Type Checking and Type Inference in the Presence of an Adversary

Palle Raabjerg
Aalborg University

Title:
Type Checking and Type Inference in the Presence of an Adversary

Author:
Palle Raabjerg

Supervisor:
Hans Hüttel

Date:
June 10, 2009

Pages in Report:
59

Pages in Appendix:
64

Pages in total:
123

Abstract:

Correspondence assertions are used as a means to express authentication properties in communicating protocols expressed in the π and Spi-calculi. A protocol is considered *safe* if assertions in the protocol always hold. It has been shown by Gordon and Jeffrey that a type system can provide a sound approximation to the *safety* property. A paper by Hüttel shows that sound and complete type inference for such a system can be achieved by encoding constraints to a subset of propositional logic, ALFP, which can be satisfied by the Succinct Solver, a tool developed by Nielson, Nielson and Seidl. In 2007, Fournet, Gordon and Maffeis generalised the notion of correspondence assertions by using Datalog, a logic programming language utilising a subset of ALFP, and developed a type system for verifying these assertions. An existing implementation of the type system with Datalog correspondences was mentioned, but apparantly not published. Likewise, the type inference method of Hüttel has not seen an actual implementation yet.

We aim to implement a type checker for the type system with Datalog correspondences, as presented by Fournet, Gordon and Maffeis. We also aim to implement the type inference method for a type system with non-injective correspondences, as presented by Hüttel.

We use Objective Caml for both implementations. For the type checker implementation, the Succinct Solver is used to interpret the Datalog correspondences, as Datalog logic expressions are a subset of ALFP. For the type inference implementation, the Succinct Solver is used as the last stage of the inference algorithm.

We prove that the algorithm used to implement the type checker is sound and complete. We provide a few correctly typed examples with the type checker. We do not complete the type inference implementation. But we provide a partial implementation as a framework for a complete implementation of the method. We point to issues with the method and show how they can be, or might be, solved.

Preface

This DAT6 report was written as my master's thesis on my 10th semester at Computer Science at Aalborg University.

The code for the type checker can be found at `http://www.cs.aau.dk/~raabjerg/masters/policychecker-0.9.tar.gz`. The code for the type inference implementation can be found at `http://www.cs.aau.dk/~raabjerg/masters/inference-svn100609.tar.gz`. Both programs require a compiled Succinct Solver heap to be located in the `ssolver/` directory. The type inference implementation must be executed using the command: `ocaml -I ssolver/ alfpgen.ml`.

I would like to thank F-klub and everyone involved in it for providing distractions from this project from time to time. I believe the effects have been mostly beneficial in the end.

Contents

Chapter 1

Introduction

1.1 The Dolev-Yao Assumptions

In [3], Dolev and Yao present a number of assumptions that can be used as a framework to reason about the security of cryptographic communication protocols. To focus on the protocol itself, they remove considerations of any cryptographic algorithms the protocol may use, by assuming that any ciphertext is unbreakable, its plaintext only accessible by use of a decryption key. For the purposes of capturing protocol security, they also assume that any attacker on the protocol has access to any message passing through the network, can initiate communications with any user on the network, and has the opportunity to be receiver to any initiation of communication. So while the assumptions do not allow for the possible cracking of a ciphertext, they allow the attacker the opportunity of breaking the protocol in any other way he or she can imagine within the scope of the network. Such an attacker is frequently referred to as a "Dolev-Yao attacker".

1.2 The π- and Spi-calculi

In [15] and [16], Milner, Parrow and Walker present the π-calculus, a calculus of communicating systems which can be used to model communication protocols. In [1], Abadi and Gordon extend the π-calculus with cryptographic primitives to create the spi-calculus. Today, the names "π-calculus" and "spi-calculus" both cover a number of variants of those calculi. The spi-calculi are interesting because they allow us to model cryptographic protocols under the Dolev-Yao assumptions.

1.3 Correspondence Assertions

In [19], Woo and Lam suggests the use of correspondence assertions to formally define authentication. In basic terms, correspondence assertions can be used to express that when you reach one part of a protocol, some other specific part must have been executed previously. This is formalised in the context of a spi-calculus by Gordon and Jeffrey in [8] and [7], by the use of *begin* and *end* statements. We say that correspondences hold if, in every run, every *end* statement corresponds to a previously executed *begin* statement with the same label. These first papers specified *injective correspondences*, where each *begin* statement may only satisfy a single *end* statement with the same label.

But in [6] for instance, Gordon and Jeffrey use *non-injective correspondences* instead. For *non-injective correspondences*, each *begin* statement can satisfy any number of *end* statements that have the same label.

The notion of *safety* is defined to hold for a protocol if all correspondence assertions hold under every possible execution of that protocol. *Robust safety* holds if *safety* holds under the assumption of a Dolev-Yao attacker. We note that sound and complete verification of these properties is undecidable, on the grounds that we can express the halting problem for the spi-calculus using correspondence assertions, and that the spi-calculus is Turing complete. Using type systems for verification is a sound approach to the problem.

1.4 Type Systems and Type Inference

Apart from formalising correspondence assertions in the spi-calculus, Gordon and Jeffrey also develop a type system for verification of the *safety* and *robust safety* properties in [8] and [7]. To allow the verification of *robust safety*, Gordon and Jeffrey introduce opponent types to indicate the influence of a Dolev-Yao attacker in the type system. Since then, the spi-calculus and type system has been extended and built upon to allow such concepts as: One-to-many correspondences ([6] by Gordon and Jeffrey), fractional effects ([11] by Kikuchi and Kobayashi) and Datalog correspondences for authorisation policies ([4] by Fournet, Gordon and Maffeis). Further, Gordon and Jeffrey also develop a similar type system, which uses correspondence assertions to verify *secrecy* in [9], and in [14], Maffeis, Abadi, Fournet and Gordon extend [4] to allow dynamic verification.

A type system in itself will only allow us to verify processes which have already been typed. So given a process with type annotations, we can say if that process is well-typed, and thus if the process is *safe*. Type inference will allow us to take a process without type annotations and find an annotation that makes it well-typed, if possible. This can ease the process of verification significantly.

Type inference for type and effect systems such as those treated here is a recent development, and has been undertaken by such people as: Kikuchi and Kobayashi in [11] and [12], and Hüttel in [10]. Kikuchi and Kobayashi use fractional effects to lower the time complexity of type inference in [11], for a system without opponent types. In [12], they introduce type inference for a system with opponent types, but modified to remove opponent type rules, making the type system incomparable to the systems of Gordon and Jeffrey. [10] takes a subset of the type system developed in [4], and presents an inference method using ALFP and the Succinct Solver (a logic and a solver presented by Nielson, Nielson and Seidl in [17]). The Succinct Solver was developed as a result of Nielson and Nielson's work on flow analysis, and it is worth noting that their work has resemblances to type systems.

1.5 The Contributions of this Report

This report consists of two parts. The first part of the report concerns the implementation of a type checker for the type system presented by Fournet, Gordon and Maffeis in [4]. Citing [4]: "We built a typechecker and a symbolic interpreter for our language". Even so, that implementation of the type system does not appear to have been published. In chapter 2, we outline the spi-calculus and type system described in [4]. In chapter 3, we describe a type checking algorithm and show a minor modification to the original type system. In chapter 4, we highlight some implementation details that may hold an

interest to the reader. In chapter 5, we show a few examples of typed spi processes that are accepted by the type checker.

The second part of the report changes the focus somewhat. We treat a slightly different type system, described by Hüttel in [10] as a "subsystem of that of [4]". The only important difference from the type system described in [4] is the use of correspondence assertions as described by Woo and Lam in [19], instead of Datalog correspondences. Development of type inference methods for such a system has been undertaken from more than one angle. One approach, used by Kikuchi and Kobayashi in [12] involves modifying the type system presented by Gordon and Jeffrey in [7] to remove the opponent rules ("Un" rules), and thus make the task of type inference simpler. The resulting system is still sound with relation to safety and robust safety, but not comparable to the original system in [7].

In chapter 6, we show the type system from [10] which represents a subset of the one from [4]. In chapter 7, we explain the method from [10] and its motivation. In chapter 8, we show the constraint generation part of [10] and touch on the subject of type abstraction. In chapter 9, we show the Alternation Free Fixpoint Logic (ALFP) from [17] and go into detail with the subject of encoding constraints into ALFP formulae. In chapter 10, we show the correctness theorems from [10], stating soundness and completeness of the inference method. In chapter 11, we touch on a few issues encountered during the implementation, and discuss their possible solutions. In chapter 12, we highlight som details of the implementation.

And finally, we make our conclusion in chapter 13.

Part I

Type Checking

Chapter 2

A Spi Calculus with Datalog Correspondence Assertions

In this chapter, we outline a spi-calculus with Datalog correspondence assertions for expressing authenticity properties in the form of authorisation policies, and a type system for verifying these correspondence assertions. The syntax and semantics are made to closely resemble those of the spi-calculus presented in [4], with the differences that the replication operator !, is bound to **in** statements, and M is annotated on **match** statements (the exact reason for this is explained in section 3.3).

2.1 Syntax and Semantics

Syntax for Messages

$M, N ::=$	message
x, y, z	name
ok	**ok** token
pair(M, N)	pair of messages
$\{M\}_N$	M encrypted by N

Names can be seen as the variables of a spi process. These variables may be bound in a process by an **in**, **new**, **decrypt**, **match** or **split** term.

Syntax for Datalog

X, Y, Z	logic variable
$u ::=$	term
X	logic variable
M	spi calculus message
$L ::=$	literal
$p(u_1, \ldots, u_n)$	predicate p holds for terms $u_1, \ldots, u_n$
$C ::=$	Horn clause
$L :- L_1, \ldots, L_n$	clause, with $n \geq 0$ and $\texttt{fn}(L) \subseteq \cup_i \texttt{fn}(L_i)$
$S ::=$	Datalog program
$\{C_1, \ldots, C_n\}$	set of clauses

A Horn clause with $n = 0$ (no body), where each u_i of literal L is a message, is called a *fact*. We let F range over facts.

Inference of Facts

(Infer Fact)

$$\frac{L : -L_1, \ldots, L_n \in S \quad S \models L_i\sigma \quad \forall i \in 1 \ldots n}{S \models L\sigma} \text{ for } n \geq 0$$

where σ maps logic variables to messages

Datalog use Horn clauses without function symbols for its logic expressions, which is a subset of first order logic. For comparison, an expression such as $FooBar(a, b) : -Foo(a), Bar(b)$ would be written as $\forall a(\forall b(Foo(a) \wedge Bar(b) \rightarrow FooBar(a, b)))$ in first order logic. First order logic can be used to form any Datalog logic expression, but is more powerful as a logic, as it allows constructions such as disjunctions, existential quantification and negation. In the second part of this report, section 9.1, we also present ALFP, which is a larger subset of the first order logic.

Syntax for Processes

$P, Q, R ::=$	process
in$(M, x); P$	input
!**in**$(M, x); P$	replicated input
out(M, N)	output
new $x : T; P$	restriction of x in P
nil	empty process
decrypt M **as** $(y : T)_N; P$	decryption of M with key N
match $M : T_M$ **as** $(N, y : T); P$	match first pair component
split M **as** $(x : T, y : U); P$	pair splitting
S	Datalog Program
expect F	expectation of F

Syntax for Types

$T, U ::=$	type
Un	opponent type
Ch(T)	channel type
Key(T)	key type
Pair$(x : T, U)$	dependent pair type
Ok(S)	ok type

The function `fn` defines the set of free names of a process or type:

Free Names

$$
\begin{aligned}
\texttt{fn}(\textbf{nil}) &= \emptyset \\
\texttt{fn}(\textbf{in}(M, x); P) &= \texttt{fn}(M) \cup \texttt{fn}(P) \backslash \{x\} \\
\texttt{fn}(!\textbf{in}(M, x); P) &= \texttt{fn}(\textbf{in}(M, x); P) \\
\texttt{fn}(\textbf{out}(M, N)) &= \texttt{fn}(M) \cup \texttt{fn}(N) \\
\texttt{fn}(\textbf{new}\ x : T; P) &= \texttt{fn}(T) \cup \texttt{fn}(P) \backslash \{x\} \\
\texttt{fn}(\textbf{decrypt}\ M\ \textbf{as}\ (y : T)_N; P) &= \texttt{fn}(M) \cup \texttt{fn}(T) \cup \texttt{fn}(N) \cup \texttt{fn}(P) \backslash \{y\} \\
\texttt{fn}(\textbf{match}\ M : T_M\ \textbf{as}\ (N, y : T); P) &= \texttt{fn}(M) \cup \texttt{fn}(T_M) \cup \texttt{fn}(N) \cup \texttt{fn}(T) \cup \texttt{fn}(P) \backslash \{y\} \\
\texttt{fn}(\textbf{split}\ M\ \textbf{as}\ (x : T, y : U); P) &= \texttt{fn}(M) \cup \texttt{fn}(T) \cup \texttt{fn}(U) \backslash \{x\} \cup \texttt{fn}(P) \backslash \{x, y\} \\
\texttt{fn}(\textbf{expect}\ F) &= \texttt{fn}(F) \\
&\dots
\end{aligned}
$$

The function `bn` defines the set of bound names of a process or type:

Bound Names

$$
\begin{aligned}
\texttt{bn}(\textbf{nil}) &= \emptyset \\
\texttt{bn}(\textbf{in}(M, x); P) &= \{x\} \cup \texttt{bn}(P) \\
\texttt{bn}(!\textbf{in}(M, x); P) &= \texttt{bn}(\textbf{in}(M, x); P) \\
\texttt{bn}(\textbf{out}(M, N)) &= \emptyset \\
\texttt{bn}(\textbf{new}\ x : T; P) &= \{x\} \cup \texttt{bn}(T) \cup \texttt{bn}(P) \\
\texttt{bn}(\textbf{decrypt}\ M\ \textbf{as}\ (y : T)_N; P) &= \{y\} \cup \texttt{bn}(T) \cup \texttt{bn}(P) \\
\texttt{bn}(\textbf{match}\ M : T_M\ \textbf{as}\ (N, y : T); P) &= \texttt{bn}(T_M) \cup \{y\} \cup \texttt{bn}(T) \cup \texttt{bn}(P) \\
\texttt{bn}(\textbf{split}\ M\ \textbf{as}\ (x : T, y : U); P) &= \texttt{bn}(T) \cup \texttt{bn}(U) \cup \{x, y\} \cup \texttt{bn}(P) \\
\texttt{bn}(\textbf{expect}\ F) &= \emptyset \\
&\dots
\end{aligned}
$$

For the full definition of `fn` and `bn`, see appendix A.1 and A.2.

Every x or y in the syntax denotes the *binding occurrence* of a name. **in** and **new** binds x in P, **decrypt** and **match** binds y in P, and **split** binds x in U and P, and y in P. The *free names* of a process are the names that are not bound by a binding occurrence of that name. Most binding occurrences of names are annotated with types, as can be seen in the syntax definitions of, **new**, **decrypt**, **match** and **split**. Types are checked statically, and have no effect on runtime. We explain types and the type system in section 2.2.

Alpha-conversion allows us to change the bound names of a process. An *alpha-conversion* of a process P is a process P' with the same meaning as P, but a different set of bound names.

We define $J\{M/x\}$ as the substitution of M for the free name x in J, where J is a message, effect, type or process. See appendix A.4 for the formal definition.

The operational semantics are defined by structural equivalence and reduction rules.

Rules for Structural Equivalence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv P' \Rightarrow \textbf{new}\ x : T; P \equiv \textbf{new}\ x : T; P'$	(Struct Res)
$P \equiv P' \Rightarrow P \mid R \equiv P' \mid R$	(Struct Res)
$\textbf{new}\ x : T; (P \mid Q) \equiv P \mid \textbf{new}\ x : T; Q$	(Struct Res Par) (for $x \notin \texttt{fn}(P)$)
$\textbf{new}\ x_1 : T_1; \textbf{new}\ x_2 : T_2; P \equiv$ $\textbf{new}\ x_2 : T_2; \textbf{new}\ x_1 : T_1; P$	(Struct Res Res) (for $x_1 \neq x_2$,$x_1 \notin fn(T_2)$,$x_2 \notin fn(T_1)$)
$P \mid \textbf{nil} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)

Rules for Reduction: $P \rightarrow P'$

$P \equiv Q, Q \rightarrow Q', Q' \equiv P' \Rightarrow P \rightarrow P'$	(Red Struct)
$P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q$	(Red Par)
$P \rightarrow P' \Rightarrow \textbf{new}\ x : T; P \rightarrow \textbf{new}\ x : T; P'$	(Red Res)
$\textbf{out}(m, N) \mid \textbf{in}(m, x); P \rightarrow P\{N/x\}$	(Red Com)
$\textbf{out}(m, N) \mid !\textbf{in}(m, x); P \rightarrow P\{N/x\} \mid !\textbf{in}(m, x); P$	(Red Repl)
$\textbf{decrypt}\ \{M\}_k\ \textbf{as}\ (y : T)_k; P \rightarrow P\{M/y\}$	(Red Decrypt)
$\textbf{match}\ (N, M) : T_{NM}\ \textbf{as}\ (N, y : T); P \rightarrow P\{M/y\}$	(Red Match)
$\textbf{split}\ (M, N)\ \textbf{as}\ (x : T_1, y : T_2); P \rightarrow P\{M/x\}\{N/y\}$	(Red Split)

We write $P \rightarrow^* Q$ if $P \rightarrow Q$, or if there is some R for which it holds that $P \rightarrow^* R$ and $R \rightarrow Q$. We write $P \rightarrow^*_{\equiv} Q$ if $P \equiv Q$ or $P \rightarrow^* Q$. The intuition then, is that Q is reachable from P.

We can now define the notions of *safety* and *robust safety* in this calculus:

Definition 1. A process P is *safe* iff whenever $P \rightarrow^*_{\equiv} \textbf{new}\ x; (\textbf{expect}\ F \mid P')$, we have $P' \equiv S \mid P''$ for some P'', where $S \models F$.

Definition 2. A process P is *robustly safe* iff for every opponent O we have that $P \mid O$ is *safe*

2.1.1 Example

Most of the students and staff of the Department of Computer Science at Aalborg University are members of a social club, F-klub. To participate in F-klub activities, one must be a member of F-klub. The Treo is an administrative group of F-klub, and to become a member of F-klub, one has to make a deposit to the Treo. As an example of how a spi-calculus with Datalog correspondences might be used, we will now present a process which describes how membership of the F-klub works.

As the basis for the process, we formulate a policy in Datalog with the Horn clauses A and B:

$$A = \mathsf{Factivity}(U) : -\mathsf{Fmember}(U)$$
$$B = \mathsf{Fmember}(U) : -\mathsf{Treo}(V), \mathsf{Deposit}(U, V)$$

A tells us that if U is a member of F-klub, then U may participate in F-klub activities. B tells us that if V is a member of the Treo, and U has made a deposit to V, then U is a member of F-klub. To reason about Treo members, we use the process P_T:

$$\begin{aligned} P_T \triangleq & !\mathbf{in}\ (\mathit{createTreo}, v); (\mathsf{Treo}(v) \mid !\mathbf{in}\ (\mathit{deposit}, u); \\ & (\mathsf{Deposit}(u, v) \mid !\mathbf{in}\ (\mathit{checkdeposit}, p); \mathbf{match}\ \mathbf{pair}(p, \mathit{nothing})\ \mathbf{as}\ (u, \mathit{nothing}); \\ & \mathbf{out}\ (\mathit{confirm}, \{\mathbf{pair}(u, \mathbf{pair}(v, \mathbf{ok})\}_{kf})))); \end{aligned}$$

Sending v on the *createTreo* channel, will make v a Treo member by stating the fact $\mathsf{Treo}(v)$. In turn, this spawns a process which will take and confirm deposits. To handle the question of whether someone may participate in F-klub activities or not, we use the process P_F:

$$\begin{aligned} P_F \triangleq & !\mathbf{in}\ (\mathit{activity}, u); \\ & (\mathbf{out}\ (\mathit{checkdeposit}, u) \mid \mathbf{in}\ (\mathit{confirm}, \mathit{cipher}); \\ & \mathbf{decrypt}\ \mathit{cipher}\ \mathbf{as}\ \{q\}_{kf}; \mathbf{match}\ q\ \mathbf{as}\ (u, v); \\ & (\mathbf{expect}\ \mathsf{Factivity}(u) \mid P)) \end{aligned}$$

We send u on the *activity* channel to check if u is allowed to participate in F-klub activities. P_F tries to confirm this by querying one of the processes spawned by P_T on the channels *checkdeposit* and *confirm*. By stating **expect** $\mathsf{Factivity}(u)$ in parallel with P, we say that for this process to be safe, $\mathsf{Factivity}(u)$ must hold whenever we execute P. As a finishing touch, we define the entirety as the process *Sys*, which we want to be *safe*, and say that the key kf is restricted to P_T and P_F.

$$\mathit{Sys} \triangleq A \mid B \mid (\mathbf{new}\ kf; (P_T \mid P_F))$$

2.2 Type System

The type checker we develop is based on the type system presented by Fournet, Gordon and Maffeis in [4]. The type rules are shown in table 2.1 and 2.2. To make the type checker implementation simpler, we annotate M with a type in the (Proc Match) rule. The exact reason for this is explained in section 3.3.

Good Message: $E \vdash M : T$

(Msg x)
$$\frac{E \vdash \diamond \quad x \in \texttt{dom}(E)}{E \vdash x : E(x)}$$

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

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

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash \mathbf{pair}(M, N) : \mathbf{Pair}(x : T, U)}$$

(Msg Pair Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \mathbf{pair}(M, N) : \mathbf{Un}}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad \texttt{fn}(S) \subseteq \texttt{dom}(E) \quad \texttt{clauses}(E) \models C \quad \forall C \in S}{E \vdash \mathbf{ok} : \mathbf{Ok}(S)}$$

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

Table 2.1: Good messages

Good Process: $E \vdash P$

(Proc Input)(μ is either ! or nothing)
$$\frac{E \vdash M : \mathbf{Ch}(T) \quad E, x : T \vdash P}{E \vdash \mu\mathbf{in}(M, x); P}$$

(Proc Input Un)(μ is either ! or nothing)
$$\frac{E \vdash M : \mathbf{Un} \quad E, x : \mathbf{Un} \vdash P}{E \vdash \mu\mathbf{in}(M, x); P}$$

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

(Proc Output Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \mathbf{out}(M, N)}$$

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

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

(Proc Split)
$$\frac{E \vdash M : \mathbf{Pair}(x : T, U) \quad E, x : T, y : U \vdash P}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P}$$

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

(Proc Match)
$$\frac{E \vdash M : \mathbf{Pair}(x : T, U) \quad E \vdash N : T \quad E, y : U\{N/x\} \vdash P}{E \vdash \mathbf{match}\ M : \mathbf{Pair}(x : T, U)\ \mathbf{as}\ (N, y : U\{N/x\}); P}$$

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

(Proc Res)
$$\frac{E, x : T \vdash P \quad \texttt{generative}(T)}{E \vdash \mathbf{new}\ x : T; P}$$

(Proc Par)
$$\frac{E, \texttt{env}(P_2) \vdash P_1 \quad E, \texttt{env}(P_1) \vdash P_2 \quad \texttt{fn}(P \mid Q) \subseteq \texttt{dom}(E)}{E \vdash P_1 \mid P_2}$$

(Proc Nil)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{nil}}$$

(Proc Fact)
$$\frac{(E, C) \vdash \diamond}{E \vdash C}$$

(Proc Expect)
$$\frac{(E, C) \vdash \diamond \quad \texttt{clauses}(E) \models C}{E \vdash \mathbf{expect}\ C}$$

Table 2.2: Good Processes

Chapter 3

Type Checking

In this chapter, we will treat the theoretical side of developing the type checker. In section 3.1 we show the algorithms the type checker implements. In section 3.2, we discuss why the algorithms are structured as they are. In section 3.3, we present the reason for making an additional type annotation on **match** statements. In section 3.4, we present correctness lemmas for the algorithms.

3.1 Algorithms

Algorithm 1, 2 and 3 are the pseudocode for an algorithm that performs type-checking according to the type system described in section 2.2. The basic structure of these algorithms is simple, and consists of a switch statement each.

The algorithms take the form of three recursive functions: `env_judgment`, `msg_judgment` and `proc_judgment`. The functions consist of a switch on the structure of an environment, a message and a process, respectively. The cases of each switch cover all possible syntactic structures of the element, and will check the element in accordance with the type rules of section 2.2. Some constructions are covered by more than one case, to handle specific permutations of the construction. The reason for having more than one case for some constructions has to do with the fact that some types are generative and have no message constructors, and the fact that we must often choose between either a normal or an **Un** rule in the type system. We give a more detailed reasoning for this in section 3.2. The output of each algorithm is a boolean value. If at any point an element does not conform to the type rules, the algorithm returns false.

3.2 Cases of Generativity

Cases of the (Msg Encr) and (Msg Encr Un) rules are implemented in algorithm 2, starting at line 2.10 and 2.21. The two cases correspond to the situation where the key N is a name, and the situation where N is not a name. The reason for this distinction is that the type **Key**(T) is *generative*. There are no message constructors for the **Key**(T) type. So the only way N can be of type **Key**(T) is if N is a name, and the type can be extracted from the environment. In line 2.10, we use the (Msg Encr) rule only if we extract a **Key**(T) type. In line 2.21, we default on the (Msg Encr Un) rule if N is not a name, because **Key**(T) has no message constructor.

This approach to checking is repeated for several other rules, such as (Proc Decrypt*), (Proc Out*) and (Proc In*) where we can make decisions based on the form and type of keys and channels, which are both *generative types*.

The (Proc Split*) rules (lines 3.50, 3.58 and 3.60) are a bit more challenging, because they operate on pair messages. Pair messages are non-generative and we must therefore handle the extra case of the message M being a pair constructor for which we do not, strictly speaking, know the type. The type of $\mathbf{pair}(M, N)$ can be either $\mathbf{Pair}(x : T, U)$ or $\mathbf{Un}$. The solution for this case is to observe that if both T and U are $\mathbf{Un}$ (which is the only case where we can use the (Proc Split Un) rule), using $\mathbf{Pair}(x : \mathbf{Un}, \mathbf{Un})$ as the type of M has the exact same result as using $\mathbf{Un}$. So in the case of M being a pair constructor, we can choose to always apply (Proc Split).

3.3 Annotating Match

During the implementation, a minor flaw was found in the (Proc Match) rule, from a type checking perspective. Citing [4], the original (Proc Match) is presented as:

$$\text{(Proc Match)}\quad \frac{E \vdash M : \mathbf{Pair}(x : T, U) \quad E \vdash N : T \quad E, y : U\{N/x\} \vdash P}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y : U\{N/x\}); P}$$

If M is a name, we can extract the type of M from environment E. Thus, we can know the types T and U. However, M may not be a name. M may legally be a pair constructor, in which case the exact type of M is unknown. In this case, neither T nor U is known. To further complicate the issue, the type we annotate y with is the substituted form of U. So any U we find must make $U\{N/x\}$ possible.

In [5], Andersen implements a type checker for a similar type system, with a (Proc Match) rule identical to the one shown here. Looking at the code however, it appears that the possibility of M being a pair message was ignored, and that the type checker will fail in that instance.

Our solution is a simple one, however: Instead of attempting to infer T and U, as was the intention of Andersen in [5], we chose to annotate M with a type, so that T and U are always known.

Algorithm 1: env_judgment(env)

Data: Environment to be judged: *env*
Result: **true** if environment *env* is well-formed, **false** otherwise
switch *env* **do**
1.2 **case** $\emptyset$
return true
1.4 **case** $E, x : T$
return `env_judgment(`E`)` $\wedge$ (`fn(`T`)` $\subseteq$ `dom(`E`)`) $\wedge$ ($x \notin$ `dom(`E`)`)
1.6 **case** $E, \ell(M)$
return `env_judgment(`E`)` $\wedge$ (`fn(`$\ell(M)$`)` $\subseteq$ `dom(`E`)`)

Algorithm 2: msg_judgment(E, msg, typ)

Data: Environment of message: E,
Message to be judged: msg,
Assumed type of message: typ
Result: **true** if message msg has type typ in environment E, **false** otherwise
switch msg **do**
2.2 **case** **pair**(M, N)
switch typ **do**
2.4 **case** **Pair**$(x : T, U)$
return `msg_judgment(`E, M, T`)` $\wedge$ `msg_judgment(`E, N, $U\{M/x\}$`)`
2.6 **case** **Un**
return `msg_judgment(`E, M, **Un**`)` $\wedge$ `msg_judgment(`E, N, **Un**`)`
otherwise
return false
2.10 **case** $\{M\}_{Name(x)}$
if $typ =$ **Un** **then**
switch $E(x)$ **do**
2.13 **case** **Key**(T)
return `msg_judgment(`E, M, T`)` $\wedge$
`msg_judgment(`E, $Name(x)$, **Key**(T)`)`
2.15 **case** **Un**
return `msg_judgment(`E, M, **Un**`)` $\wedge$ `msg_judgment(`E, N, **Un**`)`
otherwise
return false
else
return false
2.21 **case** $\{M\}_N$
if $typ =$ **Un** **then**
return `msg_judgment(`E, M, **Un**`)` $\wedge$ `msg_judgment(`E, N, **Un**`)`
else
return false
2.26 **case** $Name(x)$
return $(E(x) = typ) \wedge (x \in$ `dom(`E`)`$) \wedge$ `env_judgment(`E`)`
2.28 **case** **ok**
switch typ **do**
2.30 **case** **Un**
return `env_judgment(`E`)`
2.32 **case** **Ok**(S)
return (`clauses(`E`)` $\models C \quad \forall C \in S$) $\wedge$
(`fn(`S`)` $\subseteq$ `dom(`E`)`) $\wedge$ `env_judgment(`E`)`
otherwise
return false

Algorithm 3: proc_judgment(E, proc)

Data: Environment of process: E,
Process to be judged: *proc*
Result: **true** if process *proc* is well-typed in environment E, **false** otherwise

```
switch proc do
3.2   case new x : T; P
        return generative(T) ∧ proc_judgment((E, x : T), P)
3.4   case (P|Q)
        return fn((P|Q)) ⊆ dom(E) ∧
        proc_judgment((env(Q), E), P) ∧ proc_judgment((env(P), E), Q)
3.6   case nil
        return env_judgment(E)
3.8   case C
        return env_judgment((E, C))
3.10  case expect C
        return (clauses(E) ⊨ C) ∧ env_judgment((E, C))
3.12  case decrypt M as (y : T)_Name(x); P
        switch E(x) do
3.14      case Key(T)
            return
            msg_judgment(E, M, Un)∧msg_judgment(E, Name(x), Key(T))∧
            proc_judgment((E, y : T), P)
3.16      case Un
            return msg_judgment(E, M, Un) ∧
            msg_judgment(E, Name(x), Un) ∧
            proc_judgment((E, y : Un), P)
          otherwise
            return false
3.20  case decrypt M as (y : T)_N; P
        return msg_judgment(E, M, Un) ∧
        msg_judgment(E, N, Un) ∧
        proc_judgment((E, y : Un), P)
3.22  case match M : T_M as (N, y : U_sub); P
        switch T_M do
3.24      case Pair(x : T, U)
            return msg_judgment(E, M, T_M) ∧ msg_judgment(E, N, T) ∧
            proc_judgment((E, y : U{N/x})) ∧ U{N/x} = U_sub
3.26      case Un
            return msg_judgment(E, M, Un) ∧ msg_judgment(E, N, Un) ∧
            proc_judgment((E, y : Un), P)
```

(* Switch statement continued on next page *)

Algorithm 3: proc_judgment(E, proc)

```
(* Continuation of switch statement *)
3.30  case out(Name(x), N)
        switch E(x) do
3.32      case Ch(T)
            return
            msg_judgment(E, Name(x), Ch(T)) ∧ msg_judgment(E, N, T)
3.34      case Un
            return msg_judgment(E, M, Un) ∧
            msg_judgment(E, N, Un)
          otherwise
            return false
3.38  case out(M, N)
        return msg_judgment(E, M, Un) ∧
        msg_judgment(E, N, Un)
3.40  case μin(Name(y), x); P
        switch E(y) do
3.42      case Ch(T)
            return msg_judgment(E, Name(y), Ch(T)) ∧
            proc_judgment((E, x : T), P)
3.44      case Un
            return
            msg_judgment(E, Name(y), Un) ∧ proc_judgment((E, x : Un), P)
          otherwise
            return false
3.48  case μin(M, x); P
        return msg_judgment(E, M, Un) ∧
        proc_judgment((E, x : Un), P)
3.50  case split Name(z) as (x : T, y : U); P
        switch E(z) do
3.52      case Pair(x : T, U)
            return msg_judgment(E, Name(z), Pair(x : T, U)) ∧
            proc_judgment((E, x : T, y : U), P)
3.54      case Un
            return msg_judgment(E, Name(z), Un) ∧
            proc_judgment((E, x : Un, y : Un), P)
          otherwise
            return false
3.58  case split pair(M1, M2) as (x : T, y : U); P
        return msg_judgment(E, pair(M1, M2), Pair(x : T, U)) ∧
        proc_judgment((E, x : T, y : U), P)
3.60  case split M as (x : T, y : U); P
        return
        msg_judgment(E, M, Un) ∧ proc_judgment((E, x : Un, y : Un), P)
    end
```

3.4 Correctness

To prove the correctness of these algorithms in accordance with the type system, we present the following five lemmas, asserting that the functions of the algorithm implement the corresponding type judgments correctly.

