# Computing effects for correspondence types

Hüttel, Hans

*Published in:*
Proceedings of FCS'09

*Publication date:*
2010

*Document Version*
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

*Citation for published version (APA):*
Hüttel, H. (2010). Computing effects for correspondence types. In V. Cortier (Ed.), *Proceedings of FCS'09* (pp. 75-89) http://www.loria.fr/~cortier/FCS09/papersFCS09.pdf

# Computing effects for correspondence types

Hans Hüttel[⋆]

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

**Abstract.** We show that type and effect inference is possible for a type and effect system for authenticity using non-injective correspondences, opponent types and a spi-calculus with symmetric encryption. We do this by a general account of how effects can be computed given knowledge of how and where they appear in type judgments.

## 1 Introduction

Authenticity properties of cryptographic protocols can be described using labelled correspondence assertions due to Woo and Lam [13]. A protocol $P$ is safe if in every run of $P$, whenever an end-assertion is encountered, a corresponding begin-event must have occurred earlier.

A series of papers introduce type/effect systems [5, 6, 8] that are sound approximations of this notion of safety. In all of them, a protocol is described as a process calculus term. If the entities in a protocol $P$ can be assigned types that make $P$ well-typed, such that all free names have an opponent type, then $P$ is safe in the presence of any Dolev-Yao attacker. The methodology is based on *type checking*; a central part of the approach is to provide the type information that makes $P$ safe. Several protocols have since been analyzed with the Cryptyc type checker [2].

Recent work has considered *type reconstruction* in this setting. Gordon et al. consider type and effect inference in a polarized $\pi$-calculus [7] but without cryptography and, notably, without opponent types. Kikuchi and Kobayashi show [9, 10] that type inference is possible for injective correspondences, if effects are modelled as rational numbers and the constraints to be solved are inequalities over the rationals. In [9] the authors describe a variation of a type system of Gordon and Jeffrey that allows type inference in the presence of opponent types. However, the type system of [9] is not equivalent to the original. For instance, the subject reduction property does not hold.

In this paper we show that type inference is possible for non-injective correspondences in the presence of opponent types and a spi-calculus with symmetric encryption; no changes to the type system, which is a subsystem of that of [4], are needed. Moreover, we fix a source of incompleteness found in [7]. We do this by a general account of how effects can be computed given knowledge of how they appear in type judgments. Our method populates all effects in such a way that every end can be traced back to the inhabitants of some effect variable.

We then reduce constraint solving to the problem of finding a *failure-free minimal model* for a formula in the ALFP fragment of first-order logic [11] introduced by Nielson et al. in the setting of *control flow analysis* [11] . Constraints in ALFP can then be checked using the Succinct Solver [12].

[⋆] e-mail: hans@cs.aau.dk

The rest of our paper is organized as follows. We introduce the spi-calculus with symmetric encryption (Section 2) and a typesystem which is a subset of the one introduced by Fournet et al. [4] (Section 3). We then show how to generate a constraint $\psi$ (Section 4), given a process $P$ and a type environment $E$. In order to be able to compute a type/effect assignment, we need to establish a series of results on the origin of effects in type judgments (Section 5). Finally we introduce ALFP (Section 7) and outline how to translate the constraint $\psi$ into ALFP. The translation into ALFP gives rise to a so-called failure-free model if and only if $P$ is well-typed under $E$ (Section 8).

## 2 A spi-calculus with correspondence assertions

We consider a subset of the spi-calculus introduced in [4]with only simple begin/end-correspondences.

The syntax of message terms $M$ is given by the syntax

$$M ::= a, b, m, n, \ldots \mid x, y, z... \mid \mathbf{ok} \mid \mathbf{pair}(M_1, M_2) \mid \{M_1\}_{M_2}$$

Here $a, b, m, n, \ldots$ range over a countable set of names and $x, y, z \ldots$ range over a countable set of variables. The term **ok** is a designated effect term; its only purpose is to populate ok-types (see e.g. [6, 4]), introduced in section 3. The set of process terms is defined by

| $P, Q, R ::=$ | process |
|---|---|
| $\mathbf{in}(M, x); P$ | input |
| $!\mathbf{in}(M, x); P$ | replicated input |
| $\mathbf{out}(M, N)$ | output |
| $\mathbf{new}\ a : T; P$ | restriction of name $a$ within $P$ |
| $(P \mid Q)$ | parallel composition |
| $\mathbf{0}$ | empty process |
| $\mathbf{decrypt}\ M\ \mathbf{as}\ \{y : T\}_N; P$ | decryption of $M$ with key $N$ |
| $\mathbf{match}\ M\ \mathbf{as}\ (N, y : T); P$ | matching first pair component |
| $\mathbf{split}\ M\ \mathbf{as}\ (x : T_1, y : T_2); P$ | pair splitting |
| $\mathbf{begin}\ \ell(M)$ | begin event |
| $\mathbf{end}\ \ell(M)$ | end event |

Here $T$ ranges over the set of types, defined in section 3, and $\ell$ ranges over a countable set of labels. We let $\mathrm{fn}(M)$ and $\mathrm{fv}(M)$ denote the sets of free names and variables for messages – and similarly for $\mathrm{fn}(P)$ and $\mathrm{fv}(P)$ for processes.

In the semantics, the definitions of structural congruence $\equiv$ and the reduction relation $\rightarrow$ follow those of [4]. We write $P \rightarrow^*_{\equiv} Q$ if $P \rightarrow^* Q$ or $P \equiv Q$.

A process is *safe* wrt. its correspondence annotations if in every run, every end-assertion encountered is preceded by a begin-assertion with the same label.

**Definition 1 (Safety).** *A process $P$ is* safe *if whenever $P \rightarrow^*_{\equiv}$ $\mathbf{new}\ \boldsymbol{a}; (\mathbf{end}\ \ell(M) \mid P')$ we have that $P' \equiv \mathbf{begin}\ \ell(M) \mid P''$ for some $P''$.*

A process is *robustly safe* if it is safe for every *opponent*, i.e. any spi calculus process without correspondence assertions.

**Definition 2 (Robust safety).** *A process $P$ is* safe *if for every opponent $O$ we have that $P \mid O$ is safe.*

## 3 A type/effect system

Our type/effect system, a subset of that of [4], soundly approximates robust safety.

### 3.1 Effects and types

An *effect* $S$ represents knowledge of a collection of begin-events and is therefore a set of labelled messages. In general, effects may contain effect variables $R$; the syntax of the set of effects $\mathcal{E}$ is thus

$$S ::= \ell(M) \mid \varnothing \mid S_1, S_2 \mid R$$

Here $\ell$ belongs to a countable set of *labels*. We denote effect inclusion by $S_1 \leq S_2$.

