Good Environment: $E \vdash \diamond$

(Env Empty)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env Typing)
$$\frac{E \vdash \diamond \quad a \notin dom(E), fv(T) \subseteq dom(E)}{E, a : T \vdash \diamond}$$

(Env Effect)
$$\frac{E \vdash \diamond \; fv(S) \subseteq dom(E)}{E, S \vdash \diamond}$$

(Env Assign)
$$\frac{E \vdash \diamond \quad fv(M) \subseteq dom(E) \quad x \in dom(E) \quad \vdash \beta \quad x \notin dom(\beta) \quad \beta = \mathrm{subs}(E)}{E, x = M \vdash \diamond}$$

The type rules for messages are given in Table 1; in (Msg Ok) we instantiate all names occurring in the effect with respect to the substitutions in the environment.

Typed Message: $E \vdash M : T$

(Msg Var)
$$\frac{E \vdash \diamond \quad E \vdash M : X \quad \Delta(X) = T}{E \vdash M : T}$$

(Msg Name)
$$\frac{E \vdash \diamond \quad E = E', a : T, E''}{E \vdash a : T}$$

(Msg Fst)
$$\frac{E \vdash M : \mathbf{Pair}(x : T, T'(x))}{E \vdash \mathbf{fst}(M) : T}$$

(Msg Snd)
$$\frac{E \vdash M : \mathbf{Pair}(x : T, T'(x))}{E \vdash \mathbf{snd}(M) : T'(\mathbf{fst}(M))}$$

(Msg Capa)
$$\frac{E \vdash M : \mathbf{Ch}(T)}{E \vdash M^{\kappa} : \mathbf{Ch}^{\kappa}(T)}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash M' : T'(M)}{E \vdash \mathbf{pair}(M, M') : \mathbf{Pair}(x : T, T'(x))}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad S \leq (\mathit{effect}(E))\beta \quad \beta = \mathrm{subs}(E)}{E \vdash \mathbf{ok} : \mathbf{Ok}(S)}$$

Table 1
Type rules for messages

The type system for guaranteeing safety for processes is specified in Table 2. Note that in the (Proc If) rule terms M and N need not have the same type; they may contain different effects but have the same syntactic structure. To capture this, we require that M and N are unifiable.

Example 3.4 Let $E = b : B$ for some type B and let

$$\begin{aligned}\Delta &= \{T = \mathbf{Ch}(\mathbf{Pair}(x : B, \mathsf{Ok}(\{\ell(x)\})))\} \\ P &= \mathbf{new}\ a : T; (\mathbf{begin}\ \ell(b) \mid \mathbf{out}(a^{+}, (b, \mathbf{ok}\,)) \\ &\quad \mid \mathbf{in}(a^{-}, x); \mathbf{exercise}(\mathbf{snd}(x)); \mathbf{end}\ \ell(\mathbf{fst}(x)))\end{aligned}$$

Good Process: $E \vdash P$

(Proc In) (μ either ! or nothing)

$$\frac{E \vdash M : \mathbf{Ch}^{-}(T) \quad E, a : T \vdash P}{E \vdash \mu\mathbf{in}(M, a); P}$$

(Proc Out)

$$\frac{E \vdash M : \mathbf{Ch}^{+}(T) \quad E \vdash N : T}{E \vdash \mathbf{out}(M, N)}$$

(Proc Par)

$$\frac{E, \mathit{begins}(P_2) \vdash P_1 \quad E, \mathit{begins}(P_1) \vdash P_2}{E \vdash P_1 \mid P_2}$$

(Proc Zero)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{0}}$$

(Proc Res)

$$\frac{E, a : T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ a : T; P}$$

(Proc Begin)

$$\frac{E \vdash \diamond \quad \mathit{fv}(M) \subseteq \mathit{dom}(E) \quad E \vdash M : T}{E \vdash \mathbf{begin}\ \ell(M)}$$

(Proc End)

$$\frac{E \vdash \diamond \quad \mathit{fv}(M) \subseteq \mathit{dom}(E) \quad \ell(M) \leq \mathit{effect}(E)}{E \vdash \mathbf{end}\ \ell(M)}$$