Lemma 1. `env_judgment(`*env*`)` $\iff$ $env \vdash \diamond$

Proof. Proof by induction in the structure of *env*. See appendix B. □

Lemma 2. `msg_judgment(`*E, msg, typ*`)` $\Rightarrow$ $E \vdash msg : typ$

Proof. Proof by induction in the structure of *msg*. See appendix B. □

Lemma 3. `proc_judgment(`*E, proc*`)` $\Rightarrow$ $E \vdash proc$

Proof. Proof by induction in the structure of *proc*. See appendix B. □

Lemma 4. $E \vdash msg : typ \Rightarrow$ `msg_judgment(`*E, msg, typ*`)`

Proof. Proof by induction in the structure of *msg*. See appendix B. □

Lemma 5. $E \vdash proc \Rightarrow$ `proc_judgment(`*E, proc*`)`

Proof. Proof by induction in the structure of *proc*. See appendix B. □

By Lemmas 1, 2, 3, 4 and 5, the algorithms are both sound and complete in relation to the type system. The boolean output of the algorithms correspond to acceptance or rejection by the type system. This is interesting because our actual implementation of these algorithms correspond very closely to the pseudocode. So barring any outright bugs in the code itself, the implementation should be correct.

Chapter 4

Implementing the Type Checker

The type checker was implemented in Objective Caml, a multi-paradigm language supporting both functional, imperative and object oriented constructions ([13]). The type checker was written using only functional constructions, with one exception. In `spichecker.ml` for instance, we use the only object in the entire program to keep track of command line arguments. This prevalence of functional constructions is partly because the process of syntax-tree traversals, as used in a type checker, is well suited for the functional paradigm. But it is also partly because of the author's wish to train himself in the practical use of functional programming.

In this chapter, we first present the general purpose and use of each module, and how they fit together in a dependency hierarchy. We then proceed to present some specific implementation details that we believe may hold an interest to the reader.

4.1 Hierarchy

Figure 4.1 shows the hierarchy of module dependencies in the type checker. Following is a short description of each module.

- `spichecker.ml`: Main function of the type checker. This is the part of the code which reflects the pseudocode in section 3.1.
- `spitree.ml`: Syntax tree structure for the spi calculus. Contains `string_of` functions for all elements of the structure.
- `spilexer.mll`, `spiparser.mly`: Lexer and parser for the spi calculus. Parses to the structure of `spitree.ml`.
- `aconv.ml`: Alpha conversion. Contains a function to perform alpha conversion on a spi process such that every binding name of the process is distinct. This function is called before using the checker function in `spichecker.ml`. This simplifies the process of type checking.
- `auxiliary.ml`: Contains functions used by the type checking function of `spichecker.ml`, such as `fn`, `dom` and `clauses`.

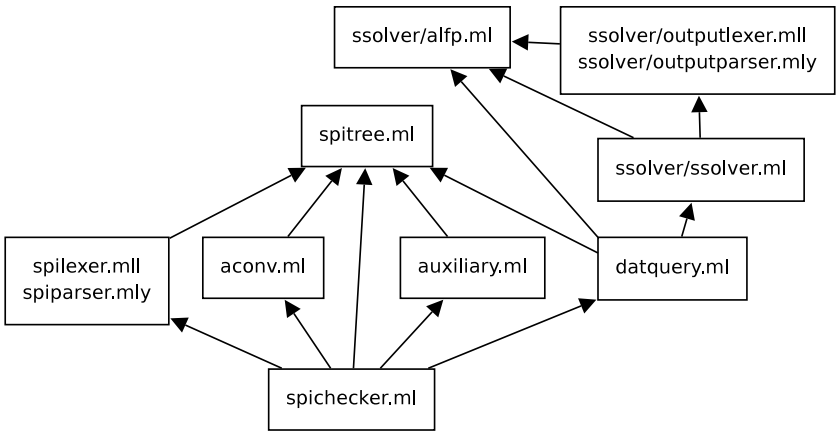

Figure 4.1: Module hierarchy of the type checker. An arrow indicates a dependency.

- `ssolver/alfp.ml`: Syntax tree structure for ALFP. Contains `string_of` functions for all elements of the structure.
- `ssolver/outputlexer.mll`, `ssolver/outputparser.mly`: Lexer and parser for the output of the Succinct Solver.
- `ssolver/ssolver.ml`: Simple OCaml wrapper for the Succinct Solver. Contains a function which returns the minimal solution to an ALFP formula represented by the tree structure of `ssolver/alfp.ml`. It needs SML and a compiled Succinct Solver heap to function.
- `datquery.ml`: Datalog interface to Succinct Solver wrapper. Contains functions to convert from the Datalog substructure in `spitree.ml` to ALFP. Performs queries on Datalog programs to resolve expect statements.

When `spichecker.ml` is executed with a spi process as input, the process is first parsed, and then alpha converted through `aconv.ml`, and finally type checked. Whenever a Datalog query needs to be performed, `datquery.ml` is called, which uses the Succinct Solver through the modules in `ssolver/` as its back-end for resolving queries.

4.2 Alpha Conversion

An alpha conversion of a process is another process with a different set of bound names, but with the same meaning as the original process. The alpha conversion returned by `proc_subst` in `aconv.ml`, ensures that every binding name is distinct, and that no bound name is the same as a free name. This is handled by prefixing each name with a natural number which is incremented for each new binding or free name. Being able to make these assumptions makes type checking easier in general, not least because it makes the process of capture-avoiding substitution trivial.

Three simple functions are responsible for the actual name substitution: `subst_name`, `bind` and `unbind`. A substitution map and an accumulator is passed with each call to one of the traversal functions. An updated substitution map and accumulator is returned (`us`, `ua`), together with the alpha conversion of the element. All names in the alpha conversion are on the form `#_name`, where `#` is a natural number and `name` is the name

Listing 4.1: Function `subst_name`

```
let subst_name x accum substMap =
  if SMap.mem x substMap then
    (SMap.find x substMap, accum, substMap)
  else
    let newname = (string_of_int accum) ^ "_" ^ x in
    let newMap = SMap.add x newname substMap in
      (newname, accum + 1, newMap)
```

Listing 4.2: Function `bind`

```
let bind x accum substMap =
  let newname = (string_of_int accum) ^ "_" ^ x in
  let newMap = SMap.add x newname substMap in
    if SMap.mem x substMap then
      let oldname = SMap.find x substMap in
        (newname, accum + 1, newMap, (x, oldname))
    else
      (newname, accum + 1, newMap, (x, ""))
```

from the original process. The accumulator ensures that every fresh name is unique by increasing `#` for each name.

The algorithm itself can be seen from the "viewpoint" of the substitution map, which at any given point during the execution maps names from the original process to names in the alpha conversion.

If we see a free name, we call the `subst_name` function (listing 4.1). If the name is in the map, we substitute. If not, we create a fresh name, substitute, and add the substitution to the map. This way, we catch and substitute any free names to make sure that bound and free names do not clash.

If we see a binding occurrence of a name, we call `bind` (listing 4.2) before traversing into the scope it binds, and `unbind` (listing 4.3) afterwards. `bind` returns not only the new name, accumulator and map, but also an old mapping that we may have replaced in the substitution map, and which needs to be reinserted after the traversal of the bound scope. Calling `unbind` with the old mapping will handle this reinsertion.

A feature of this approach is that the substitution map returned from the alpha conversion represents the mappings of all the free names of the process. Any bound name is removed by `unbind` when exiting the bound scope, while free names are left in the map. To check for robust safety, we declare these names to be of type **Un** in the environment passed to the type checker.

Having written these three functions, the matter of handling an alpha conversion becomes the relatively simple, if tedious, task of writing the traversal functions and remembering when to call which function. In hindsight, since the accumulator and substitution map are used as persistent structures in the traversal functions of `aconv.ml`, one might consider using mutable values to represent these, and avoid passing too many arguments with each call.

Listing 4.3: Function `unbind`

```
let unbind (x, oldname) substMap =
  if oldname = "" then
    SMap.remove x substMap
  else
    SMap.add x oldname substMap
```

4.3 The Succinct Solver Wrapper

The folder `ssolver/` contains the wrapper for the Succinct Solver, a program developed by Nielson, Nielson and Seidl for the purpose of calculating models to ALFP formulae ([17]). ALFP is a subset of first order logic, and will be explained in detail in section 9.1.

The wrapper was developed as a "shortcut" to handle the evaluation of Datalog queries. Several other tools might have been used for this purpose. The SWI-Prolog interpreter was considered, for instance, as Datalog is a subset of Prolog. In the end, we settled on using the Succinct Solver, since we would have to find a way to interface with it anyway, in the type inference part of this project.

The type checker was written using Objective Caml, and is interpreted or compiled with ocaml. The Succinct Solver was written using SML, and is interpreted or compiled with smlnj (SML New Jersey). Perhaps surprisingly, directly interfacing between the two does not yet appear possible, despite the fact that OCaml and SML are both considered dialects of ML.

The solution used instead is conceptually simple, and consists of the type checker executing the Succinct Solver command and parsing the output. In truth, this type of solution should be considered a last resort, used in the absence of a more sensible approach. It is very error-prone, and requires some care when formatting input and parsing output. In this particular case, we should take note of the fact that the atoms we use for terms in Datalog are Spi messages.

As part of the wrapper, we wrote `ssolver/alfp.ml` which contains a data-structure for ALFP, with accompanying `string_of` functions that facilitate the generation of strings readable by the Succinct Solver. `datquery.ml` contains the function `alfp_of_dat`, for converting from the Spi-representation of Datalog to ALFP. One reason that we cannot use the ALFP data-structure for the Spi-representation, is that in Spi, we use Spi messages as the atoms of Datalog terms. These messages must be converted to strings to fit into the standard ALFP syntax. To make sure that these strings are unique for unique messages, and that they are not misinterpreted in any way by either the parser of the Succinct Solver or our parser for the resulting output, we make the `string_of_msg` function in `spitree.ml` capable of using an alternative syntax, clearly marking names, and using a set of characters that could not potentially be confused with the syntax of ALFP.

Default message syntax

message ::=	
`ok`	ok token
{*message*}*message*	encrypted message
`pair(`*message*`,`*message*`)`	pair message
`['a'-'z']['a'-'z' 'A'-'Z' '0'-'9']`$*$	name

Listing 4.4: Function `type_equal`

```
let rec type_equal ?(acc=0) t1 t2 =
  match (t1, t2) with
    | (Un, Un) -> true
    | (Key(t), Key(u)) -> (type_equal ~acc:acc t u)
    | (Ch(t), Ch(u)) -> (type_equal ~acc:acc t u)
    | (TPair(x, left1, right1), TPair(y, left2, right2)) ->
        if (type_equal ~acc:acc left1 left2) then
          let r1_subst = subst_name right1 x (Name(string_of_int acc)) in
          let r2_subst = subst_name right2 y (Name(string_of_int acc)) in
            type_equal ~acc:(acc + 1) r1_subst r2_subst
        else false
    | (TOk(d1), TOk(d2)) ->
        (List.for_all (fun d1elt ->
                         List.mem d1elt d2
                       ) d1) &&
        (List.for_all (fun d2elt ->
                         List.mem d2elt d1
                       ) d2)
    | _ -> false
```

Alternative message syntax

message ::=	
`ok`	ok token
{*message*}*message*	encrypted message
`pair<`*message*`,`*message*`>`	pair message
`Name<['a'-'z']['a'-'z' 'A'-'Z' '0'-'9']*>`	name

4.4 Type Equality

The problem of type equality was one problem that appeared trivial at first sight. One might be tempted to think that types can be compared by the standard operator for structural equality, "=". But one should be reminded that types can contain both dependent pair types and Datalog facts in the form of the clauses D in a **Ok**(D) type. A Datalog program is handled in our data structure as a list of clauses, and equality in this case is just a matter of checking if all elements in one list is contained in the other, and vice versa. While perhaps not as efficient as it could be, the implementation of case `(TOk(d1), TOk(d2))` in listing 4.4 is at least simple.

The problem with dependent pair types is that we can have two types, **Pair**($x : Un$, **Ok**(D_1)) and **Pair**($y : Un$, **Ok**(D_2)) that have the same meaning but are not structurally equal, because the binding name x differs from y. The solution is simply to perform alpha-conversion to make the names equal whenever we need to check equality between two dependent pair types. The fresh names for equality check are natural numbers, which are accumulated recursively.

4.5 Exception Handling

Calling what we do in the code for "exception handling" may be a stretch of the definition of "handling". If at any type we encounter a type error, we raise an exception. This being a type checker, we are not supposed to recover from a type error. So any exception we catch in relation to a type error, is only caught to raise another exception with more information accessible from that particular scope. In the main call to the type checker in `spichecker.ml`, we simply catch and print the associated string of any `TypeFailure` exception. The `TypeFailure` exception is raised if we catch a `MsgTypeFailure` exception from the main body of the checker functions (`env_judgment`, `msg_judgment` and `proc_judgment`). `MsgTypeFailure` is raised on any type failure with an appropriate explanatory string. `TypeFailure` adds the information of exactly which process caused that exception.

Chapter 5

Examples

To demonstrate the type checker we have included a few sample spi processes with the checker, in the `spisamples/` directory. We show a few of them here, together with a short explanation. Since these examples are taken from text files, the formatting is a bit different from other spi samples in this report. Each example shown here are well-typed according to the type checker. To test the type checker, one simply has to make small changes to the types or correspondences to introduce type errors, and see if the checker catches the error.

5.1 Sign and Decrypt

The following process asserts a simple correspondence between two sub-processes, where one sends a message to the other.

```
new ka:Key(Ok(send(a)));
(
((begin send(a)| out(pub, {ok}ka)) |
in(pub, crypt); decrypt crypt as {o:Ok(send(a))}ka; end send(a)
)
```

5.2 Reviewer

The following example is an adaptation of an example presented by Fournet, Gordon and Maffeis in [4]. It models a situation where a reviewer can file an opinion on a report, or delegate that responsibility to another reviewer. If the reviewer is a referee, he may write an opinion on a report assigned to him, or delegate that responsibility to someone else. If the reviewer is a programme committee member (pcmember), he may file an opinion on any report.

As a seemingly very large and complex process, this example highlights the possibility of improving the type checker by implementing shorthand forms (syntactic sugar) for common syntactic constructions, akin to the ones used in [4].

The policy presented as part of the process consists of three Horn clauses. The first states that if principal U has been asked to review paper ID, and U has written opinion R on paper ID, then R attaches to paper ID as the opinion of U.

The second Horn clause states that if U is a programme committee member and R is U's opinion on ID, then R attaches to paper ID as the opinion of U. So anyone who is a programme committee member may file opinions on papers.

The third Horn clause states that if principal U has been asked to review ID, and U delegates ID to V, then V has been asked to review ID, and may do so.

```
({report(U,ID,R) :- referee(U,ID), opinion(U,ID,R);
 report(U,ID,R) :- pcmember(U), opinion(U,ID,R);
 referee(V,ID) :- referee(U,ID), delegate(U,V,ID)}
|
 new pwdb:Ch(Pair(
      u:
      Un,
      Pair(none:
           Key(Pair(v:Un, Pair(id:Un, Ok({delegate(u,v,id)})))),
           Key(Pair(id:Un, Pair(report:Un, Ok({opinion(u,id,report)})))))
          )))
;
 (!in(createReviewer, v);
    new kdv: Key(Pair(z:Un, Pair(id:Un, Ok({delegate(v,z,id)})))));
    new krv: Key(Pair(id:Un, Pair(report:Un, Ok({opinion(v,id,report)})))));
    (out (pwdb, pair(v, pair(kdv, krv)))|
     (!in (sendreportonline, inrep);
          match inrep:Un as (v,inrest:Un);
          split inrest as (id:Un, report:Un);
      ({opinion(v,id,report)}
          |out (filereport, pair(v,{pair(id, pair(report, ok))}krv)))
     |!in (delegateonline, inrep);
          match inrep:Un as (v, inrest:Un);
          split inrest as (w:Un, id:Un);
      ({delegate(v,w,id)}
          |out (filedelegate, pair(v,{pair(w,pair(id,ok))}kdv)))
     )
    )
 |new refereedb:Ch(Pair(u:Un, Pair(id:Un, Ok({referee(u,id)}))));
  (!in (filereport, indat);split indat as (v:Un,e:Un);
       in(pwdb, indat);
          match indat:
            Pair(u:Un,
                 Pair(none:Key(Pair(v:Un,
                                    Pair(id:Un,
                                         Ok({delegate(u,v,id)})))),
                                Key(Pair(id:Un,
                                         Pair(report:Un,
                                              Ok({opinion(u,id,report)}))))))
          as (v, inrest:Pair(none:Key(Pair(w:Un,
                                           Pair(id:Un,
                                                Ok({delegate(v,w,id)})))),
                                      Key(Pair(id:Un,
                                               Pair(report:Un,
```

```
                                                    Ok({opinion(v,id,report)})))))));
        split inrest as (kdv:Key(Pair(w:Un,
                                      Pair(id:Un,
                                           Ok({delegate(v,w,id)})))),
                         krv:Key(Pair(id:Un,
                                      Pair(report:Un,
                                           Ok({opinion(v,id,report)}))))));
        decrypt e as {edec:Pair(id:Un,
                                Pair(report:Un,
                                     Ok({opinion(v,id,report)})))}krv;
        split edec as (id:Un,
                       rest:Pair(report:Un, Ok({opinion(v,id,report)})));
        split rest as (report:Un, o:Ok({opinion(v,id,report)}));
      in(refereedb, indat);
        match indat:Pair(u:Un, Pair(id:Un, Ok({referee(u,id)})))
         as (v, rest:Pair(id:Un, Ok({referee(v,id)})));
        match rest:Pair(id:Un, Ok({referee(v,id)}))
         as (id, o:Ok({referee(v,id)}));
        expect report(v,id,report)
|!in (filedelegate, indat); split indat as (v:Un, sigd:Un);
      in(pwdb, indat);
        match indat:Pair(u:Un,
                         Pair(none:Key(Pair(v:Un,
                                            Pair(id:Un,
                                                 Ok({delegate(u,v,id)})))),
                              Key(Pair(id:Un,
                                       Pair(report:Un,
                                            Ok({opinion(u,id,report)}))))))
        as (v, inrest:Pair(none:Key(Pair(w:Un,
                                         Pair(id:Un,
                                              Ok({delegate(v,w,id)})))),
                           Key(Pair(id:Un,
                                    Pair(report:Un,
                                         Ok({opinion(v,id,report)})))))));
        split inrest as (kdv:Key(Pair(w:Un,
                                      Pair(id:Un,
                                           Ok({delegate(v,w,id)})))),
                         krv:Key(Pair(id:Un,
                                      Pair(report:Un,
                                           Ok({opinion(v,id,report)}))))));
        decrypt sigd as {edec:Pair(w:Un,
                                   Pair(id:Un,
                                        Ok({delegate(v,w,id)})))}kdv;
        split edec as (w:Un, rest:Pair(id:Un, Ok({delegate(v,w,id)})));
        split rest as (id:Un, o:Ok({delegate(v,w,id)}));
      in(refereedb, indat);
        match indat:Pair(u:Un,
                         Pair(id:Un, Ok({referee(u,id)})))
         as (v, rest:Pair(id:Un, Ok({referee(v,id)})));
        match rest:Pair(id:Un, Ok({referee(v,id)}))
```

```
            as (id, o:Ok({referee(v,id)}));
           out(refereedb, pair(w, pair(id, ok)))
 )
|new kp:Key(Pair(u:Un,Ok({pcmember(u)})));
 (!in (createpcmember, indat);
     split indat as (u:Un, pc:Un);({pcmember(u)}
                                   |out(pc, {pair(u,ok)}kp))
 |!in (filepcreport, indat);
     split indat as (v:Un, indat:Un);
     split indat as (e:Un, pctoken:Un);
     in (pwdb, indat);
     match indat:Pair(u:Un,
                      Pair(none:Key(Pair(v:Un,
                                         Pair(id:Un,
                                              Ok({delegate(u,v,id)})))),
                           Key(Pair(id:Un,
                                    Pair(report:Un,
                                         Ok({opinion(u,id,report)}))))))
     as (v, inrest:Pair(none:Key(Pair(w:Un,
                                      Pair(id:Un,
                                           Ok({delegate(v,w,id)})))),
                        Key(Pair(id:Un,
                                 Pair(report:Un,
                                      Ok({opinion(v,id,report)}))))));
     split inrest as (kdv:Key(Pair(w:Un,
                                   Pair(id:Un,
                                        Ok({delegate(v,w,id)})))),
                      krv:Key(Pair(id:Un,
                                   Pair(report:Un,
                                        Ok({opinion(v,id,report)}))))));
     decrypt e as {edec:Pair(id:Un,
                             Pair(report:Un,
                                  Ok({opinion(v,id,report)})))}krv;
     split edec as (id:Un,
                    rest:Pair(report:Un, Ok({opinion(v,id,report)})));
     split rest as (report:Un,
                    o:Ok({opinion(v,id,report)}));
     decrypt pctoken as {dcrypt:Pair(u:Un,Ok({pcmember(u)}))}kp;
     match dcrypt:Pair(u:Un,Ok({pcmember(u)}))
      as (v, o:Ok({pcmember(v)}));
     expect report(v,id,report)
 )
)
)
```

Part II

Type Inference

Chapter 6

A Type System with Simple Correspondence Assertions

The spi-calculus and type system we use for the part of the project concerning type inference is slightly different from the ones used in the first part. For the calculus itself, we can simply substitute **begin** $\ell(M)$ for S and **end** $\ell(M)$ for **expect** F in the syntax for processes in section 2.1. For the semantics of the type system, we reproduce the system of [10] here, in table 6.1 and 6.2. Although there are certain cosmetic differences, the only real differences in what they express are related to the different kinds of correspondences.

We add the Δ notation, which denotes a *type/effect assignment*. If we say that $TVar$ and $EVar$ denote the sets of type/effect variables, respectively, we can cite the definition of Δ from [10]:

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

Since we define another type of correspondence assertions, we also need to redefine *safety* and restate the definition of *robust safety*:

Definition 4. A process P is *safe* iff whenever $P \rightarrow_{\equiv}^{*}$ **new** x; (**end** $\ell(M) \mid P'$), we have $P' \equiv$ **begin** $\ell(M) \mid P''$ for some P''

Definition 5. A process P is *robustly safe* iff for every opponent O we have that $P \mid O$ is *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 Name)
$$\frac{E \vdash \diamond \quad E = E', a : T, E''}{E \vdash_\Delta a : T}$$

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

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

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

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

Table 6.1: Type rules for messages

Good Process: $E \vdash_\Delta P$

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

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

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

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

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

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

(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)
$$\frac{E \vdash_\Delta M : \mathbf{Pair}(x : T, U) \quad E \vdash_\Delta N : T \quad E, y : U(N) \vdash_\Delta P}{E \vdash_\Delta \mathbf{match}\ M : \mathbf{Pair}(x : T, U)\ \mathbf{as}\ (N, y : U(N)); P}$$

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

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

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

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

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

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

Table 6.2: Good Processes

Chapter 7

Inferring Types and Effects by ALFP

7.1 Method

In chapters 8 and 9, we describe some of the details of the inference method presented in [10]. We devote this section to a more informal overview of that method. The basic steps of the method are as follows:

1. Generate constraint φ from process P
2. Encode φ to the ALFP formula ψ
3. Feed the ALFP formula $\psi \wedge \psi_{ax}$ to the Succinct Solver, where ψ_{ax} is a formula of axioms

The correctness properties of this method are stated through two theorems in [10]. These theorems can informally be stated as follows:

- **Soundness:** Some type assignment makes P well-typed if $\psi \wedge \psi_{ax}$ is satisfiable in a *failure-free* model.
- **Completeness:** $\psi \wedge \psi_{ax}$ is satisfiable in a *failure-free* model if some type assignment makes P well-typed.

We will come back to the formal statement of those theorems in chapter 10, when the definitions to describe them is in place. For now, we will consider their implications informally.

Informally, the satisfaction of an ALFP formula works by populating relations to find the least solution of the formula (for more information, see section 9.1. In this case, we use the Fail relation to indicate if a process is *failure-free* or not. If the least solution of the encoded formula does not require us to populate Fail then the process is *failure-free*, and vice versa. The two theorems we cited from [10] tells us that a process is *failure-free* if and only if that process can be well-typed according to the type system.

One could say that extracting types from the result of the Succinct Solver and assigning them to the process is a fourth, optional stage to the algorithm presented above. In theory, it is enough to look at the output of the Solver and confirm that the Fail

relation is empty to know that inference has succeded and the process is robustly safe. But in practice, it is not a bad idea to implement this fourth stage and feed the typed process to an external type checker to verify the result. Incidentally, the type checker developed in the first part of this project should suffice for that purpose. Although the type system was built for Datalog correspondences, it is compatible with traditional one-to-many correspondences. As Fournet, Gordon and Maffeis indicate in [4], one-to-many correspondences has a simple translation to Datalog correspondences. We built the type checker to accept the syntax for one-to-many correspondences, and perform the translation in those cases.

The first stage of the inference algorithm is described in chapter 8, through a number of rules defining which constraints to extract from each process and message primitive. Each constraint rule reflects type rules from the type system in chapter 6, and defines constraints from the conditions of each of those type rules.

In the second stage, the constraints are directly encoded to a conjunction of ALFP formulae. A number of axioms are defined as part of the conjunction. These axioms describe how to populate the relations using the information from the initial assertions and Horn clauses encoded from the constraints.

In the third stage, we use the Succinct Solver to populate the relations. The relations describe such information as type variables and type assignments, and should when populated correctly, describe a correct typing of the process with an empty Fail relation if and only if the process is robustly safe.

7.2 Reasoning

The task of type inference in a type system like the one Hüttel treats in [10] has been undertaken by a few other people. In [2], Dahl develops a type inference algorithm where type constraints are solved by unification, and effect constraints are solved by a non-deterministic algorithm. In [11], Kikuchi and Kobayashi show a form of type inference for a similar type system, but the focus is on reducing time complexity of inference, and the type system is one without opponent types. So they can only check for *safety*, not *robust safety*. In [12], Kikuchi and Kobayashi show an extension of the type inference method of [11] which allows for opponent types. But while that type system is similar to other type systems for verifying correspondence assertions, it is not comparable. To make the task simpler in [12], the system was made to avoid separate opponent type judgments. As a result, subject reduction does not hold, and some processes that can be verified by the type system used in [10] cannot be verified by the type system used in [12], and vice versa.

The method proposed by Hüttel in [10] and described above, qualifies in this context as a new, experimental approach to the task of inferring types and effects in a type system for verifying correspondence assertions. It also has the advantage of being sound and complete with respect to a type system that is a proper subset of the system presented previously in [4].

Chapter 8

Constraint Generation

In this chapter, we will discuss the aspect of constraint generation presented by Hüttel in [10]. In section 8.1, we present the constraint language itself. In section 8.2, we present the constraint generation method. In section 8.3, we present some of the general constraints implied by the use of axioms in the encoding to ALFP formulae. In section 8.4, we discuss the usage and meaning of type abstraction in the constraints.

8.1 Constraint Language

In [10], Hüttel presents a constraint language, intended as an intermediate step between the type rules and the ALFP formulae. Below, we present this constraint language. φ is the formula syntax, ϕ_T represents type constraints, ϕ_S effect constraints and ϕ_E environment constraints. It is worth noting that the constraint language is more powerful than ALFP, so not everything that can be expressed with constraints can be encoded to ALFP. A negation, for instance, can only be used as a query in the precondition of a Horn clause in ALFP. So we can only use $U_T \neq \mathbf{Un}$ in φ_1 of $\varphi_1 \Rightarrow \varphi_2$.

Equality, $=$ denotes a *necessary* equality and $\stackrel{?}{=}$ denotes a *possible* equality. The distinction is relevant because many of the type rules have two incarnations: A normal rule and an Un rule, making it possible to type many constructions with either normal or **Un** types. Necessary and possible equalities are used to express this by denoting **Un** types as *necessary* whenever we must apply them, and other types as *possible*, to be applied as *necessary* types to any type variable that did not receive the **Un** type.

Fail is intended to indicate failure of the formula, if it cannot be satisfied to represent a *safe* assignment of types.

$$\varphi ::= \phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x.\varphi_1$$
$$\phi ::= \phi_T \mid \phi_S \mid \phi_E$$

$$\begin{aligned}\phi_T ::= {} & U_T \stackrel{?}{=} \mathbf{Ch}(U_{T_1}) \mid U_T \stackrel{?}{=} \mathbf{Pair}(x : U_{T_1}, U_{T_2}) \mid U_T \stackrel{?}{=} \mathbf{Un} \\ & \mid U_T \stackrel{?}{=} \mathbf{Key}(U_{T_1}) \mid U_T \stackrel{?}{=} U_{T_1}^M(x) \mid U_T \stackrel{?}{=} U_{T_1}(M) \\ & \mid U_T = \mathbf{Ch}(U_{T_1}) \mid U_T = \mathbf{Pair}(x : U_{T_1}, U_{T_2}) \mid U_T = \mathbf{Un} \\ & \mid U_T = \mathbf{Key}(U_{T_1}) \mid U_T = U_{T_1}^M(x) \mid U_T = U_{T_1}(M) \\ & \mid U_T \neq \mathbf{Un} \mid T \text{ generative} \mid \mathsf{Fail} \mid M : U_T\end{aligned}$$

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

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

8.2 Constraint Generation

Table 8.1 and 8.2 shows how we generate constraints from messages and processes, respectively.

8.3 General Constraints

The constraint generation shown in section 8.2 is a presentation of the one shown in [10]. In truth, there are two kinds of constraints: Syntax based constraints and general constraints. The ones we show in section 8.2 are only the syntax based constraints, the constraints we extract from a spi process. In section 7.1, we discussed the inference method, and mentioned the ALFP formula ψ_{ax} as representing a number of axioms. Even though they are not explicitly presented, ψ_{ax} can be seen as the encoding of a number of general constraints which express properties and operations such as type equality and unification.