The set of types $\mathcal{T}$ is given by the syntax

**Types and Type Variables**

| $T ::=$ | Type |
|---|---|
| $U$ | type variable |
| $\mathbf{Ch}(T)$ | channel type |
| $\mathbf{Ok}(S)$ | ok-type |
| $\mathbf{Pair}(x : T_1, T_2)$ | dependent pair type |
| $\mathbf{Key}(T)$ | key type |
| $\mathbf{Un}$ | opponent type |

The types **Un**, $\mathbf{Key}(T)$ and $\mathbf{Ch}(T)$ are called *generative*, since only these can appear in restrictions.

Ok-types were introduced in [8] to provide a clear separation between types and effects; it is straightforward to encode effect type systems with latent effects such as that of [10] using ok-types.

Dependent pair types are necessary in order to describe the type of pairs whose second coordinate contains an effect that refers to the first coordinate. For instance, the term $\mathbf{pair}(a, \mathbf{ok})$ can be assigned type $\mathbf{Pair}(x : T, \mathbf{Ok}(\ell(x)))$.

### 3.2 Type/effect assignments

We assume that types are defined by a *type/effect assignment*, where $TVar$ and $EVar$ denote the sets of type/effect variables, respectively.

**Definition 3.** *A* type/effect assignment $\Delta$ *is a finite function $\Delta : TVar \cup EVar \to \mathcal{T} \cup \mathcal{E}$ such that $\Delta(U) \in \mathcal{T}$ for $U \in TVar$ and $\Delta(R) \in \mathcal{E}$ for $R \in EVar$.*

We represent a type/effect assignment as a set of equations $\Delta = \{ U_i = T_i \mid 1 \leq i \leq m\} \cup \{ R_j = S_j \mid 1 \leq j \leq n\}$; we assume that the equations are non-recursive and that for any equation $U = T$, $T$ has exactly one type constructor occurrence. It is easy to see that any non-recursive type/effect assignment can be rewritten in this way.

### 3.3 Assigning types to terms

Type judgments are relative to a type environment that contains type information about the types of free names and variables on the form $u : T$ and collects effect assumptions $S$. The syntax of environments is

$$E ::= \varnothing \mid E, u : T$$

where $\varnothing$ denotes the empty environment and $u$ ranges over names and variables. An environment is well-formed, written $E \vdash \diamond$, if it is a finite function, i.e. if every name bound in $E$ is assigned a type only once. We then write $\mathrm{dom}(E)$ for the domain of $E$.

We denote the set of effects found in an environment $E$ by $\mathit{effects}(E)$ and define it by the clauses

$$\begin{aligned} \mathit{effects}(\varnothing) &= \varnothing \\ \mathit{effects}(E, u : \mathbf{Ok}(S)) &= \mathit{effects}(E), S \\ \mathit{effects}(E, u : T) &= \mathit{effects}(E) \quad \text{otherwise} \\ \mathit{effects}(E, S) &= \mathit{effects}(E), S \end{aligned}$$

As seen from the above, the presence of terms of type $\mathbf{Ok}(S)$ allows us to add knowledge of available begins to an environment.

The type rules for messages and processes are given in Tables 1 and 2.

The rule (Msg Ok) describes how **Ok**-types arise; an occurrence of the term **ok** can get type $\mathbf{Ok}(S)$ for some set of labelled messages from environment $E$.

The rules (Proc Decrypt), (Proc Split), (Proc Match) and (Proc In) may add type bindings with non-opponent types to the environment and thereby account for the effects of an **Ok**-type. For instance, in the rule (Proc Split) the type $T_2$ may be $\mathbf{Ok}(S)$ for some $S$. When typing $P$ in environment $E, x : T_1, y : T_2$, the available effect will then be that of $E$ along with $S$. The effect of the environment is used in the rule (Proc End), which is central to the type system – here, we check if $\ell(M)$ occurs among the labelled messages that form the effect of $E$.

The Un-rules describe opponent typability and directly capture how a Dolev-Yao style attacker [3] can create new messages from old. For instance, the rule (Msg Pair Un) states that if $M$ and $N$ have both been seen or created by the opponent and thus have type **Un**, then $\mathbf{pair}(M, M')$ can also be seen or created by the opponent.

In our typing rule for parallel composition we collect the effects that can be used in each of the parallel components. Here, we must record the bound names that may occur. So whenever a bound name appears in $E$, we must add it as well. We define $\mathit{env}(P)^{(\boldsymbol{n})}$, where $\boldsymbol{n}$ is a sequence of distinct names, by the clauses

$$\begin{aligned} \mathit{env}(P \mid Q)^{(\boldsymbol{n},\boldsymbol{m})} &= \mathit{env}(P)^{\boldsymbol{n}}, \mathit{env}(Q)^{\boldsymbol{m}} \text{ where } \{\boldsymbol{n}, \boldsymbol{m}\} \cap fn(P \mid Q) = \varnothing \\ \mathit{env}(\mathbf{new}\ x : T; P)^{(x,\boldsymbol{n})} &= x : T, \mathit{env}(P)^{\boldsymbol{n}} \text{ where } \{\boldsymbol{n}\} \cap fn(P) = \varnothing \\ \mathit{env}(\mathbf{begin}\ \ell(M))^{\varnothing} &= \ell(M) \\ \mathit{env}(P)^{\boldsymbol{n}} &= \varnothing \text{ otherwise} \end{aligned}$$

We let $\mathit{env}(P)$ stand for $\mathit{env}(P)^{\boldsymbol{n}}$ for some $\boldsymbol{n}$ where $\mathit{env}(P)^{\boldsymbol{n}}$ is well-defined.