(Proc Exercise)

$$\frac{E \vdash M : \mathbf{Ok}(S) \quad E, S \vdash P}{E \vdash \mathbf{exercise}(M); P}$$

(Proc Let)

$$\frac{E \vdash M : T \quad E \vdash P\{^{M}/_{x}\}}{E \vdash \mathbf{let}\ x = M\ \mathbf{in}\ P}$$

(Proc If)

$$\frac{E \vdash v_1 : T \quad E \vdash v_2 : T' \quad (\mathit{mgu}(v_1, v_2)\ \text{exists} \Rightarrow E \vdash P(\mathit{mgu}(v_1, v_2))) \quad E \vdash Q}{E \vdash \mathbf{if}\ v_1 = v_2\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

Table 2
Type rules for processes

Then $E \vdash P$. Notice the use of a dependent pair type for typing the pair $(b, \mathbf{ok})$; the effect of **ok** depends upon b.

Typability implies safety; here, we say that a type is *generative* iff it is a channel type. An environment E is *generative* iff $E(x)$ is generative for all $x \in \mathit{dom}(E)$.

Theorem 3.5 (Safety) *If $E \vdash P$ and E is generative and $\mathit{effect}(E) \equiv \varnothing$ then P is safe.*

Note that our type system is stronger than systems with latent effects such as [13]. In these systems, all channels have a latent effect. In our case, since we allow dependent pair types and effect terms, the usage of a channel a with type $\mathbf{Ch}(T)[S]$, where S is a latent effect can be encoded as

$$[\![\mathbf{out}(M, N)]\!] = \mathbf{out}(M, (N, \mathbf{ok}))$$
$$[\![\mu\mathbf{in}(M, x); P]\!] = \mu\mathbf{in}(M, x); \mathbf{exercise}(\mathbf{snd}(x)); [\![P\{^{\mathbf{fst}(x)}/_{x}\}]\!]$$

4 Type inference

We now show how to extract information from a process expression P and a type environment E that lets us construct a type and effect assignment Δ such that P is well-typed in E iff this is possible.

4.1 The type inference problem

Given a process P and a type context E, is it possible to construct a type and effect assignment Δ such that $E \vdash P$?

Type inference consists in 'running the type system backwards' and collecting the set of constraints that arise from the conditions that must be satisfied in order for the typing rules to apply.

4.2 Expressing constraints

We first express our constraints in a high-level constraint language that corresponds to the side conditions of the typing rules.

High-level constraints

$\phi ::=$	atomic high-level constraint
$S_1 \leq S_2$	effect inclusion
$S_1 = S_2$	effect equality
$T_1 = T_2$	simple type equality
$T = U^m(x)$	abstracted type equality
$T = U(n)$	applied type equality
$M_1 = M_2$	term equality
$\mathtt{Fresh}(x, N)$	x is a fresh name with respect to N
$M \in \mathsf{Pair}(M_1, M_2)$	witness of pair type
$M \in \mathsf{Ch}(N)$	witness of channel type
$\varphi ::=$	composite constraint
$\varphi_1 \wedge \varphi_2$	conjunction
$\varphi_1 \Rightarrow \varphi_2$	implication
$\forall x.\varphi_1$	universal quantification

These constraints are then translated into the ALFP logic, which is a fragment of first-order logic introduced in [17]. In the syntax below we assume a fixed countable set of variables $\mathcal{X}$, and a finite set $\mathcal{R}$ of relation symbols.

$$t ::= c \mid x$$
$$\phi ::= R(x_1, \ldots, x_k) \mid \neg R(x_1, \ldots, x_k) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x : \phi \mid \forall x : \phi$$
$$\psi ::= R(x_1, \ldots, x_k) \mid \mathbf{t} \mid \psi_1 \wedge \psi_2 \mid \phi \Rightarrow \psi \mid \forall x : \psi$$

Here $R(x_1, \ldots, x_k)$ is a arbitrary k-place relation symbol. Formulae ψ are called *clauses*, while formulae ϕ are called *preconditions*; here, $\exists$ is allowed.

ALFP formulae are interpreted over first-order structures.