As an example, consider type equality. If we allow for a modification of the constraint language syntax, we can express type equality as follows, saying that if two type are equal, their inner types must also be equal:

$$\begin{aligned}\mathbf{Ch}(T_1) = \mathbf{Ch}(T_2) &\Rightarrow T_1 = T_2 \\ \mathbf{Key}(T_1) = \mathbf{Key}(T_2) &\Rightarrow T_1 = T_2 \\ \mathbf{Pair}(x_1 : T_{11}, T_{12}) = \mathbf{Pair}(x_2 : T_{21}, T_{22}) &\Rightarrow T_{11} = T_{21} \wedge T_{12} = T_{22} \\ \mathbf{Ok}(S_1) = \mathbf{Ok}(S_2) &\Rightarrow S_1 = S_2\end{aligned}$$

And unification may be represented by the following constraints:

$$\begin{aligned}T_1 = \mathbf{Ch}(T_2) \wedge T_3 = T_1 &\Rightarrow T_3 = \mathbf{Ch}(T_2) \\ T_1 = \mathbf{Key}(T_2) \wedge T_3 = T_1 &\Rightarrow T_3 = \mathbf{Key}(T_2) \\ T_1 = \mathbf{Pair}(x : T_2, T_3) \wedge T_4 = T_1 &\Rightarrow T_4 = \mathbf{Pair}(x : T_2, T_3) \\ T_1 = \mathbf{Ok}(S) \wedge T_2 = T_1 &\Rightarrow T_2 = \mathbf{Ok}(S)\end{aligned}$$

Constraints from messages: $E \vdash_\Delta M \rightsquigarrow T; \varphi_1$

(Msg Name)
$$\frac{E = E', x : U_T, E''}{E \vdash_\Delta x \rightsquigarrow U_T \wedge \mathrm{wf}(E)}$$

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

(Msg Ok)
$$\frac{}{E \vdash_\Delta \mathbf{ok} \rightsquigarrow U_T; \left(\begin{array}{l} \mathbf{ok} : U_T \wedge \\ (U_T = \mathbf{Un} \Rightarrow \mathrm{wf}(E)) \wedge \\ U_T \stackrel{?}{=} \mathbf{Ok}(R) \wedge \\ (U_T \neq \mathbf{Un} \Rightarrow (E \vdash R) \wedge \mathrm{wf}(E)) \end{array} \right)}$$

(Msg Encrypt)
$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E \vdash_\Delta N \rightsquigarrow U_{T_2}; \varphi_2}{E \vdash_\Delta \{M\}_N \rightsquigarrow U_T; \left(\begin{array}{l} U_T = \mathbf{Un} \wedge U_{T_2} \stackrel{?}{=} \mathbf{Key}(U_{T_1}) \wedge \\ (U_{T_2} = \mathbf{Un} \Rightarrow U_{T_1} = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \end{array} \right)}$$

(Msg Pair)
$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E \vdash_\Delta N \rightsquigarrow U_{T_2}; \varphi_2}{E \vdash_\Delta \mathbf{pair}(M, N) \rightsquigarrow U_T; \left(\begin{array}{l} U_T \stackrel{?}{=} \mathbf{Pair}(x : U_{T_1}, U'_{T_2}) \wedge U'_{T_2} \stackrel{?}{=} U^M_{T_2}(x) \wedge \\ (U_{T_1} = \mathbf{Un} \wedge U_{T_2} = \mathbf{Un} \Rightarrow U_T = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \end{array} \right)}$$

Table 8.1: Constraint generation from messages

Constraints from processes: $E \vdash_\Delta P \rightsquigarrow \varphi_1$

(Proc In) (μ is either ! or nothing)

$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E, x : U_{T_2} \vdash_\Delta P \rightsquigarrow \varphi_2 \quad U_{T_2} \text{ fresh}}{E \vdash_\Delta \mu\mathbf{in}(M, x); P \rightsquigarrow \left(\begin{array}{l} U_{T_1} \stackrel{?}{=} \mathbf{Ch}(U_{T_2}) \wedge \\ (U_{T_1} = \mathbf{Un} \Rightarrow U_{T_2} = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \end{array} \right)}$$

(Proc Out)

$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E \vdash_\Delta N \rightsquigarrow U_{T_2}; \varphi_2}{E \vdash_\Delta \mathbf{out}(M, N) \rightsquigarrow \left(\begin{array}{l} U_{T_1} \stackrel{?}{=} \mathbf{Ch}(U_{T_2}) \wedge \\ (U_{T_1} = \mathbf{Un} \Rightarrow U_{T_2} = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \end{array} \right)}$$

(Proc Res)

$$\frac{E, x : U_T \vdash_\Delta P \rightsquigarrow \varphi_1 \quad U_T \text{ fresh}}{E \vdash_\Delta \mathbf{new}\ x; P \rightsquigarrow x \notin \mathrm{dom}(E) \wedge (\neg(T \text{ generative}) \Rightarrow \mathsf{Fail}) \wedge \varphi_1}$$

(Proc Par)

$$\frac{E, \mathrm{env}(P_2) \vdash_\Delta P_1 \rightsquigarrow \varphi_1 \quad E, \mathrm{env}(P_1) \vdash_\Delta P_2 \rightsquigarrow \varphi_2}{E \vdash_\Delta (P_1 \mid P_2) \rightsquigarrow \varphi_1 \wedge \varphi_2}$$

(Proc Zero)

$$\frac{}{E \vdash_\Delta \mathbf{nil} \rightsquigarrow}$$

(Proc Begin)

$$\frac{}{E \vdash_\Delta \mathbf{begin}\ \ell(M) \rightsquigarrow \mathrm{wf}(E) \wedge \mathrm{fn}(M) \subseteq \mathrm{dom}(E)}$$

(Proc End)

$$\frac{}{E \vdash_\Delta \mathbf{end}\ \ell(M) \rightsquigarrow \ell(M) \leq \mathrm{effects}_{\mathrm{req}}(E) \wedge \mathrm{wf}(E)}$$

(Proc Decrypt)

$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E \vdash_\Delta N \rightsquigarrow U_{T_2}; \varphi_2 \quad E, y : U_{T_3} \vdash_\Delta P \rightsquigarrow \varphi_3 \quad U_{T_3} \text{ fresh}}{E \vdash_\Delta \mathbf{decrypt}\ M\ \mathbf{as}\ (y)_N; P \rightsquigarrow \left(\begin{array}{l} U_{T_1} = \mathbf{Un} \wedge U_{T_2} \stackrel{?}{=} \mathbf{Key}(U_{T_3}) \wedge \\ (U_{T_2} = \mathbf{Un} \Rightarrow U_{T_3} = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \end{array} \right)}$$

(Proc Split)

$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_0}; \varphi_1 \quad E, x : U_{T_1}, y : U_{T_2} \vdash_\Delta P \rightsquigarrow \varphi_2 \quad U_{T_1}, U_{T_2} \text{ fresh}}{E \vdash_\Delta \mathbf{split}\ M\ \mathbf{as}\ (x, y); P \rightsquigarrow \left(\begin{array}{l} U_{T_0} \stackrel{?}{=} \mathbf{Pair}(x : U_{T_1}, U_{T_2}) \wedge \\ (U_{T_0} = \mathbf{Un} \Rightarrow U_{T_1} = \mathbf{Un} \wedge U_{T_2} = \mathbf{Un}) \wedge \\ \varphi_1 \wedge \varphi_2 \end{array} \right)}$$

(Proc Match)

$$\frac{E \vdash_\Delta M \rightsquigarrow U_{T_1}; \varphi_1 \quad E \vdash_\Delta N \rightsquigarrow U_{T_2}; \varphi_2 \quad E, y : U_{T_3} \vdash_\Delta P \rightsquigarrow \varphi_3 \quad U_{T_3}, U_{T_0}, U_{T_4}, x \text{ fresh}}{E \vdash_\Delta \mathbf{match}\ M\ \mathbf{as}\ (N, y); P \rightsquigarrow \left(\begin{array}{l} U_{T_1} \stackrel{?}{=} \mathbf{Pair}(x : U_{T_0}, U_{T_4}) \wedge U_{T_3} \stackrel{?}{=} U_{T_4}(N) \wedge \\ (U_{T_1} = \mathbf{Un} \Rightarrow U_{T_2} = \mathbf{Un} \wedge U_{T_0} = \mathbf{Un} \wedge U_{T_3} = \mathbf{Un} \wedge U_{T_4}) \\ \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \end{array} \right)}$$

Table 8.2: Constraint generation from processes

With the provision that inconsistent types should result in failure. For instance:

$$T_1 = \mathbf{Ch}(T_2) \wedge T_3 = \mathbf{Key}(T_4) \wedge T_1 = T_3 \Rightarrow \mathsf{Fail}$$

8.4 Type Abstraction

Abstraction and application of types are related to the dependent pair types of the type system. The type $\mathbf{Pair}(x : T_1, T_2)$ is the type of a pair message whose left element may be substituted for occurrences of x in T_2. Consider the (Proc Match) rule:

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

$T_2(N)$ is an *application*, and is the type resulting from a substitution of N for x in T_2. The question, which we will try to answer is: How do we find $\mathbf{Pair}(x : T_1, T_2)$, and thus $T_2(N)$? The type system is of course built on the assumption that we know both $\mathbf{Pair}(x : T_1, T_2)$ and the application $T_2(N)$ for the purpose of type checking. But for the purpose of type inference, it is our job to find those types. The task of deducing a dependent pair-type does not appear trivial, however. The primitive we must deduce dependent pair types from is, of course, pair messages. Consider the (Msg Pair) rule:

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

We could easily find a fitting pair type by deducing T and $T'(M)$, and then creating the type $\mathbf{Pair}(x : T, T'(M))$. But to make this a dependent pair type which can be used in an application of (Proc Match), we have to find $\mathbf{Pair}(x : T, T'(x))$. To illustrate this, consider the following process:

$$\begin{array}{l}((\mathbf{begin}\ l(a) \mid \mathbf{begin}\ k(a)) \\ \quad \mid (\mathbf{out}\ (c, \mathbf{pair}(a, \mathbf{ok})) \\ \quad\quad \mid \mathbf{in}\ (c, y); \mathbf{match}\ y\ \mathbf{as}\ (a, z); \\ \quad\quad\ (\mathbf{end}\ l(a) \mid \mathbf{end}\ k(a)) \\ ))\end{array}$$

Deducing the type of $\mathbf{pair}(a, \mathbf{ok})$ we may get $\mathbf{Pair}(x : \mathbf{Un}, \mathbf{Ok}(l(a), k(a)))$. But for the benefit of later applications, we can choose to abstract away either one of the a messages or both of them. Both $\mathbf{Pair}(x : \mathbf{Un}, \mathbf{Ok}(l(x), k(a)))$ and $\mathbf{Pair}(x : \mathbf{Un}, \mathbf{Ok}(l(x), k(x)))$ would be legal in this case.

By saying $U^M_{T_2}(x)$, we express the substitution of x for all occurrences of M, and thus we abstract away M completely. Unfortunately, it is still an open problem whether this kind of total abstraction works as intended, or if it makes the inference method incomplete. If a typeable process exists whose types cannot be inferred by this kind of abstraction, then it makes the method incomplete, and we would have to consider enumerating all possible abstractions, for instance.

But the only place in the constraint rules where we use application is in (Proc Match), and the semantic condition for reduction of a match statement is that N must match the left element of M. Thus, in constructing the dependent type, we believe we may rely on the assumption that N, and the message that was abstracted away are equal.

Chapter 9

Encoding to ALFP

In this chapter, we define and explain the Alternation Free Least Fixpoint Logic (ALFP) from [17] by Nielson, Nielson and Seidl. In section 9.1.1, we explain the syntax of ALFP. In section 9.1.2, we explain the semantics of ALFP. In section 9.1.3, we explain the concept of stratification, and why it is used.

9.1 ALFP

ALFP is interesting because it defines a useable set of logic where:

1. Every expressible formula is satisfiable, and has a unique least solution
2. The least solution to any formula can be found by a simple fixed-point algorithm

This particular description of ALFP is cited, with few changes, from [18].

9.1.1 Syntax

ALFP was introduced by Nielson, Nielson and Seidl in [17] together with the Succinct Solver tool for solving ALFP formulae. ϕ signifies a *precondition* and ψ signifies a *clause*. A formula is a clause (ψ). R ranges over $\mathcal{R}$, which is the set of relation symbols. x ranges over a countable set of variables $\mathcal{X}$:

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

$R(t_1, \ldots, t_k)$ is a *predicate*. Predicates and negated predicates in preconditions (ϕ) are called queries and negated queries, respectively. Any other occurrences are called assertions. An important distinction, as a query asks "is this predicate true?", while an assertion states that "this predicate must be true".

9.1.2 Semantics

Which brings us to the semantics of ALFP. We define the universe $\mathcal{U}$ of atomic values to be non-empty and finite. ρ and σ are interpretations of relation symbols R and free

$(\rho,\sigma) \models R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \neg R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \notin \rho(R)$
$(\rho,\sigma) \models \phi_1 \wedge \phi_2$	iff	$(\rho,\sigma) \models \phi_1$ and $(\rho,\sigma) \models \phi_2$
$(\rho,\sigma) \models \phi_1 \vee \phi_2$	iff	$(\rho,\sigma) \models \phi_1$ or $(\rho,\sigma) \models \phi_2$
$(\rho,\sigma) \models \exists x.\phi$	iff	$(\rho,\sigma[x \mapsto a]) \models \phi$ for some $a \in \mathcal{U}$
$(\rho,\sigma) \models \forall x.\phi$	iff	$(\rho,\sigma[x \mapsto a]) \models \phi$ for all $a \in \mathcal{U}$
$(\rho,\sigma) \models R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \mathbf{true}$		always
$(\rho,\sigma) \models \psi_1 \wedge \psi_2$	iff	$(\rho,\sigma) \models \psi_1$ and $(\rho,\sigma) \models \psi_2$
$(\rho,\sigma) \models \phi \Rightarrow \psi$	iff	$(\rho,\sigma) \models \psi$ whenever $(\rho,\sigma) \models \phi$
$(\rho,\sigma) \models \forall x.\psi$	iff	$(\rho,\sigma[x \mapsto a]) \models \psi$ for all $a \in \mathcal{U}$

Table 9.1: Semantics of preconditions (top) and clauses(bottom).

variables x, respectively. ρ maps relations to sets of k-tuples. σ maps variables to atoms of $\mathcal{U}$. We define two satisfaction relations

$$(\rho,\sigma) \models \phi \quad \text{and} \quad (\rho,\sigma) \models \psi$$

for preconditions and clauses in table 9.1. We write $\rho(R)$ for the set of k-tuples $(a_1,\cdots,a_k)$ from $\mathcal{U}$ associated with the k-ary predicate R. We write $\sigma(x)$ for the atom of $\mathcal{U}$ bound to x, and $\sigma[x \mapsto a]$ for the mapping of σ together with the mapping of x to a.

Variables are bound by the universal and existential quantifiers. Free variables in a formula are viewed as constants from $\mathcal{U}$. Given an interpretation σ_0 of constant symbols in a clause ψ, an interpretation ρ of predicate symbols $\mathcal{R}$ is called a *solution* to ψ, provided $(\rho,\sigma_0) \models \psi$.

We let Δ_{ALFP} be the set of interpretations ρ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$. Δ_{ALFP} is then a complete lattice with relation to the lexicographical ordering $\sqsubseteq$ defined by $\rho_1 \sqsubseteq \rho_2$, if and only if there is some $1 \leq j \leq s$ such that the following properties hold:

- $\rho_1(R) = \rho_2(R)$ for all $R \in \mathcal{R}$ with $\mathit{rank}(R) < j$
- $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$ with $\mathit{rank}(R) = j$
- either $j = s$ or $\rho_1(R) \subset \rho_2(R)$ for at least one $R \in \mathcal{R}$ with $\mathit{rank}(R) = j$

In [17], the authors also prove the following proposition:

Proposition 1. Assume ψ is an ALFP formula and σ_0 is an interpretation of the free variables in ψ. Then the set of all ρ with $(\rho,\sigma_0) \models \psi$ forms a Moore family, i.e., is closed under greatest lower bounds (wrt. $\sqsubseteq$).

Most importantly, this means that for every initial interpretation ρ_0 of predicate symbols, there is a lexicographically least solution ρ of ψ (given a fixed σ_0) such that $\rho_0 \subseteq \rho$. ρ is then called the *optimal* solution of ψ exceeding ρ_0.

When we talk about the "least solution" to some formula, we assume the least solution exceeding $\rho_0 = \emptyset$, unless otherwise specified.

9.1.3 Stratification

In ALFP, formulae must be ordered in s strata, so $\psi = \psi_1 \wedge \ldots \wedge \psi_s$. The rank of a relation corresponds to the stratum in which it may be populated. So only the formula ψ_j may populate relations R if $\mathit{rank}(R) = j$. Assuming this, we can then populate the relations in the order of their ranks. Queries may only be performed on relations with a rank equal to or lower than the one that is being populated, as we do not know the contents of higher ranked relations. Negated queries may only be performed on relations with a rank strictly lower than the one that is being populated, since we have to know the final population of a relation before we can know its negation.

Definition 6. By the syntax, a formula ψ can be expressed as a conjunction of clauses, $\psi = \psi_1 \wedge \ldots \wedge \psi_s$. The set of relation symbols $\mathcal{R}$ is ranked by a function rank:$\mathcal{R} \rightarrow \mathbb{N}$, which satisfies the following for all $1 \leq j \leq s$:

- For every assertion R in ψ_j we have $\mathit{rank}(R) = j$
- For every query R in ψ_j we have $\mathit{rank}(R) \leq j$
- For every negated query R in ψ_j we have $\mathit{rank}(R) < j$

Every index j is a *stratum*

To understand the reasoning for the syntax of ALFP formulae, note first that by excluding the possibility of asserting falsehoods and negations, we exclude the possibility of contradicting formulae, and thus the existence of unsatisfiable formulae. So to find a solution to an ALFP formula, you do not have to deal with the satisfiability problem. There is always a solution. By restricting the existential quantifier and disjunctions to preconditions, we guarantee that there is always a unique least solution. This solution corresponds to the least fixed point of a monotone function, and can be found by applying a fixed point algorithm for each succeeding stratum.

9.2 Axioms

In this section we present samples of some of the axioms that accompany any encoding of formulae from the constraint language.

One good example would be the axioms for populating the Subs relation. The Subs relation is a quaternary relation presented as $\mathsf{Subs}(n', n, m', m)$, with the intended meaning that substituting m' for m in the message denoted by n, results in the message denoted by m'. The following four formulae axiomatise Subs for the OkTerm, Name, TermPair and Enc relations, each of which denote the kind of a message.

$\forall \mathtt{n}.\forall \mathtt{m'}.\forall \mathtt{m}.(\mathsf{OkTerm}(\mathtt{n})) \Rightarrow \mathsf{Subs}(\mathtt{n},\mathtt{n},\mathtt{m'},\mathtt{m})$
$\forall \mathtt{m'}.\forall \mathtt{m}.(\mathsf{Name}(\mathtt{m})) \Rightarrow \mathsf{Subs}(\mathtt{m'},\mathtt{m},\mathtt{m'},\mathtt{m})$
$\forall \mathtt{n}.\forall \mathtt{n1'}.\forall \mathtt{n1}.\forall \mathtt{m'}.\forall \mathtt{m}.\forall \mathtt{n2'}.\forall \mathtt{n2}.\forall \mathtt{n'}.(\mathsf{Subs}(\mathtt{n1'},\mathtt{n1},\mathtt{m'},\mathtt{m}) \wedge \mathsf{Subs}(\mathtt{n2'},\mathtt{n2},\mathtt{m'},\mathtt{m}) \wedge$
$\mathsf{TermPair}(\mathtt{n},\mathtt{n1},\mathtt{n2}) \wedge \mathsf{TermPair}(\mathtt{n'},\mathtt{n1'},\mathtt{n2'})) \Rightarrow \mathsf{Subs}(\mathtt{n'},\mathtt{n},\mathtt{m'},\mathtt{m})$
$\forall \mathtt{n}.\forall \mathtt{n1'}.\forall \mathtt{n1}.\forall \mathtt{m'}.\forall \mathtt{m}.\forall \mathtt{n2'}.\forall \mathtt{n2}.\forall \mathtt{n'}.(\mathsf{Subs}(\mathtt{n1'},\mathtt{n1},\mathtt{m'},\mathtt{m}) \wedge \mathsf{Subs}(\mathtt{n2'},\mathtt{n2},\mathtt{m'},\mathtt{m}) \wedge$
$\mathsf{Enc}(\mathtt{n},\mathtt{n1},\mathtt{n2}) \wedge \mathsf{Enc}(\mathtt{n'},\mathtt{n1'},\mathtt{n2'})) \Rightarrow \mathsf{Subs}(\mathtt{n'},\mathtt{n},\mathtt{m'},\mathtt{m})$

Another one would be the representation of free names, the Fn relation. Fn is a binary relation presented as $\mathsf{Fn}(n, m)$, with the intended meaning that n is a free name of m. n is a name, but m may be either a message or a type. For message terms, the following

three formulae axiomatise Fn for the Name, TermPair and Enc relations. OkTerm is not a part of these, as there are no free names in an **ok** message.

$\forall \mathtt{n}.(\mathsf{Name}(\mathtt{n})) \Rightarrow \mathsf{Fn}(\mathtt{n},\mathtt{n})$
$\forall \mathtt{n}.\forall \mathtt{n1}.\forall \mathtt{n2}.\forall \mathtt{m}.((\mathsf{Fn}(\mathtt{m},\mathtt{n1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{n2})) \wedge \mathsf{TermPair}(\mathtt{n},\mathtt{n1},\mathtt{n2})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{n})$
$\forall \mathtt{n}.\forall \mathtt{n1}.\forall \mathtt{n2}.\forall \mathtt{m}.((\mathsf{Fn}(\mathtt{m},\mathtt{n1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{n2})) \wedge \mathsf{Enc}(\mathtt{n},\mathtt{n1},\mathtt{n2})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{n})$

The first axiom "initiates" the population of Fn by stating that any name is a free name of itself. The two other axioms continue the population by stating, for pair messages and encrypted messages, that if some name is free in either term of the message, then it is free in the message itself. The axioms for free names of types is more of the same.

$\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{m}.(\mathsf{Ch}(\mathtt{t},\mathtt{t1}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{t1})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
$\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{m}.(\mathsf{Key}(\mathtt{t},\mathtt{t1}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{t1})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
$\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{t2}.\forall \mathtt{x}.\forall \mathtt{m}.(\mathsf{Pair}(\mathtt{t},\mathtt{x},\mathtt{t1},\mathtt{t2}) \wedge (\mathsf{Fn}(\mathtt{m},\mathtt{t1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{t2})) \wedge \mathsf{neq}(\mathtt{m},\mathtt{x})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
$\forall \mathtt{t}.\forall \mathtt{s}.\forall \mathtt{m}.(\mathsf{Ok}(\mathtt{t},\mathtt{s}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{s})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
$\forall \mathtt{m}.\forall \mathtt{n}.\forall \mathtt{l}.\forall \mathtt{s}.(\mathsf{Fn}(\mathtt{n},\mathtt{m}) \wedge \mathsf{Effect}(\mathtt{l},\mathtt{m},\mathtt{s})) \Rightarrow \mathsf{Fn}(\mathtt{n},\mathtt{s})$

Except for the axiom for the Pair relation. The neq relation simply indicates here that m and x may not be equal. If they are, then the name is bound.

9.3 Encodings

9.3.1 Message Terms

To reason about messages in the ALFP formulae, we need to know which names are messages, and which kinds of messages they are. Below, we show how messages are encoded. Lowercase characters denote proper names of their uppercase counterparts. n is the proper name of the message we encode. This works as a way of representing the abstract syntax tree of a message in ALFP.

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

If $\mathtt{messages}(P)$ returns all messages of a process, then we add the conjunction

$$\bigwedge_{M \in \mathtt{messages}(P)} [\![M]\!]$$

to the formula generated from P.

9.3.2 Type Constraints

Type equalities and possible type equalities are simply encoded to corresponding relations. So for instance,

$$\begin{aligned}
[\![U_T = \mathbf{Key}(U_{T_1})]\!] &= \mathsf{Key}(t, t_1) \\
[\![U_T = \mathbf{Ch}(U_{T_1})]\!] &= \mathsf{Ch}(t, t_1) \\
[\![U_T = \mathbf{Pair}(x : U_{T_1}, U_{T_2})]\!] &= \mathsf{Pair}(t, x, t_1, t_2) \\
[\![U_T = \mathbf{Un}]\!] &= \mathsf{Un}(t) \\
[\![U_T = \mathbf{Ok}(R_S)]\!] &= \mathsf{Ok}(s)
\end{aligned}$$

are the encodings for type equalities. Encodings for possible equalities are nearly identical, and can be obtained by replacing $\overset{?}{=}$ for $=$ on the left hand side and postfixing relation names with Q on the right hand side.

Chapter 10

Correctness Theorems

In section 7.1, we briefly touched on two correctness theorems stated in [10], asserting that the inference method is both sound and complete. We now present the formal statements of those theorems. A few definitions are needed to understand this. Some definitions have already been stated elsewhere, such as Δ (chapter 7), σ and ρ (section 9.1.2). We will present a few more definitions along with the theorems.

Definition 7. Iff $[x \mapsto y] \in \sigma$ for a mapping function σ, some variable x and some atom y, then $x \in \texttt{dom}(\sigma)$. Likewise, $\texttt{dom}(E)$ will capture the domain of environment E (see appendix A.3).

Definition 8. If some statement St can be derived from $\mathcal{M}$, we say that $\mathcal{M} \models St$.

Definition 9. Assume $E \vdash \diamond$ and let $\mathcal{M} = (\sigma, \rho)$ be a first-order structure. We say that $\mathcal{M} \prec E$ if $\texttt{dom}(\sigma) = \texttt{dom}(E)$ and we have $\mathcal{M} \models (x, t) \in \mathsf{Type}$ iff $E(x) = T$. If also for a given Δ 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 1. *If $E \vdash_\Delta P \rightsquigarrow \psi$ and $E \vdash_\Delta P$ where $\texttt{dom}(\Delta) = \texttt{Var}(\psi)$, then there exists a failure-free model $\mathcal{M}$ where $\mathcal{M} \prec (E, \Delta)$ and $\mathcal{M} \models [\![\psi \wedge \psi_{Ax}]\!]$.*

Definition 10. Let Δ 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 2. *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 $[\![\psi \wedge \psi_{Ax}]\!]$, there exists a type/effect-assignment Δ such that $E \vdash_\Delta P$, $\mathcal{M} \prec (E, \Delta)$ and $\mathcal{M} \models \Delta$.*

Chapter 11

Issues and Solutions

Hans Hüttel presents in [10] an attempt to achieve type inference in a type system with both opponent types, dependent types and correspondence assertions, by encoding constraints to ALFP and feeding them to the Succinct Solver. We have outlined this method in previous chapters, and we have, in turn, made the first attempt to implement this method of type inference. Considering the novelty of this approach, we were bound to find some technical issues in [10], which we describe in this chapter together with possible solutions.

11.1 Possible Types

In [10], possible types are included in the constraint rules seen in table 8.1 and 8.2, and encoded to the *may* relations as seen in section 9.3. Apart from that however, no axioms show how to deal with those relations. But this is more an oversight than an actual issue, and finding the necessary axioms is not difficult once the intent behind the *may* relations are clear: The **Un** relations must be populated in a stratum prior to the other type relations. The *may* relations should then be used to populate the real type relation whenever a type variable is not **Un**. This results in the possible type axioms in section 9.2.

11.2 Abstraction and Application

The relations Abs and App are intended to represent abstractions ($T^M(x)$) and applications ($U(N)$). We found that the exact implementations of Abs and App, as described in [10] do not appear workable. The actual abstraction and application of types, for instance, should yield new types. Consider the types $U_{T_2} = U^M_{T_1}(x)$ and $U_{T_3} = U_{T_2}(N)$, and suppose we have $U_{T_1} = \mathbf{Ch}(U_T)$. U_{T_2} may be an abstraction of U_{T_1}, but they are also two distinct types, and if we want to propagate the abstraction into U_{T_1} to describe U_{T_2} properly, we need to create new type variables to describe the new type tree and, ultimately, any abstractions on effects in the leaves of that tree. The same goes for applications, as seen in U_{T_3}. U_{T_3} is an application of U_{T_2}, but both are distinct types, and to describe the type of U_{T_3} we need a set of variables for the new tree, with applications on effect variables at the leaves. While it is possible that some form of function application in ALFP would allow us to generate new type variables through the Succinct Solver it

would be somewhat involved, and the current set of axioms and encodings would need to be modified to allow for it.

Unfortunately, we did not get far enough to implement and test a possible solution for this, but we do have one that may work. Instead of implementing this through ALFP, it may be possible to handle the abstraction and application of types in the constraints to some extent, before we start encoding the ALFP formula.

If we let U_F and S_F denote *fresh* variables, we can define a reduction $\stackrel{p}{\Rightarrow}$ on a constraint formula ψ:

$$\begin{aligned}
U_{T_2} \stackrel{?}{=} U^M_{T_1}(x) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Ch}(U_{T_3}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_F = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Ch}(U_F) \wedge U_F \stackrel{?}{=} U^M_{T_3}(x) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U^M_{T_1}(x) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Key}(U_{T_3}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_F = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Key}(U_F) \wedge U_F \stackrel{?}{=} U^M_{T_3}(x) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U^M_{T_1}(x) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Pair}(x_1 : U_{T_3}, U_{T_4}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_{F_1} = \mathbf{Un}) \wedge (U_{T_4} = \mathbf{Un} \Rightarrow U_{F_2} = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Pair}(x_1 : U_{F_1}, U_{F_2}) \wedge U_{F_1} \stackrel{?}{=} U^M_{T_3}(x) \wedge \\
&\quad U_{F_2} \stackrel{?}{=} U^M_{T_4}(x) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U^M_{T_1}(x) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Ok}(S) \wedge \psi_1 &\stackrel{p}{\Rightarrow} U_{T_2} \stackrel{?}{=} \mathbf{Ok}(S_F) \wedge S_F \stackrel{?}{=} S^M(x) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U_{T_1}(N) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Ch}(U_{T_3}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_F = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Ch}(U_F) \wedge U_F \stackrel{?}{=} U_{T_3}(N) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U_{T_1}(N) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Key}(U_{T_3}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_F = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Key}(U_F) \wedge U_F \stackrel{?}{=} U_{T_3}(N) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U_{T_1}(N) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Pair}(x : U_{T_3}, U_{T_4}) \wedge \psi_1 &\stackrel{p}{\Rightarrow} (U_{T_3} = \mathbf{Un} \Rightarrow U_{F_1} = \mathbf{Un}) \wedge (U_{T_4} = \mathbf{Un} \Rightarrow U_{F_2} = \mathbf{Un}) \wedge \\
&\quad U_{T_2} \stackrel{?}{=} \mathbf{Pair}(x : U_{F_1}, U_{F_2}) \wedge U_{F_1} \stackrel{?}{=} U_{T_3}(N) \wedge \\
&\quad U_{F_2} \stackrel{?}{=} U_{T_4}(N) \wedge \psi_1 \\
U_{T_2} \stackrel{?}{=} U_{T_1}(N) \wedge U_{T_1} \stackrel{?}{=} \mathbf{Ok}(S) \wedge \psi_1 &\stackrel{p}{\Rightarrow} U_{T_2} \stackrel{?}{=} \mathbf{Ok}(S_F) \wedge S_F \stackrel{?}{=} S(M) \wedge \psi_1
\end{aligned}$$

In the closure of $\stackrel{p}{\Rightarrow}$, this will let both abstractions and applications become "real" types (one of $\mathbf{Ch}(U_T)$, $\mathbf{Key}(U_T)$, $\mathbf{Pair}(x : U_{T_1}, U_{T_2})$, $\mathbf{Un}$ or $\mathbf{Ok}(S)$), while propagating them to become abstractions and applications directly on effect variables. This will hopefully make the handling of these concepts simpler in the encoding to ALFP.

As an example, if we have that $U_{T_2} \stackrel{?}{=} U^M_{T_1}(x)$, $U_{T_1} \stackrel{?}{=} \mathbf{Ch}(U_{T_3})$ and $U_{T_3} \stackrel{?}{=} \mathbf{Ok}(S)$ in the constraints, we get:

$$\begin{aligned}
U_{T_2} &\stackrel{?}{=} U^M_{T_1}(x) \\
U_{T_2} &\stackrel{?}{=} \mathbf{Ch}(U_{T_4}) \\
U_{T_4} &\stackrel{?}{=} \mathbf{Ok}(S_1) \\
S_1 &\stackrel{?}{=} S^M(x)
\end{aligned}$$

Where U_{T_4}, U_{T_5} and S_1 are fresh. Likewise, if $U_{T_6} = U_{T_2}(N)$, the application is propa-

gated:

$$U_{T_5} \stackrel{?}{=} U_{T_2}(N)$$
$$U_{T_5} \stackrel{?}{=} \mathbf{Ch}(U_{T_6})$$
$$U_{T_6} \stackrel{?}{=} \mathbf{Ok}(S_2)$$
$$S_2 \stackrel{?}{=} S_1(N)$$

This can all be performed in the space of possible types, which will only become real types if they are not **Un**.

11.3 Relation Size Explosions

During the initial testing of the inference method, it was discovered that the Subs relation would explode and become, apparently, intractably large. This happened because of the Subs axioms for the OkTerm and Un relations, presented in [10] as:

$$\forall n.\forall n'.\forall m.\forall m'.\mathsf{OkTerm}(n) \Rightarrow \mathsf{Subs}(n', n, m', m)$$
$$\forall t.\forall t'.\forall m.\forall m'.\mathsf{Un}(t) \wedge \mathsf{Un}(t') \Rightarrow \mathsf{Subs}(t', t, m', m)$$

If a is the size of the OkTerm relation, and b is the number of names in the ALFP formula, then the Subs axiom for the OkTerm relation would contribute with $a * b^3$ predicates in Subs. Fortunately, the OkTerm axiom turned out to be an error. As no substitution can make an **ok** term change, we can say $\forall n.\forall m.\forall m'.\mathsf{OkTerm}(n) \Rightarrow \mathsf{Subs}(n, n, m', m)$ instead, and reduce the contribution of the axiom to $a * b^2$.

If we say a is the size of the Un relation, and b is the number of names in the ALFP formula, then The Subs axiom for the Un relation would contribute with $a^2 * b^2$ predicates in Subs. Fortunately, making the changes discussed in section 11.2, removes the only axioms requiring the Subs relation for types, and thus remove the Subs axiom for Un.

Chapter 12

Implementing Type Inference

As the type checker of the first part of the report, the implementation of type inference was also made in Objective Caml. This has even allowed some reuse of code from the type checker. The main parser and the Succinct Solver has only been slightly modified to strip out type annotations, and to make minor adaptations.

As with the type checker implementation, we show here the dependency hierarchy, and then go into detail with some specific implementation details.

12.1 Hierarchy

Figure 12.1 shows the hierarchy of module dependencies in the type inference implementation. Following is a short description of each module.

- `spideducer.ml`: Main function of the implementation. It uses the other modules to open and parse the input, alpha convert the process, generate constraints, encode a formula, and feed that formula to the Succinct Solver.
- `spitree.ml`: Syntax tree structure for the spi calculus. Unmodified copy of the same file of the type checker.
- `spilexer.mll`, `spiparser.mly`: Lexer and parser for the spi calculus. Parses to the structure of `spitree.ml`. Changes from the type checker version includes: Removal of type annotations from the syntax. Every type is designated as "Unknown".
- `aconv.ml`: Alpha conversion. Unmodified copy of the same file of the type checker.
- `ssolver/`: Succinct Solver wrapper. Slightly modified from the type checker for adaptation purposes.
- `constraints.ml`: Contains the data types for the constraint language.
- `constraintgen.ml`: Contains constraint generation functions. Will generate a set of constraints from a process.
- `axioms.ml`: Contains the axiom formulae written in the ALFP data type.
- `latex_axioms.ml`: While it may not be hard to write axioms in `axioms.ml`, reading, understanding and maintaining an overview of them in that form would be a

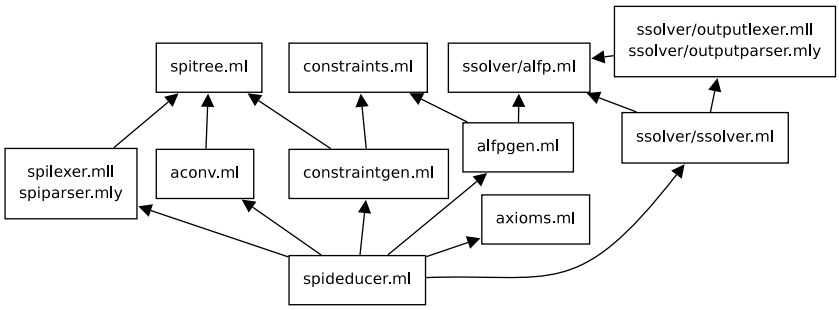

Figure 12.1: Module hierarchy of the type inference implementation. An arrow indicates a dependency.

thankless task. `latex_axioms.ml` will print the axioms in a more readable LaTeX form in the file `axioms.tex`, which can be a tremendous help for readability.

- `alfpgen.ml`: Contains encoding functions from the constraint language to ALFP.

Unfortunately, because of time constraints, and the fact that this is a first attempt at an implementation of this particular method, we did not complete the implementation. But the ground work is complete. Adding, removing and modifying the axioms in `axioms.ml` is close to trivial. The constraint generation of `constraintgen.ml` is equally well-structured, and should not be difficult to modify either. The structure of the encoding definitions in `alfpgen.ml` suffers slightly from the difference in syntax and expressive power between the constraints and ALFP, and may need a bit of extra work to attain the same level of structure.

12.2 Writing Up Axioms

Hüttel presents a great number of axioms in [10], written as ALFP formulae. These axioms must be integrated in the implementation in a sensible way. Preferably one that allows us a readable overview of the axioms.

We chose simply to write the axioms directly in the data type provided by `ssolver/alfp.ml`, with a few helper functions to make the actual writing bearable. This way, we benefit directly from the Objective Caml type checker, which will tell us if a formula is not well-formed. We could have allowed writing the axioms in a more bearable syntax, but that would require us to write our own parser with error reporting facilities for that syntax.