**Theorem 1.** *[4] If $E \vdash_\Delta P$ and for all $x \in \mathrm{dom}(E)$, $E \vdash_\Delta x : \mathbf{Un}$ then $P$ is robustly safe.*

**Typed Message:** $E \vdash_\Delta M : T$

(Msg Def)
$$\frac{E \vdash_\Delta M : T \quad \Delta(X) = T}{E \vdash_\Delta M : X}$$

(Msg Encrypt)
$$\frac{E \vdash_\Delta M : T \quad E \vdash_\Delta N : \mathbf{Key}(T)}{E \vdash_\Delta \{M\}_N : \mathbf{Un}}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad \Delta(R) \leq \mathit{effects}(()E)}{E \vdash_\Delta \mathbf{ok} : \mathbf{Ok}(R)}$$

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

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

(Msg Encrypt Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta N : \mathbf{Un}}{E \vdash_\Delta \{M\}_N : \mathbf{Un}}$$

(Msg Pair Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta M' : \mathbf{Un}}{E \vdash_\Delta \mathbf{pair}(M, M') : \mathbf{Un}}$$

(Msg Ok Un)
$$\frac{E \vdash \diamond}{E \vdash_\Delta \mathbf{ok} : \mathbf{Un}}$$

**Table 1.** Type rules for messages

## 4 Generating constraints

The goal of type inference is given $E$ and $P$ to compute a type/effect declaration $\Delta$ such that $E \vdash_\Delta P$, if one exists. From $P$ and $E$ we extract a constraint $\phi$ and then show how a *failure-free* solution to $\phi$ can be found iff $P$ is well-typed and how to use a failure-free solution to extract a type/effect declaration $\Delta$.

### 4.1 A high-level constraint language

The constraint $\phi$ is built from simpler constraints that correspond to the side conditions of the type rules and constraints that describe other, structural requirements on types and effects. In these constraints we use a different notation for type/effect variables to indicate that these variables are the unknowns of our constraints; type variables are denoted by $U_T$ and effect variables by $R_S$.

There are three kinds of atomic constraints. *Type constraints* $\phi_T$ describe conditions on types, *effect constraints* $\phi_S$ describe constraints on effects and *environment constraints* $\phi_E$ describe constraints on type environments.

Type constraints can be equations that equate a type variable and a composite type. Since there are two possible type rules for most syntactic constructs, the type equations that we can extract can be either *possible* or *necessary*. Possible equations are denoted with $\overset{?}{=}$, necessary equalities by $=$. See Section 4.3 for further explanation.

We also introduce constraints that describe if a type is a type abstracted wrt. a term $M$ (written $T = U^M(x)$) or is a type abstraction that has been applied (written $T = U(M)$). Finally, we introduce constraints describing conditions on names and introduce a constraint $\mathsf{Fail}$ meant to indicate failure (see section 6).

The effect constraints describe that an atomic effect is found in the *required total effect* of an environment (written $\ell(M) \leq \mathit{effects}_{\mathrm{req}}(E)$), that an effect

**Good Process:** $E \vdash_\Delta P$

(Proc In) ($\mu$ either ! or nothing)
$$\frac{E \vdash_\Delta M : \mathbf{Ch}(T) \quad E, x : T \vdash_\Delta P}{E \vdash_\Delta \mu\mathbf{in}(M, x); P}$$

(Proc Out)
$$\frac{E \vdash_\Delta M : \mathbf{Ch}(T) \quad E \vdash_\Delta N : T}{E \vdash_\Delta \mathbf{out}(M, N)}$$

(Proc Par)
$$\frac{E, \mathit{env}(P_2) \vdash_\Delta P_1 \quad E, \mathit{env}(P_1) \vdash_\Delta P_2}{E \vdash_\Delta P_1 \mid P_2}$$

(Proc Zero)
$$\frac{E \vdash \diamond}{E \vdash_\Delta \mathbf{0}}$$

(Proc Res)
$$\frac{E, a : T \vdash_\Delta P \quad T \text{ generative}}{E \vdash_\Delta \mathbf{new}\ a : T; P}$$

(Proc Begin)
$$\frac{E \vdash \diamond \quad \mathrm{fv}(M) \subseteq \mathrm{dom}(E)}{E \vdash_\Delta \mathbf{begin}\ \ell(M)}$$

(Proc End)
$$\frac{E \vdash \diamond \quad \mathrm{fv}(M) \subseteq \mathrm{dom}(E) \quad \ell(M) \leq \mathit{effects}(E)}{E \vdash_\Delta \mathbf{end}\ \ell(M)}$$

(Proc Decrypt)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta N : \mathbf{Key}(T) \quad E, y : T \vdash_\Delta P}{E \vdash_\Delta \mathbf{decrypt}\ M\ \mathbf{as}\ \{y : T\}_N; P}$$

(Proc Split)
$$\frac{E \vdash_\Delta M : \mathbf{Pair}(x : T_1, T_2) \quad E, x : T_1, y : T_2 \vdash_\Delta P}{E \vdash_\Delta \mathbf{split}\ M\ \mathbf{as}\ (x : T_1, y : T_2); P}$$

(Proc Match)
$$\frac{E \vdash_\Delta M : \mathbf{Pair}(x : T_1, T_2) \quad E \vdash_\Delta N : T_1 \quad E, y : T_2(N) \vdash_\Delta P}{E \vdash_\Delta \mathbf{match}\ M\ \mathbf{as}\ (N, y : T_2(N)); P}$$

(Proc In Un) ($\mu$ either ! or nothing)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E, a : \mathbf{Un} \vdash_\Delta P}{E \vdash_\Delta \mu\mathbf{in}(M, a); P}$$

(Proc Decrypt Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta N : \mathbf{Un} \quad E, y : \mathbf{Un} \vdash_\Delta P}{E \vdash_\Delta \mathbf{decrypt}\ M\ \mathbf{as}\ \{y : \mathbf{Un}\}_N; P}$$

(Proc Split Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E, x : \mathbf{Un}, y : \mathbf{Un} \vdash_\Delta P}{E \vdash_\Delta \mathbf{split}\ M\ \mathbf{as}\ (x : \mathbf{Un}, y : \mathbf{Un}); P}$$

(Proc Match Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta N : \mathbf{Un} \quad E, y : \mathbf{Un} \vdash_\Delta P}{E \vdash_\Delta \mathbf{match}\ M\ \mathbf{as}\ (N, y : \mathbf{Un}); P}$$

(Proc Out Un)
$$\frac{E \vdash_\Delta M : \mathbf{Un} \quad E \vdash_\Delta N : \mathbf{Un}}{E \vdash_\Delta \mathbf{out}(M, N)}$$

**Table 2.** Type rules for processes

variable is instantiated by an environment ($E \vdash R$) or that an effect variable $R$ occurs in environment $E$ as $x : \mathbf{Ok}(R)$ for some $x$ ($R \in E$).