Definition 4.1 A *first-order structure* is a triple $\mathcal{M} = (\mathcal{U}, \rho, \sigma)$ where $\mathcal{U}$ is a finite universe of values and $\rho : \mathcal{R} \to \bigcup_{k \geq 0} \mathcal{P}(\mathcal{U}^k)$ and $\sigma : \mathcal{X} \to \mathcal{P}(\mathcal{U})$ are interpretations

of relation symbols and variables. We often omit mention of the universe and write a first-order structure as a pair (ρ, σ).

Theorem 4.2 ([17]) *Given a σ, the least solution of a $\psi = \psi_1 \wedge \ldots \psi_m$ can be found in time $O(\sum_{i=1}^{m} n_i N^{r_i})$ where n_i is the size of ψ_i, r_i is the maximal depth of quantifiers in ψ_i and N is the size of the universe.*

4.3 Encoding terms and types

The translation of high-level constraints into ALFP makes use of an encoding of message terms, types and effects.

Terms

We encode a message term by representing its abstract syntax tree. Each term constructor has a corresponding relation, so we introduce the relations Name, $\mathsf{TermPair}$, OkTerm, Capa, Fst and Snd.

For every term M the encoding $[\![M]\!]$ is a pair (m, ψ) where m is called the *root name* and ψ is a constraint that describes which relation m must inhabit along with any constraints caused by subterms of M. We can then encode an equality $M = N$ as $[\![M = N]\!] = m = n \wedge \psi \wedge \psi'$ where $[\![M]\!] = (m, \psi), [\![N]\!] = (n, \psi')$, and we encode the congruence rules for message terms by the conjuction of the following clauses:

$$\forall m.\forall n.\forall n_1.\forall n_2.m \in \mathsf{Fst}(n) \wedge n \in \mathsf{TermPair}(n_1, n_2) \Rightarrow m = n_1$$
$$\forall m.\forall n.\forall n_1.\forall n_2.m \in \mathsf{Snd}(n) \wedge n \in \mathsf{TermPair}(n_1, n_2) \Rightarrow m = n_2$$

Types

We encode types as relations such that a term M has type T if and only if the relation T is inhabited by the root name of M.

We introduce three *relation constructors*, **Pair**, **Ch**, and **Ok**, that correspond to the type constructors. The set of type relation symbols is given by the syntax

$$R ::= U \mid \mathbf{Pair}_{X_1,X_2} \mid \mathbf{Ch}_{X_1} \mid \mathbf{Ok}_R$$

where $U \in TVar$ and $R \in EVar$.

For type environment E we let the relation E_e stand for the effects in E.

Dependent pair types

To every dependent type $U^M(x)$ obtained by abstracting out a term M of type T, we associate a relation $U^m(x)$, where m is the root name of M. $U^m(x)$ is inhabited by elements whose type is found by replacing all occurrences of m in the definition of U by an arbitrary element m' of type T. For any type U we can capture this by the clause

$$\begin{aligned}\forall n.\forall n'.\forall m.\forall m'.(m \in T) \Rightarrow (m' \in T)\\ \Rightarrow((n \in U \Rightarrow n' = n\{^{m'}/_{m}\} \Rightarrow n' \in U^m(x))\\ \wedge(n' \in U^m(x) \Rightarrow n' = \{^{m'}/_{m}\} \Rightarrow n \in U))\end{aligned}$$

Here, $n = n'\{^{m'}/_{m}\}$ is syntactic sugar for a four-place relation $R(n, n', m, m')$, defined by a set of auxiliary clauses. The intended meaning is that $n = n'\{^{m'}/_{m}\}$ if n and n' are root names of terms M and M' such that $E \vdash M : T$ and $E \vdash M' : T'$ where $T' = T\{^{m'}/_{m}\}$.