Small functions allows us to write frequently used elements as lists, such as `aquantify`, which makes it possible to write the universal quantifiers of a formula as a list:

```
(* Prefix a clause with a number of universal quantifiers *)
let rec aquantify qlist clause =
  match qlist with
    | [] -> clause
    | (hd::tl) -> Forall(hd, aquantify tl clause)
```

Thus, the formula for substitution on pairs:
$\forall$n.$\forall$n1'.$\forall$n1.$\forall$m'.$\forall$m.$\forall$n2'.$\forall$n2.$\forall$n'.
(Subs(n1',n1,m',m)$\wedge$Subs(n2',n2,m',m)$\wedge$TermPair(n,n1,n2)$\wedge$TermPair(n',n1',n2')) $\Rightarrow$
Subs(n',n,m',m)

Can be written in the code as:

```
let ax_tpair_subs = aquantify ["n"; "n1'"; "n1"; "m'"; "m"; "n2'"; "n2"; "n'"]
  (Leadsto(preand_of_list
             [PPredicate(subsrel, varlist ["n1'"; "n1"; "m'"; "m"]);
              PPredicate(subsrel, varlist ["n2'"; "n2"; "m'"; "m"]);
              PPredicate(tpairrel, varlist ["n"; "n1"; "n2"]);
              PPredicate(tpairrel, varlist ["n'"; "n1'"; "n2'"])],
           CPredicate(subsrel, varlist ["n'"; "n"; "m'"; "m"])
          )
  )
```

It is still somewhat verbose, but it is a bearable way of writing the formulae. All the formulae are concatenated into the list `axioms`, which may contain both clauses and comment strings. This way, we can precede a conjunction of formulae in the list with a comment which may be included in any printout of the formulae, and omitted when we feed it to the Succinct Solver.

This brings us to the readability of the axioms. Clearly, maintaining an overview of the axioms from the code would be difficult. For this purpose, we can present `latex_axioms.ml`, and a number of `latex_of_*` functions in `ssolver/alfp.ml`. Executing `latex_axioms.ml` will result in the file `axioms.tex`, which can be compiled by LaTeX to show a much more readable overview of the axioms, together with comments.

12.3 Encoding Constraint Formulae

The encoding of constraint formulae proved to be one of the more non-intuitive parts of the implementation. Part of this is because the constraint language is more expressive than ALFP logic. There are expressions which are possible in the given constraint language, but have no equivalent in ALFP logic, such as $\varphi_1 \Rightarrow (\varphi_2 \vee \varphi_3)$ (disjunction in a clause), or $\varphi_1 \Rightarrow U_T \neq \mathbf{Un}$ (negation in a clause). But if the constraint generation is good, we should never encounter such constraints. If we do, it is either a bug in the implementation, or a problem with the constraint generation seen earlier in tables 8.1 and 8.2. This justifies the use of a `UnsupportedByALFP` exception in certain places in `alfpgen.ml`. Fortunately, we have not encountered a case that raises such an exception yet.

Another part of the reason that this part of the implementation seems non-intuitive, is because of the need to encode such elements as messages and name-inclusion for environments, which we decided to separate from the rest of the encoding. This is embodied in the function `formula_of_env`, which encodes every name-inclusion of an environment, and the function `alfp_of_msg`, which encodes the kind of a message (see section 9.3). In the process of encoding the main formula, we collect each environment and each message. Afterwards, we pass each environment and each message to `formula_of_env` and `alfp_of_msg`, respectively, and add the resulting formulae to the conjunction of the main formula.

Chapter 13

Conclusion and Future Work

In this report, we set out to develop a type checker for the type system presented in [4] by Fournet, Gordon and Maffeis, and an implementation of the type inference method developed in [10] by Hüttel. The type checker is in a functional state, but should be considered a beta-version and may still contain significant bugs. The type inference implementation was not completed, in part because of a relatively late start on that project, and in part because of the need to address some technical issues with the method itself. But the ground work has been laid, and completing the implementation is mostly a question of modifying constraint generation and axioms in the code.

Further work on the type checker would involve some more thorough testing. We also believe the utility of the checker could be greatly improved by the implementation of syntactic sugar in the form of shorthand expressions akin to those already found in [4]. Furthermore, we suspect we created some unnecessary redundancy in the type annotations by introducing the extra annotation on **match** statements. We also believe there may be other annotation redundancies in the type system. As seen in the reviewer example in section 5.2, types have a tendency to become large and complex, so finding and removing any such redundancies could greatly reduce the work involved in typing a process.

As we mention previously, the implementation of type inference is not finished. In theory, what is missing is not much, but in practice it depends on whether or not we find any more technical issues with the method as described in [10], and how involved the solutions would be. Future work on the implementation would involve completing it. As it stands, the status is as follows:

- **Constraint generation:** Implemented, with the exception of the modification to handle abstraction and application as described in section 11.2.
- **Constraint encoding:** Implemented.
- **Axioms:** The axioms governing types have been implemented, but the axioms governing effects have not, so far. Most of that work amounts to simply writing the axioms in from [10], although we believe the axioms for effects could be subject to minor modifications to reflect the modification of section 11.2.

A point we would like to highlight is that in general, the most significant issues we have encountered during this project were in some way related to dependent pair types (the **match** annotation, section 3.3. Abstraction and application, section 8.4 and 11.2).

We imagine a focus for further theoretical work could be a more thorough investigation of how dependent pair types affect the task of type inference, and perhaps what impact they have on type and effect systems in general.

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, New York, NY, USA, 1997. ACM.

[2] Morten Dahl. Security Made Easy - Proving Security using Type Inference. *Master's thesis, not published, Aalborg University*, 2008.

[3] Danny Dolev and Andrew C. Yao. On the security of public key protocols. In *SFCS '81: Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[4] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A Type Discipline for Authorization Policies. *ACM Trans. Program. Lang. Syst.*, 29(5):25, 2007.

[5] Jens Kristian Søgaard Andersen. The Implementation of a Type Checker for a Spi-Calculus with Time Stamps. *Master's thesis, not published, Aalborg University*, 2005.

[6] Andrew D. Gordon and Alan Jeffrey. Typing One-to-One and One-to-Many Correspondences in Security Protocols. In *ISSS*, pages 263–282, 2002.

[7] Andrew D. Gordon and Alan Jeffrey. Authenticity by Typing for Security Protocols. *J. Comput. Secur.*, 11(4):451–519, 2003.

[8] Andrew D. Gordon and Alan Jeffrey. Typing Correspondence Assertions for Communication Protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003.

[9] Andrew D. Gordon and Alan Jeffrey. Secrecy Despite Compromise: Types, Cryptography, and the Pi-calculus. In *CONCUR 2005 - Concurrency Theory*, pages 186–201, London, UK, 2005. Springer-Verlag.

[10] Hans Hüttel. Computing effects for correspondence assertions. *FCS'09*, 2009.

[11] Daisuke Kikuchi and Naoki Kobayashi. Type-Based Verification of Correspondence Assertions for Communication Protocols. *Programming Languages and Systems*, pages 191–205, 2007.

[12] Daisuke Kikuchi and Naoki Kobayashi. Type-Based Automated Verification of Authenticity in Cryptographic Protocols. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 222–236, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.11*, 2008.

[14] Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. Code-Carrying Authorization. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 563–579, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[16] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

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

[18] Palle Raabjerg. Inferring Correspondence Types With Facts In A Spi-Calculus With Horn Clauses. *DAT5 project report, not published, Aalborg University*, 2008.

[19] Thomas Y. C. Woo and Simon S. Lam. A Semantic Model for Authentication Protocols. In *SP '93: Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 178, Washington, DC, USA, 1993. IEEE Computer Society.

Appendix A

Full Function Definitions

A.1 Free Names (fn)

Find the free names of a process.

$$
\begin{aligned}
\texttt{fn}(\textbf{ok}) &= \emptyset \\
\texttt{fn}(x) &= \{x\} \\
\texttt{fn}(\textbf{pair}(M, N)) &= \texttt{fn}(M) \cup \texttt{fn}(N) \\
\texttt{fn}(\{M\}_N) &= \texttt{fn}(M) \cup \texttt{fn}(N) \\
\texttt{fn}(\textbf{nil}) &= \emptyset \\
\texttt{fn}(\textbf{in}(M, x); P) &= \texttt{fn}(M) \cup \texttt{fn}(P) \backslash \{x\} \\
\texttt{fn}(!\textbf{in}(M, x); P) &= \texttt{fn}(\textbf{in}(M, x); P) \\
\texttt{fn}(\textbf{out}(M, N)) &= \texttt{fn}(M) \cup \texttt{fn}(N) \\
\texttt{fn}(\textbf{new}\ x : T; P) &= \texttt{fn}(T) \cup \texttt{fn}(P) \backslash \{x\} \\
\texttt{fn}(\textbf{decrypt}\ M\ \textbf{as}\ (y : T)_N; P) &= \texttt{fn}(M) \cup \texttt{fn}(T) \cup \texttt{fn}(N) \cup \texttt{fn}(P) \backslash \{y\} \\
\texttt{fn}(\textbf{match}\ M : T_M\ \textbf{as}\ (N, y : T); P) &= \texttt{fn}(M) \cup \texttt{fn}(T_M) \cup \texttt{fn}(N) \cup \texttt{fn}(T) \cup \texttt{fn}(P) \backslash \{y\} \\
\texttt{fn}(\textbf{split}\ M\ \textbf{as}\ (x : T, y : U); P) &= \texttt{fn}(M) \cup \texttt{fn}(T) \cup \texttt{fn}(U) \backslash \{x\} \cup \texttt{fn}(P) \backslash \{x, y\} \\
\texttt{fn}(\textbf{expect}\ F) &= \texttt{fn}(F) \\
\texttt{fn}(\textbf{Un}) &= \emptyset \\
\texttt{fn}(\textbf{Ok}(S)) &= \texttt{fn}(S) \\
\texttt{fn}(\textbf{Pair}(x : T, U)) &= \texttt{fn}(T) \cup \texttt{fn}(U) \backslash \{x\} \\
\texttt{fn}(\textbf{Key}(T)) &= \texttt{fn}(T) \\
\texttt{fn}(\textbf{Ch}(T)) &= \texttt{fn}(T) \\
\texttt{fn}(X) &= \emptyset \\
\texttt{fn}(p(u_1, \ldots, u_n)) &= \bigcup_{k=1}^{n} \texttt{fn}(u_k) \\
\texttt{fn}(L :- L_1, \ldots, L_n) &= \texttt{fn}(L) \cup \left(\bigcup_{k=1}^{n} \texttt{fn}(L_k) \right) \\
\texttt{fn}(\{C_1, \ldots, C_n\}) &= \bigcup_{k=1}^{n} \texttt{fn}(C_k)
\end{aligned}
$$

A.2 Bound Names (bn)

Find the bound names of a process.

$$
\begin{aligned}
\mathtt{bn}(\mathbf{nil}) &= \emptyset \\
\mathtt{bn}(\mathbf{in}(M, x); P) &= \{x\} \cup \mathtt{bn}(P) \\
\mathtt{bn}(!\mathbf{in}(M, x); P) &= \mathtt{bn}(\mathbf{in}(M, x); P) \\
\mathtt{bn}(\mathbf{out}(M, N)) &= \emptyset \\
\mathtt{bn}(\mathbf{new}\ x : T; P) &= \{x\} \cup \mathtt{bn}(T) \cup \mathtt{bn}(P) \\
\mathtt{bn}(\mathbf{decrypt}\ M\ \mathbf{as}\ (y : T)_N; P) &= \{y\} \cup \mathtt{bn}(T) \cup \mathtt{bn}(P) \\
\mathtt{bn}(\mathbf{match}\ M : T_M\ \mathbf{as}\ (N, y : T); P) &= \mathtt{bn}(T_M) \cup \{y\} \cup \mathtt{bn}(T) \cup \mathtt{bn}(P) \\
\mathtt{bn}(\mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P) &= \mathtt{bn}(T) \cup \mathtt{bn}(U) \cup \{x, y\} \cup \mathtt{bn}(P) \\
\mathtt{bn}(\mathbf{expect}\ F) &= \emptyset \\
\mathtt{bn}(\mathbf{Un}) &= \emptyset \\
\mathtt{bn}(\mathbf{Ok}(S)) &= \emptyset \\
\mathtt{bn}(\mathbf{Pair}(x : T, U)) &= \{x\} \cup \mathtt{bn}(T) \cup \mathtt{bn}(U) \\
\mathtt{bn}(\mathbf{Key}(T)) &= \mathtt{bn}(T) \\
\mathtt{bn}(\mathbf{Ch}(T)) &= \mathtt{bn}(T)
\end{aligned}
$$

A.3 Domain (dom)

Capture the domain of an environment.

$$
\begin{aligned}
\mathtt{dom}(E, x : T) &= \mathtt{dom}(E) \cup \{x\} \\
\mathtt{dom}(E, \mathbf{begin}\ \ell(M)) &= \mathtt{dom}(E) \\
\mathtt{dom}(x : T) &= \{x\} \\
\mathtt{dom}(\mathbf{begin}\ \ell(M)) &= \emptyset
\end{aligned}
$$

A.4 Substitution

Substitute a message for a name.

$$
\begin{aligned}
\mathbf{ok}\{N/x\} &= \mathbf{ok} \\
x\{N/x\} &= N \\
x\{N/y\} &= x \\
\mathbf{pair}(M_1, M_2)\{N/x\} &= \mathbf{pair}(M_1\{N/x\}, M_2\{N/x\}) \\
\{M_1\}_{M_2}\{N/x\} &= \{M_1\{N/x\}\}_{M_2\{N/x\}} \\
\mathbf{nil}\{N/x\} &= \mathbf{nil} \\
(\mathbf{in}(M, x); P)\{N/x\} &= \mathbf{in}(M, x); P \\
(\mathbf{in}(M, x); P)\{N/y\} &= \mathbf{in}(M, x); P\{N/y\} \\
\mathbf{out}(M_1, M_2)\{N/x\} &= \mathbf{out}(M_1\{N/x\}, M_2\{N/x\}) \\
(\mathbf{new}\ x : T; P)\{N/x\} &= \mathbf{new}\ x : T\{N/x\}; P \\
(\mathbf{new}\ x : T; P)\{N/y\} &= \mathbf{new}\ x : T\{N/x\}; P\{N/x\} \\
(\mathbf{decrypt}\ M_1\ \mathbf{as}\ (y : T)_{M_2}; P)\{N/y\} &= \mathbf{decrypt}\ M_1\{N/y\}\ \mathbf{as}\ (y : T\{N/y\})_{M_2\{N/y\}}; P \\
(\mathbf{decrypt}\ M_1\ \mathbf{as}\ (y : T)_{M_2}; P)\{N/x\} &= \mathbf{decrypt}\ M_1\{N/x\}\ \mathbf{as}\ (y : T\{N/x\})_{M_2\{N/x\}}; P\{N/x\} \\
(\mathbf{match}\ M_1 : T_{M_1}\ \mathbf{as}\ (M_2, y : T); P)\{N/y\} &= \mathbf{match}\ M_1\{N/y\} : M_2\{N/y\}\ \mathbf{as}\ (T_{M_1}\{N/y\}, y : T\{N/y\}); P \\
(\mathbf{match}\ M_1 : T_{M_1}\ \mathbf{as}\ (M_2, y : T); P)\{N/x\} &= \mathbf{match}\ M_1\{N/x\} : M_2\{N/x\}\ \mathbf{as}\ (T_{M_1}\{N/x\}, y : T\{N/x\}); P\{N/
\\
(\mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P)\{N/x\} &= \mathbf{split}\ M\{N/x\}\ \mathbf{as}\ (x : T\{N/x\}, y : U); P \\
(\mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P)\{N/y\} &= \mathbf{split}\ M\{N/y\}\ \mathbf{as}\ (x : T\{N/y\}, y : U\{N/y\}); P \\
(\mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P)\{N/z\} &= \mathbf{split}\ M\{N/z\}\ \mathbf{as}\ (x : T\{N/z\}, y : U\{N/z\}); P\{N/z\} \\
(\mathbf{expect}\ F)\{N/x\} &= \mathbf{expect}\ F\{N/x\} \\
\mathbf{Un}\{N/x\} &= \mathbf{Un} \\
\mathbf{Ok}(S)\{N/x\} &= \mathbf{Ok}(S\{N/x\}) \\
\mathbf{Pair}(x : T, U)\{N/x\} &= \mathbf{Pair}(x : T\{N/x\}, U) \\
\mathbf{Pair}(x : T, U)\{N/y\} &= \mathbf{Pair}(x : T\{N/y\}, U\{N/y\}) \\
\mathbf{Key}(T)\{N/x\} &= \mathbf{Key}(T\{N/x\}) \\
\mathbf{Ch}(T)\{N/x\} &= \mathbf{Ch}(T\{N/x\}) \\
X\{N/y\} &= X \\
(p(u_1, \ldots, u_n))\{N/x\} &= p(u_1\{N/x\}, \ldots, u_n\{N/x\}) \\
(L : -L_1, \ldots, L_n)\{N/x\} &= L : -L_1\{N/x\}, \ldots, L_n\{N/x\} \\
\{C_1, \ldots, C_n\}\{N/x\} &= \{C_1\{N/x\}, \ldots, C_n\{N/x\}\}
\end{aligned}
$$

Appendix B

Type Checker Correctness Proofs

Lemma 1. `env_judgment(`env`)` $\iff$ $env \vdash \diamond$

Proof. Proof by induction in the structure of env.

- **Base case:** `env_judgment(`$\emptyset$`)` $\iff$ $E \vdash \emptyset$, line 1.2
  This call is always **true**. The corresponding judgment states that $\emptyset \vdash \diamond$ is always good:

  (Env $\emptyset$)

  $$\frac{}{\emptyset \vdash \diamond}$$

- `env_judgment(`E, x:T`)` $\iff$ $E, x : T \vdash \diamond$, line 1.4
  This call is **true** if `env_judgment(`E`)` $\wedge$ (`fn(`T`)` $\subseteq$ `dom(`E`)`) $\wedge$ ($x \notin$ `dom(`E`)`).
  The inclusion conditions are identical to the ones in (Env x),
  by the induction hypothesis, `env_judgment(`E`)` $\iff$ $E \vdash \diamond$,
  thus it holds that `env_judgment(`E, x:T`)` $\iff$ $E, x : T \vdash \diamond$

  (Env x)
  $$\frac{E \vdash \diamond \quad \texttt{fn(}T\texttt{)} \subseteq \texttt{dom(}E\texttt{)} \quad x \notin \texttt{dom(}E\texttt{)}}{E, x : T \vdash \diamond}$$

- `env_judgment(`E, $\ell(M)$`)` $\iff$ $E, \ell(M) \vdash \diamond$, line 1.6
  This call is **true** if `env_judgment(`E`)` $\wedge$ (`fn(`$\ell(M)$`)` $\subseteq$ `dom(`E`)`).
  The inclusion condition is identical to the one in (Env $\ell(M)$),
  by the induction hypothesis, `env_judgment(`E`)` $\iff$ $E \vdash \diamond$,
  thus it holds that `env_judgment(`E, $\ell(M)$`)` $\iff$ $E, \ell(M) \vdash \diamond$

  (Env $\ell(M)$)
  $$\frac{E \vdash \diamond \quad (\texttt{fn(}\ell(M)\texttt{)} \subseteq \texttt{dom(}E\texttt{)})}{E, \ell(M) \vdash \diamond}$$

□

Lemma 2. `msg_judgment(`E, msg, typ`)` $\Rightarrow$ $E \vdash msg : typ$

Proof. Proof by induction in the structure of msg. From lemma 1, we know that $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$.

- $msg = \mathbf{pair}(M, N)$, line 2.2
  - When $typ = \mathbf{Pair}(x : T, U)$, line 2.4
    By the induction hypothesis
    $\texttt{msg\_judgment}(E,\ M,\ T) \Rightarrow E \vdash M : T$ and
    $\texttt{msg\_judgment}(E,\ N,\ U\{M/x\}) \Rightarrow E \vdash N : U\{M/x\}$ hold,
    thus by the (Msg Pair) rule,
    $\texttt{msg\_judgment}(E,\ \mathbf{pair}(M, N),\ \mathbf{Pair}(x : T, U)) \Rightarrow E \vdash \mathbf{pair}(M, N) : \mathbf{Pair}(x : T, U)$
    holds.
  - When $typ = \mathbf{Un}$, line 2.6
    By the induction hypothesis
    $\texttt{msg\_judgment}(E,\ M,\ \mathbf{Un}) \Rightarrow E \vdash M : \mathbf{Un}$ and
    $\texttt{msg\_judgment}(E,\ N,\ \mathbf{Un}) \Rightarrow E \vdash N : \mathbf{Un}$ hold,
    thus by the (Msg Pair Un) rule,
    $\texttt{msg\_judgment}(E,\ \mathbf{pair}(M, N),\ \mathbf{Un}) \Rightarrow E \vdash \mathbf{pair}(M, N) : \mathbf{Un}$ holds.
- $msg = \{M\}_{Name(x)}$, line 2.10
  - When $typ = \mathbf{Un}$ and $E(x) = \mathbf{Key}(T)$, line 2.13
    By the induction hypothesis
    $\texttt{msg\_judgment}(E,\ M,\ T) \Rightarrow E \vdash M : T$ and
    $\texttt{msg\_judgment}(E,\ Name(x),\ \mathrm{T}) \Rightarrow E \vdash Name(x) : T$ hold,
    thus by the (Msg Encrypt) rule,
    $\texttt{msg\_judgment}(E,\ \{M\}_{Name(x)},\ \mathbf{Un}) \Rightarrow E \vdash \{M\}_{Name(x)} : \mathbf{Un}$ holds.
  - When $typ = \mathbf{Un}$ and $E(x) = \mathbf{Un}$, line 2.15
    By the induction hypothesis
    $\texttt{msg\_judgment}(E,\ M,\ \mathbf{Un}) \Rightarrow E \vdash M : \mathbf{Un}$ and
    $\texttt{msg\_judgment}(E,\ Name(x),\ \mathbf{Un}) \Rightarrow E \vdash Name(x) : \mathbf{Un}$ hold,
    thus by the (Msg Encrypt Un) rule,
    $\texttt{msg\_judgment}(E,\ \{M\}_{Name(x)},\ \mathbf{Un}) \Rightarrow E \vdash \{M\}_{Name(x)} : \mathbf{Un}$ holds.
- $msg = \{M\}_N$, when $typ = \mathbf{Un}$ and $N \neq Name(x)$, line 2.21
  By the induction hypothesis
  $\texttt{msg\_judgment}(E,\ M,\ \mathbf{Un}) \Rightarrow E \vdash M : \mathbf{Un}$ and
  $\texttt{msg\_judgment}(E,\ N,\ \mathbf{Un}) \Rightarrow E \vdash N : \mathbf{Un}$ hold,
  thus by the (Msg Encrypt Un) rule,
  $\texttt{msg\_judgment}(E,\ \{M\}_N,\ \mathbf{Un}) \Rightarrow E \vdash \{M\}_N : \mathbf{Un}$ holds
- $msg = Name(x)$, when $typ = T$, line 2.26
  If $E(x) = T$, the (Msg x) rule is applicable.
  By lemma 1, $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$ holds.
  $(x \in \texttt{dom}(E)) \Rightarrow (x \in \texttt{dom}(E))$ is trivial.
  Thus, $\texttt{msg\_judgment}(E,\ Name(x),\ \mathrm{T}) \Rightarrow E \vdash Name(x) : T$ holds.
- $msg = \mathbf{ok}$, line 2.28
  - When $typ = \mathbf{Un}$, line 2.30
    By lemma 1, $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$ holds.
    Thus, by the (Msg Ok Un) rule, $\texttt{msg\_judgment}(E,\ \mathbf{ok},\ \mathbf{Un}) \Rightarrow E \vdash \mathbf{ok} : \mathbf{Un}$
    holds.

– When $typ = \mathbf{Ok}(S)$, line 2.32
By lemma 1, $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$ holds.
$(\texttt{clauses}(E) \models C \quad \forall C \in S) \Rightarrow (\texttt{clauses}(E) \models C \quad \forall C \in S)$ and
$\texttt{fn}(S) \subseteq \texttt{dom}(E) \Rightarrow \texttt{fn}(S) \subseteq \texttt{dom}(E)$ are both trivial.
Thus, by the (Msg Ok) rule, $\texttt{msg\_judgment}(E, \mathbf{ok}, \mathbf{Ok}(S)) \Rightarrow E \vdash \mathbf{ok} : \mathbf{Ok}(S)$ holds.

□

Lemma 3. $\texttt{proc\_judgment}(E, proc) \Rightarrow E \vdash proc$

Proof. Proof by induction in the structure of *proc*.
From lemma 1, we know that $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$.
From lemma 2, we know that $\texttt{msg\_judgment}(E, M, T) \Rightarrow E \vdash M : T$.

- $proc = \mathbf{new}\ x : T; P$, line 3.2
$\texttt{generative}(T) \Rightarrow \texttt{generative}(T)$ is trivial.
By the induction hypothesis
$\texttt{proc\_judgment}((E, x : T), \mathrm{P}) \Rightarrow (E, x : T) \vdash P$ holds,
thus by the (Proc Res) rule,
$\texttt{proc\_judgment}(E, \mathbf{new}\ x : T; P) \Rightarrow E \vdash \mathbf{new}\ x : T; P$ holds.

- $proc = (P|Q)$, line 3.4
$\texttt{fn}((P|Q)) \subseteq \texttt{dom}(E) \Rightarrow \texttt{fn}((P|Q)) \subseteq \texttt{dom}(E)$ is trivial.
By the induction hypothesis
$\texttt{proc\_judgment}((\texttt{env}(Q), E), \mathrm{P}) \Rightarrow (\texttt{env}(Q), E) \vdash P$ and
$\texttt{proc\_judgment}((\texttt{env}(P), E), \mathrm{Q}) \Rightarrow (\texttt{env}(P), E) \vdash Q$ hold,
thus by the (Proc Par) rule,
$\texttt{proc\_judgment}(E, (P|Q)) \Rightarrow E \vdash (P|Q)$ holds.

- $proc = \mathbf{nil}$, line 3.6
By lemma 1, $\texttt{env\_judgment}(E) \Rightarrow E \vdash \diamond$ holds.
thus by the (Proc Nil) rule,
$\texttt{proc\_judgment}(E, \mathbf{nil}) \Rightarrow E \vdash \mathbf{nil}$ holds.

- $proc = C$, line 3.8
By lemma 1, $\texttt{env\_judgment}((E, C)) \Rightarrow (E, C) \vdash \diamond$ holds.
thus by the (Proc Begin) rule,
$\texttt{proc\_judgment}(E, C) \Rightarrow E \vdash C$ holds.

- $proc = \mathbf{expect}\ C$, line 3.10
By lemma 1, $\texttt{env\_judgment}((E, C)) \Rightarrow (E, C) \vdash \diamond$ holds.
$\texttt{clauses}(E) \models C \Rightarrow \texttt{clauses}(E) \models C$ is trivial.
thus by the (Proc End) rule,
$\texttt{proc\_judgment}(E, \mathbf{expect}\ C) \Rightarrow E \vdash \mathbf{expect}\ C$ holds.

- $proc = \mathbf{decrypt}\ M\ \mathbf{as}\ (y : T)_{Name(x)}; P$, line 3.12

  – When $E(x) = \mathbf{Key}(T)$, line 3.14
  By lemma 2,
  $\texttt{msg\_judgment}(E, M, \mathbf{Un}) \Rightarrow E \vdash M : \mathbf{Un}$ and
  $\texttt{msg\_judgment}(E, Name(x), \mathbf{Key}(T)) \Rightarrow E \vdash Name(x) : \mathbf{Key}(T)$ hold.
  By the induction hypothesis,

`proc_judgment`$((E, y : T)$, P$) \Rightarrow (E, y : T) \vdash P$ holds.
thus by the (Proc Decrypt) rule,
`proc_judgment`$(E,$ **decrypt** M **as** $(y : T)_{Name(x)}; P) \Rightarrow$
$E \vdash$ **decrypt** M **as** $(y : T)_{Name(x)}; P$ holds.

- When $E(x) =$ **Un**, line 3.16
  By lemma 2,
  `msg_judgment`$(E,$ $M,$ **Un**$) \Rightarrow E \vdash M :$ **Un** and
  `msg_judgment`$(E,$ $Name(x),$ **Un**$) \Rightarrow E \vdash Name(x) :$ **Un** hold.
  By the induction hypothesis,
  `proc_judgment`$((E, y :$ **Un**$)$, P$) \Rightarrow (E, y :$ **Un**$) \vdash P$ holds.
  thus by the (Proc Decrypt Un) rule,
  `proc_judgment`$(E,$ **decrypt** M **as** $(y :$ **Un**$)_{Name(x)}; P) \Rightarrow$
  $E \vdash$ **decrypt** M **as** $(y :$ **Un**$)_{Name(x)}; P$ holds.

- $proc =$ **decrypt** M **as** $(y : T)_N; P$, when $N \neq Name(x)$, line 3.20
  By lemma 2,
  `msg_judgment`$(E,$ $M,$ **Un**$) \Rightarrow E \vdash M :$ **Un** and
  `msg_judgment`$(E,$ $N,$ **Un**$) \Rightarrow E \vdash N :$ **Un** hold.
  By the induction hypothesis,
  `proc_judgment`$((E, y :$ **Un**$)$, P$) \Rightarrow (E, y :$ **Un**$) \vdash P$ holds.
  thus by the (Proc Decrypt Un) rule,
  `proc_judgment`$(E,$ **decrypt** M **as** $(y :$ **Un**$)_N; P) \Rightarrow E \vdash$ **decrypt** M **as** $(y :$ **Un**$)_N; P$
  holds.

- $proc =$ **match** $M : T_M$ **as** $(N, y : U_{sub}); P$, line 3.22

  - When $T_M =$ **Pair**$(x : T, U)$, line 3.24
    By lemma 2,
    `msg_judgment`$(E,$ $M,$ **Pair**$(x : T, U)) \Rightarrow E \vdash M :$ **Pair**$(x : T, U)$ and
    `msg_judgment`$(E,$ $N,$ $T) \Rightarrow E \vdash M : T$ hold.
    By the induction hypothesis,
    `proc_judgment`$((E, y : U\{N/x\})$, P$) \Rightarrow (E, y : U\{N/x\}) \vdash P$
    Use of the (Proc Match) rule implies that $U\{N/x\} = U_{sub}$
    Thus by the (Proc Match) rule,
    `proc_judgment`$(E,$ **match** $M : T_M$ **as** $(N, y : U_{sub}); P) \Rightarrow$
    $E \vdash$ **match** $M : T_M$ **as** $(N, y : U_{sub}); P$
  - When $T_M =$ **Un**, line 3.26
    By lemma 2,
    `msg_judgment`$(E,$ $M,$ **Un**$) \Rightarrow E \vdash M :$ **Un** and
    `msg_judgment`$(E,$ $N,$ **Un**$) \Rightarrow E \vdash M :$ **Un** hold.
    By the induction hypothesis,
    `proc_judgment`$((E, y :$ **Un**$))$
    Thus by the (Proc Match) rule,
    `proc_judgment`$(E,$ **match** $M :$ **Un as** $(N, y :$ **Un**$); P) \Rightarrow$
    $E \vdash$ **match** $M :$ **Un as** $(N, y :$ **Un**$); P$

- $proc =$ **out**$(Name(x), N)$, line 3.30

  - When $E(x) =$ **Ch**(T), line 3.32
    By lemma 2,
    `msg_judgment`$(E,$ $Name(x),$ **Ch**$(T)) \Rightarrow E \vdash Name(x) :$ **Ch**(T) and

`msg_judgment(`E, N, T`)` $\Rightarrow E \vdash N : T$ hold.
thus by the (Proc Output) rule,
`proc_judgment(`E, $\mathbf{out}(Name(x), N)$`)` $\Rightarrow E \vdash \mathbf{out}(Name(x), N)$ holds.

– When $E(x) = \mathbf{Un}$, line 3.34
By lemma 2,
`msg_judgment(`E, $Name(x)$, $\mathbf{Un}$`)` $\Rightarrow E \vdash Name(x) : \mathbf{Un}$ and
`msg_judgment(`E, N, $\mathbf{Un}$`)` $\Rightarrow E \vdash N : \mathbf{Un}$ hold.
thus by the (Proc Output Un) rule,
`proc_judgment(`E, $\mathbf{out}(Name(x), N)$`)` $\Rightarrow E \vdash \mathbf{out}(Name(x), N)$ holds.

- $proc = \mathbf{out}(M, N)$, line 3.38
By lemma 2,
`msg_judgment(`E, M, $\mathbf{Un}$`)` $\Rightarrow E \vdash M : \mathbf{Un}$ and
`msg_judgment(`E, N, $\mathbf{Un}$`)` $\Rightarrow E \vdash N : \mathbf{Un}$ hold.
thus by the (Proc Output Un) rule,
`proc_judgment(`E, $\mathbf{out}(M, N)$`)` $\Rightarrow E \vdash \mathbf{out}(M, N)$ holds.

- $proc = \mathbf{in}(Name(y), x); P$, line 3.40

– When $E(y) = \mathbf{Ch}(T)$, line 3.42
By lemma 2,
`msg_judgment(`E, $Name(y)$, $\mathbf{Ch}(T)$`)` $\Rightarrow E \vdash Name(y) : \mathbf{Ch}(T)$ holds.
By the induction hypothesis,
`proc_judgment(`$(E, x : T)$, P`)` $\Rightarrow (E, x : T) \vdash P$ holds.
thus by the (Proc Input) rule,
`proc_judgment(`E, $\mathbf{in}(Name(y), x); P$`)` $\Rightarrow E \vdash \mathbf{in}(Name(y), x); P$

– When $E(y) = \mathbf{Un}$, line 3.44
By lemma 2,
`msg_judgment(`E, $Name(y)$, $\mathbf{Un}$`)` $\Rightarrow E \vdash Name(y) : \mathbf{Un}$ holds.
By the induction hypothesis,
`proc_judgment(`$(E, x : \mathbf{Un})$, P`)` $\Rightarrow (E, x : \mathbf{Un}) \vdash P$ holds.
thus by the (Proc Input Un) rule,
`proc_judgment(`E, $\mathbf{in}(Name(y), x); P$`)` $\Rightarrow E \vdash \mathbf{in}(Name(y), x); P$

- $proc = \mathbf{in}(M, x); P$, line 3.48
By lemma 2,
`msg_judgment(`E, $Name(y)$, $\mathbf{Un}$`)` $\Rightarrow E \vdash Name(y) : \mathbf{Un}$ holds.
By the induction hypothesis,
`proc_judgment(`$(E, x : \mathbf{Un})$, P`)` $\Rightarrow (E, x : \mathbf{Un}) \vdash P$ holds.
thus by the (Proc Input Un) rule,
`proc_judgment(`E, $\mathbf{in}(M, x); P$`)` $\Rightarrow E \vdash \mathbf{in}(M, x); P$ holds.