The syntax of constraints is given by the formation rules

$$\begin{aligned} \varphi &::= \phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x.\varphi_1 \\ \phi &::= \phi_T \mid \phi_S \mid \phi_E \end{aligned}$$

$$\begin{aligned} \phi_T ::= {} & U_{T_1} \overset{?}{=} \mathbf{Ch}(U_{T_2}) \mid U_{T_1} \overset{?}{=} \mathbf{Pair}(x : U_{T_2}, U_{T_3}) \mid U_T \overset{?}{=} \mathbf{Un} \mid U_T \overset{?}{=} \mathbf{Key}(U_{T_1}) \\ & \mid U_{T_1} \overset{?}{=} {U_{T_2}}^{M}(x) \mid U_{T_1} \overset{?}{=} U_{T_2}(M) \mid\mid U_T = \mathbf{Key}(U_{T_1}) \mid U_T = \mathbf{Ch}(U_{T_1}) \mid U_T = \mathbf{Un} \\ & \mid U_T = \mathbf{Pair}(x : U_{T_1}, U_{T_2}) \mid U_{T_1} = {U_{T_2}}^{M}(x) \mid U_{T_1} = U_{T_2}(M) \\ & \mid T \text{ generative} \mid n \notin \mathrm{n}(P) \mid \mathrm{Fail} \mid M : U_T \end{aligned}$$

$$\phi_S ::= \ell(M) \leq \mathit{effects}_{\mathrm{req}}(E) \mid E \vdash R \mid S \in E$$

$$\phi_E ::= \mathrm{wf}(E) \mid \mathrm{fn}(M) \subseteq \mathrm{dom}(E)$$

### 4.2 Generating constraints

We describe constraint generation as a big-step semantics. For *messages*, transitions are on the form $E \vdash M \rightsquigarrow U_T; \psi_1$, where $U_T$ is a fresh type variable and $\psi_1$ is the constraint associated with $M$. For *processes*, transitions are on the form $E \vdash_\Delta P \rightsquigarrow \psi_1$, where $\psi_1$ is the constraint associated with $P$.

We present a selection of the rules in Tables 3 and 4. We assume that a type environment $E$ is given by the syntax

$$E ::= \varnothing \mid E, u : U_T \mid E, u : \mathbf{Un}$$

Thus, names are either assigned to type variables or to **Un**, while both effect variables and actual effects may appear in $E$.

**Horn Clause Constraints from Messages:** $E \vdash M \rightsquigarrow T; \psi_1$

(Msg Name)

$$\frac{E = E', x : U_T, E''}{E \vdash x \rightsquigarrow U_T; x : U_T \wedge \mathrm{wf}(E)}$$

(Msg Name Un)

$$\frac{E = E', x : \mathbf{Un}, E''}{E \vdash x \rightsquigarrow U_T; x : U_T \wedge U_T = \mathbf{Un} \wedge \mathrm{wf}(E)}$$

(Msg Ok)

$$\frac{}{E \vdash \mathbf{ok} \rightsquigarrow U_T; \begin{pmatrix} \mathbf{ok} : U_T \wedge \\ U_T = \mathbf{Un} \Rightarrow \mathrm{wf}(E) \wedge \\ (U_T \overset{?}{=} \mathbf{Ok}(R) \wedge \\ \ U_T \neq \mathbf{Un} \Rightarrow (E \vdash R) \ \wedge \ \mathrm{wf}(E)) \end{pmatrix}}$$

(Msg Encrypt)

$$\frac{E \vdash M \rightsquigarrow U_{T_1}; \psi_1 \quad E \vdash N \rightsquigarrow U_{T_2}; \psi_2}{E \vdash \{M\}_N \rightsquigarrow U_{T_3}; U_{T_3} = \mathbf{Un} \ \wedge \begin{pmatrix} U_{T_2} \overset{?}{=} \mathbf{Key}(U_{T_1}) \wedge \\ (U_{T_2} = \mathbf{Un} \Rightarrow U_{T_1} = \mathbf{Un}) \wedge \psi_1 \wedge \psi_2 \end{pmatrix}}$$

**Table 3.** Selection of rules for generating constraints from messages

### 4.3 Dealing with opponent types

Many of the syntactic constructs have more than one type rule. This requires us to build constraints that can be used to determine when a type must be an opponent type and when it must be a non-opponent type.

Firstly, a type $T$ can be a necessary opponent type if some name of type $T$ was declared to have type **Un** in the type environment. This is described by the rule (Msg Name Un). If the type of $x$ is not known when types are generated, the rule (Msg Name) must be used. If the type of $x$ is known to be **Un**, we add the constraint that the type of $x$ is **Un**.

Secondly, a type $T$ can be a necessary opponent type if this will resolve conflicting possible type equalities. Suppose for instance that $T \overset{?}{=} \mathbf{Key}(T_1)$ and $T \overset{?}{=} \mathbf{Ch}(T_2)$. Then $T$ has to **Un** for the constraints to be satisfiable.

Moreover, since there are two type rules for most syntactic constructs, the constraints must describe the possible choices. For instance, consider the rules

**Horn Clause Constraints from Processes:** $E \vdash P \leadsto \psi_1$

(Proc In) (where $\mu$ is either ! or nothing)

$$\frac{E \vdash M \leadsto U_{T_1}; \psi_1 \quad E, x : U_{T_2} \vdash P \leadsto \psi_2 \quad U_{T_2} \text{ fresh}}{E \vdash \mu\mathbf{in}(M, x); P \leadsto \psi_1 \wedge \psi_2 \left( \begin{array}{l} U_{T_1} \overset{?}{=} \mathbf{Ch}(U_{T_2}) \wedge \\ (U_{T_1} = \mathbf{Un} \Rightarrow U_{T_2} = \mathbf{Un}) \end{array} \right)}$$

(Proc Res)

$$\frac{E, x : U_T \vdash P \leadsto \psi_1 \quad U_T \text{ fresh}}{E \vdash \mathbf{new}\ x; P \leadsto \forall x (x \notin \mathrm{n}(P) \wedge U_T \text{ generative} \Rightarrow \psi_1)}$$

(Proc Begin)

$$\frac{}{E \vdash \mathbf{begin}\ \ell(M) \leadsto \mathrm{wf}(E) \wedge \mathrm{fn}(M) \subseteq \mathrm{dom}(E)}$$

(Proc End)

$$\frac{}{E \vdash \mathbf{end}\ \ell(M) \leadsto \ell(M) \leq \mathit{effects}_{\mathrm{req}}(E) \wedge \mathrm{wf}(E)}$$

(Proc Decrypt)

$$\frac{E \vdash M \leadsto U_{T_1}; \psi_1 \quad E \vdash N \leadsto U_{T_2}; \psi_2 \quad E, y : U_{T_3} \vdash P \leadsto \psi_3 \quad U_{T_3} \text{ fresh}}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y\}_N; P \leadsto \left( \begin{array}{l} \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge\ U_{T_1} = \mathbf{Un} \wedge \\ U_{T_2} \overset{?}{=} \mathbf{Key}(U_{T_3}) \wedge \\ (U_{T_2} = \mathbf{Un} \Rightarrow U_{T_3} = \mathbf{Un}) \end{array} \right)}$$

**Table 4.** Selected rules for generating constraints from processes

(Msg Encrypt) and (Msg Encrypt Un) for typing an encryption $\{M\}_N$. Here the type of $M$ depends on the type of the key $N$. It is always possible that $M$ has type $\mathbf{Key}(U_T)$ and $N$ has type $T$. If we know that $N$ necessarily has type $\mathbf{Un}$, then $M$ must also have type $\mathbf{Un}$. This is captured in the constraint generation rule (Msg Encrypt).