We encode the application $V(n)$ of a dependent type to a term by a conjuction of auxiliary clauses for each type abstraction $U^M(x)$ found in the constraints generated; note that V must be a dependent type for the application to make sense.

$$\begin{aligned}\forall n.\forall m'.\forall m.\forall m_1.\forall m_2.\forall m_3.[\![V = U^M(x)]\!] \Rightarrow \\ (m_3 = m_2\{^{m'}/_{m}\} \wedge m_3 \in U^m(x)) \Rightarrow \\ m_1 = m_2\{^{n}/_{m}\} \Rightarrow m_1 \in V(n)\end{aligned}$$

$$\begin{aligned}\forall n.\forall m'.\forall m.\forall m_1.\forall m_2.\forall m_3.[\![V = U^M(x)]\!] \Rightarrow \\ (m_1 \in V(n) \wedge m_1 = m_2\{^{m}/_{n}\}) \Rightarrow \\ m_3 = m_2\{^{m'}/_{m}\} \Rightarrow m_3 \in U^m(x)\end{aligned}$$

4.4 Translating high-level constraints into ALFP

Type and effect constraints

If T_1 and T_2 are type relation symbols, we translate a type equation as follows:

$$[\![T_1 = T_2]\!] = \forall y.(y \in T_1 \Rightarrow y \in T_2) \wedge (y \in T_2 \Rightarrow y \in T_1)$$

If R_1 and R_2 are effect relation symbols, we translate *effect inclusions* as:

$$[\![R_1 \leq R_2]\!] = \forall x.\forall \ell.(\ell, x) \in R_1 \Rightarrow (\ell, x) \in R_2$$

For equations on the form $V = \mathsf{Ok}(S)$ we construct the formula

$$\begin{aligned}[\![V = \mathsf{Ok}(S)]\!] = (\forall \ell.\forall x.(\ell, x) \in V \Rightarrow (\ell, x) \in S) \\ \wedge(\forall \ell.\forall x.(\ell, x) \in S \Rightarrow (\ell, x) \in V)\end{aligned}$$

4.5 Generating constraints

We describe constraint generation by two big-step semantics, whose transition rules are found in Table 3.

Constraints from messages

In the judgment $E \vdash M \rightsquigarrow T; \psi; \tilde{\mathbf{U}}$, ψ denotes the generated constraint. T denotes the primary relation symbol introduced (corresponding to a type variable) and $\tilde{\mathbf{U}}$ denotes the set of auxiliary relation symbols introduced.

In the constraints generated for messages, we record the witnesses in (Msg Pair), where the subterms must inhabit the witness relation Pair for pair types. This is needed in the account of dependent types.

Constraints from processes

In the judgment $E \vdash P \rightsquigarrow \psi_1; \tilde{\mathbf{U}}$, ψ_1 denotes the constraint generated. We again let $\tilde{\mathbf{U}}$ denote the set of auxiliary relation symbols introduced.

In the rules (Proc In) and (Proc Out) we record the witness in the witness relation Chan. This, too, is needed in our account of dependent types.

Example 4.3 For the process $P = \mathbf{new}\ a : T; P_1$ and environment E from Example 3.4, we have $E \vdash P \rightsquigarrow \psi_P; \tilde{\mathbf{U}}$, where $\psi_P = [\![N = \mathrm{n}(P_1)]\!] \wedge \forall a.(\mathtt{Fresh}(a, N) \Rightarrow \psi_1; \tilde{\mathbf{U_1}}; T)$ where $E, a : T \vdash P_1 \rightsquigarrow \psi_1; \tilde{\mathbf{U_1}}$. Conjuncts from ψ_1 include

$$\begin{aligned}\psi_{12} = {} & [\![a^+ \in \mathsf{Chan}^+(m)]\!] \wedge [\![m = (b, \mathbf{ok}\ )]\!] \wedge [\![m \in T_2]\!] \\ & \wedge [\![T = \mathbf{Ch}^+(T_2)]\!] \wedge \psi_{121}\end{aligned}$$

The constraints tell us that the polarized channel a^+ must carry terms of type T and that m, the root name of $(b, \mathbf{ok}\ )$ inhabits this type.

4.6 Constraints representing unification conditions

A solution must respect the normal requirements of term unification, namely that composite types are equal if and only if their corresponding immediate constituents are. For this reason we introduce additional clauses for each type constructor.