- $proc = \mathbf{split}\ Name(z)\ \mathbf{as}\ (x : T, y : U); P$, line 3.50

– When $E(z) = \mathbf{Pair}(x : T, U)$, line 3.52
By lemma 2,
`msg_judgment(`E, $Name(z)$, $\mathbf{Pair}(x : T, U)$`)` $\Rightarrow E \vdash Name(z) : \mathbf{Pair}(x : T, U)$
holds.
By the induction hypothesis,
`proc_judgment(`$(E, x : T, y : U)$, P`)` $\Rightarrow (E, x : T, y : U) \vdash P$ holds.
thus by the (Proc Split) rule,

`proc_judgment(`E, **split** $Name(z)$ **as** $(x : T, y : U); P$`)` $\Rightarrow$
$E \vdash$ **split** $Name(z)$ **as** $(x : T, y : U); P$ holds.

– When $E(z) =$ **Un**, line 3.54
By lemma 2,
`msg_judgment(`E, $Name(z)$, **Un**`)` $\Rightarrow E \vdash Name(z) :$ **Un** holds.
By the induction hypothesis,
`proc_judgment(`$(E, x :$ **Un**$, y :$ **Un**$)$, P`)` $\Rightarrow (E, x :$ **Un**$, y :$ **Un**$) \vdash P$ holds.
thus by the (Proc Split Un) rule,
`proc_judgment(`E, **split** $Name(z)$ **as** $(x : T, y : U); P$`)` $\Rightarrow$
$E \vdash$ **split** $Name(z)$ **as** $(x : T, y : U); P$ holds.

- $proc =$ **split pair**(M_1, M_2) **as** $(x : T, y : U); P$, line 3.58
By lemma 2,
`msg_judgment(`E, **pair**(M_1, M_2), **Pair**$(x : T, U)$`)` $\Rightarrow E \vdash$ **pair**$(M_1, M_2) :$ **Pair**$(x : T, U)$ holds.
By the induction hypothesis,
`proc_judgment(`$(E, x : T, y : U)$, P`)` $\Rightarrow (E, x : T, y : U) \vdash P$ holds.
thus by the (Proc Split) rule,
`proc_judgment(`E, **split pair**(M_1, M_2) **as** $(x : T, y : U); P$`)` $\Rightarrow$
$E \vdash$ **split pair**(M_1, M_2) **as** $(x : T, y : U); P$ holds.

- $proc =$ **split** M **as** $(x : T, u : U); P$, line 3.60
By lemma 2,
`msg_judgment(`E, M, **Un**`)` $\Rightarrow E \vdash M :$ **Un** holds.
By the induction hypothesis,
`proc_judgment(`$(E, x :$ **Un**$, y :$ **Un**$)$, P`)` $\Rightarrow (E, x :$ **Un**$, y :$ **Un**$) \vdash P$ holds.
thus by the (Proc Split Un) rule,
`proc_judgment(`E, **split** M **as** $(x : T, y : U); P$`)` $\Rightarrow$
$E \vdash$ **split** M **as** $(x : T, y : U); P$ holds.

□

Lemma 4. $E \vdash msg : typ \Rightarrow$ `msg_judgment(`E, msg, typ`)`

Proof. Proof by induction in the structure of msg. From lemma 1, we know that $E \vdash \diamond \Rightarrow$ `env_judgment(`E`)`.

- $msg =$ **pair**(M, N), (Msg Pair) and (Msg Pair Un) are applicable.

  – When using (Msg Pair), $typ =$ **Pair**$(x : T, U)$
  By the induction hypothesis,
  $E \vdash M : T \Rightarrow$ `msg_judgment(`E, M, T`)` and
  $E \vdash N : U\{M/x\} \Rightarrow$ `msg_judgment(`E, N, $U\{M/x\}$`)` hold,
  thus by the case of line 2.4,
  $E \vdash$ **pair**$(M, N) :$ **Pair**$(x : T, U) \Rightarrow$ `msg_judgment(`E, **pair**(M, N), **Pair**$(x : T, U)$`)` holds.

  – When using (Msg Pair Un), $typ =$ **Un**
  By the induction hypothesis,
  $E \vdash M :$ **Un** $\Rightarrow$ `msg_judgment(`E, M, **Un**`)` and
  $E \vdash N :$ **Un** $\Rightarrow$ `msg_judgment(`E, N, **Un**`)` hold,
  thus by the case of line 2.6,
  $E \vdash$ **pair**$(M, N) :$ **Un** $\Rightarrow$ `msg_judgment(`E, **pair**(M, N), **Un**`)` holds.

- $msg = \{M\}_N$, (Msg Encrypt) and (Msg Encrypt Un) are applicable.
  - When using (Msg Encrypt), $typ = \mathbf{Un}$ and $E \vdash N : \mathbf{Key}(T)$
    Because $\mathbf{Key}(T)$ is generative and has no message constructor, $N = Name(x)$.
    By the induction hypothesis,
    $E \vdash M : T \Rightarrow$ `msg_judgment(`E, M, T`)` and
    $E \vdash Name(x) : T \Rightarrow$ `msg_judgment(`E, $Name(x)$, `T)` hold,
    thus by the case of line 2.13,
    $E \vdash \{M\}_{Name(x)} : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, $\{M\}_{Name(x)}$, $\mathbf{Un}$`)` holds.
  - When using (Msg Encrypt Un), $typ = \mathbf{Un}$ and $E \vdash N : \mathbf{Un}$ By the induction hypothesis,
    $E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Un}$`)` and
    $E \vdash N : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, N, $\mathbf{Un}$`)` hold,
    thus by the cases of both line 2.15, which is used when $N = Name(x)$ and line 2.21, which is used otherwise,
    $E \vdash \{M\}_N : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, $\{M\}_N$, $\mathbf{Un}$`)` holds.
- $msg = Name(x)$, (Msg x) is applicable, $typ = T$. (Msg x) is only applicable if $E(x) = T$.
  By lemma 1, $E \vdash \diamond \Rightarrow$ `env_judgment(`E`)` holds.
  $(x \in$ `dom(`E`)`$) \Rightarrow (x \in$ `dom(`E`)`$)$ is trivial.
  Thus by the case of line 2.26,
  Thus, $E \vdash Name(x) : T \Rightarrow$ `msg_judgment(`E, $Name(x)$, `T)` holds.
- $msg = \mathbf{ok}$, (Msg Ok) and (Msg Ok Un) are applicable.
  - When using (Msg Ok Un), $typ = \mathbf{Un}$
    By lemma 1, $E \vdash \diamond \Rightarrow$ `env_judgment(`E`)` holds.
    Thus, by the case of line 2.30, $E \vdash \mathbf{ok} : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, $\mathbf{ok}$, $\mathbf{Un}$`)` holds.
  - When using (Msg Ok), $typ = \mathbf{Ok}(S)$
    By lemma 1, $E \vdash \diamond \Rightarrow$ `env_judgment(`E`)` holds.
    (`clauses(`E`)` $\models C \quad \forall C \in S) \Rightarrow$ (`clauses(`E`)` $\models C \quad \forall C \in S$) and
    `fn(`S`)` $\subseteq$ `dom(`E`)` $\Rightarrow$ `fn(`S`)` $\subseteq$ `dom(`E`)` are both trivial.
    Thus, by the case of line 2.32, $E \vdash \mathbf{ok} : \mathbf{Ok}(S) \Rightarrow$ `msg_judgment(`E, $\mathbf{ok}$, $\mathbf{Ok}(S)$`)` holds.

□

Lemma 5. $E \vdash proc \Rightarrow$ `proc_judgment(`E, $proc$`)`

Proof. Proof by induction in the structure of $proc$.
From lemma 1, we know that `env_judgment(`E`)` $\Rightarrow E \vdash \diamond$.
From lemma 4, we know that $E \vdash M : T \Rightarrow$ `msg_judgment(`E, M, T`)`.

- $proc = \mathbf{new}\ x : T; P$, (Proc Res) is applicable.
  `generative(`T`)` $\Rightarrow$ `generative(`T`)` is trivial.
  By the induction hypothesis,
  $(E, x : T) \vdash P \Rightarrow$ `proc_judgment(`$(E, x : T)$, `P)` holds,
  thus by the case of line 3.2,
  $E \vdash \mathbf{new}\ x : T; P \Rightarrow$ `proc_judgment(`E, $\mathbf{new}\ x : T; P$`)` holds.

- $proc = (P|Q)$, (Proc Par) is applicable.
  $\texttt{fn}((P|Q)) \subseteq \texttt{dom}(E) \Rightarrow \texttt{fn}((P|Q)) \subseteq \texttt{dom}(E)$ is trivial.
  By the induction hypothesis,
  $(\texttt{env}(Q), E) \vdash P \Rightarrow \texttt{proc\_judgment}((\texttt{env}(Q), E), \texttt{P})$ and
  $(\texttt{env}(P), E) \vdash Q \Rightarrow \texttt{proc\_judgment}((\texttt{env}(P), E), \texttt{Q})$ hold,
  thus by the case of line 3.4,
  $E \vdash (P|Q) \Rightarrow \texttt{proc\_judgment}(E, (P|Q))$ holds.

- $proc = \textbf{nil}$, (Proc Nil) is applicable.
  By lemma 1, $E \vdash \diamond \Rightarrow \texttt{env\_judgment}(E)$ holds.
  thus by the case of line 3.6,
  $E \vdash \textbf{nil} \Rightarrow \texttt{proc\_judgment}(E, \textbf{nil})$ holds.

- $proc = C$, (Proc Begin) is applicable.
  By lemma 1, $(E, C) \vdash \diamond \Rightarrow \texttt{env\_judgment}((E, C))$ holds.
  thus by the case of line 3.8,
  $E \vdash C \Rightarrow \texttt{proc\_judgment}(E, C)$ holds.

- $proc = \textbf{expect}\ C$, (Proc End) is applicable.
  By lemma 1, $(E, C) \vdash \diamond \Rightarrow \texttt{env\_judgment}((E, C))$ holds.
  $\texttt{clauses}(E) \models C \Rightarrow \texttt{clauses}(E) \models C$ is trivial.
  thus by the case of line 3.10,
  $E \vdash \textbf{expect}\ C \Rightarrow \texttt{proc\_judgment}(E, \textbf{expect}\ C)$ holds.

- $proc = \textbf{decrypt}\ M\ \textbf{as}\ (y : T)_N; P$, (Proc Decrypt) and (Proc Decrypt Un) are applicable.

  - When using (Proc Decrypt), $E \vdash N : \textbf{Key}(T)$.
    Because $\textbf{Key}(T)$ is generative and has no message constructor, $N = Name(x)$.
    By lemma 4,
    $E \vdash M : \textbf{Un} \Rightarrow \texttt{msg\_judgment}(E, M, \textbf{Un})$ and
    $E \vdash Name(x) : \textbf{Key}(T) \Rightarrow \texttt{msg\_judgment}(E, Name(x), \textbf{Key}(T))$ hold.
    By the induction hypothesis,
    $(E, y : T) \vdash P \Rightarrow \texttt{proc\_judgment}((E, y : T), \texttt{P})$ holds.
    thus by the case of line 3.14,
    $E \vdash \textbf{decrypt}\ M\ \textbf{as}\ (y : T)_{Name(x)}; P \Rightarrow$
    $\texttt{proc\_judgment}(E, \textbf{decrypt}\ M\ \textbf{as}\ (y : T)_{Name(x)}; P)$ holds.
  - When using (Proc Decrypt Un), $E \vdash N : \textbf{Un}$. By lemma 4,
    $E \vdash M : \textbf{Un} \Rightarrow \texttt{msg\_judgment}(E, M, \textbf{Un})$ and
    $E \vdash N : \textbf{Un} \Rightarrow \texttt{msg\_judgment}(E, N, \textbf{Un})$ hold.
    By the induction hypothesis,
    $(E, y : \textbf{Un}) \vdash P \Rightarrow \texttt{proc\_judgment}((E, y : \textbf{Un}), \texttt{P})$ holds.
    thus by the cases of both line 3.16, which is used when $N = Name(x)$, and line 3.20, which is used otherwise,
    $E \vdash \textbf{decrypt}\ M\ \textbf{as}\ (y : \textbf{Un})_N; P \Rightarrow$
    $\texttt{proc\_judgment}(E, \textbf{decrypt}\ M\ \textbf{as}\ (y : \textbf{Un})_N; P)$ holds.

- $proc = \textbf{match}\ M : T_M\ \textbf{as}\ (N, y : U_{sub}); P$, (Proc Match) and (Proc Match Un) are applicable.

  - When using (Proc Match), $T_M = \textbf{Pair}(x : T, U)$ and $U_{sub} = U\{N/x\}$.
    By lemma 4,

$E \vdash M : \mathbf{Pair}(x : T, U) \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Pair}(x : T, U)$`)` and
$E \vdash M : T \Rightarrow$ `msg_judgment(`E, N, T`)` hold.
By the induction hypothesis,
$(E, y : U\{N/x\}) \vdash P \Rightarrow$ `proc_judgment(`$(E, y : U\{N/x\})$, P`)`
Thus by the case of line 3.24,
$E \vdash \mathbf{match}\ M : \mathbf{Pair}(x : T, U)\ \mathbf{as}\ (N, y : U\{N/x\}); P \Rightarrow$
`proc_judgment(`E, $\mathbf{match}\ M : \mathbf{Pair}(x : T, U)\ \mathbf{as}\ (N, y : U\{N/x\}); P$`)` holds.

– When using (Proc Match Un), $T_M = \mathbf{Un}$ and $U_{sub} = \mathbf{Un}$.
By lemma 4,
$E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Un}$`)` and
$E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, N, $\mathbf{Un}$`)` hold.
By the induction hypothesis,
$(E, y : \mathbf{Un}) \vdash P \Rightarrow$
`proc_judgment(`$(E, y : \mathbf{Un})$, P`)`
Thus by the case of line 3.26,
$E \vdash \mathbf{match}\ M : \mathbf{Un}\ \mathbf{as}\ (N, y : \mathbf{Un}); P \Rightarrow$ `proc_judgment(`E, $\mathbf{match}\ M : \mathbf{Un}\ \mathbf{as}\ (N, y : \mathbf{Un});$`)`
holds.

- $proc = \mathbf{out}(M, N)$, (Proc Output) and (Proc Output Un) are applicable.

  – When using (Proc Output), $E \vdash M : \mathbf{Ch}(T)$.
  Because $\mathbf{Ch}(T)$ is generative and has no message constructor, $M = Name(x)$.
  By lemma 4,
  $E \vdash Name(x) : \mathbf{Ch}(T) \Rightarrow$ `msg_judgment(`E, $Name(x)$, $\mathbf{Ch}(T)$`)` and
  $E \vdash N : T \Rightarrow$ `msg_judgment(`E, N, T`)` hold.
  thus by the case of line 3.32,
  $E \vdash \mathbf{out}(Name(x), N) \Rightarrow$ `proc_judgment(`E, $\mathbf{out}(Name(x), N)$`)` holds.

  – When using (Proc Output Un), $E \vdash M : \mathbf{Un}$.
  By lemma 4,
  $E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Un}$`)` and
  $E \vdash N : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, N, $\mathbf{Un}$`)` hold.
  thus by the cases of both line 3.34, which is used when $M = Name(x)$, and line 3.38, which is used otherwise,
  $E \vdash \mathbf{out}(M, N) \Rightarrow$ `proc_judgment(`E, $\mathbf{out}(M, N)$`)` holds.

- $proc = \mathbf{in}(M, x); P$, (Proc Input) and (Proc Output Un) are applicable.

  – When using (Proc Input), $E \vdash M : \mathbf{Ch}(T)$.
  Because $\mathbf{Ch}(T)$ is generative and has no message constructor, $M = Name(x)$.
  By lemma 4,
  $E \vdash Name(y) : \mathbf{Ch}(T) \Rightarrow$ `msg_judgment(`E, $Name(y)$, $\mathbf{Ch}(T)$`)` holds.
  By the induction hypothesis,
  $(E, x : T) \vdash P \Rightarrow$ `proc_judgment(`$(E, x : T)$, P`)` holds.
  thus by the case of line 3.42,
  $E \vdash \mathbf{in}(Name(y), x); P \Rightarrow$ `proc_judgment(`E, $\mathbf{in}(Name(y), x); P$`)`

  – When using (Proc Input Un), $E \vdash M : \mathbf{Un}$.
  By lemma 4,
  $E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Un}$`)` holds.
  By the induction hypothesis,
  $(E, x : \mathbf{Un}) \vdash P \Rightarrow$ `proc_judgment(`$(E, x : \mathbf{Un})$, P`)` holds.

thus by the cases of both line 3.44, which is used when $M = Name(x)$, and line 3.48, which is used otherwise,
$E \vdash \mathbf{in}(M, x); P \Rightarrow$ `proc_judgment(`E, $\mathbf{in}(M, x); P$`)`

- $proc = \mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P$, (Proc Split) and (Proc Split Un) are applicable.
  - When using (Proc Split), $E \vdash M : \mathbf{Pair}(x : T, U)$.
    Because $\mathbf{Pair}(x : T, U)$ is not generative, either $M = Name(z)$ or $M = \mathbf{pair}(M_M, N_M)$.
    By lemma 4,
    $E \vdash M : \mathbf{Pair}(x : T, U) \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Pair}(x : T, U)$`)` holds.
    By the induction hypothesis,
    $(E, x : T, y : U) \vdash P \Rightarrow$ `proc_judgment(`$(E, x : T, y : U)$, P`)` holds.
    thus by the cases of both line 3.52, which is used when $M = Name(z)$, and line 3.58, which is used when $M = \mathbf{pair}(M_M, N_M)$,
    $E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P \Rightarrow$
    `proc_judgment(`E, $\mathbf{split}\ M\ \mathbf{as}\ (x : T, y : U); P$`)` holds.
  - When using (Proc Split Un), $E \vdash M : \mathbf{Un}$, $T = \mathbf{Un}$ and $U = \mathbf{Un}$.
    By lemma 4,
    $E \vdash M : \mathbf{Un} \Rightarrow$ `msg_judgment(`E, M, $\mathbf{Un}$`)` holds.
    By the induction hypothesis,
    $(E, x : \mathbf{Un}, y : \mathbf{Un}) \vdash P \Rightarrow$ `proc_judgment(`$(E, x : \mathbf{Un}, y : \mathbf{Un})$, P`)` holds.
    thus by the cases of both line 3.54, which is used when $M = Name(z)$, and line 3.60, which is used otherwise,
    $E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x : \mathbf{Un}, y : \mathbf{Un}); P \Rightarrow$
    `proc_judgment(`E, $\mathbf{split}\ M\ \mathbf{as}\ (x : \mathbf{Un}, y : \mathbf{Un}); P$`)` holds.

□

Appendix C

Relations and Axioms

C.1 Relations

The ALFP formulae that follow in section 9.2 and 9.3 populate a number of relations. We shortly explain the meaning of each relation in this section.

Names represent elements of the process and type system. We operate with a notion of *proper names* of messages and environments. A proper name is an encoding representing uniquely a message or environment in the process.

C.1.1 Equality and Inequality

In some cases, we need to know if two names are equal. Thus, the equality and inequality relations:

- $\mathsf{eq}(x, y)$: The names x and y are equal
- $\mathsf{neq}(x, y)$: The names x and y are not equal

C.1.2 Type Relations

These relations indicate the content of a type variable:

- $\mathsf{Un}(t)$: The type variable $T = \mathbf{Un}$
- $\mathsf{Key}(t_1, t_2)$: The type variable $T_1 = \mathbf{Key}(T_2)$
- $\mathsf{Ch}(t_1, t_2)$: The type variable $T_1 = \mathbf{Ch}(T_2)$
- $\mathsf{Pair}(t_1, x, t_2, t_3)$: The type variable $T_1 = \mathbf{Pair}(x : T_2, T_3)$
- $\mathsf{Ok}(t, s)$: The type variable $T = \mathbf{Ok}(S)$

Appending the postfix "Q" we get the *may* relations, indicating what may be contained in a variable if it is not **Un**.

- $\mathsf{UnQ}(t)$: The type variable $T \stackrel{?}{=} \mathbf{Un}$
- $\mathsf{KeyQ}(t_1, t_2)$: The type variable $T_1 \stackrel{?}{=} \mathbf{Key}(T_2)$
- $\mathsf{ChQ}(t_1, t_2)$: The type variable $T_1 \stackrel{?}{=} \mathbf{Ch}(T_2)$

- $\mathsf{PairQ}(t_1, x, t_2, t_3)$: The type variable $T_1 \stackrel{?}{=} \mathbf{Pair}(x : T_2, T_3)$
- $\mathsf{OkQ}(t, s)$: The type variable $T \stackrel{?}{=} \mathbf{Ok}(S)$

We have relations indicating generativity, type equality, as well as type assignments.

- $\mathsf{Gen}(t)$: Type variable T is generative
- $\mathsf{NonGen}(t)$: Type variable T is not generative
- $\mathsf{Teq}(t_1, t_2)$: Type variables T_1 and T_2 are equal
- $\mathsf{Type}(x, t)$: The type of name x is type variable T

C.1.3 Message Relations

We indicate what kind of message each proper name is related to.

- $\mathsf{Name}(m)$: Message M is a name
- $\mathsf{OkTerm}(m)$: Message M is an **ok** token
- $\mathsf{TermPair}(m, m_1, m_2)$: Message M is the pair message $\mathbf{pair}(M_1, M_2)$
- $\mathsf{Enc}(m, m_1, m_2)$: Message M is the encrypted message $\{M_1\}_{M_2}$

Subs indicates exactly what happens to any proper name of a message when a submessage is substituted. Fn indicates the free names of an entity.

- $\mathsf{Subs}(z', z, m', m)$: The name z' results from substituting all occurences M with M' in the entity represented by z. z and z' may be proper names of either terms, effects or types.
- $\mathsf{Fn}(x, z)$: The name x is a free name in entity z. z may be the name of either a term, effect or type

C.1.4 Effect Relations

The Effect relation indicates the content of effect variables.

- $\mathsf{Effect}(l, m, s)$: The effect $\ell(M) \in S$
- $\mathsf{Eff}(t)$: Type T contains effects
- $\mathsf{NonEff}(t)$: Type T does not contain effects

C.2 Axioms

Following is the list of axioms currently in the type inference implementation.

- Equality and inequality on names. We simply describe that every distinct name is equal to itself, and not equal to any other name.
  $\forall \mathtt{x}.\mathsf{eq}(\mathtt{x}, \mathtt{x}) \wedge \forall \mathtt{x}.\forall \mathtt{y}.(\neg \mathsf{eq}(\mathtt{x}, \mathtt{y})) \Rightarrow \mathsf{neq}(\mathtt{x}, \mathtt{y})$

- Generativity and non-generativity.These formulae map types to the generativity relations.
  $\forall \mathtt{t}.(\mathsf{Un}(\mathtt{t})) \Rightarrow \mathsf{Gen}(\mathtt{t})$
  $\forall \mathtt{t1}.\forall \mathtt{t2}.(\mathsf{Key}(\mathtt{t1},\mathtt{t2})) \Rightarrow \mathsf{Gen}(\mathtt{t1})$
  $\forall \mathtt{t1}.\forall \mathtt{t2}.(\mathsf{Ch}(\mathtt{t1},\mathtt{t2})) \Rightarrow \mathsf{Gen}(\mathtt{t1})$
  $\forall \mathtt{t1}.\forall \mathtt{x}.\forall \mathtt{t2}.\forall \mathtt{t3}.(\mathsf{Pair}(\mathtt{t1},\mathtt{x},\mathtt{t2},\mathtt{t3})) \Rightarrow \mathsf{NonGen}(\mathtt{t1})$
  $\forall \mathtt{t}.\forall \mathtt{r}.(\mathsf{Ok}(\mathtt{t},\mathtt{r})) \Rightarrow \mathsf{NonGen}(\mathtt{t})$

- Type equality. We say that if two types are equal, then their inner types are also equal.
  $\forall \mathtt{x1}.\forall \mathtt{x2}.\forall \mathtt{t1'}.\forall \mathtt{t1''}.\forall \mathtt{t2}.\forall \mathtt{t3}.\forall \mathtt{t4}.\forall \mathtt{t5}.(\mathsf{Pair}(\mathtt{t1'},\mathtt{x1},\mathtt{t2},\mathtt{t3}) \wedge \mathsf{Pair}(\mathtt{t1''},\mathtt{x2},\mathtt{t4},\mathtt{t5}) \wedge \mathsf{Teq}(\mathtt{t1'},\mathtt{t1''})) \Rightarrow \mathsf{Teq}(\mathtt{t2},\mathtt{t4}) \wedge \mathsf{Teq}(\mathtt{t3},\mathtt{t5})$
  $\forall \mathtt{t1'}.\forall \mathtt{t1''}.\forall \mathtt{t2}.\forall \mathtt{t3}.(\mathsf{Key}(\mathtt{t1'},\mathtt{t2}) \wedge \mathsf{Key}(\mathtt{t1''},\mathtt{t3}) \wedge \mathsf{Teq}(\mathtt{t1'},\mathtt{t1''})) \Rightarrow \mathsf{Teq}(\mathtt{t2},\mathtt{t3})$
  $\forall \mathtt{t1'}.\forall \mathtt{t1''}.\forall \mathtt{t2}.\forall \mathtt{t3}.(\mathsf{Ch}(\mathtt{t1'},\mathtt{t2}) \wedge \mathsf{Ch}(\mathtt{t1''},\mathtt{t3}) \wedge \mathsf{Teq}(\mathtt{t1'},\mathtt{t1''})) \Rightarrow \mathsf{Teq}(\mathtt{t2},\mathtt{t3})$
  $\forall \mathtt{t1'}.\forall \mathtt{t1''}.\forall \mathtt{s1}.\forall \mathtt{s2}.(\mathsf{Ok}(\mathtt{t1'},\mathtt{s1}) \wedge \mathsf{Ok}(\mathtt{t1''},\mathtt{s2}) \wedge \mathsf{Teq}(\mathtt{t1'},\mathtt{t1''})) \Rightarrow \mathsf{Teq}(\mathtt{s1},\mathtt{s2})$

- Type equality is both reflexive and transitive.
  $\forall \mathtt{t}.\mathsf{Teq}(\mathtt{t},\mathtt{t})$
  $\forall \mathtt{t1}.\forall \mathtt{t2}.(\mathsf{Teq}(\mathtt{t1},\mathtt{t2})) \Rightarrow \mathsf{Teq}(\mathtt{t2},\mathtt{t1})$
  $\forall \mathtt{t1}.\forall \mathtt{t2}.\forall \mathtt{t3}.(\mathsf{Teq}(\mathtt{t1},\mathtt{t2}) \wedge \mathsf{Teq}(\mathtt{t2},\mathtt{t3})) \Rightarrow \mathsf{Teq}(\mathtt{t1},\mathtt{t3})$

- Substitution on terms
  $\forall \mathtt{n}.\forall \mathtt{m'}.\forall \mathtt{m}.(\mathsf{OkTerm}(\mathtt{n})) \Rightarrow \mathsf{Subs}(\mathtt{n},\mathtt{n},\mathtt{m'},\mathtt{m})$
  $\forall \mathtt{m'}.\forall \mathtt{m}.(\mathsf{Name}(\mathtt{m})) \Rightarrow \mathsf{Subs}(\mathtt{m'},\mathtt{m},\mathtt{m'},\mathtt{m})$
  $\forall \mathtt{n}.\forall \mathtt{n1'}.\forall \mathtt{n1}.\forall \mathtt{m'}.\forall \mathtt{m}.\forall \mathtt{n2'}.\forall \mathtt{n2}.\forall \mathtt{n'}.(\mathsf{Subs}(\mathtt{n1'},\mathtt{n1},\mathtt{m'},\mathtt{m}) \wedge \mathsf{Subs}(\mathtt{n2'},\mathtt{n2},\mathtt{m'},\mathtt{m}) \wedge \mathsf{TermPair}(\mathtt{n},\mathtt{n1},\mathtt{n2}) \wedge \mathsf{TermPair}(\mathtt{n'},\mathtt{n1'},\mathtt{n2'})) \Rightarrow \mathsf{Subs}(\mathtt{n'},\mathtt{n},\mathtt{m'},\mathtt{m})$
  $\forall \mathtt{n}.\forall \mathtt{n1'}.\forall \mathtt{n1}.\forall \mathtt{m'}.\forall \mathtt{m}.\forall \mathtt{n2'}.\forall \mathtt{n2}.\forall \mathtt{n'}.(\mathsf{Subs}(\mathtt{n1'},\mathtt{n1},\mathtt{m'},\mathtt{m}) \wedge \mathsf{Subs}(\mathtt{n2'},\mathtt{n2},\mathtt{m'},\mathtt{m}) \wedge \mathsf{Enc}(\mathtt{n},\mathtt{n1},\mathtt{n2}) \wedge \mathsf{Enc}(\mathtt{n'},\mathtt{n1'},\mathtt{n2'})) \Rightarrow \mathsf{Subs}(\mathtt{n'},\mathtt{n},\mathtt{m'},\mathtt{m})$

- Substitution on effects
  $\forall \mathtt{n}.\forall \mathtt{n'}.\forall \mathtt{l}.\forall \mathtt{m}.\forall \mathtt{m'}.\forall \mathtt{s}.\forall \mathtt{s'}.(\mathsf{Effect}(\mathtt{l},\mathtt{n},\mathtt{s'}) \wedge \mathsf{Subs}(\mathtt{n'},\mathtt{n},\mathtt{m'},\mathtt{m})) \Rightarrow \mathsf{Effect}(\mathtt{l},\mathtt{n'},\mathtt{s})$

- Free names of terms. A name is always free in itself. For pairs and encryptions, if a name is free in either element, it is free for the pair, or encryption.
  $\forall \mathtt{n}.(\mathsf{Name}(\mathtt{n})) \Rightarrow \mathsf{Fn}(\mathtt{n},\mathtt{n})$
  $\forall \mathtt{n}.\forall \mathtt{n1}.\forall \mathtt{n2}.\forall \mathtt{m}.((\mathsf{Fn}(\mathtt{m},\mathtt{n1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{n2})) \wedge \mathsf{TermPair}(\mathtt{n},\mathtt{n1},\mathtt{n2})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{n})$
  $\forall \mathtt{n}.\forall \mathtt{n1}.\forall \mathtt{n2}.\forall \mathtt{m}.((\mathsf{Fn}(\mathtt{m},\mathtt{n1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{n2})) \wedge \mathsf{Enc}(\mathtt{n},\mathtt{n1},\mathtt{n2})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{n})$

- Free names of types
  $\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{m}.(\mathsf{Ch}(\mathtt{t},\mathtt{t1}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{t1})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
  $\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{m}.(\mathsf{Key}(\mathtt{t},\mathtt{t1}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{t1})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
  $\forall \mathtt{t}.\forall \mathtt{t1}.\forall \mathtt{t2}.\forall \mathtt{x}.\forall \mathtt{m}.(\mathsf{Pair}(\mathtt{t},\mathtt{x},\mathtt{t1},\mathtt{t2}) \wedge (\mathsf{Fn}(\mathtt{m},\mathtt{t1}) \vee \mathsf{Fn}(\mathtt{m},\mathtt{t2})) \wedge \mathsf{neq}(\mathtt{m},\mathtt{x})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
  $\forall \mathtt{t}.\forall \mathtt{s}.\forall \mathtt{m}.(\mathsf{Ok}(\mathtt{t},\mathtt{s}) \wedge \mathsf{Fn}(\mathtt{m},\mathtt{s})) \Rightarrow \mathsf{Fn}(\mathtt{m},\mathtt{t})$
  $\forall \mathtt{m}.\forall \mathtt{n}.\forall \mathtt{l}.\forall \mathtt{s}.(\mathsf{Fn}(\mathtt{n},\mathtt{m}) \wedge \mathsf{Effect}(\mathtt{l},\mathtt{m},\mathtt{s})) \Rightarrow \mathsf{Fn}(\mathtt{n},\mathtt{s})$

- Possible types. These axioms were added to the ones of [10], to describe how the $=$ and $\stackrel{?}{=}$ type relations interact. The Un relation is populated by the generation rules in a stratum prior to all the other type relations. In that stratum, all the types that must be **Un** are found, and a stratum later, the other type relations are populated from the $\stackrel{?}{=}$ relations to fill in those types that are not **Un**.
  $\forall \mathtt{t}.\forall \mathtt{t1}.(\mathsf{KeyQ}(\mathtt{t},\mathtt{t1}) \wedge \neg\mathsf{Un}(\mathtt{t})) \Rightarrow \mathsf{Key}(\mathtt{t},\mathtt{t1})$
  $\forall \mathtt{t}.\forall \mathtt{x}.\forall \mathtt{t1}.\forall \mathtt{t2}.(\mathsf{PairQ}(\mathtt{t},\mathtt{x},\mathtt{t1},\mathtt{t2}) \wedge \neg\mathsf{Un}(\mathtt{t})) \Rightarrow \mathsf{Pair}(\mathtt{t},\mathtt{x},\mathtt{t1},\mathtt{t2})$

$\forall t.\forall t1.(\mathsf{ChQ}(t,t1) \wedge \neg\mathsf{Un}(t)) \Rightarrow \mathsf{Ch}(t,t1)$
$\forall t.\forall s.(\mathsf{OkQ}(t,s) \wedge \neg\mathsf{Un}(t)) \Rightarrow \mathsf{Ok}(t,s)$
$\forall v.\forall u.\forall m.\forall t.(\mathsf{AbsQ}(v,u,m,t) \wedge \neg\mathsf{Un}(v)) \Rightarrow \mathsf{Abs}(v,u,m,t)$
$\forall v.\forall u.\forall n.(\mathsf{AppQ}(v,u,n) \wedge \neg\mathsf{Un}(v)) \Rightarrow \mathsf{App}(v,u,n)$