## 5 Computing effects

We now describe how the inhabitants of effect variables can be found.

### 5.1 Actual vs. required effects

The constraints of Tables 3 and 4 do not suffice. Consider the simple process $\mathbf{end}\ \ell(M)$, where $M$ is arbitrary and a type environment $E = x : \mathbf{Ok}(S)$. Then the constraints generated thus far will allow for a solution where $\ell(M)$ occurs in $\mathit{effects}_{\mathrm{req}}(E)$ but not in $S$, since the solution may set $S$ to $\varnothing$; none of the constraints presented so far relate $S$ to $\mathit{effects}_{\mathrm{req}}(E)$. However, $S$ should contain $\ell(M)$. Therefore the constraint $\ell(M) \leq \mathit{effects}_{\mathrm{req}}(E)$ is not sufficient; we must compare the *required effects* of $E$ to the *actual effects* found in $E$.

### 5.2 How and where do actual effects arise?

Actual inhabitants of effects are determined from the type rules (Msg Ok) and (Proc End). We first consider the demands laid upon us by (Msg Ok), since this rule provides an upper bound on the contents of $S$ where we know that $E \vdash \mathbf{ok} : \mathbf{Ok}(S)$ for some $E$.

For any type environment $E$, the set $\mathit{inits}(E)$ is the set of immediately available effects, i.e. the ones not supplied by some $\mathbf{Ok}$-type.

**Definition 4.**

$$\begin{aligned} \mathit{inits}(\varnothing) &= \varnothing \\ \mathit{inits}(x : T, E) &= \mathit{inits}(E) \\ \mathit{inits}(\ell(M), E) &= \ell(M), \mathit{inits}(E) \end{aligned}$$

The set $\mathit{others}(E)$ contains the effect in $E$ that is only available in effect variables within ok-types.

**Definition 5.**

$$\begin{aligned} \mathit{others}(\varnothing) &= \varnothing \\ \mathit{others}(x : \mathbf{Ok}(S), E) &= S, \mathit{others}(E) \\ \mathit{others}(x : T, E) &= \mathit{others}(E) \\ \mathit{others}(\ell(M), E) &= \mathit{others}(E) \end{aligned}$$

**Definition 6.** *Let $E$ be a type environment.*

- $E$ *is* simple *if* $\mathit{others}(E) = \varnothing$
- $E$ *is* semisimple *if* $\mathit{others}(E) \neq \varnothing$ *and* $\mathit{inits}(E) \neq \varnothing$
- $E$ *is* nonsimple *if* $\mathit{others}(E) \neq \varnothing$ *and* $\mathit{inits}(E) = \varnothing$

A type environment $E$ is thus simple if its effects can be directly determined from its immediate effects – so no additional effects are hidden in ok-types found in $E$. Note that if $E$ only contains generative types, then $E$ is simple.

In what follows we write $E \vdash_\Delta \mathcal{J}$ to denote that we are dealing with an arbitrary type judgement $\mathcal{J}$ which can be either $E \vdash_\Delta M : T$ or $E \vdash_\Delta P$.

**Definition 7.** *Let $E$ be a type environment. and assume that $E \vdash_\Delta \mathcal{J}$. We say that an effect variable $R$ is* initial *in the derivation of $E \vdash_\Delta \mathcal{J}$ if there exists an instance of (Msg Ok) $E' \vdash_\Delta$ **ok** $: \mathbf{Ok}(R)$ where $E'$ is simple. Otherwise we say that $R$ is* noninitial*.*

*We call any $E' \vdash_\Delta$: **ok** $: \mathbf{Ok}(S)$ where $R$ is initial an* initial instance *of $R$.*

## 5.3 Admissible names in effects

We need to ensure that literals in an effect only use names that are mentioned in the associated type environment. For an effect variable $R$ occuring in the derivation of $E \vdash \mathcal{J}$ we say that $R$ originates in $E$, written $E \heartsuit R$, if $E \vdash$ **ok** $: \mathbf{Ok}(R)$. Any $R$ can originate in at most one $E$, since our strategy for generating constraints introduces a fresh effect variable for each occurrence of an **ok**.

We must ensure that equivalent effect variables can only make use of the same names. Let us write $R_j \sim R_i$ if $\Delta(R_j) = \Delta(R_i)$.

The names that can appear in terms found in an effect variable $R$ are then those that are known by every type environment containing a binding $x : \mathbf{Ok}(R')$ prior to this occurrence of $R'$, where $R'$ is an effect variable equivalent to $R$.

**Definition 8.** *Assume that we have a derivation $\mathcal{D}$ for a judgment $E \vdash \mathcal{J}$. The set of admissible names $\heartsuit(R)$ is*

$$\heartsuit(R) = \bigcap_{R' \sim R, E'_R \heartsuit R'} \mathrm{dom}(E'_R) \cap \bigcap_{E'_i, x_i : \mathbf{Ok}(R'), E''_i \vdash \mathcal{J}_1 \in \mathcal{D}, \quad R' \sim R} \mathrm{dom}(E'_i)$$

If $\mathrm{fn}(\ell(M)) \subseteq \heartsuit(R)$, we write $\ell(M) \heartsuit R$. Every $\ell(M)$ that inhabits $\Delta(R)$ must satisfy this condition.

### 5.4 Default effects

Three kinds of effects can safely be added to any effect assignment; these *default effects* are effects inherited via applications of (Msg Ok) and (Proc End).

In what follows we always assume that we have a type derivation for the judgments that we consider; this derivation tree will determine the exact dependencies between effects. For the purposes of type inference we of course do not have access to a type derivation tree yet but due to the structure of the rules, a type derivation tree will be isomorphic to the constraint derivation tree.

*Initial effect variables* Initial instances impose conditions on the possible inhabitants of ok-types via (Msg Ok).