As an example, for every pair of relation symbols $\mathbf{Ch}^\kappa(T_1), \mathbf{Ch}^\kappa(T_2)$ introduced in ψ we introduce the clauses

$$[\![\mathbf{Ch}_{T_1} = \mathbf{Ch}_{T_2}]\!] \Rightarrow [\![T_1 = T_2]\!] \wedge [\![T_1 = T_2]\!] \Rightarrow [\![\mathbf{Ch}_{T_1} = \mathbf{Ch}_{T_2}]\!]$$

We represent failure of unification by a relation Fail which is inhabited when a name inhabits types with different type constructors. A sample clause is

$$\forall x.x \in \mathbf{Pair}_{T_1,T_2} \wedge x \in \mathbf{Ch}_{T_3} \Rightarrow x \in \mathsf{Fail}$$

Given the total constraint ψ_P that has been generated, the conjuction of ψ_P and the auxiliary clauses is called the *saturation* of ψ_P, denoted $\mathcal{S}(\psi_P)$.

Example 4.4 For the process $P = \mathbf{new}\ a : T; P_1$ and environment E from Example 3.4, we have $E \vdash P \rightsquigarrow \psi_P; \tilde{\mathbf{U}}$, we have $\psi_P = [\![N = \mathrm{n}(P_1)]\!] \wedge \forall a.(\mathtt{Fresh}(a, N) \Rightarrow \psi_1; \tilde{\mathbf{U_1}}; T)$ where $E, a : T \vdash P_1 \rightsquigarrow \psi_1; \tilde{\mathbf{U_1}}$. Conjuncts from ψ_1 include

$$\begin{aligned}\psi_{12} = {} & [\![a^+ \in \mathsf{Chan}^+(m)]\!] \wedge [\![m = (b, \mathbf{ok}\ )]\!] \wedge [\![m \in T_2]\!] \\ & \wedge [\![T = \mathbf{Ch}^+(T_2)]\!] \wedge \psi_{121}\end{aligned}$$

The constraints tell us that the polarized channel a^+ must carry terms of type T and that m, the root name of $(b, \mathbf{ok}\ )$ inhabits this type. Other constraints in ψ_1 are

$$\begin{aligned}\psi_{121} = {} & [\![b \in B]\!] \wedge [\![S \leq E_3^2]\!] \wedge [\![V = \mathsf{Ok}(S)]\!] \wedge [\![E_e^2 = \{\ell(b)\}]\!] \wedge [\![X = \mathbf{Pair}(B, U')]\!] \\ & \wedge [\![U' = V^b(x)]\!] \wedge [\![(b, \mathbf{ok}\ ) \in \mathsf{Pair}(b, ok)]\!]\end{aligned}$$

that describe that $(b, \mathbf{ok}\ )$ must be well-typed. Here, $E^2 = b : B, a : T, \mathbf{begin}\ \ell(b)$.

Horn Clause Constraints from Messages: $E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}}$

(Msg Name)
$$\frac{E \vdash \diamond \quad E = E', x : T, E''}{E \vdash x \rightsquigarrow T; x \in T; \varnothing}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad n \text{ fresh} \quad E_e \text{ fresh}}{E \vdash \mathbf{ok} \rightsquigarrow V; n \in V \wedge [\![S \leq E_e]\!] \wedge [\![V = \mathsf{Ok}(S)]\!] \wedge [\![E_e = \mathit{effects}(E)]\!] \wedge [\![\mathrm{subs}(E)]\!]; S, E_e}$$

(Msg Pair)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{V}}, E \vdash N \rightsquigarrow U; \psi_2; \tilde{\mathbf{W}}}{E \vdash \mathbf{pair}(M,N) \rightsquigarrow X; \begin{array}{l}[\![X = \mathbf{Pair}(x : T, U')]\!] \wedge \psi_1 \wedge \psi_2 \wedge [\![U' = U^M(x)]\!] \wedge \\ ([\![N \in U']\!] \wedge [\![(M,N) \in X]\!] \wedge [\![(M,N) \in \mathsf{Pair}(M,N)]\!])\end{array}; \tilde{\mathbf{V}}, \tilde{\mathbf{W}}}$$

(Msg Fst)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}}}{E \vdash \mathbf{fst}(M) \rightsquigarrow V; \psi_1 \wedge [\![T = \mathbf{Pair}(V, W_2)]\!]; \tilde{\mathbf{U}}, W_1}$$