- Axioms for deciding which types contain effects, and which do not.
  $\forall t.\forall s.(\mathsf{Ok}(t,s)) \Rightarrow \mathsf{Eff}(t)$
  $\forall x.\forall t.\forall t1.\forall t2.(\mathsf{Pair}(t,x,t1,t2)) \Rightarrow \mathsf{NonEff}(t)$
  $\forall t.\forall t1.(\mathsf{Key}(t,t1)) \Rightarrow \mathsf{NonEff}(t)$
  $\forall t.\forall t1.(\mathsf{Ch}(t,t1)) \Rightarrow \mathsf{NonEff}(t)$
  $\forall t.(\mathsf{Un}(t)) \Rightarrow \mathsf{NonEff}(t)$
  $\forall t1.\forall t2.(\mathsf{Eff}(t1) \wedge \mathsf{Teq}(t1,t2)) \Rightarrow \mathsf{Eff}(t2)$
  $\forall t1.\forall t2.(\mathsf{NonEff}(t1) \wedge \mathsf{Teq}(t1,t2)) \Rightarrow \mathsf{NonEff}(t2)$

- We decide which environments are simple and which are non-simple.
  $\forall e.\forall t.\forall x.(\mathsf{Env}(x,t,e) \wedge \mathsf{Eff}(t)) \Rightarrow \mathsf{Simple}(e)$
  $\forall e.(\exists t.\exists x.\mathsf{Env}(x,t,e) \wedge \mathsf{NonEff}(t)) \Rightarrow \mathsf{NonSimple}(e)$

- We find dependencies between effect variables.
  $\forall e.\forall x.\forall t.\forall s.\forall s'.(\mathsf{Env}(x,t,e) \wedge \mathsf{Ok}(t,s') \wedge \mathsf{EnvDep}(s,e)) \Rightarrow \mathsf{Dep}(s,s')$
  $\forall s1.\forall s2.\forall s3.(\mathsf{Dep}(s1,s2) \wedge \mathsf{Dep}(s2,s3)) \Rightarrow \mathsf{Dep}(s1,s3)$

- We find initial and non-initial effect variables.
  $\forall s.\forall e.(\mathsf{Simple}(e) \wedge \mathsf{EnvDep}(s,e)) \Rightarrow \mathsf{Initial}(s)$
  $\forall s.\forall e.(\mathsf{EnvDep}(s,e) \wedge \mathsf{NonSimple}(e)) \Rightarrow \mathsf{NotInitial}(s)$

- We find which messages are admitted by environments and effects
  $\forall m.\forall e.\forall n.((\neg\mathsf{Fn}(n,m) \vee \mathsf{Dom}(n,e))) \Rightarrow \mathsf{Admits}(e,m)$
  $\forall m.\forall e.\forall s.(\mathsf{Admits}(e,m) \wedge \mathsf{EnvDep}(s,e)) \Rightarrow \mathsf{EffAdmits}(s,m)$

- Axioms for abstraction. Propagates abstraction into abstracted type.
  $\forall t1.\forall t1'.\forall t2.\forall m.\forall x.(\mathsf{Ch}(t1,t2) \wedge \mathsf{Abs}(t1',t1,m,x)) \Rightarrow \mathsf{Abs}(t1',t2,m,x)$
  $\forall t1.\forall t1'.\forall t2.\forall m.\forall x.(\mathsf{Key}(t1,t2) \wedge \mathsf{Abs}(t1',t1,m,x)) \Rightarrow \mathsf{Abs}(t1',t2,m,x)$
  $\forall t1.\forall t1'.\forall t2.\forall t3.\forall t2'.\forall m.\forall x1.\forall x2.(\mathsf{Pair}(t1,x1,t2,t3) \wedge \mathsf{Abs}(t1',t1,m,x2)) \Rightarrow \mathsf{Abs}(t1',t2,m,x2) \wedge \mathsf{Abs}(t1',t3,m,x2)$
  $\forall t1.\forall t1'.\forall s.\forall m.\forall x.(\mathsf{Ok}(t1,s) \wedge \mathsf{Abs}(t1',t1,m,x)) \Rightarrow \mathsf{Abs}(t1',s,m,x)$

- Axioms for application. Indicates application of a type abstraction.
  $\forall t1.\forall t2.\forall t3.\forall t4.\forall m1.\forall m2.\forall x.(\mathsf{Ch}(t1,t2) \wedge \mathsf{Abs}(t1,t3,m1,x) \wedge \mathsf{App}(t4,t1,m2)) \Rightarrow \mathsf{Ch}(t4,t2) \wedge \mathsf{Apply}(t4,t4,m2,m1)$
  $\forall t1.\forall t2.\forall t3.\forall t4.\forall m1.\forall m2.\forall x.(\mathsf{Key}(t1,t2) \wedge \mathsf{Abs}(t1,t3,m1,x) \wedge \mathsf{App}(t4,t1,m2)) \Rightarrow \mathsf{Key}(t4,t2) \wedge \mathsf{Apply}(t4,t4,m2,m1)$
  $\forall t1.\forall t2.\forall t3.\forall t4.\forall t5.\forall m1.\forall m2.\forall x.\forall x1.(\mathsf{Pair}(t1,x1,t2,t5) \wedge \mathsf{Abs}(t1,t3,m1,x) \wedge \mathsf{App}(t4,t1,m2)) \Rightarrow \mathsf{Pair}(t4,x1,t2,t5) \wedge \mathsf{Apply}(t4,t4,m2,m1)$
  $\forall t1.\forall s.\forall t3.\forall t4.\forall m1.\forall m2.\forall x.(\mathsf{Ok}(t1,s) \wedge \mathsf{Abs}(t1,t3,m1,x) \wedge \mathsf{App}(t4,t1,m2)) \Rightarrow \mathsf{Ok}(t4,s) \wedge \mathsf{Apply}(t4,t4,m2,m1)$

- Axioms for apply propagation. Propagates an application to inner types.
  $\forall t.\forall t1.\forall t2.\forall m1.\forall m2.(\mathsf{Ch}(t1,t2) \wedge \mathsf{Apply}(t,t1,m2,m1)) \Rightarrow \mathsf{Apply}(t,t2,m2,m1)$
  $\forall t.\forall t1.\forall t2.\forall m1.\forall m2.(\mathsf{Key}(t1,t2) \wedge \mathsf{Apply}(t,t1,m2,m1)) \Rightarrow \mathsf{Apply}(t,t2,m2,m1)$
  $\forall t.\forall t1.\forall t2.\forall t3.\forall m1.\forall m2.\forall x.(\mathsf{Pair}(t1,x,t2,t3) \wedge \mathsf{Apply}(t,t1,m2,m1)) \Rightarrow \mathsf{Apply}(t,t2,m2,m1) \wedge \mathsf{Apply}(t,t3,m2,m1)$
  $\forall t.\forall t1.\forall s.\forall m1.\forall m2.(\mathsf{Ok}(t1,s) \wedge \mathsf{Apply}(t,t1,m2,m1)) \Rightarrow \mathsf{Apply}(t,s,m2,m1)$

- Unification axioms. This is also a way of assigning types to isolated type variables.
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.(\mathsf{Ch}(\texttt{t1},\texttt{t2}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{Ch}(\texttt{t1'},\texttt{t2})$
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.(\mathsf{Key}(\texttt{t1},\texttt{t2}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{Key}(\texttt{t1'},\texttt{t2})$
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.\forall \texttt{t3}.\forall \texttt{x}.(\mathsf{Pair}(\texttt{t1},\texttt{x},\texttt{t2},\texttt{t3}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{Pair}(\texttt{t1'},\texttt{x},\texttt{t2},\texttt{t3})$
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.(\mathsf{Ok}(\texttt{t1},\texttt{t2}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{Ok}(\texttt{t1'},\texttt{t2})$
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.\forall \texttt{m}.(\mathsf{App}(\texttt{t1},\texttt{t2},\texttt{m}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{App}(\texttt{t1'},\texttt{t2},\texttt{m})$
  $\forall \texttt{t1}.\forall \texttt{t1'}.\forall \texttt{t2}.\forall \texttt{m}.\forall \texttt{x}.(\mathsf{Abs}(\texttt{t1},\texttt{t2},\texttt{m},\texttt{x}) \wedge \mathsf{Teq}(\texttt{t1},\texttt{t1'})) \Rightarrow \mathsf{Abs}(\texttt{t1'},\texttt{t2},\texttt{m},\texttt{x})$

Appendix D

Type Checker Code

D.1 spiparser.mly

```
/* spiparser.mly */
/* Spi Calculus parser */
%{
open Printf
let parse_error s = (* Called by the parser function on error *)
  print_endline ("spiparser: " ^ s);
  flush stdout;;
%}
%token <string> ID
%token <string> LVAR
%token <char> CHAR
%token NIL OK
%token BEGIN END BANG PAR OUT IN NEW DECRYPT SPLIT MATCH
%token CHAN UN KEY TOK TPAIR UNKNOWN
%token EMPTY PAIR COMMA LPAREN RPAREN LBRACE RBRACE COLON SCOLON AS DASH EXPECT

%start process
%start msgtype
%start resource
%start message
%type <Spitree.proc> process
%type <Spitree.typ> msgtype
%type <Spitree.datalog> resource
%type <Spitree.msg> message

%start dprogram
%type <Spitree.term> term
%type <Spitree.literal> literal
%type <Spitree.term list> restlit
%type <Spitree.hclause> hclause
%type <Spitree.literal list> restcl
%type <Spitree.datalog> dprogram
%type <Spitree.hclause list> restprog

%%
process:    NIL  { Spitree.PNil }
          | OUT LPAREN message COMMA message RPAREN  { Spitree.Out($3, $5) }
          | IN LPAREN message COMMA ID RPAREN SCOLON process  { Spitree.In($3, $5, $8) }
          | BANG IN LPAREN message COMMA ID RPAREN SCOLON process  { Spitree.InRepl($4, $6, $9) }
          | NEW ID COLON msgtype SCOLON process  { Spitree.Nu($2, $4, $6) }
          | LPAREN process parrest { if $3 = Spitree.PNil then
                                       $2
                                     else
                                       Spitree.PPara($2, $3) }
          | BEGIN ID LPAREN message RPAREN  { Spitree.Datprog([[($2, [Spitree.SpiMsg($4)])]]) }
          | END ID LPAREN message RPAREN  { Spitree.Expect($2, [Spitree.SpiMsg($4)]) }
          | DECRYPT message AS LBRACE ID COLON msgtype RBRACE message SCOLON process
{ Spitree.Decrypt($2, $5, $7, $9, $11) }
```

```
          | SPLIT message AS LPAREN ID COLON msgtype COMMA ID COLON msgtype RPAREN SCOLON process
            { Spitree.Split($2, $5, $7, $9, $11, $14) }
          | MATCH message COLON msgtype AS LPAREN message COMMA ID COLON msgtype RPAREN SCOLON process
            { Spitree.Match($2, $4, $7, $9, $11, $14) }
          | EXPECT ID LPAREN restlit { if (List.for_all (fun term ->
                                                           match term with
                                                             | Spitree.SpiMsg(_) -> true
                                                             | _ -> false
                                                        ) $4)
                                       then (Spitree.Expect($2, $4))
                                       else raise Parsing.Parse_error }
          | dprogram { Spitree.Datprog($1) }
;
parrest:    PAR process parrest { if $3 = Spitree.PNil then
                                    $2
                                  else
                                    Spitree.PPara($2, $3) }
          | RPAREN { Spitree.PNil }
;
msgtype:    UN  { Spitree.Un }
          | UNKNOWN  { Spitree.Unknown }
          | KEY LPAREN msgtype RPAREN  { Spitree.Key($3) }
          | TPAIR LPAREN ID COLON msgtype COMMA msgtype RPAREN  { Spitree.TPair($3, $5, $7) }
          | TOK LPAREN dprogram RPAREN  { Spitree.TOk($3) }
          | TOK LPAREN resource RPAREN  { Spitree.TOk($3) }
          | CHAN LPAREN msgtype RPAREN  { Spitree.Ch($3) }
;
resource:   EMPTY  { [] }
          | LPAREN resource COMMA resource RPAREN  { List.append $2 $4 }
          | ID LPAREN message RPAREN  { [[($1, [Spitree.SpiMsg($3)])]] }
;
message:    OK  { Spitree.Ok }
          | LBRACE message RBRACE message  { Spitree.Encr($2, $4) }
          | PAIR LPAREN message COMMA message RPAREN  { Spitree.MPair($3, $5) }
          | ID  { Spitree.Name($1) }
;

term:       LVAR { Spitree.Var($1) }
          | message { Spitree.SpiMsg($1) }
;
literal:    ID LPAREN restlit { ($1, $3) };
restlit:    term RPAREN { [$1] }
          | term COMMA restlit { $1::$3 };
hclause:    literal COLON DASH restcl { ($1::$4) }
          | literal { [$1] };
restcl:     literal COMMA restcl { $1::$3 }
          | literal { [$1] };
dprogram:   LBRACE restprog { $2 }
restprog:   hclause SCOLON restprog { $1::$3 }
          | hclause RBRACE { [$1] }
          | RBRACE { [] };
%%
```

D.2 spilexer.mll

```
{
  open Printf
  open Spiparser
}

let id = ['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9']*
let logvar = ['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']*

rule spi_tokens = parse
  | [' ' '\t' '\n'] { spi_tokens lexbuf } (* Skip whitespaces *)
  | "begin"     { BEGIN }
  | "end"       { END }
  | '!'         { BANG }
  | '|'         { PAR }
  | "nil"       { NIL }
  | "ok"        { OK }
  | '('         { LPAREN }
```

```
  | ')'          { RPAREN }
  | '{'          { LBRACE }
  | '}'          { RBRACE }
  | ':'          { COLON }
  | ';'          { SCOLON }
  | ','          { COMMA }
  | "-"          { DASH }
  | "out"        { OUT }
  | "in"         { IN }
  | "new"        { NEW }
  | "decrypt"    { DECRYPT }
  | "split"      { SPLIT }
  | "match"      { MATCH }
  | "expect"     { EXPECT }
  | "as"         { AS }
  | "pair"       { PAIR }
  | "Ch"         { CHAN }
  | "Un"         { UN }
  | "Key"        { KEY }
  | "Ok"         { TOK }
  | "Pair"       { TPAIR }
  | '*'          { UNKNOWN }
  | "Empty"      { EMPTY }
  | id as str    { ID str }
  | logvar as str { LVAR str }
  | _ as c       { printf "Unrecognized character: %c\n" c; CHAR c}
  | eof          { raise End_of_file }
```

D.3 spitree.ml

```
(* Spi-calculus data-structure *)
type name = string;;
type label = string;;
type msg = Name of string
           | Encr of msg*msg
           | MPair of msg*msg
           | Ok;;

(* Datalog structure *)
type pred = string;;
type term = Var of string
            | SpiMsg of msg;;
type literal = pred*(term list);;
type hclause = (literal list);;
type datalog = (hclause list);;

(* Spi-calculus data-structure, continued *)
type reso = Reso of label*msg
            | RPair of reso*reso
            | Empty;;
type typ = Un
           | Key of typ
           | Ch of typ
           | TPair of name*typ*typ
           | TOk of datalog
           | Unknown;;
type proc = Out of msg*msg
            | In of msg*name*proc
            | InRepl of msg*name*proc
            | Nu of name*typ*proc
            | PPara of proc*proc
            | PNil
(*          | Begin of label*msg
            | End of label*msg*)
            | Decrypt of msg*name*typ*msg*proc
            | Split of msg*name*typ*name*typ*proc
            | Match of msg*typ*msg*name*typ*proc
            | Datprog of datalog
            | Expect of literal;;

(* Each element in an environment is either a type or an effect *)
type env_element = Type of name*typ
```

```
          | Effect of datalog;;

(* "Pretty-printing" functions for error-messages and similar.
 * Note that these functions should, theoretically, print in a
 * syntax accepted by the parser.
 *)
let rec string_of_msg msg =
  match msg with
    | Name(x) -> x
    | Encr(m, n) -> Printf.sprintf "{%s}%s" (string_of_msg m) (string_of_msg n)
    | MPair(m, n) -> Printf.sprintf "pair(%s,_%s)" (string_of_msg m) (string_of_msg n)
    | Ok -> "ok";;

let rec string_of_msg_altsyntax msg =
  match msg with
    | Name(x) -> "Name<" ^ x ^ ">"
    | Encr(m, n) -> Printf.sprintf "{%s}%s" (string_of_msg m) (string_of_msg n)
    | MPair(m, n) -> Printf.sprintf "pair<%s,_%s>" (string_of_msg m) (string_of_msg n)
    | Ok -> "ok";;

let rec string_of_reso reso =
  match reso with
    | Reso(l, m) -> Printf.sprintf "%s(%s)" l (string_of_msg m)
    | RPair(r, s) -> Printf.sprintf "(%s,_%s)" (string_of_reso r) (string_of_reso s)
    | Empty -> "Empty";;

(* This went a bit out of hand. Maybe there's a more readable way
 * of formatting lists than using fold?
 * It appears to work this way, but it's something to consider
 * for future work. *)
let string_of_term ?(altsyn = false) term =
  match term with
    | Var(x) -> x
    | SpiMsg(msg) ->
        if altsyn then
          string_of_msg_altsyntax msg
        else
          string_of_msg msg;;

let string_of_literal ?(altsyn = false) (pred, termlist) =
  pred ^ "(" ^ (string_of_term ~altsyn:altsyn (List.hd termlist)) ^
    (List.fold_left (
      fun termstring term ->
        termstring ^ ",_" ^ (string_of_term ~altsyn:altsyn term)) "" (List.tl termlist))
  ^ ")";;

let string_of_hclause ?(altsyn = false) literals =
  if (List.length literals) = 1 then
    string_of_literal ~altsyn:altsyn (List.hd literals)
  else
    (string_of_literal ~altsyn:altsyn (List.hd literals)) ^ "_:-_" ^
        (string_of_literal ~altsyn:altsyn (List.nth literals 1)) ^
      List.fold_left (
        fun litstring lit ->
          litstring ^ ",_" ^ (string_of_literal ~altsyn:altsyn lit)) ""
            (List.tl (List.tl literals));;

let string_of_datalog ?(altsyn = false) clauses =
  if (List.length clauses) > 0 then
    "{" ^ (string_of_hclause (List.hd clauses)) ^
      (List.fold_left (
         fun clstring clause ->
           clstring ^ ";\n" ^ (string_of_hclause ~altsyn:altsyn clause)) "" (List.tl clauses))
    ^ "}"
  else
    "{}";;

let rec string_of_typ typ =
  match typ with
    | Un -> "Un"
    | Key(t) -> Printf.sprintf "Key(%s)" (string_of_typ t)
    | Ch(t) -> Printf.sprintf "Ch(%s)" (string_of_typ t)
    | TPair(x, t, u) -> Printf.sprintf "Pair(%s:%s,_%s)" x (string_of_typ t) (string_of_typ u)
    | TOk(r) -> Printf.sprintf "Ok(%s)" (string_of_datalog r)
```

```
    | Unknown -> "*";;

let rec string_of_proc proc =
  match proc with
    | Out(m, n) -> Printf.sprintf "out(%s, %s)" (string_of_msg m) (string_of_msg n)
    | In(m, x, p) -> Printf.sprintf "in(%s, %s); %s" (string_of_msg m) x (string_of_proc p)
    | InRepl(m, x, p) -> Printf.sprintf "!in(%s, %s); %s" (string_of_msg m) x (string_of_proc p)
    | Nu(x, t, p) -> Printf.sprintf "new %s:%s; %s" x (string_of_typ t) (string_of_proc p)
    | PPara(p, q) -> Printf.sprintf "(%s|\n %s)" (string_of_proc p) (string_of_proc q)
    | PNil -> "nil"
(*  | Begin(l, m) -> Printf.sprintf "begin %s(%s)" l (string_of_msg m)
    | End(l, m) -> Printf.sprintf "end %s(%s)" l (string_of_msg m)*)
    | Decrypt(m, x, t, n, p) -> Printf.sprintf "decrypt %s as {%s:%s}%s; %s"
        (string_of_msg m) x (string_of_typ t) (string_of_msg n) (string_of_proc p)
    | Split(m, x, t, y, u, p) -> Printf.sprintf "split %s as (%s:%s, %s:%s); %s"
        (string_of_msg m) x (string_of_typ t) y (string_of_typ u) (string_of_proc p)
    | Match(m, t, n, x, usub, p) -> Printf.sprintf "match %s:%s as (%s, %s:%s); %s"
        (string_of_msg m) (string_of_typ t) (string_of_msg n) x (string_of_typ usub) (string_of_proc
    | Datprog(d) -> Printf.sprintf "%s" (string_of_datalog d)
    | Expect(f) -> Printf.sprintf "expect %s" (string_of_literal f);;

let rec string_of_env_element env_element =
  match env_element with
    | Type(x, t) -> Printf.sprintf "%s:%s" x (string_of_typ t)
    | Effect(d) -> (string_of_datalog d);;

let rec string_of_env env =
  List.fold_right (fun element env_str ->
                     (string_of_env_element element) ^ ", " ^ env_str) env "";;
```

D.4 aconv.ml

```
open Spitree;;

(* SMap is the basis for the substitution map of the
 * alpha conversion function. The substitution map
 * indicates what each name in the current scope must
 * be substituted with.
 *)
module SMap = Map.Make(String)

(* --- subst_name ---
 * Called on free names
 * Given some name, we do the following:
 * If the name is in the map of substitutions
 * (It is either bound, or an already found free variable):
 * - Substitute the name
 * Else (we have found a new free variable!):
 * - Create a new name, and update the map and accumulator
 *)
let subst_name x accum substMap =
  if SMap.mem x substMap then
    (SMap.find x substMap, accum, substMap)
  else
    let newname = (string_of_int accum) ^ "_" ^ x in
    let newMap = SMap.add x newname substMap in
    (newname, accum + 1, newMap)

(* --- bind ---
 * Called for the binding occurrence of a name:
 * We generate a new unique name and update
 * the substitution map.
 * We return the new name, an updated accumulator
 * and the updated map.
 * We also return a tuple (x, oldname) or (x, ""),
 * containing the mapping we replace in this scope,
 * if any.
 *)
let bind x accum substMap =
  let newname = (string_of_int accum) ^ "_" ^ x in
  let newMap = SMap.add x newname substMap in
  if SMap.mem x substMap then
```

```
    let oldname = SMap.find x substMap in
    (newname, accum + 1, newMap, (x, oldname))
  else
    (newname, accum + 1, newMap, (x, ""))

(* --- unbind ---
 * When exiting a scope, we must call unbind
 * with any mappings from the outer scope that must
 * be reinserted in the substitution map.
 *)
let unbind (x, oldname) substMap =
  if oldname = "" then
    SMap.remove x substMap
  else
    SMap.add x oldname substMap

(* Input: a - accumulator, s - substitution map *)
(* Returns: *_dat - updated datastructure,
            ua - updated accumulator,
            us - updated substitution map *)
let rec msg_subst msg a s = match msg with
  | Name (x) ->
      let (x_dat, ua, us) = (subst_name x a s) in
      (Name (x_dat), ua, us)
  | Encr (m, n) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (n_dat, ua, us) = (msg_subst n ua us) in
      (Encr (m_dat, n_dat), ua, us)
  | MPair (m, n) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (n_dat, ua, us) = (msg_subst n ua us) in
      (MPair (m_dat, n_dat), ua, us)
  | Ok -> (Ok, a, s);;

let rec reso_subst reso a s = match reso with
  | Reso (l, m) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      (Reso (l, m_dat), ua, us)
  | RPair (r, t) ->
      let (r_dat, ua, us) = (reso_subst r a s) in
      let (t_dat, ua, us) = (reso_subst t ua us) in
      (RPair (r_dat, t_dat), ua, us)
  | Empty -> (Empty, a, s);;

let literal_subst (pred, termlist) a s =
  let (subst_termlist, ua, us) =
    List.fold_right (fun term (data_accum, ua, us) ->
                       match term with
                         | Var(x) -> (term::data_accum, ua, us)
                         | SpiMsg(m) ->
                             let (term_dat, ua, us) = (msg_subst m ua us) in
                               (SpiMsg(term_dat)::data_accum, ua, us)
                    ) termlist ([], a, s)
  in
    ((pred, subst_termlist), ua, us);;

let hclause_subst clause a s =
  List.fold_right (fun literal (data_accum, ua, us) ->
                     let (lit_dat, ua, us) = (literal_subst literal ua us) in
                       (lit_dat::data_accum, ua, us)
                  ) clause ([], a, s);;

let datalog_subst data a s =
  List.fold_right (fun clause (data_accum, ua, us) ->
                     let (cl_dat, ua, us) = (hclause_subst clause ua us) in
                       (cl_dat::data_accum, ua, us)
                  ) data ([], a, s);;

let rec typ_subst typ a s = match typ with
  | Un -> (Un, a, s)
  | Key (t) ->
      let (t_dat, ua, us) = (typ_subst t a s) in
      (Key (t_dat), ua, us)
  | Ch (t) ->
```

```
      let (t_dat, ua, us) = (typ_subst t a s) in
      (Ch (t_dat), ua, us)
  | TPair (x, t, u) ->
      let (t_dat, ua, us) = (typ_subst t a s) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (u_dat, ua, us) = (typ_subst u ua us) in
      (TPair (x_dat, t_dat, u_dat), ua, (unbind binding us))
  | TOk (d) ->
      let (d_dat, ua, us) = (datalog_subst d a s) in
      (TOk (d_dat), ua, us)
  | Unknown -> (Unknown, a, s);;

let rec proc_subst proc a s = match proc with
  | Out (m, n) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (n_dat, ua, us) = (msg_subst n ua us) in
      (Out (m_dat, n_dat), ua, us)
  | In (m, x, p) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (In (m_dat, x_dat, p_dat), ua, (unbind binding us))
  | InRepl (m, x, p) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (InRepl (m_dat, x_dat, p_dat), ua, (unbind binding us))
  | Nu (x, t, p) ->
      let (t_dat, ua, us) = (typ_subst t a s) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (Nu (x_dat, t_dat, p_dat), ua, (unbind binding us))
  | PPara (p, q) ->
      let (p_dat, ua, us) = (proc_subst p a s) in
      let (q_dat, ua, us) = (proc_subst q ua us) in
      (PPara (p_dat, q_dat), ua, us)
  | PNil -> (PNil, a, s)
(*  | Begin (l, m) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      (Begin (l, m_dat), ua, us)
  | End (l, m) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      (End (l, m_dat), ua, us)*)
  | Decrypt (m, x, t, n, p) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (t_dat, ua, us) = (typ_subst t ua us) in
      let (n_dat, ua, us) = (msg_subst n ua us) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (Decrypt (m_dat, x_dat, t_dat, n_dat, p_dat), ua, (unbind binding us))
  | Split (m, x, t, y, u, p) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (t_dat, ua, us) = (typ_subst t ua us) in
      let (x_dat, ua, us, xbinding) = (bind x ua us) in
      let (u_dat, ua, us) = (typ_subst u ua us) in
      let (y_dat, ua, us, ybinding) = (bind y ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (Split (m_dat, x_dat, t_dat, y_dat, u_dat, p_dat), ua,
         (unbind xbinding (unbind ybinding us)))
  | Match (m, t, n, x, usub, p) ->
      let (m_dat, ua, us) = (msg_subst m a s) in
      let (t_dat, ua, us) = (typ_subst t ua us) in
      let (n_dat, ua, us) = (msg_subst n ua us) in
      let (usub_dat, ua, us) = (typ_subst usub ua us) in
      let (x_dat, ua, us, binding) = (bind x ua us) in
      let (p_dat, ua, us) = (proc_subst p ua us) in
      (Match (m_dat, t_dat, n_dat, x_dat, usub_dat, p_dat), ua, (unbind binding us))
  | Datprog(d) ->
      let (d_dat, ua, us) = (datalog_subst d a s) in
      (Datprog(d_dat), ua, us)
  | Expect(l) ->
      let (l_dat, ua, us) = (literal_subst l a s) in
      (Expect(l_dat), ua, us);;
```

D.5 auxiliary.ml

```
(* "Auxiliary" functions. Used by the main checker functions. *)

open Spitree

module SSet = Set.Make(String);;

(* Name substitution for dependent pair-types
 * We assume types from an alpha-converted process.
 * Input:
 * typ typ - Type to perform substitution in
 * name current - Name to substitute
 * msg newmsg - Message to substitute name with
 * Output: typ - Substituted type
 *)
let rec subst_name typ current newmsg =
  let rec in_msg msg =
    match msg with
      | Name(x) ->
          if x = current then
            newmsg
          else
            msg
      | Encr(m, n) ->
          Encr(in_msg m, in_msg n)
      | MPair(m, n) ->
          MPair(in_msg m, in_msg n)
      | Ok -> Ok
  in
  let in_literal (pred, termlist) =
    let subst_termlist =
      List.map (fun term ->
                  match term with
                    | Var(x) -> term
                    | SpiMsg(msg) -> SpiMsg(in_msg msg)
                ) termlist
    in
      (pred, subst_termlist)
  in
  let in_hclause clause =
    List.map (fun literal ->
                in_literal literal
              ) clause
  in
  let in_data data =
    List.map (fun hclause ->
                in_hclause hclause
              ) data
(*  let rec in_reso reso =
    match reso with
      | Reso(l, m) -> Reso(l, in_msg m)
      | RPair(r, s) -> RPair(in_reso r, in_reso r)
      | Empty -> Empty*)
  in
  match typ with
    | Un -> Un
    | Key(t) -> Key(subst_name t current newmsg)
    | Ch(t) -> Ch(subst_name t current newmsg)
    | TPair(x, t, u) -> TPair(x, subst_name t current newmsg,
                              subst_name u current newmsg)
    | TOk(d) -> TOk(in_data d)
    | Unknown -> Unknown;;

(* Decide type equality. Dependent pairs are taken into account using subst_name
 * Input:
 * int acc - Accumulator for name substitution in dependent pair types.
 * typ t1, t2 - Types for comparison.
 * Output: bool, indicating equality between t1 and t2 *)
(*exception TypesNotEqual of typ*typ;;*)
let rec type_equal ?(acc=0) t1 t2 =
  match (t1, t2) with
    | (Un, Un) -> true
```

```
    | (Key(t), Key(u)) -> (type_equal ~acc:acc t u)
    | (Ch(t), Ch(u)) -> (type_equal ~acc:acc t u)
    | (TPair(x, left1, right1), TPair(y, left2, right2)) ->
        if (type_equal ~acc:acc left1 left2) then
          let r1_subst = subst_name right1 x (Name(string_of_int acc)) in
          let r2_subst = subst_name right2 y (Name(string_of_int acc)) in
            type_equal ~acc:(acc + 1) r1_subst r2_subst
        else false (*raise (TypesNotEqual (t1, t2))*)
    | (TOk(d1), TOk(d2)) ->
        (List.for_all (fun d1elt ->
                          List.mem d1elt d2
                       ) d1) &&
          (List.for_all (fun d2elt ->
                            List.mem d2elt d1
                         ) d2)
    | _ -> false;;

(* Find the domain of an environment
 * Input: list env_element env - Environment
 * Output: String Set - A set of strings representing the domain of env
 * Note on list-folding:
 * The reason we use fold_left here, is
 * because fold_left is tail-recursive, and because we are
 * folding into a set, order becomes irrelevant.
 * fold_right is used in the effects function to preserve the
 * order when folding into a new list.
 *)
let dom env =
  List.fold_left (fun sset typ ->
                    match typ with
                      | Type(x, _) -> SSet.add x sset
                      | _ -> sset) SSet.empty env;;

(* Flatten effect to a list. Makes it easier to test for
 * effect inclusion and extract effects.
 * Input: reso s - Effect
 * Output: list reso - Flattened list of effects (type Reso)
 *)
(*let rec effect_to_list s =
  match s with
    | Reso(_, _) -> [s]
    | RPair(r, t) ->
        List.append (effect_to_list r) (effect_to_list t)
    | Empty -> [];;*)

(* Extract effects from environment
 * Input: list env_element env - Environment
 * Output: list reso - List of effects from environment
 *)
(*let effects env =
  List.fold_right (fun eff list ->
                     match eff with
                       | Effect(r) -> r::list
                       | Type(_, TOk(s)) -> List.append (effect_to_list s) list
                       | _ -> list
                  ) env [];;*)

let clauses env =
  List.fold_right (fun eff list ->
                     match eff with
                       | Effect(d) -> List.append d list
                       | Type(_, TOk(d)) -> List.append d list
                       | _ -> list
                  ) env [];;

(* Generative type?
 * Input: typ typ - Type
 * Output: Boolean indicating if type typ is generative *)
exception NotGenerative of typ;;
let generative typ =
  match typ with
    | Un -> true
    | Key(_) -> true
    | Ch(_) -> true
```

```
    | _ -> false;;

(* Extract environment from a process. Corresponds to definition of env function
 * Input: proc proc - Process
 * Output: Environment of process proc *)
let rec env_of proc =
  match proc with
    | PPara(p, q) -> List.append (env_of q) (env_of p)
    | Nu(x, t, p) -> List.append (env_of p) [Type(x, t)]
    | Datprog(d) -> [Effect(d)]
(*    | Begin(l, m) -> [Effect(Reso(l, m))]*)
    | _ -> [];;

(* Free name functions
 * fn_msg, fn_res, fn_typ:
 * Input: msg msg, reso res, typ typ, respectively.
 * Output: String Set - Set of strings indicating the free names of the input. *)
let rec fn_msg msg =
  match msg with
    | Name(x) ->
        if x = "*" then
          SSet.empty
        else
          SSet.add x SSet.empty
    | Encr(m, n) -> SSet.union (fn_msg m) (fn_msg n)
    | MPair(m, n) -> SSet.union (fn_msg m) (fn_msg n)
    | _ -> SSet.empty;;
(*let rec fn_res res =
  match res with
    | Reso(_, m) -> fn_msg m
    | RPair(r, s) -> SSet.union (fn_res r) (fn_res s)
    | _ -> SSet.empty;;*)
let fn_literal (_, termlist) =
  List.fold_left (fun accum_set term ->
                    match term with
                      | SpiMsg(msg) ->
                          SSet.union accum_set (fn_msg msg)
                      | _ -> accum_set
                ) SSet.empty termlist;;
let fn_hclause clause =
  List.fold_left (fun accum_set literal ->
                    SSet.union accum_set (fn_literal literal)
                ) SSet.empty clause;;
let fn_data data =
  List.fold_left (fun accum_set clause ->
                    SSet.union accum_set (fn_hclause clause)
                ) SSet.empty data;;
let rec fn_typ typ =
  match typ with
    | Key(t) -> fn_typ t
    | Ch(t) -> fn_typ t
    | TPair(x, t, u) ->
        SSet.union (fn_typ t) (fn_typ (subst_name u x (Name("*"))))
    | TOk(d) -> fn_data d
    | _ -> SSet.empty;;

(* Effect Inclusion
 * Input: list Reso - List of effects. Important: Flatten before passing.
 * Output: bool, indicating if all effects in s1 is included in s2.
 *)
exception EffectInclusion of string;;
let rec effectinclusion s1 s2 =
  match s1 with
    | [] -> true
    | (x::effectrest) ->
        if (List.mem x s2) then
          effectinclusion effectrest s2
        else
          raise (EffectInclusion ("Effect_inclusion_failed:_" ^
                 (string_of_reso x) ^ "_not_permitted"));;

(* Name extraction
 * Input:
 * name x - Name
```

```
 * list env_element env - Environment
 * Output: typ - Indicating the type of x in environment env
 *)
exception NoName of string;;
let rec nametype x env =
  match env with
    | [] -> raise (NoName ("Name " ^ x ^ " not found in environment"))
    | Type(name, typ)::restenv ->
        if x = name then
          typ
        else
          nametype x restenv
    | Effect(_)::restenv ->
        nametype x restenv;;
```