The central observation is that if $E_i \vdash \mathbf{ok} : \mathbf{Ok}(R)$ is a initial instance of $R$ and $E_i$ simple, and there exists an effect $\ell(M)$ such that $\ell(M) \in \mathit{inits}(E_j)$ for every $R_j$ where $R_j \sim R_i$, then we can safely assume that $\ell(M)$ appears in $R$ and all effects that are equivalent to or inherit from $R$, if $M$ only contains admissible names.

Next, we define a relation that describes when an effect can depend on another. In the following definitions we assume that $E \vdash_\Delta \mathcal{J}$ was concluded by some derivation $\mathcal{D}$.

**Definition 9 (Depends upon).** *Let $E' \vdash M : \mathbf{Ok}(R)$ be an application of (Msg Ok) found in $\mathcal{D}$ and let $\mathit{others}(E) = R_1, \ldots, R_k$. We then write $R \sqsubseteq_0 S_i$ for $1 \leq i \leq k$. Now let $\sqsubseteq$ be the transitive closure of $\sqsubseteq_0$. We then define*

$$\mathrm{dep}(R) = \{\ R_i \mid R \sqsubseteq R_i, R_i \textit{ initial}\}$$

**Definition 10.** *Let $\Delta$ be a type/effect assignment and $\psi(R)$ a Boolean-valued condition on elements of* $\mathrm{dom}(\Delta)$*. The updated assignment $\Delta' = \Delta[R_1 \mapsto V \mid \psi]$ is defined by*

$$\Delta'(R_1) = \begin{cases} \Delta(R_1) & \textit{if not } \psi(R_1) \\ V & \textit{if } \psi(R_1) \end{cases}$$

The following important proposition is a simple consequence of the properties of set inclusion; $\sqsubseteq^*$ denotes the closure of $\sqsubseteq$ up to $\sim$.

**Proposition 1.** *Suppose $E \vdash_\Delta \mathcal{J}$ and that $R$ is an initial effect with initial instance $E_i \vdash \mathbf{ok} : \mathbf{Ok}(R)$. Let $\ell(M) \in \mathit{inits}(E_j)$ for every $R_j \sim R_i$ with $E_j \heartsuit R_j$. Then the type/effect assignment*

$$\Delta' = \Delta[R_1 \mapsto \Delta(R_1) \cup \ell(M) \mid R_1 \sqsubseteq^* R, \ell(M) \heartsuit (R_1)$$

*satisfies $E \vdash_{\Delta'} \mathcal{J}$.*

*Noninitial effect variables* Noninitial effect variables impose conditions on the inhabitants of effects through the (Proc End) rule. The following propositions are immediate consequences of the inclusion property for entailment.

**Proposition 2.** *If there exists a noninitial effect* $R$ *where* $\{\ R_1 \mid R_1 \sqsubseteq R\} = \varnothing$*, then* $R$ *can only occur in a type environment of some judgment.*

**Proposition 3.** *If there exists a noninitial effect* $R$ *where* $\{\ R_1 \mid R_1 \sqsubseteq R\} \neq \varnothing$ *and* $\mathrm{dep}(R) = \varnothing$*, then there exists a noninitial effect* $R_2$ *where* $R_2 \sqsubseteq R_2$*.*

First we consider effect variables $R$ where $\{\ R_1 \mid R \sqsubseteq R_1\} = \varnothing$. An $R$ where this is the case can only appear in type environments. If $R$ appears in $E$ where $E = E', x : \mathbf{Ok}(R), E''$ then by well-formedness we must have $\mathrm{fn}(R) \subseteq \mathrm{dom}(E')$.

If $R$ appears in $E$ where $E \vdash_{\Delta} \mathbf{end}\ \ell(M)$ and $\ell(M) \notin \mathit{inits}(E)$, then we can add $\ell(M)$ to $R$ and to any other noninitial effect variable $R_1$ where $R_1 \sqsubseteq R$ if $\ell(M)$ is admissible in these. The resulting new effect assignment will still make the process under consideration well-typed.

Alternatively, an effect variable $R$ can depend on other effect variables, so $\{\ R_1 \mid R \sqsubseteq R_1\} \neq \varnothing$.

There are now two subcases. First, note that if $R$ is noninitial and we have $\mathrm{dep}(R) \neq \varnothing$, then there is some initial $R_1$ such that $R \sqsubseteq R_1$. But then we use Proposition 1 to populate $R_1$ and thereby we also populate $R$. So in the following we assume that $\mathrm{dep}(R) = \varnothing$.

Let $\ell(M)$ be a labelled term and $E$ a type environment where $R \in \mathit{others}(E)$ and an instance of (Proc End) requires that $\ell(M) \in \mathit{effects}(E)$. For any $R'$ where $R' \sqsubseteq R'$ (and these exist by Proposition 3) we can add $\ell(M)$ if this is admissible. We will still have a $\Delta'$ that satisfies our constraints.

**Proposition 4.** *Suppose* $E \vdash_{\Delta} P$ *and that* $R$ *is a noninitial effect where* $\mathrm{dep}(R) = \varnothing$ *or* $\{\ R_1 \mid R \sqsubseteq R_1\} = \varnothing$*. Let* $E' \vdash_{\Delta'} \mathbf{end}\ \ell(M)$ *be an application of (Proc End) in the derivation tree for* $E \vdash_{\Delta} P$ *where* $\ell(M) \notin \mathit{inits}(E')$*. Suppose for all* $R_1$ *where* $R \sqsubseteq^{*} R_1$ *or* $R_1 \sim R$ *we have* $\ell(M) \heartsuit (R_1)$*. Then the type/effect assignment* $\Delta' = \Delta[R_1 \mapsto \Delta(R_1) \cup \ell(M) \mid R \sqsubseteq^{*} R_1, \mathrm{fn}(M) \subseteq \heartsuit(R_1)]$ *also satisfies* $E \vdash_{\Delta'} P$

*Semiinitial effect variables* Some noninitial variables may contain initial effects, and in this case we may be able to add additional effects. Let us say that a noninitial effect variable $R$ is *semiinitial* if for some $E \vdash \mathbf{ok} : \mathbf{Ok}(R)$ we have that $E$ is semisimple. Any $E \vdash \mathbf{ok} : \mathbf{Ok}(R)$ where $E$ is semisimple is then called a semisimple instance of $R$.