(Msg Capa)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}}}{E \vdash M^\kappa \rightsquigarrow V; \psi_1 \wedge [\![T = \mathbf{Ch}(U)]\!] \wedge [\![V = \mathbf{Ch}^\kappa(U)]\!]; \tilde{\mathbf{U}}, U}$$

(Msg Snd)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}}}{E \vdash \mathbf{snd}(M) \rightsquigarrow V; \psi_1 \wedge [\![T = \mathbf{Pair}(W_1, W_2)]\!] \wedge \forall y.([\![y = \mathbf{snd}(M)]\!] \Rightarrow [\![V = W_2(y)]\!]); \tilde{\mathbf{U}}, W_1, W_2}$$

Horn Clause Constraints from Processes: $E \vdash P \rightsquigarrow \psi_1; \tilde{\mathbf{U}}$

(Proc In) (where μ either ! or nothing)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}} \quad E, x : V \vdash P \rightsquigarrow \psi_2; \tilde{\mathbf{U}} \quad V \text{ fresh}}{E \vdash \mu\mathbf{in}(M,x); P \rightsquigarrow \psi_1 \wedge \forall x(\psi_2 \wedge ([\![M \in \mathsf{Chan}^-(x)]\!] \iff x \in V)) \wedge [\![T = \mathbf{Ch}^-(V)]\!]; V, \tilde{\mathbf{U}}}$$

(Proc Out)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{V}} \quad E \vdash N \rightsquigarrow U; \psi_2; \tilde{\mathbf{W}} \quad V \text{ fresh}}{E \vdash \mathbf{out}(M,N) \rightsquigarrow \psi_1 \wedge \psi_2 \wedge [\![U = V]\!] \wedge [\![T = \mathbf{Ch}^+(V)]\!] \wedge ([\![M \in \mathsf{Chan}^+(N)]\!] \wedge [\![N \in U]\!]); \tilde{\mathbf{V}}, \tilde{\mathbf{W}}, V}$$

(Proc Res)
$$\frac{E, x : T \vdash P \rightsquigarrow \psi_1; \tilde{\mathbf{U}} \quad \boldsymbol{x} = \mathit{dom}(E); T, N \text{ fresh}}{E \vdash \mathbf{new}\ x; P \rightsquigarrow [\![N = \mathrm{n}(E)]\!] \wedge \forall x(\mathtt{Fresh}(x,N) \Rightarrow \psi_1); \tilde{\mathbf{U}}, T}$$

(Proc Par)
$$\frac{E, \mathit{begins}(P_2) \vdash P_1 \rightsquigarrow \psi_2; \tilde{\mathbf{U}} \quad E, \mathit{begins}(P_1) \vdash P_2 \rightsquigarrow \psi_2'; \tilde{\mathbf{V}}}{E \vdash (P_1 \mid P_2) \rightsquigarrow \psi_2 \wedge \psi_2'; \tilde{\mathbf{U}}, \tilde{\mathbf{V}}}$$

(Proc Zero)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{0} \rightsquigarrow \mathbf{t}; \varnothing}$$

(Proc Exercise)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{U}} \quad E, R \vdash P \rightsquigarrow \psi_2; \tilde{\mathbf{V}} \quad R \text{ fresh}}{E \vdash \mathbf{exercise}(M); P \rightsquigarrow \psi_1 \wedge \psi_2 \wedge [\![T = \mathbf{Ok}(R)]\!]; \tilde{\mathbf{U}}, \tilde{\mathbf{V}}}$$

(Proc Begin)
$$\frac{E \vdash \diamond \quad \mathit{fv}(M) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{begin}\ \ell(M) \rightsquigarrow \mathbf{t}; \varnothing}$$

(Proc End)
$$\frac{E \vdash \diamond \quad \mathit{fv}(M) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{end}\ \ell(M) \rightsquigarrow ([\![m = M]\!] \Rightarrow (\ell, m) \in E_e); \varnothing}$$