D.6 ssolver/alfp.ml

```
module SMap = Map.Make(String);;
module SSet = Set.Make(String);;

type term = Const of string
            | Var of string
            | FuncApp of string*(term list);;
type pre = PPredicate of string*(term list)
           | NPredicate of string*(term list)
           | PConjunction of pre*pre
           | Disjunction of pre*pre
           | Exists of string*pre
           | Equal of term*term
           | NEqual of term*term;;
type cl = CPredicate of string*(term list)
          | Truth
          | CConjunction of cl*cl
          | Leadsto of pre*cl
          | Forall of string*cl;;

let freevar_from_term ?(bound = SSet.empty) term =
  match term with
    | Var(x) ->
        if (SSet.mem x bound) then
          SSet.empty
        else
          SSet.add x SSet.empty
    | _ -> SSet.empty;;

let freevars_in_termlist ?(bound = SSet.empty) termlist =
  List.fold_left (fun varset term ->
                    let freevar = (freevar_from_term ~bound:bound term) in
                      SSet.union freevar varset
                  ) SSet.empty termlist;;

let rec freevars_in_pre ?(bound = SSet.empty) pre =
  match pre with
    | (PPredicate(_, termlist) | NPredicate(_, termlist)) ->
        freevars_in_termlist ~bound:bound termlist
    | (PConjunction(pre1, pre2) | Disjunction(pre1, pre2)) ->
        SSet.union (freevars_in_pre ~bound:bound pre1) (freevars_in_pre ~bound:bound pre2)
    | Exists(x, pre) ->
        freevars_in_pre ~bound:(SSet.add x bound) pre
    | (Equal(term1, term2) | NEqual(term1, term2)) ->
        SSet.union
          (freevar_from_term ~bound:bound term1)
          (freevar_from_term ~bound:bound term2);;

let rec string_of_term term =
  match term with
    | Const(c) -> c
    | Var(x) -> x
    | FuncApp(func, termlist) ->
        string_of_predicate func termlist
and string_of_predicate rel termlist =
  let (term, trest) = ((List.hd termlist), (List.tl termlist)) in
```

```
    rel ^ "(" ^ (string_of_term term) ^
      (List.fold_left (
         fun termstring term ->
           termstring ^ ", " ^ (string_of_term term)) "" trest)
    ^ ")";;

let rec string_of_pre pred =
  match pred with
    | PPredicate(rel, termlist) ->
        string_of_predicate rel termlist
    | NPredicate(rel, termlist) ->
        "!" ^ (string_of_predicate rel termlist)
    | PConjunction(pre1, pre2) ->
        Printf.sprintf "%s & %s" (string_of_pre pre1) (string_of_pre pre2)
    | Disjunction(pre1, pre2) ->
        Printf.sprintf "%s | %s" (string_of_pre pre1) (string_of_pre pre2)
    | Exists(str, pre) ->
        Printf.sprintf "(E %s.%s)" str (string_of_pre pre)
    | Equal(term1, term2) ->
        Printf.sprintf "%s = %s" (string_of_term term1) (string_of_term term2)
    | NEqual(term1, term2) ->
        Printf.sprintf "%s != %s" (string_of_term term1) (string_of_term term2);;

let rec string_of_cl clause =
  match clause with
    | CPredicate(rel, termlist) ->
        string_of_predicate rel termlist
    | Truth -> "1"
    | CConjunction(cl1, cl2) ->
        Printf.sprintf "%s & %s" (string_of_cl cl1) (string_of_cl cl2)
    | Leadsto(pre, cl) ->
        Printf.sprintf "(%s => %s)" (string_of_pre pre) (string_of_cl cl)
    | Forall(str, cl) ->
        Printf.sprintf "(A %s.%s)" str (string_of_cl cl);;
```

D.7 ssolver/outputparser.mly

```
/* outputparser.mly */
/* Parser for output of the Succinct Solver */
%{
open Printf
let parse_error s = (* Called by the parser function on error *)
  print_endline ("SSolver outputparser: " ^ s);
  flush stdout;;
%}
%token <string> ID
%token <int> NUMBER
%token <char> CHAR
%token RELATION SLASH COLON
%token COMMA LPAREN RPAREN ENDING

%start output
%type <Alfp.term list list Alfp.SMap.t> output
%type <Alfp.term list list> tuples
%type <Alfp.term list> tuple
%type <Alfp.term list> termlist
%type <Alfp.term> term
%%

output:   RELATION ID SLASH NUMBER COLON tuples output { Alfp.SMap.add $2 $6 $7 }
        | RELATION ID SLASH NUMBER COLON output { $6 }
        | RELATION ID SLASH NUMBER COLON tuples ENDING { Alfp.SMap.add $2 $6 Alfp.SMap.empty }
        | RELATION ID SLASH NUMBER COLON ENDING {Alfp.SMap.empty};
tuples:   tuple tuples { $1::$2 }
        | tuple { [$1] };
tuple:    LPAREN termlist RPAREN COMMA { $2 };
termlist: term COMMA termlist { $1::$3 }
        | term { [$1] };
term:     ID { Alfp.Const($1) }
        | ID LPAREN termlist RPAREN { Alfp.FuncApp($1,$3) };
```

D.8 ssolver/outputlexer.mll

```
{
  open Printf
  open Outputparser
}

let id = ['a'-'z' 'A'-'Z' '{'] ['0'-'9' 'a'-'z' 'A'-'Z' '_' '-' '<' '>' '{' '}']*
let number = ['0'-'9']+

rule out_tokens = parse
  | [' ' '\t' '\n'] { out_tokens lexbuf } (* Skip whitespaces *)
  | "Relation"      { RELATION }
  | "/"             { SLASH }
  | ":"             { COLON }
  | ","             { COMMA }
  | "("             { LPAREN }
  | ")"             { RPAREN }
  | "#"             { ENDING }
  | id as str       { ID str }
  | number as num   { NUMBER (int_of_string num) }
  | _ as c          { printf "Unrecognized character: %c\n" c; CHAR c }
  | eof             { raise End_of_file }
```

D.9 datquery.ml

```
let terms_of_terms ?(quotes = true) termlist =
  List.map (fun spiterm ->
              match spiterm with
                | Spitree.Var(x) -> Alfp.Var(x)
                | Spitree.SpiMsg(msg) ->
                    if quotes then
                      Alfp.Const("\"" ^ (Spitree.string_of_msg_altsyntax msg) ^ "\"")
                    else
                      Alfp.Const(Spitree.string_of_msg_altsyntax msg)
           ) termlist;;

let cpred_of_lit (rel, termlist) =
  Alfp.CPredicate(rel, (terms_of_terms termlist));;
let ppred_of_lit (rel, termlist) =
  Alfp.PPredicate(rel, (terms_of_terms termlist));;
let conj_of_ppreds literals =
  List.fold_left (fun ppred litfromlist ->
                    Alfp.PConjunction(ppred, ppred_of_lit litfromlist)
                 ) (ppred_of_lit (List.hd literals)) (List.tl literals)

let alfp_of_dat datprog =
  let clauses =
    List.map (fun hornclause ->
                if List.length hornclause = 1 then
                  let literal = List.hd hornclause in
                    cpred_of_lit literal
                else
                  let literal = List.hd hornclause in
                  let litlist = List.tl hornclause in
                  let clpred = cpred_of_lit literal in
                  let conj = conj_of_ppreds litlist in
                  let freevars = Alfp.freevars_in_pre conj in
                  let hornclause = Alfp.Leadsto(conj, clpred) in
                    Alfp.SSet.fold (fun var clause ->
                                      Alfp.Forall(var, clause)
                                   ) freevars hornclause
             ) datprog
  in
    List.fold_left (fun cl clfromlist ->
                      Alfp.CConjunction(cl, clfromlist)
                   ) (List.hd clauses) (List.tl clauses);;

exception IllegalHornClause of string;;
let clauses_to_literals datprog =
  List.map (fun clause ->
```

```
            if List.length clause = 1 then
              List.hd clause
            else
              raise (IllegalHornClause "Expected fact, but got a Horn Clause")
        ) datprog;;

(*exception ThisIsABug of string;;*)
exception QueryFail of Spitree.datalog*Spitree.hclause;;
let querydat datprog queries ssolverheap =
  let alfpclause = alfp_of_dat datprog in
  let deducedclause = Ssolver.deducible_facts alfpclause ssolverheap
  in
    if
      (List.for_all (fun query ->
                       let cpred_of_lit_mod (rel, termlist) =
                         Alfp.CPredicate(rel, (terms_of_terms ~quotes:false termlist)) in
                       let query_alfp = cpred_of_lit_mod query in
                         (*print_endline ("Query: " ^ (Alfp.string_of_cl query_alfp));*)
                         List.mem query_alfp deducedclause) queries)
    then true else
      raise (QueryFail (datprog, queries))
;;
```

D.10 spichecker.ml

```
open Spitree
open Auxiliary
open Datquery
open Printf
open Unix;;

let arguments =
object
  val mutable heap = "ssolver/heap"
  val mutable version = false
  val mutable help = false
  method give_version = version <- true
  method give_help = help <- true
  method print_vers = version
  method print_help = help
  method set_heap h = heap <- h
  method get_heap = heap
end;;

(* Judgment of well-formedness of an environment
 * Input: list env_element env - Environment to judge
 * Output: bool, indicating well-formedness
 *)
exception EnvFailure of string;;
exception TypeDomainInclusionFailure of typ*(env_element list);;
exception DuplicateDeclaration of string*(env_element list);;
exception EffectDomainInclusionFailure of datalog*(env_element list);;
let rec env_judgment env =
  match env with
    | [] -> true
    | Type(x, t)::env_rest ->
        let typdominclusion = (SSet.subset (fn_typ t) (dom env_rest)) in
        let notduplicate = (not (SSet.mem x (dom env_rest))) in
          if not typdominclusion then
            raise (TypeDomainInclusionFailure (t, env_rest));
          if not notduplicate then
            raise (DuplicateDeclaration (x, env_rest));
          (env_judgment env_rest)
    | Effect(d)::env_rest ->
        let effdominclusion = (SSet.subset (fn_data d) (dom env_rest)) in
          if not effdominclusion then
            raise (EffectDomainInclusionFailure (d, env_rest));
          (env_judgment env_rest);;

(* Judgment of the goodness of a message
 * Input:
 * list env_element env - Environment
```

```
 * msg msg - Message
 * typ typ - Type
 * Output: bool, indicating if msg is good with type typ in environment env
 *)
exception MsgTypeFailure of string;;
exception MessageTypeMismatch of msg*typ;;
exception NameDomainInclusionFailure of string*(env_element list);;
let rec msg_judgment env msg typ =
  let pair_judgment env m t n u =
    (msg_judgment env m t) &
    (msg_judgment env n u) in
  let encrypt_judgment env m t n ntype =
    (msg_judgment env m t) &
    (msg_judgment env n ntype) in
    try
      match msg with
        | MPair(m, n) ->
          (match typ with
             | TPair(x, t, u) ->
                pair_judgment env m t n (subst_name u x m)
             | Un ->
                pair_judgment env m Un n Un
             | _ -> raise (MsgTypeFailure (sprintf "Expected type Pair or Un, not type %s"
                                             (string_of_typ typ))))
        | Encr(m, Name(x)) ->
          (if (typ = Un) then
             let n, keytype = (Name(x), (nametype x env)) in
               (match keytype with
                  | Key(t) ->
                     encrypt_judgment env m t n keytype
                  | Un ->
                     encrypt_judgment env m Un n Un
                  | _ -> raise (MsgTypeFailure
                          (sprintf "Expected type Key or Un for message %s, not type %s"
                           (string_of_msg n) (string_of_typ keytype))))
           else
             raise (MsgTypeFailure (sprintf "Expected type Un, not type %s"
                                      (string_of_typ typ))))
        | Encr(m, n) ->
          (if (typ = Un) then
             encrypt_judgment env m Un n Un
           else
             raise (MsgTypeFailure (sprintf "Expected type Un, not type %s"
                                      (string_of_typ typ))))
        | Name(x) ->
          let ntype = nametype x env in
          let domaininc = (SSet.mem x (dom env)) in
          let equality = (type_equal ntype typ) in
            if not domaininc then
              raise (MsgTypeFailure (sprintf "Name %s not declared in environment %s"
                                       x (string_of_env env)));
            if not equality then
              raise (MsgTypeFailure
                       (sprintf "Given type %s does not match type %s from environment."
                        (string_of_typ typ) (string_of_typ ntype)));
            (env_judgment env)
        | Ok ->
          (match typ with
             | Un ->
                env_judgment env
             | TOk(d) ->
                let domsubset = SSet.subset (fn_typ typ) (dom env) in
                  if not domsubset then
                    raise (MsgTypeFailure(
                      sprintf "Free names of type %s not a subset of domain of environment %s"
                                (string_of_typ typ) (string_of_env env)))
                  else
                    (querydat (clauses env) (clauses_to_literals d) arguments#get_heap) &
                      (env_judgment env)
             | _ -> raise (MsgTypeFailure "ok token must have type Un or Ok(S)"))
    with
      | TypeDomainInclusionFailure(t, subenv) ->
        raise
        (MsgTypeFailure(sprintf
```

```
  "Message %s:\nAll free names in type %s not included in sub-environment\n%s of environment\n%s"
    (string_of_msg msg) (string_of_typ t) (string_of_env subenv) (string_of_env env)))
    | DuplicateDeclaration(x, subenv) ->
        raise (MsgTypeFailure
                (sprintf "Message %s:\nDuplicate declaration of name \"%s\" in environment\n%s"
                 (string_of_msg msg) x (string_of_env env)))
    | MsgTypeFailure(str) ->
        raise (MsgTypeFailure (sprintf "Message %s:\n%s" (string_of_msg msg) str))
    | QueryFail(datprog,queryclause) ->
        raise (MsgTypeFailure
              (sprintf "Message %s:\nDatalog program %s does not entail %s"
               (string_of_msg msg) (string_of_datalog datprog) (string_of_hclause queryclause)))
;;

(* Judgment of the goodness of a process
 * Input:
 * list env_element env - Environment
 * proc proc - Process
 * Output: bool, indicating if process proc is good in environment env
 *)
exception ProcTypeFailure of string;;
exception TypeFailure of string;;
let rec proc_judgment env proc =
  let out_judgment env m mtype n t =
    (msg_judgment env m mtype) &
      (msg_judgment env n t) in
  let in_judgment env m mtype x t p =
    (msg_judgment env m mtype) &
      (proc_judgment (Type(x, t)::env) p) in
  let decrypt_judgment env m n ntype y t p =
    (msg_judgment env m Un) &
      (msg_judgment env n ntype) &
      (proc_judgment (Type(y, t)::env) p) in
  let split_judgment env m mtype x t y u p =
    (msg_judgment env m mtype) &
      (proc_judgment (Type(y, u)::Type(x, t)::env) p) in
  let match_judgment env m mtype n t y substu p =
    (msg_judgment env m mtype) &
      (msg_judgment env n t) &
      (proc_judgment (Type(y, substu)::env) p) in
    try
      match proc with
        | Out(Name(x), n) ->
            let m, chantype = (Name(x), (nametype x env)) in
              (match chantype with
                 | Ch(t) ->
                     out_judgment env m chantype n t
                 | Un ->
                     out_judgment env m Un n Un
                 | _ -> raise (MsgTypeFailure
                                 (sprintf "Expected type Ch or Un for message %s, not type %s"
                                  (string_of_msg m) (string_of_typ chantype))))
        | Out(m, n) ->
            out_judgment env m Un n Un
        | (In(Name(y), x, p) | InRepl(Name(y), x, p)) ->
            let m, chantype = (Name(y), (nametype y env)) in
              (match chantype with
                 | Ch(t) ->
                     in_judgment env m chantype x t p
                 | Un ->
                     in_judgment env m Un x Un p
                 | _ -> raise (MsgTypeFailure
                                 (sprintf "Expected type Ch or Un for message %s, not type %s"
                                  (string_of_msg m) (string_of_typ chantype))))
        | (In(m, x, p) | InRepl(m, x, p)) ->
            in_judgment env m Un x Un p
        | Nu(x, t, p) ->
            (generative t) &
              (proc_judgment (Type(x, t)::env) p)
        | PPara(p, q) ->
            (proc_judgment (List.append (env_of q) env) p) &&
              (proc_judgment (List.append (env_of p) env) q)
        | PNil -> true
        | Datprog(d) ->
```

```
    let domsubset = SSet.subset (fn_data d) (dom env) in
      if not domsubset then
        raise (MsgTypeFailure
          (sprintf "Free names of program %s not a subset of domain of environment %s"
           (string_of_datalog d) (string_of_env env)))
      else
        (env_judgment env)
| Expect(l) ->
    let domsubset = SSet.subset (fn_literal l) (dom env) in
      if not domsubset then
        raise (MsgTypeFailure
          (sprintf "Free names of literal %s not a subset of domain of environment %s"
           (string_of_literal l) (string_of_env env)))
      else
        (querydat (clauses env) [l] arguments#get_heap) &&
          (env_judgment env)
| Decrypt(m, y, t1, Name(x), p) ->
    let n, keytype = (Name(x), (nametype x env)) in
      (match keytype with
         | Key(t2) ->
             if type_equal t1 t2 then
               decrypt_judgment env m n keytype y t1 p
             else
               raise (MsgTypeFailure
                (sprintf "Decrypt failure: Key type %s incompatible with message type %s"
                 (string_of_typ keytype) (string_of_typ t1)))
         | Un ->
             decrypt_judgment env m n Un y Un p
         | _ -> raise (MsgTypeFailure
                         (sprintf "Expected type Key or Un for message %s, not type %s"
                          (string_of_msg n) (string_of_typ keytype))))
| Decrypt(m, y, t, n, p) ->
    decrypt_judgment env m n Un y Un p
| Split(Name(z), x, t, y, u, p) ->
    let m, pairtype = (Name(z), (nametype z env)) in
      (match pairtype with
         | TPair(x_dep, t_dep, u_dep) ->
             let subst_u = subst_name u_dep x_dep (Name(x)) in
             let subst_pairtype = TPair(x, t, subst_u) in
               if (type_equal t_dep t) then
                 if (type_equal subst_u u) then
                   (split_judgment env m subst_pairtype x t y u p)
                 else
                   raise (MsgTypeFailure
                            (sprintf "Split failure: Type %s incompatible with type %s"
                             (string_of_typ u_dep) (string_of_typ u)))
               else
                 raise (MsgTypeFailure
                          (sprintf "Split failure: Type %s incompatible with type %s"
                           (string_of_typ t_dep) (string_of_typ t)))
         | Un ->
             split_judgment env m Un x Un y Un p
         | _ -> raise (MsgTypeFailure
                         (sprintf "Expected type Pair or Un for message %s, not type %s"
                          (string_of_msg m) (string_of_typ pairtype))))
| Split(MPair(m1, m2), x, t, y, u, p) ->
    let m, pairtype = (MPair(m1, m2), TPair(x, t, u)) in
      split_judgment env m pairtype x t y u p
| Split(m, x, t, y, u, p) ->
    split_judgment env m Un x Un y Un p
| Match(m, mtype, n, y, usub, p) ->
    (match mtype with
       | Un ->
           match_judgment env m Un n Un y Un p
       | TPair(x, t, u) ->
           if type_equal (subst_name u x n) usub then
             (match_judgment env m mtype n t y usub p)
           else
             raise (MsgTypeFailure
                      (sprintf "type\n%s does not match substituted type\n%s."
                       (string_of_typ usub) (string_of_typ (subst_name u x n))));
       | _ ->
           raise (MsgTypeFailure
                    (sprintf "Expected type Pair or Un for message %s, not type %s"
```

```
                                (string_of_msg m) (string_of_typ mtype))))
    with
      | MsgTypeFailure(str) ->
          raise (TypeFailure (sprintf "In process:\n%s\n%s" (string_of_proc proc) str))
      | QueryFail(datprog,queryclause) ->
          raise (TypeFailure (sprintf "In process %s:\nDatalog program %s does not entail %s"
                                (string_of_proc proc) (string_of_datalog datprog)
                                   (string_of_hclause queryclause)))
;;

let name = "Spi Policy Checker" in
let version = "0.9" in

let help_msg =
  "Usage: " ^ Sys.argv.(0) ^ " [options] filename\n\n" ^
    "Options:\n" ^
    "  --solverheap <heapfile>    compiled heap of the Succinct Solver\n" ^
    "                               (default: ssolver/heap)\n" ^
    "  --help                     display this help and exit\n" ^
    "  --version                  output version information and exit"
in

  if (Array.length Sys.argv) = 1 then
    print_endline (help_msg)
  else
    let infile = Array.fold_left (fun prev_opt this_opt ->
                                match prev_opt with
                                  | "" -> this_opt
                                  | "--solverheap" ->
                                      arguments#set_heap this_opt;
                                      ""
                                  | "--version" ->
                                      arguments#give_version;
                                      this_opt
                                  | _ ->
                                      arguments#give_help;
                                      this_opt
                             ) "" (Array.sub Sys.argv 1 ((Array.length Sys.argv) - 1))
    in
      if infile = "--version" then
        arguments#give_version
      else if infile = "--help" then
        arguments#give_help;
      if ((not arguments#print_help) && (not arguments#print_vers)) then
        let filechan = open_in infile in
        let lexbuf = Lexing.from_channel filechan in
        let ast = Spiparser.process Spilexer.spi_tokens lexbuf in
        let (ast_substituted, accum, substMap) = Aconv.proc_subst ast 0 Aconv.SMap.empty in
        let freevars =
          Aconv.SMap.fold (fun _ name list ->
                             Type(name, Un) :: list) substMap [] in
          print_endline "Alpha converted process:";
          print_endline ((string_of_proc ast_substituted) ^ "\n");
          print_endline "Free names:";
          Aconv.SMap.iter (Printf.printf "%s: %s\n") substMap;
          try
            let judgment = proc_judgment freevars ast_substituted in
              if judgment then
                Printf.printf "\nCongratulations! This process is robustly safe.\n";
          with
            | TypeFailure(str) -> print_endline ("Failure:\n" ^ str)
            (*| ProcTypeFailure(str) -> print_endline ("Failure:\n" ^ str)
            | MsgTypeFailure(str) -> print_endline ("Failure:\n" ^ str)*)

      else
        if arguments#print_vers then
          print_endline (name ^ " " ^ version)
        else
          print_endline (help_msg);;
```

Appendix E

Implementation of Type Inference

E.1 spiparser.mly

Modification of `spiparser.mly` from D.1.

```
/* spiparser.mly */
/* Spi Calculus parser */
%{
open Printf
let parse_error s = (* Called by the parser function on error *)
  print_endline ("spiparser: " ^ s);
  flush stdout;;
%}
%token <string> ID
%token <string> LVAR
%token <char> CHAR
%token NIL OK
%token BEGIN END BANG PAR OUT IN NEW DECRYPT SPLIT MATCH
%token CHAN UN KEY TOK TPAIR UNKNOWN
%token EMPTY PAIR COMMA LPAREN RPAREN LBRACE RBRACE COLON SCOLON AS DASH

%start process
%start message
%type <Spitree.proc> process
%type <Spitree.msg> message

%%
process:    NIL  { Spitree.PNil }
          | OUT LPAREN message COMMA message RPAREN  { Spitree.Out($3, $5) }
          | IN LPAREN message COMMA ID RPAREN SCOLON process  { Spitree.In($3, $5, $8) }
          | BANG IN LPAREN message COMMA ID RPAREN SCOLON process  { Spitree.InRepl($4, $6, $9) }
          | NEW ID SCOLON process  { Spitree.Nu($2, Spitree.Unknown, $4) }
          | LPAREN process parrest { if $3 = Spitree.PNil then
                                       $2
                                     else
                                       Spitree.PPara($2, $3) }
          | BEGIN ID LPAREN message RPAREN  { Spitree.Begin($2, $4) }
          | END ID LPAREN message RPAREN  { Spitree.End($2, $4) }
          | DECRYPT message AS LBRACE ID RBRACE message SCOLON process
              { Spitree.Decrypt($2, $5, Spitree.Unknown, $7, $9) }
          | SPLIT message AS LPAREN ID COMMA ID RPAREN SCOLON process
              { Spitree.Split($2, $5, Spitree.Unknown, $7, Spitree.Unknown, $10) }
          | MATCH message AS LPAREN message COMMA ID RPAREN SCOLON process
              { Spitree.Match($2, Spitree.Unknown, $5, $7, Spitree.Unknown, $10) }
;
parrest:    PAR process parrest { if $3 = Spitree.PNil then
                                    $2
                                  else
```

```
                                        Spitree.PPara($2, $3) }
          | RPAREN { Spitree.PNil }
;
message:    OK  { Spitree.Ok }
          | LBRACE message RBRACE message  { Spitree.Encr($2, $4) }
          | PAIR LPAREN message COMMA message RPAREN  { Spitree.MPair($3, $5) }
          | ID  { Spitree.Name($1) }
;

%%
```

E.2 spilexer.mll

Modification of `spilexer.mll` from D.2.

```
{
  open Printf
  open Spiparser
}

let id = ['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9']*
let logvar = ['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']*

rule spi_tokens = parse
  | [' ' '\t' '\n'] { spi_tokens lexbuf } (* Skip whitespaces *)
  | "begin"     { BEGIN }
  | "end"       { END }
  | '!'         { BANG }
  | '|'         { PAR }
  | "nil"       { NIL }
  | "ok"        { OK }
  | '('         { LPAREN }
  | ')'         { RPAREN }
  | '{'         { LBRACE }
  | '}'         { RBRACE }
  | ':'         { COLON }
  | ';'         { SCOLON }
  | ','         { COMMA }
  | "-"         { DASH }
  | "out"       { OUT }
  | "in"        { IN }
  | "new"       { NEW }
  | "decrypt"   { DECRYPT }
  | "split"     { SPLIT }
  | "match"     { MATCH }
(*  | "expect"    { EXPECT }*)
  | "as"        { AS }
  | "pair"      { PAIR }
  | "Ch"        { CHAN }
  | "Un"        { UN }
  | "Key"       { KEY }
  | "Ok"        { TOK }
  | "Pair"      { TPAIR }
  | '*'         { UNKNOWN }
  | "Empty"     { EMPTY }
  | id as str   { ID str }
  | logvar as str { LVAR str }
  | _ as c      { printf "Unrecognized character: %c\n" c; CHAR c}
  | eof         { raise End_of_file }
```

E.3 spitree.ml

Modification of `spitree.ml` from D.3.

```
(* Spi-calculus data-structure *)
type name = string;;
type label = string;;
type msg = Name of string
```

```
            | Encr of msg*msg
            | MPair of msg*msg
            | Ok;;

(* Spi-calculus data-structure, continued *)
type reso = Reso of label*msg
            | RPair of reso*reso
            | Empty;;
type typ = Un
            | Key of typ
            | Ch of typ
            | TPair of name*typ*typ
            | TOk of reso
            | Unknown;;
type proc = Out of msg*msg
            | In of msg*name*proc
            | InRepl of msg*name*proc
            | Nu of name*typ*proc
            | PPara of proc*proc
            | PNil
            | Begin of label*msg
            | End of label*msg
            | Decrypt of msg*name*typ*msg*proc
            | Split of msg*name*typ*name*typ*proc
            | Match of msg*typ*msg*name*typ*proc
(*          | Datprog of datalog
            | Expect of literal*);;

(* Each element in an environment is either a type or an effect *)
type env_element = Type of name*typ
            | Effect of reso;;

(* "Pretty-printing" functions for error-messages and similar.
 * Note that these functions should, theoretically, print in a
 * syntax accepted by the parser.
 *)
let rec string_of_msg msg =
  match msg with
    | Name(x) -> x
    | Encr(m, n) -> Printf.sprintf "{%s}%s" (string_of_msg m) (string_of_msg n)
    | MPair(m, n) -> Printf.sprintf "pair(%s,_%s)" (string_of_msg m) (string_of_msg n)
    | Ok -> "ok";;

let rec string_of_msg_altsyntax msg =
  match msg with
    | Name(x) -> "Name<" ^ x ^ ">"
    | Encr(m, n) -> Printf.sprintf "{%s}%s" (string_of_msg m) (string_of_msg n)
    | MPair(m, n) -> Printf.sprintf "pair<%s,_%s>" (string_of_msg m) (string_of_msg n)
    | Ok -> "ok";;

let rec string_of_reso reso =
  match reso with
    | Reso(l, m) -> Printf.sprintf "%s(%s)" l (string_of_msg m)
    | RPair(r, s) -> Printf.sprintf "(%s,_%s)" (string_of_reso r) (string_of_reso s)
    | Empty -> "Empty";;

let rec string_of_typ typ =
  match typ with
    | Un -> "Un"
    | Key(t) -> Printf.sprintf "Key(%s)" (string_of_typ t)
    | Ch(t) -> Printf.sprintf "Ch(%s)" (string_of_typ t)
    | TPair(x, t, u) -> Printf.sprintf "Pair(%s:%s,_%s)" x (string_of_typ t) (string_of_typ u)
    | TOk(r) -> Printf.sprintf "Ok(%s)" (string_of_reso r)
    | Unknown -> "*";;

let rec string_of_proc proc =
  match proc with
    | Out(m, n) -> Printf.sprintf "out(%s,_%s)" (string_of_msg m) (string_of_msg n)
    | In(m, x, p) -> Printf.sprintf "in(%s,_%s);_%s" (string_of_msg m) x (string_of_proc p)
    | InRepl(m, x, p) -> Printf.sprintf "!in(%s,_%s);_%s" (string_of_msg m) x (string_of_proc p)
    | Nu(x, t, p) -> Printf.sprintf "new_%s:%s;_%s" x (string_of_typ t) (string_of_proc p)
    | PPara(p, q) -> Printf.sprintf "(%s|\n_%s)" (string_of_proc p) (string_of_proc q)
    | PNil -> "nil"
    | Begin(l, m) -> Printf.sprintf "begin_%s(%s)" l (string_of_msg m)
```

```
    | End(l, m) -> Printf.sprintf "end %s(%s)" l (string_of_msg m)
    | Decrypt(m, x, t, n, p) -> Printf.sprintf "decrypt %s as {%s:%s}%s; %s"
        (string_of_msg m) x (string_of_typ t) (string_of_msg n) (string_of_proc p)
    | Split(m, x, t, y, u, p) -> Printf.sprintf "split %s as (%s:%s, %s:%s); %s"
        (string_of_msg m) x (string_of_typ t) y (string_of_typ u) (string_of_proc p)
    | Match(m, t, n, x, usub, p) -> Printf.sprintf "match %s:%s as (%s, %s:%s); %s"
        (string_of_msg m) (string_of_typ t) (string_of_msg n) x (string_of_typ usub) (string_of_proc p)
(*    | Datprog(d) -> Printf.sprintf "%s" (string_of_datalog d)
    | Expect(f) -> Printf.sprintf "expect %s" (string_of_literal f)*);;

let rec string_of_env_element env_element =
  match env_element with
    | Type(x, t) -> Printf.sprintf "%s:%s" x (string_of_typ t)
    | Effect(r) -> (string_of_reso r);;

let rec string_of_env env =
  List.fold_right (fun element env_str ->
                     (string_of_env_element element) ^ "," ^ env_str) env "";;
```

E.4 aconv.ml

Identical to aconv.ml from D.4.