**Proposition 5.** *Suppose* $E \vdash_{\Delta} \mathcal{J}$ *and* $R$ *is a semiinitial effect with semiinitial instance* $E_i \vdash \mathbf{ok} : \mathbf{Ok}(R)$*. Let* $\ell(M) \in \mathit{inits}(E_j)$ *for every* $R_j \sim R_i$ *with* $E_j \heartsuit R_j$*. Then the type/effect assignment* $\Delta' = \Delta[R_1 \mapsto \Delta(R_1) \cup \ell(M) \mid R_1 \sqsubseteq^{*} R, \mathrm{fn}(M) \heartsuit (R_1)]$ *satisfies* $E \vdash_{\Delta'} \mathcal{J}$*.*

### 5.5 Saturated effect assignments

In the above propositions, if $\Delta' \neq \Delta$ we write $\Delta \rightarrow \Delta'$ and say that $\Delta'$ *extends* $\Delta$. An effect $\ell(M)$ that can be added by an application of the propositions of the previous sections is called a *default effect*.

The significance of the propositions is that we can add all default effects to the type/effect assignment and obtain a new satisfying assignment.

**Definition 11.** *A $\Delta$ such that $\Delta \not\rightarrow$ is* saturated*.*

The following theorem follows from the propositions that we now have:

**Theorem 2.** *$E \vdash_\Delta \mathcal{J}$ iff there exists a saturated $\Delta_1$ such that $E \vdash_{\Delta_1} \mathcal{J}$.*

### 5.6 Minimal solutions

It will be particularly convenient if we can show that it is enough to find a solution that only contains default effects.

**Definition 12.** *Let $\Delta_1$ and $\Delta_2$ be type/effect assignments. We say that $\Delta_1 \leq \Delta_2$ if $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$ and for every type variable $T$ we have $\Delta_1(T) = \Delta_2(T)$, and for every effect variable $S$ we have $\Delta_1(S) \subseteq \Delta_2(S)$.*

From now on we only consider type/effect assignments with the same domain.

**Definition 13.** *$\Delta$ is* minimal *wrt. $E \vdash \mathcal{J}$ if $E \vdash_\Delta \mathcal{J}$ but there does not exist a $\Delta'$ such that $\Delta' < \Delta$ and $E \vdash_\Delta \mathcal{J}$.*

Any minimal type/effect assignment will only contain default effects.

**Theorem 3.** *Let $E \vdash_\Delta \mathcal{J}$ . If $\Delta$ is minimal wrt. $E \vdash \mathcal{J}$, then whenever $\ell(M) \in \Delta(R)$, $\ell(M)$ is a default effect.*

## 6 General constraints

Some constraints do not arise directly from the description of a process but capture general properties of the type system. These general constraints include a description of the default effects (cf. the previous section) and unification conditions. An example of a unification constraint is

$$\forall T_1.\forall T_2.\mathbf{Ch}(T_1) = \mathbf{Ch}(T_2) \Rightarrow T_1 = T_2$$

Another special class of general constraints is comprised of the failure constraints that apply. For effects, the constraints will cause failure if an effect term $\ell(M)$ is required to be in the required effect $\mathit{effects}_{\mathrm{req}}(E)$ for some $E$ but is not found among the actual effects in $E$, $\mathit{effects}(E)$:

$$\exists \ell(M).\exists E.(\ell(M) \in \mathit{effects}_{\mathrm{req}}(E) \wedge \ell(M) \notin \mathit{effects}(E)) \Rightarrow \mathsf{Fail}$$

We denote the conjunction of all general constraints by $\psi_{\mathbf{Ax}}$.

## 7 Encoding constraints in ALFP

We encode our high-level constraints of the previous section in the ALFP fragment of first-order logic, introduced by Nielson et al. in [12].

### 7.1 The ALFP logic

In the syntax of ALFP, we let $R$ range over $\mathcal{R}$, the set of relation symbols, let $x$ range over a countable set of variables $\mathcal{X}$ and let $c$ range over a finite set of constants. Formulae are either preconditions ($\phi$) or clauses ($\psi$).

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

Note that existential quantification and disjunction are only allowed in preconditions.

A relation symbol which occurs in a precondition is called a *query*. A negated relations symbol is called a *negated query*. Finally, any other occurrence of a relation symbol is called an *assertion*.

In the interpretation of ALFP we assume given a first-order structure.

**Definition 14.** *A* first-order structure *is a triple $\mathcal{M} = (\mathcal{U}, \rho, \sigma)$ where $\mathcal{U}$ is a finite universe of values and $\rho$ and $\sigma$ are interpretations of variables: $\sigma : \mathcal{X} \to \mathcal{U}$ and relation symboles $\rho : \mathcal{R} \to \bigcup_{k\geq 0} \mathcal{P}(\mathcal{U}^k)$. We often omit mention of the universe and write a first-order structure as a pair $(\rho, \sigma)$.*

Let $\Delta_\sigma = \{\ \rho \mid (\rho, \sigma) \models \psi\}$. Then we can find the least solution of a conjunctive Horn clause in time exponential in the maximal quantifier depth, as told by the following theorem from [11]:

**Theorem 4.** *Given a $\sigma$, the least solution of $\psi = \psi_1 \wedge \ldots \psi_m$ can be found in time $O(\sum_{i=1}^{m} n_i N^{r_i})$ where $n_i$ is the size of $\psi_i$ and $r_i$ is the maximal depth of quantifiers in $\psi_i$.*

### 7.2 Encoding constraints

We can define an encoding of atomic constraints in our high-level constraint language into ALFP. To define this encoding we also need to

- encode message terms and substitution on terms
- encode types and effects and operations on types and effects. This includes an ALFP representation of dependent types
- represent inhabitants of types and effects as members of relations $\mathsf{Type}$ and $\mathsf{Effect}$
- represent failure by a relation $\mathsf{Fail}$

In our encoding of the constraints in ALFP we often need to quantify over and compare such entities. A convenient approach is to introduce *proper names*, so quantification over e.g. types translates to first-order quantification over these.

To illustrate our approach, here is a part of the encoding that encodes message terms. Every term $M$ is enoded as $(n, \psi)$, where $m$ is a corresponding proper name and $\psi$ is a consistency condition; the term constructors are represented by relations $\mathsf{Name}, \mathsf{TermPair}, \mathsf{OkTerm}$, and $\mathsf{Enc}$.

$$\begin{aligned}
[\![(M_1, M_2)]\!] &= (n, n \in \mathsf{TermPair}(m_1, m_2) \wedge \psi_1 \wedge \psi_2) \\
&\quad \text{where } [\![M_1]\!] = (m_1, \psi_1), [\![M_2]\!] = (m_2, \psi_2), \\
[\![\{M_1\}_{M_2}]\!] &= (n, n \in \mathsf{Enc}(m_1, m_2) \wedge \psi_1 \wedge \psi_2) \\
&\quad \text{where } [\![M_1]\!] = (m_1, \psi_1), [\![M_2]\!] = (m_2, \psi_2), \\
[\![n]\!] &= (n, n \in \mathsf{Name}) \\
[\![\mathbf{ok}]\!] &= (n, n \in \mathsf{OkTerm})
\end{aligned}$$

## 8 Correctness properties

The constraints generated are the correct ones. A process $P$ is well-typed under $E$ and $\Delta$ iff we there is a *failure-free* model of the generated constraints. A model is failure-free if the $\mathsf{Fail}$ relation is empty.