(Proc Let)
$$\frac{E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{V}}' \quad E, x : T, x = M \vdash P \rightsquigarrow \psi_2, \tilde{\mathbf{W}}'}{E \vdash \mathbf{let}\ x = M\ \mathbf{in}\ P \rightsquigarrow \psi_1 \wedge [\![x = M]\!] \wedge \psi_2; \tilde{\mathbf{V}}', \tilde{\mathbf{W}}'}$$

(Proc If)
$$\frac{\begin{array}{l}E \vdash M \rightsquigarrow T; \psi_1; \tilde{\mathbf{V}}' \quad E \vdash N \rightsquigarrow T'; \psi_1'; \tilde{\mathbf{V}}'' \\ \mathit{mgu}(M,N) \text{ exists} \Rightarrow E \vdash P(\mathit{mgu}(M,N)) \rightsquigarrow \psi_2, \tilde{\mathbf{W}}' \quad E \vdash Q \rightsquigarrow \psi_2', \tilde{\mathbf{W}}''\end{array}}{E \vdash \mathbf{if}\ M = N\ \mathbf{then}\ P\ \mathbf{else}\ Q \rightsquigarrow \psi_1 \wedge \psi_1' \wedge \psi_2 \wedge \psi_2'; \tilde{\mathbf{V}}', \tilde{\mathbf{V}}'', \tilde{\mathbf{W}}', \tilde{\mathbf{W}}''}$$

Table 3
Rules for generating constraints

5 Soundness, completeness and finiteness

The constraints generated are sufficient to infer the existence of a type and effect assignment, if one exists.

Definition 5.1 Let E be a type environment where $E \vdash \diamond$ and let $\mathcal{M} = (\sigma, \rho)$ be a first-order structure. We say that $\mathcal{M} \prec E$ if $dom(\sigma) = dom(E)$, and

- $\mathcal{M} \models x \in T$ iff $E(x) = T$, and
- $\rho(E_e) = \{(\ell, M) \mid \ell(M) \in effects(E)\}$
- $\mathcal{M} \models [\![x = M]\!]$ if $E(x) = M$

Theorem 5.2 (Soundness of type inference for processes) *Suppose* $E \vdash P \rightsquigarrow \psi_P; \mathbf{U}$ *and that* $E \vdash P$. *Then there exists a first-order structure* $\mathcal{M}$ *such that* $\mathcal{M} \prec E$ *and such that* $\mathcal{M}$ *is a failsafe model of* ψ.

Proof. (Sketch) Construct a model where m inhabits the relation T if m is the root name of M and $E' \vdash M : T$, where $E' \vdash M : T$ is a type judgment in the derivation tree for $E \vdash P$. □

Definition 5.3 Let E be a type environment, and $\mathcal{M} = (\sigma, \rho)$ be a first-order structure with universe $\mathcal{U}$ such that $\mathcal{M} \prec E$. If $\mathcal{M}' = (\sigma, \rho[E_e \mapsto \rho(E_e) \cup Z])$ we say that $\mathcal{M}'$ is an effect extension of $\mathcal{M}$ and write $\mathcal{M} \leq \mathcal{M}'$.

Let ψ be a constraint. We say that $\mathcal{M} = (\sigma, \rho)$ is *failsafe for* ψ if $\rho(\mathsf{Fail}) = \varnothing$.

Theorem 5.4 (Completeness of type inference for processes) *Suppose* $E \vdash P \rightsquigarrow \psi; \mathbf{U}$. *Let* $\psi = \psi_P \wedge \mathcal{S}(\psi_P)$. *Whenever a first-order structure* $\mathcal{M}$ *satisfies that* $\mathcal{M} \prec E$ *and is a failsafe model of* ψ, *there exists a type and effect assignment* Δ *such that* $E \vdash P$ *and such that* $\mathcal{M}' \models \Delta$ *for some* $\mathcal{M}'$ *where* $\mathcal{M} \leq \mathcal{M}'$.

Proof. (Sketch) Induction in the derivation tree of $E \vdash P \rightsquigarrow \psi; \mathbf{U}$. As a lemma, one needs to show the same property for message terms: that if $E \vdash M \rightsquigarrow T, \psi; \mathbf{V}$ and $\mathcal{M} \models \psi$ for some failsafe $\mathcal{M}$, then $E \vdash M : T$. □

Theorem 5.5 (Finite model property) *Suppose* $E \vdash P \rightsquigarrow \psi_P; \tilde{\mathbf{U}}$. *Let* $\psi = \psi_P \wedge \mathcal{S}(\psi_P)$. ψ *is satisfiable if and only if* ψ *has a finite model* $\mathcal{M}$ *such that* $\mathcal{M} \prec E$.

Proof. (Sketch) Notice that the model constructed in the proof of Theorem 5.2 is finite. □

6 Conclusions

We describe a method for establishing authenticity, namely that of automated type inference for correspondence types in an effect type system for a polarized π-calculus with pairs. Given process P and initial type environment E we show how to generate an ALFP formula ψ such that ψ has a failsafe model if and only if $E \vdash P$. The proof that a model of ψ implies that $E \vdash P$ is constructive, yielding an algorithm for finding a type and effect assignment Δ if one exists. This should be contrasted with the approach of [3] which does not guarantee termination.

It should also be contrasted with the approach of [13], where constraints are inequalities over the rational numbers. Our type system is stronger; usage of a channel a with type $\mathbf{Ch}(T)[S]$, where S is a latent effect can be encoded as

$$[\![\mathbf{out}(M,N)]\!] = \mathbf{out}(M,(N,\mathbf{ok}\,))$$
$$[\![\mu\mathbf{in}(M,x);P]\!] = \mu\mathbf{in}(M,x);\mathbf{exercise}(\mathbf{snd}(x));[\![P\{^{\mathbf{fst}(x)}/_{x}\}]\!]$$

Most importantly, our approach is general in nature; the encoding of types and terms does not depend on the rules of the type system. For this reason, our approach appears a natural candidate for obtaining similar type inference results for type systems such as [9], where correspondences concern formulas in an arbitrary authorization logic and the underlying process calculus includes cryptographic operations, and type systems for secrecy properties such as [12]. The possibility of such ramifications is currently under investigation.

Finally, and importantly, our approach allows for an implementation of type inference using the Succinct Solver suite.

An important next step is to consider *robust safety*. A process is robustly safe P if for every opponent process O we have that $P \mid O$ is safe. We can capture robust safety by introducing the opponent type **Un** and a set of extra type rules for typing terms containing subterms of type **Un**. The type system should then guarantee that if P is typable under the assumption that all its free names have type **Un**, then P will be robustly safe.

Type inference becomes more complicated in the presence of opponent types; the reconstruction of a type is no longer purely syntax-directed and the constraints consequently need to express that a term can have either a proper type or an opponent type. This kind of disjunctive property is not directly expressible in the ALFP logic. The solution to this problem is the topic of a forthcoming paper.

References

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

[2] M. Abadi and A. Gordon. A calculus of cryptographic protocols: The spi calculus. *Inf. and Comp.*, 148:1–70, 1999.

[3] Bruno Blanchet. Automatic verification of correspondences for security protocols. Unpublished manuscript, updating papers at CSFW'01 and SAS'02, October 2007.

[4] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[5] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Security analysis using flow logics. In *Current Trends in Theoretical Computer Science*, pages 525–542. World Scientific, 2001.

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

[7] The Cryptyc type checking tool. http://www.cryptyc.org.

[8] Riccardo Focardi, Matteo Maffei, and Francesco Placella. Inferring authentication tags. In *Proceedings of the 2005 workshop on Issues in the Theory of Security (WITS'05)*, pages 41–49. ACM, 2005.

[9] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2005.

[10] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[11] A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2003.

[12] A. D. Gordon and A. S. A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. Concur 2005, LNCS 3653*, volume 3653 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag, 2005.

[13] Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2007.

[14] R. Milner. *The π-calculus*. Cambridge University Press, 2000.

[15] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Cryptographic analysis in cubic time. *ENTCS*, 62, 2001.

[16] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The succinct solver suite. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2004.

[17] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A Succinct Solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.

[18] Martin Odersky. Polarized name passing. In P. S. Thiagarajan, editor, *FSTTCS*, volume 1026 of *Lecture Notes in Computer Science*, pages 324–337. Springer, 1995.

[19] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.