**Definition 15 (Agrees with).** *Assume $E \vdash \diamond$ and let $\mathcal{M} = (\sigma, \rho)$ be a first-order structure. We say that $\mathcal{M} \prec E$ if $\mathrm{dom}(\sigma) = \mathrm{dom}(E)$ and we have $\mathcal{M} \models (x, t) \in \mathsf{Type}$ iff $E(x) = T$. If also for a given $\Delta$ we have $\mathcal{M} \models (\ell, m, s) \in \mathsf{Effect}$ iff $E(x) : \mathbf{Ok}(S)$ and $\ell(M) \in \Delta(S)$, we write $\mathcal{M} \prec (E, \Delta)$.*

**Theorem 5 (Soundness).** *If $E \vdash_\Delta P \rightsquigarrow \psi$ and $E \vdash_\Delta P$ where $\mathrm{dom}(\Delta) = Var(\psi)$, then there exists a failure-free model $\mathcal{M}$ where $\mathcal{M} \prec (E, \Delta)$ and $\mathcal{M} \models [\![\psi \wedge \psi_{\mathbf{Ax}}]\!]$.*

**Definition 16.** *Let $\Delta$ be a type/effect assignment and $\mathcal{M}$ be a first-order structure. We write $\mathcal{M} \models \Delta$ if for every $X = T \in \Delta$ we have that $\mathcal{M} \models [\![X = T]\!]$.*

**Theorem 6 (Completeness).** *Suppose $E \vdash_\Delta P \rightsquigarrow \psi$. Whenever a first-order structure $\mathcal{M}$ satisfies that $\mathcal{M} \prec E$ and is a failure-free model of $[\![\psi \wedge \psi_{\mathrm{Ax}}]\!]$, there exists a type/effect-assignment $\Delta$ such that $E \vdash_\Delta P$, $\mathcal{M} \prec (E, \Delta)$ and $\mathcal{M} \models \Delta$.*

As an immediate consequence of our results, we can implement a type inference algorithm that proceeds as follows, given process $P$ and environment $E$:

1. Use the constraint generation rules to find $\psi_P$, where $E \vdash_\Delta P \rightsquigarrow \psi_P$.
2. Construct the general constraints $\psi_{\mathrm{Ax}}$
3. Give $[\![\psi_P \wedge \psi_{\mathrm{Ax}}]\!]$ as input to the Succinct Solver.
4. If the Succinct Solver produces a model that is not failure-free, then $P$ cannot be typed under $E$.
5. Otherwise, if a failure-free model $\mathcal{M}$ is returned, then use the construction of the proof of Theorem 6 to build a type/effect assignment $\Delta$.

## 9 Conclusion

We have described a strategy for type reconstruction in a type and effect system for robust safety in a spi calculus with opponent types and non-injective correspondences. The strategy consists in generating a constraint $\phi$ from process $P$ and type environment $E$ such that it translates to formula in the ALFP logic that has a failsafe model if and only if $E \vdash_{\Delta} P$. A central part of the construction of $\psi$ is a characterization of how effects arise in a well-typed process.

A decision procedure that constructs a typing if and only if one exists can be built around a solver for ALFP such as the Succinct Solver [12]. From the constraint rule (Proc Res) we see that every level of restriction introduces a level of quantification, and by Theorem 4 we see that our type inference algorithm is exponential in the maximal restriction depth of $P$.

The example of Section 5.1 shows that effect constraints need not have a unique solution. Consider $P = \mathbf{new}\ a : T_1; \mathbf{new}\ b : T_2; \mathbf{in}(a, x); \mathbf{in}(b, y); \mathbf{end}\ \ell(M)$, where $M$ is arbitrary; $P$ can be typed with any of the following type/effect assignments, where $\Delta_3$ is saturated and $\Delta_1$ and $\Delta_2$ are incomparable:

$$\begin{aligned}
\Delta_1 &= \{T_1 = \mathbf{Ch}(S_1), T_2 = \mathbf{Ch}(S_2), S_1 = \varnothing, S_2 = \ell(M)\}\\
\Delta_2 &= \{T_1 = \mathbf{Ch}(S_1), T_2 = \mathbf{Ch}(S_2), S_1 = \ell(M), S_2 = \varnothing\}\\
\Delta_3 &= \{T_1 = \mathbf{Ch}(S_1), T_2 = \mathbf{Ch}(S_2), S_1 = \ell(M), S_2 = \ell(M)\}
\end{aligned}$$

A topic of future work is to apply the inference strategy to the full type system of [4]. Here, effects are sets of Datalog facts, so populating an effect $S$ amounts to adding facts to Datalog clauses to deduce all in any expectation **expect** $C$. This is the problem of *abduction*, studied by Becker and Nanz in [1]. Since abduction may produce an infinite set of valid abduction candidates, it cannot be internalized in ALFP and a strategy for solving constraints must therefore step outside the Succinct Solver.

## References

1. Moritz Y. Becker and Sebastian Nanz. The role of abduction in declarative authorization policies. In *Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2008.
2. The Cryptyc type checking tool. http://www.cryptyc.org.
3. D. Dolev and A.C. Yao. On the security of public key protocols. *Transactions on Information Theory*, IT-29(2):198–208, 1983.
4. Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
5. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.
6. A. D. Gordon and A. S. A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.
7. Andrew D. Gordon, Hans Hüttel, and René Rydhof Hansen. Type inference for correspondence types. In S. Kremer and P. Panangaden, editors, *Proceedings of SecCo'08*, 2008. To appear.
8. Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003.

9. D. Kikuchi and N. Kobayashi. Type-based automated verification of authenticity in cryptographic protocols. In *Proceedings of ESOP'09*, Lecture Notes in Computer Science. Springer Verlag, 2009.
10. 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.
11. Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Cryptographic analysis in cubic time. *ENTCS*, 62, 2001.
12. Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A Succinct Solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.
13. 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.