E.5 constraints.ml

```
open Printf;;

type label = string;;
type name = string;;
type typevariable = string;;
type effectvariable = string;;

type reso = Reso of label*Spitree.msg
            | RPair of reso*reso
            | Empty;;

type env_element = Type of name*typevariable
                   | UnType of name
(*                 | EffectVar of effectvariable*)
                   | Effect of reso;;

type env = (env_element list);;

type basetype = Ch of typevariable
           | Pair of name*typevariable*typevariable
           | Un
           | Key of typevariable
           | Ok of effectvariable (* Addition to the syntax *)
           | Abstraction of typevariable*Spitree.msg*typevariable
           | Application of typevariable*Spitree.msg;;

type typeconstraint = MayEqual of typevariable*basetype
                      | Equal of typevariable*basetype
                      | NotUn of typevariable (* Addition to the syntax *)
                      | Generative of typevariable
                      | NotGenerative of typevariable
                      (*| NotFN of name*Spitree.proc*)
                      | HasType of name*typevariable
                      | Fail;;

type effectconstraint = EffectIncl of label*Spitree.msg*env
                        | Instantiates of env*effectvariable
                        | InEnv of Spitree.reso*env;;

type envconstraint = WellFormed of env
                     | NamesInDom of Spitree.msg*env
                     | NotInDom of name*env;;
```

```
type formula = TypeC of typeconstraint
             | EffectC of effectconstraint
             | EnvC of envconstraint
             | And of formula*formula
             | Or of formula*formula
             | Leadsto of formula*formula
             | Forall of name*formula
             | Conjunction of (formula list)
             | CommentConjunction of string*(formula list)
             | True
;;

let rec string_of_reso reso =
  match reso with
    | Reso(l, m) -> sprintf "%s(%s)" l (Spitree.string_of_msg m)
    | RPair(s1, s2) ->
        sprintf "(%s, %s)" (string_of_reso s1) (string_of_reso s2)
    | Empty -> "Empty"
;;

let string_of_env_element element =
  match element with
    | Type(x, t) -> sprintf "%s:%s" x t
    | UnType(x) -> sprintf "%s:Un" x
    | Effect(reso) -> sprintf "%s" (string_of_reso reso)
;;

let string_of_env env =
  if (List.length env) > 0 then
    "(" ^ (string_of_env_element (List.hd env)) ^
      (List.fold_left (
          fun envstring element ->
            envstring ^ ", " ^ (string_of_env_element element)) "" (List.tl env))
    ^ ")"
  else
    "()"
;;
```

E.6 constraintgen.ml

```
open Constraints;;

module SMap = Map.Make(String);;

class genstring prefix =
object
  val mutable counter = 0
  method fresh =
    let countval = counter in
      counter <- counter + 1;
      prefix ^ (string_of_int countval)
end;;

let typevar = new genstring "T";;
let effectvar = new genstring "R";;
let namegen = new genstring "x";;

let messages =
object
  val mutable msglist = []
  method add msg =
    msglist <- msg::msglist
  method get =
    msglist
end;;

let environments =
object
  val mutable envlist = []
  method add env =
    envlist <- env::envlist
  method get =
```

```
    envlist
end;;

exception NoName of string;;
let rec nametype x env =
  match env with
    | [] -> raise (NoName ("Name " ^ x ^ " not found in environment"))
    | Type(name, typvar)::restenv ->
        if x = name then
          Type(name, typvar)
        else
          nametype x restenv
    | UnType(name)::restenv ->
        if x = name then
          UnType(name)
        else
          nametype x restenv
    | Effect(_)::restenv ->
        nametype x restenv
(*  | EffectVar(_)::restenv ->
        nametype x restenv*)
(*  | Empty::restenv ->
        nametype x restenv*)
;;

let rec env_of proc =
  match proc with
    | Spitree.PPara(p, q) -> List.append (env_of q) (env_of p)
    | Spitree.Nu(x, _, p) -> List.append (env_of p) [Type(x, typevar#fresh)]
    | Spitree.Begin(l, m) -> [Effect(Reso(l, m))]
    | _ -> [];;

let rec collect_list_mayeq formula =
  let from_typec typec =
    match typec with
      | MayEqual(tvar, basetype) -> [(tvar, basetype)]
      | _ -> []
  in
    match formula with
      | TypeC(tconstraint) -> from_typec tconstraint
      | (EffectC(_) | EnvC(_)) -> []
      | (And(f1, f2) | Or(f1, f2) | Leadsto(f1, f2)) ->
          List.append (collect_list_mayeq f1) (collect_list_mayeq f2)
      | Forall(_, f) -> collect_list_mayeq f
      | (Conjunction(flist) | CommentConjunction(_, flist)) ->
          List.fold_left (fun coll_list f ->
                            List.append (collect_list_mayeq f) coll_list
                         ) [] flist
      | True -> []

let table_of_list_mayeq may_list =
  let may_table = Hashtbl.create 100 in
    List.iter (fun (tvar, basetype) ->
                 Hashtbl.add may_table tvar basetype
              ) may_list;
    may_table

(*let propagate_abstraction formula =
  let may_table = table_of_list_mayeq (collect_list_mayeq formula) in
  let in_typec typec =

    match formula with
      |
*)
exception Bug of string;;
exception TypeMismatch of string;;
let rec constraint_of_msg env msg =
  environments#add env;
  match msg with
    | Spitree.Encr(m, n) ->
        let (ut1, psi1) = constraint_of_msg env m in
        let (ut2, psi2) = constraint_of_msg env n in
        let ut3 = typevar#fresh in
          (ut3,
```

```
            CommentConjunction("(Msg_Encr)",
              [TypeC(Equal(ut3, Un));
               TypeC(MayEqual(ut2, Key(ut1)));
               Leadsto(TypeC(Equal(ut2, Un)),
                       TypeC(Equal(ut1, Un)));
               psi1;
               psi2
              ]
            ))
    | Spitree.MPair(m, n) ->
        let (ut1, psi1) = constraint_of_msg env m in
        let (ut2, psi2) = constraint_of_msg env n in
        let ut = typevar#fresh in
        let ut2p = typevar#fresh in
        let x = namegen#fresh in
          (ut,
           CommentConjunction("(Msg_Pair)",
             [TypeC(MayEqual(ut, Pair(x, ut1, ut2p)));
              TypeC(MayEqual(ut2p,Abstraction(ut2, m, x)));
              Leadsto(And(TypeC(Equal(ut1,Un)),TypeC(Equal(ut2,Un))),
                      TypeC(Equal(ut,Un)));
              psi1;
              psi2
             ]
           ))
    | Spitree.Name(x) ->
        let xtype = nametype x env in
          (match xtype with
             | Type(_, tv) ->
                 (tv,
                  CommentConjunction("(Msg_Name)",
                    [TypeC(HasType(x,tv));
                     EnvC(WellFormed(env))]
                  ))
             | UnType(_) ->
                 let ut = typevar#fresh in
                   (ut,
                    CommentConjunction("(Msg_Name_Un)",
                      [TypeC(HasType(x,ut));
                       TypeC(Equal(ut,Un));
                       EnvC(WellFormed(env))]
                    ))
             | _ -> raise (Bug ("Received_effect_or_empty_from_the_nametype_function." ^
                                "This_should_never_happen._nametype_is_not_constructed_that_way."))
          )
    | Spitree.Ok ->
        let r = effectvar#fresh in
        let ut = typevar#fresh in
          (ut,
           CommentConjunction("Msg_Ok",
             [TypeC(HasType("ok", ut));
              Leadsto(TypeC(Equal(ut,Un)),
                      EnvC(WellFormed(env)));
              TypeC(MayEqual(ut,Ok(r)));
              Leadsto(TypeC(NotUn(ut)),
                      And(EffectC(Instantiates(env, r)), EnvC(WellFormed(env))))]
           ))
;;

let rec constraint_of_proc env proc =
  environments#add env;
  match proc with
    | Spitree.Out(m, n) ->
        let (ut1, psi1) = constraint_of_msg env m in
        let (ut2, psi2) = constraint_of_msg env n in
          messages#add m; messages#add n;
          CommentConjunction("(Proc_Out)",
            [TypeC(MayEqual(ut1,Ch(ut2)));
             Leadsto(TypeC(Equal(ut1, Un)),
                     TypeC(Equal(ut2, Un)));
             psi1;
             psi2]
          )
    | (Spitree.In(m, x, p)|Spitree.InRepl(m, x, p)) ->
```

```
    let (ut1, psi1) = constraint_of_msg env m in
    let ut2 = typevar#fresh in
    let psi2 = constraint_of_proc (Type(x, ut2)::env) p in
      messages#add m;
      CommentConjunction("(Proc_In)",
        [TypeC(MayEqual(ut1,Ch(ut2)));
         Leadsto(TypeC(Equal(ut1, Un)),
                 TypeC(Equal(ut2, Un)));
         psi1;
         psi2]
      )
| Spitree.Nu(x, _, p) ->
    let ut = typevar#fresh in
    let psi1 = constraint_of_proc (Type(x, ut)::env) p in
      CommentConjunction("(Proc_Res)",
                         [EnvC(NotInDom(x, env));
                          psi1;
                          Leadsto(TypeC(NotGenerative(ut)), TypeC(Fail))
                         ]
                        )
| Spitree.PPara(p, q) ->
    let psi1 = constraint_of_proc (List.append (env_of q) env) p in
    let psi2 = constraint_of_proc (List.append (env_of p) env) q in
      CommentConjunction("(Proc_Par)",
                         [psi1; psi2])
| Spitree.PNil ->
    True
| Spitree.Begin(l, m) ->
    messages#add m;
    CommentConjunction("(Proc_Begin)",
                       [EnvC(NamesInDom(m,env));
                        EnvC(WellFormed(env))])
| Spitree.End(l, m) ->
    messages#add m;
    CommentConjunction("(Proc_End)",
                       [EffectC(EffectIncl(l,m,env));
                        EnvC(WellFormed(env))])
| Spitree.Decrypt(m, y, _, n, p) ->
    let (ut1, psi1) = constraint_of_msg env m in
    let (ut2, psi2) = constraint_of_msg env n in
    let ut3 = typevar#fresh in
    let psi3 = constraint_of_proc (Type(y, ut3)::env) p in
      messages#add m; messages#add n;
      CommentConjunction("(Proc_Decrypt)",
        [TypeC(Equal(ut1,Un));
         TypeC(MayEqual(ut2,Key(ut3)));
         Leadsto(TypeC(Equal(ut2,Un)),
                 TypeC(Equal(ut3,Un)));
         psi1;
         psi2;
         psi3
        ]
      )
| Spitree.Split(m, x, _, y, _, p) ->
    let (ut0, psi1) = constraint_of_msg env m in
    let ut1, ut2 = typevar#fresh, typevar#fresh in
    let psi2 = constraint_of_proc (Type(y, ut2)::Type(x, ut1)::env) p in
      messages#add m;
      CommentConjunction("(Proc_Split)",
        [TypeC(MayEqual(ut0,Pair(x, ut1, ut2)));
         Leadsto(TypeC(Equal(ut0,Un)),
                 And(TypeC(Equal(ut1,Un)), TypeC(Equal(ut2,Un))));
         psi1;
         psi2
        ]
      )
| Spitree.Match(m, _, n, y, _, p) ->
    let (ut1, psi1) = constraint_of_msg env m in
    let (ut2, psi2) = constraint_of_msg env n in
    let ut0, ut3, ut4 = typevar#fresh, typevar#fresh, typevar#fresh in
    let psi3 = constraint_of_proc (Type(y, ut3)::env) p in
    let x = namegen#fresh in
      messages#add m; messages#add n;
      CommentConjunction("(Proc_Match)",
```

```
            [TypeC(MayEqual(ut1,Pair(x, ut0, ut4)));
             TypeC(MayEqual(ut3,Application(ut4,n)));
             Leadsto(TypeC(Equal(ut1, Un)),
                     Conjunction([TypeC(Equal(ut2, Un));
                                  TypeC(Equal(ut0, Un));
                                  TypeC(Equal(ut3, Un));
                                  TypeC(Equal(ut4, Un))
                                 ])
                    );
             psi1;
             psi2;
             psi3
            ]
          )
;;
```

E.7 axioms.ml

```
open Alfp;;

type con_element = Clause of cl
                 | Comment of string;;
type commented_conjunction = (con_element list)

(* Shorthand functions. Using these can make axioms and encodings more readable in the code. *)
exception ClauseBuildError of string;;

(* Build a conjunction of clauses from a list *)
let rec cland_of_list flist =
  match flist with
    | (hd::[]) -> hd
    | (hd::tl) -> ClAnd(hd, cland_of_list tl)
    | [] -> raise (ClauseBuildError ("Bug: Please do not feed cland_of_list with an empty list. " ^
                                     "Be nice and give it something to work with."))

(* Build a conjunction of preconditions from a list *)
let rec preand_of_list flist =
  match flist with
    | (hd::[]) -> hd
    | (hd::tl) -> PreAnd(hd, preand_of_list tl)
    | [] -> raise (ClauseBuildError ("Bug: Please do not feed preand_of_list with an empty list. " ^
                                     "Be nice and give it something to work with."))

(* Prefix a clause with a number of universal quantifiers *)
let rec aquantify qlist clause =
  match qlist with
    | [] -> clause
    | (hd::tl) -> Forall(hd, aquantify tl clause)

(* Build a list of variables from a list of strings *)
let rec varlist strlist =
  List.fold_right (fun str variables ->
                     Var(str)::variables
                  ) strlist []

(* Build a list of constants from a list of strings *)
let rec constlist strlist =
  List.fold_right (fun str constants ->
                     Const(str)::constants
                  ) strlist []

(* - Relations - *)

(* Messages *)
let tpairrel = "TermPair"
let encrel = "Enc"
let namerel = "Name"
let tokrel = "OkTerm"

(* Types *)
let keyrel = "Key"
let pairrel = "Pair"
```

```
let chrel = "Ch"
let unrel = "Un"
let okrel = "Ok"

let typrel = "Type"

let maypf = "Q"

(* Effects *)
let effectrel = "Effect"

(* Generativity *)
let genrel = "Gen"
let nongenrel = "NonGen"

(* Equality *)
let eqrel = "eq"
let neqrel = "neq"

(* Type equality *)
let teqrel = "Teq"

(* Substitution for equality for dependent types. *)
let subsrel = "Subs"

let absrel = "Abs"
let apprel = "App"

let applyrel = "Apply"

(* Free names *)
let fnrel = "Fn"

(* Name in domain *)
let domrel = "Dom"

(* Elements of environments *)
let envrel = "Env"

(* Effects and non-effects *)
let effrel = "Eff"
let noneffrel = "NonEff"

(* Simple and non-simple environments *)
let simplerel = "Simple"
let nonsimplerel = "NonSimple"

(* Dependency *)
let deprel = "Dep"
let deprefrel = "DepRef"
let envdeprel = "EnvDep"

(* Initial and non-initial effects *)
let initrel = "Initial"
let noninitrel = "NotInitial"

(* Failure - Unary. Give it a string explaining failure. *)
let failrel = "Fail"

(* Admissibility *)
let admitsrel = "Admits"
let effadmitsrel = "EffAdmits"

(* Ranking of each relation *)
let ranklist = [(domrel, 1);      (* Domain relation *)
                (eqrel, 1);       (* Equality, rank 1 *)
                (neqrel, 2);      (* Inequality, rank 2 *)
                (tpairrel, 2);    (* Term relations, rank 2 *)
                (encrel, 2);
                (namerel, 2);
                (tokrel, 2);
                (subsrel, 2);     (* Subs relation, rank 2 *)
                (unrel, 2);       (* Un relation, rank 2 *)
                (keyrel, 3);      (* Other type relations, rank 3 *)
```

```
                  (keyrel ^ maypf, 3);
                  (pairrel, 3);
                  (pairrel ^ maypf, 3);
                  (chrel, 3);
                  (chrel ^ maypf, 3);
                  (okrel, 3);
                  (okrel ^ maypf, 3);
                  (absrel, 3);
                  (absrel ^ maypf, 3);
                  (apprel, 3);
                  (apprel ^ maypf, 3);
                  (applyrel, 3);
                  (typrel, 3);      (* Type relation *)
                  (teqrel, 3);      (* Type equality, rank 3 *)
                  (genrel, 3);      (* Generativity and non-generativity, rank 3 *)
                  (nongenrel, 3);
                  (envdeprel, 3);   (* Dependency of effect variables on environments and each other, ra
                  (deprel, 4);
                  (deprefrel, 4);
                  (envrel, 4);      (* Environment relation, rank 4 *)
                  (simplerel, 4);   (* Simplicity relations, rank 4 *)
                  (nonsimplerel, 4);
                  (effectrel, 4);   (* Effect relations, rank 4 *)
                  (effrel, 4);
                  (noneffrel, 4);
                  (initrel, 4);     (* Initial and noninitial effects, rank 4 *)
                  (noninitrel, 4);
                  (fnrel, 5);
                  (admitsrel, 6);   (* Admissibility, rank 6 *)
                  (effadmitsrel, 6);
                  (failrel, 6)
                 ]

(* - Axioms - *)

(* Equality and inequality *)
let eqneq = ClAnd(Forall("x",
                         CPredicate(eqrel, [Var("x"); Var("x")])),
                  Forall("x", Forall("y",
                                     Leadsto(NegPredicate(eqrel, [Var("x"); Var("y")]),
                                             CPredicate(neqrel, [Var("x"); Var("y")]))))));;

let axioms_eqneq = [Comment("Equality and inequality on names. We simply describe that " ^
                            "every distinct name is equal to itself, " ^
                            "and not equal to any other name.");
                    Clause(eqneq)]

(* Generativity *)
let ax_un = Forall("t",
                   Leadsto(PPredicate(unrel, [Var("t")]),
                           CPredicate(genrel, [Var("t")])))
let ax_key = Forall("t1", Forall("t2",
                                 Leadsto(PPredicate(keyrel, [Var("t1"); Var("t2")]),
                                         CPredicate(genrel, [Var("t1")]))))
let ax_ch = Forall("t1", Forall("t2",
                                Leadsto(PPredicate(chrel, [Var("t1"); Var("t2")]),
                                        CPredicate(genrel, [Var("t1")]))))
let ax_pair = Forall("t1", Forall("x", Forall("t2", Forall("t3",
                Leadsto(PPredicate(pairrel, [Var("t1"); Var("x"); Var("t2"); Var("t3")]),
                        CPredicate(nongenrel, [Var("t1")]))))))
let ax_ok = Forall("t", Forall("r",
                               Leadsto(PPredicate(okrel, [Var("t"); Var("r")]),
                                       CPredicate(nongenrel, [Var("t")]))))

let axioms_gen = [Comment("Generativity and non-generativity." ^
                          "These formulae map types to the generativity relations.");
                  Clause(ax_un);
                  Clause(ax_key);
                  Clause(ax_ch);
                  Clause(ax_pair);
                  Clause(ax_ok)
                 ]

(* Type equality *)
```

```
(* Note: Doing this kind of type equality for the Un type would be pointless,
 * as Un does not contain any type or effect. *)
let pair_pre = PreAnd(PPredicate(pairrel, [Var("t1'"); Var("x1"); Var("t2"); Var("t3")]),
                      PreAnd(PPredicate(pairrel, [Var("t1''"); Var("x2"); Var("t4"); Var("t5")]),
                             PPredicate(teqrel, [Var("t1'"); Var("t1''")])))
let pair_cl = ClAnd(CPredicate(teqrel, [Var("t2"); Var("t4")]),
                    CPredicate(teqrel, [Var("t3"); Var("t5")]))
let ax_teq_pair = aquantify ["x1"; "x2"; "t1'"; "t1''"; "t2"; "t3"; "t4"; "t5"]
  (Leadsto(pair_pre,
           pair_cl)
  )

let key_pre = PreAnd(PPredicate(keyrel, [Var("t1'"); Var("t2")]),
                     PreAnd(PPredicate(keyrel, [Var("t1''"); Var("t3")]),
                            PPredicate(teqrel, [Var("t1'"); Var("t1''")])))
let ax_teq_key = aquantify ["t1'"; "t1''"; "t2"; "t3"]
  (Leadsto(key_pre,
           CPredicate(teqrel, [Var("t2"); Var("t3")]))
  )

let ch_pre = PreAnd(PPredicate(chrel, [Var("t1'"); Var("t2")]),
                    PreAnd(PPredicate(chrel, [Var("t1''"); Var("t3")]),
                           PPredicate(teqrel, [Var("t1'"); Var("t1''")])))
let ax_teq_ch = aquantify ["t1'"; "t1''"; "t2"; "t3"]
  (Leadsto(ch_pre,
           CPredicate(teqrel, [Var("t2"); Var("t3")]))
  )

let ok_pre = PreAnd(PPredicate(okrel, [Var("t1'"); Var("s1")]),
                    PreAnd(PPredicate(okrel, [Var("t1''"); Var("s2")]),
                           PPredicate(teqrel, [Var("t1'"); Var("t1''")])))
let ax_teq_ok = aquantify ["t1'"; "t1''"; "s1"; "s2"]
  (Leadsto(ok_pre,
           CPredicate(teqrel, [Var("s1"); Var("s2")]))
  )

let axioms_teq = [Comment("Type equality. We say that if two types are equal," ^
                          "then their inner types are also equal.");
                  Clause(ax_teq_pair);
                  Clause(ax_teq_key);
                  Clause(ax_teq_ch);
                  Clause(ax_teq_ok)
                 ]

(* General type equality axioms *)
let ax_teq_reflex1 = Forall("t", CPredicate(teqrel, [Var("t"); Var("t")]))
let ax_teq_reflex2 = aquantify ["t1"; "t2"]
  (Leadsto(PPredicate(teqrel, [Var("t1"); Var("t2")]),
           CPredicate(teqrel, [Var("t2"); Var("t1")])))

let ax_teq_trans = aquantify ["t1"; "t2"; "t3"]
  (Leadsto(PreAnd(PPredicate(teqrel, [Var("t1"); Var("t2")]),
                  PPredicate(teqrel, [Var("t2"); Var("t3")])),
           CPredicate(teqrel, [Var("t1"); Var("t3")])))

let axioms_genteq = [Comment("Type equality is both reflexive and transitive.");
                     Clause(ax_teq_reflex1);
                     Clause(ax_teq_reflex2);
                     Clause(ax_teq_trans)
                    ]

(* Dependent types *)

(* Substitution on terms *)
let ax_tok_subs = aquantify ["n"; "m'"; "m"]
  (Leadsto(PPredicate(tokrel, [Var("n")]),
           CPredicate(subsrel, varlist ["n"; "n"; "m'"; "m"])))

let ax_name_subs = aquantify ["m'"; "m"]
  (Leadsto(PPredicate(namerel, [Var("m")]),
           CPredicate(subsrel, varlist ["m'"; "m"; "m'"; "m"])))

let ax_tpair_subs = aquantify ["n"; "n1'"; "n1"; "m'"; "m"; "n2'"; "n2"; "n'"]
  (cland_of_list [
```

```
    Leadsto(preand_of_list [PPredicate(subsrel, varlist ["n1'"; "n1"; "m'"; "m"]);
                            PPredicate(subsrel, varlist ["n2'"; "n2"; "m'"; "m"]);
                            PPredicate(tpairrel, varlist ["n"; "n1"; "n2"]);
                            PPredicate(tpairrel, varlist ["n'"; "n1'"; "n2'"])],
            CPredicate(subsrel, varlist ["n'"; "n"; "m'"; "m"])
           )
  ])

let ax_enc_subs = aquantify ["n"; "n1'"; "n1"; "m'"; "m"; "n2'"; "n2"; "n'"]
  (cland_of_list [
    Leadsto(preand_of_list [PPredicate(subsrel, varlist ["n1'"; "n1"; "m'"; "m"]);
                            PPredicate(subsrel, varlist ["n2'"; "n2"; "m'"; "m"]);
                            PPredicate(encrel, varlist ["n"; "n1"; "n2"]);
                            PPredicate(encrel, varlist ["n'"; "n1'"; "n2'"])],
            CPredicate(subsrel, varlist ["n'"; "n"; "m'"; "m"])
           )
  ])

let axioms_subs = [Comment("Substitution on terms");
                   Clause(ax_tok_subs);
                   Clause(ax_name_subs);
                   Clause(ax_tpair_subs);
                   Clause(ax_enc_subs)
                  ]

(* Substitution on effects *)

let ax_effect_subs = aquantify ["n"; "n'"; "l"; "m"; "m'"; "s"; "s'"]
  (Leadsto(PreAnd(PPredicate (effectrel, varlist ["l"; "n"; "s'"]),
                  PPredicate (subsrel, varlist ["n'"; "n"; "m'"; "m"])
                 ),
           CPredicate(effectrel, varlist ["l"; "n'"; "s"])
          )
  )

let axioms_effsubs = [Comment("Substitution on effects");
                      Clause(ax_effect_subs)
                     ]

(* Abstraction on types *)
(* Woah. The definitions of abstraction and application are a bit confused.
 * Hey! Subject for next supervisor meeting. *)
(*let ax_abs_subs = aquantify ["n"; "n'"; "m"; "m'"; "t"; "t'"]
  (Leadsto(preand_of_list [PPredicate(typrel, varlist ["n"; "t"]);
                           PPredicate(subsrel, varlist ["t'"; "t"; "m'"; "m"]);
                           PPredicate(typrel, varlist ["n'"; "t"])
                          ],
           CPredicate(absrel, varlist ["n'"; "t"; "m'"; "m"])
          )
  )

let axioms = List.append axioms [Comment("Abstraction on types");
                                 Clause(ax_abs_subs)
                                ]*)

(* Application on types *)
(*let ax_app_subs = aquantify ["v"; "v'"; "n"; "m"; "m1"; "m2"; "m3"]
  (
  )*)

(* Free names of terms *)
let ax_name_fn = Forall("n",
                        Leadsto(PPredicate(namerel, [Var("n")]),
                                CPredicate(fnrel, varlist ["n"; "n"])))

let ax_tpair_fn = aquantify ["n"; "n1"; "n2"; "m"]
  (Leadsto(PreAnd(Or(PPredicate(fnrel, varlist ["m"; "n1"]),
                     PPredicate(fnrel, varlist ["m"; "n2"])),
                  PPredicate(tpairrel, varlist ["n"; "n1"; "n2"])),
           CPredicate(fnrel, varlist ["m"; "n"]))
  )

let ax_enc_fn = aquantify ["n"; "n1"; "n2"; "m"]
  (Leadsto(PreAnd(Or(PPredicate(fnrel, varlist ["m"; "n1"]),
```

```
                    PPredicate(fnrel, varlist ["m"; "n2"])),
                  PPredicate(encrel, varlist ["n"; "n1"; "n2"])),
          CPredicate(fnrel, varlist ["m"; "n"]))
  )

let axioms_fn = [Comment("Free names of terms. A name is always free in itself. " ^
                         "For pairs and encryptions, if a name is free in either element, " ^
                         "it is free for the pair, or encryption.");
                 Clause(ax_name_fn);
                 Clause(ax_tpair_fn);
                 Clause(ax_enc_fn);
                ]

(* Free names of types *)

let ax_ch_fn = aquantify ["t"; "t1"; "m"]
  (Leadsto(PreAnd(PPredicate(chrel, varlist ["t"; "t1"]),
                  PPredicate(fnrel, varlist ["m"; "t1"])
                 ),
           CPredicate(fnrel, varlist ["m"; "t"])
          )
  )

let ax_key_fn = aquantify ["t"; "t1"; "m"]
  (Leadsto(PreAnd(PPredicate(keyrel, varlist ["t"; "t1"]),
                  PPredicate(fnrel, varlist ["m"; "t1"])
                 ),
           CPredicate(fnrel, varlist ["m"; "t"])
          )
  )

let ax_pair_fn = aquantify ["t"; "t1"; "t2"; "x"; "m"]
  (Leadsto(preand_of_list [PPredicate(pairrel, varlist ["t"; "x"; "t1"; "t2"]);
                           Or(PPredicate(fnrel, varlist ["m"; "t1"]),
                              PPredicate(fnrel, varlist ["m"; "t2"]));
                           PPredicate(neqrel, varlist ["m"; "x"])
                          ],
           CPredicate(fnrel, varlist ["m"; "t"])
          )
  )

let ax_ok_fn = aquantify ["t"; "s"; "m"]
  (Leadsto(PreAnd(PPredicate(okrel, varlist ["t"; "s"]),
                  PPredicate(fnrel, varlist ["m"; "s"])
                 ),
           CPredicate(fnrel, varlist ["m"; "t"])
          )
  )

let ax_effects_fn = aquantify ["m"; "n"; "l"; "s"]
  (Leadsto(PreAnd(PPredicate(fnrel, varlist ["n"; "m"]),
                  PPredicate(effectrel, varlist ["l"; "m"; "s"])
                 ),
           CPredicate(fnrel, varlist ["n"; "s"])
          )
  )

let axioms_typfn = [Comment("Free names of types");
                    Clause(ax_ch_fn);
                    Clause(ax_key_fn);
                    Clause(ax_pair_fn);
                    Clause(ax_ok_fn);
                    Clause(ax_effects_fn);
                   ]

let ax_may_key = aquantify ["t"; "t1"]
  (Leadsto(PreAnd(PPredicate(keyrel ^ maypf, varlist ["t"; "t1"]),
                  NegPredicate(unrel, [Var("t")])),
           CPredicate(keyrel, varlist ["t"; "t1"])))

let ax_may_pair = aquantify ["t"; "x"; "t1"; "t2"]
  (Leadsto(PreAnd(PPredicate(pairrel ^ maypf, varlist ["t"; "x"; "t1"; "t2"]),
                  NegPredicate(unrel, [Var("t")])),
           CPredicate(pairrel, varlist ["t"; "x"; "t1"; "t2"])))
```

```
let ax_may_ch = aquantify ["t"; "t1"]
  (Leadsto(PreAnd(PPredicate(chrel ^ maypf, varlist ["t"; "t1"]),
                  NegPredicate(unrel, [Var("t")])),
           CPredicate(chrel, varlist ["t"; "t1"])))

let ax_may_ok = aquantify ["t"; "s"]
  (Leadsto(PreAnd(PPredicate(okrel ^ maypf, varlist ["t"; "s"]),
                  NegPredicate(unrel, [Var("t")])),
           CPredicate(okrel, varlist ["t"; "s"])))

let ax_may_abs = aquantify ["v"; "u"; "m"; "t"]
  (Leadsto(PreAnd(PPredicate(absrel ^ maypf, varlist ["v"; "u"; "m"; "t"]),
                  NegPredicate(unrel, [Var("v")])),
           CPredicate(absrel, varlist ["v"; "u"; "m"; "t"])))

let ax_may_app = aquantify ["v"; "u"; "n"]
  (Leadsto(PreAnd(PPredicate(apprel ^ maypf, varlist ["v"; "u"; "n"]),
                  NegPredicate(unrel, [Var("v")])),
           CPredicate(apprel, varlist ["v"; "u"; "n"])))

let axioms_may = [Comment("Possible␣types.␣These␣axioms␣were␣added␣to␣the␣ones␣of␣\\cite{HHWIP},␣" ^
                          "to␣describe␣how␣the␣$=$␣and␣$\\mayeq$␣type␣relations␣interact.␣" ^
                          "The␣$\\Rel{" ^ unrel ^ "}$␣relation␣is␣populated␣by␣the␣generation␣" ^
                          "rules␣in␣a␣stratum␣prior␣to␣all␣the␣other␣type␣relations.␣" ^
                          "In␣that␣stratum,␣all␣the␣types␣that␣must␣be␣\\Un{}␣are␣found,␣" ^
                          "and␣a␣stratum␣later,␣the␣other␣type␣relations␣are␣populated␣from␣" ^
                          "the␣$\\mayeq$␣relations␣to␣fill␣in␣those␣types␣that␣" ^
                          "are␣not␣\\Un{}.");
                  Clause(ax_may_key);
                  Clause(ax_may_pair);
                  Clause(ax_may_ch);
                  Clause(ax_may_ok);
                  Clause(ax_may_abs);
                  Clause(ax_may_app)
                 ]

(* Effect and Non-effect axioms *)
let ax_eff_ok = aquantify ["t"; "s"]
  (Leadsto(PPredicate(okrel, varlist ["t"; "s"]),
           CPredicate(effrel, [Var("t")])))

let ax_eff_pair = aquantify ["x"; "t"; "t1"; "t2"]
  (Leadsto(PPredicate(pairrel, varlist ["t"; "x"; "t1"; "t2"]),
           CPredicate(noneffrel, [Var("t")])))

let ax_eff_key = aquantify ["t"; "t1"]
  (Leadsto(PPredicate(keyrel, varlist ["t"; "t1"]),
           CPredicate(noneffrel, [Var("t")])))

let ax_eff_ch = aquantify ["t"; "t1"]
  (Leadsto(PPredicate(chrel, varlist ["t"; "t1"]),
           CPredicate(noneffrel, [Var("t")])))

let ax_eff_un =
  Forall("t",
         Leadsto(PPredicate(unrel, [Var("t")]),
                 CPredicate(noneffrel, [Var("t")]))
        )

let ax_eff_teq = aquantify ["t1"; "t2"]
  (Leadsto(PreAnd(PPredicate(effrel, [Var("t1")]),
                  PPredicate(teqrel, varlist ["t1"; "t2"])),
           CPredicate(effrel, [Var("t2")])))

let ax_noneff_teq = aquantify ["t1"; "t2"]
  (Leadsto(PreAnd(PPredicate(noneffrel, [Var("t1")]),
                  PPredicate(teqrel, varlist ["t1"; "t2"])),
           CPredicate(noneffrel, [Var("t2")])))

let axioms_eff = [Comment("Axioms␣for␣deciding␣which␣types␣contain␣effects,␣and␣which␣do␣not.");
                  Clause(ax_eff_ok);
                  Clause(ax_eff_pair);
                  Clause(ax_eff_key);
```

```
                  Clause(ax_eff_ch);
                  Clause(ax_eff_un);
                  Clause(ax_eff_teq);
                  Clause(ax_noneff_teq);
                 ]

(* Simple and nonsimple environments *)
let ax_simple = aquantify ["e"; "t"; "x"]
  (Leadsto(PreAnd(PPredicate(envrel, varlist ["x"; "t"; "e"]),
                  PPredicate(effrel, [Var("t")])),
           CPredicate(simplerel, [Var("e")])))

let ax_nonsimple =
  Forall("e",
         Leadsto(Exists("t", Exists("x",
                                    PreAnd(PPredicate(envrel, varlist ["x"; "t"; "e"]),
                                           PPredicate(noneffrel, [Var("t")])))),
                 CPredicate(nonsimplerel, [Var("e")])))

let axioms_simplicity = [Comment("We decide which environments are simple " ^
                                 "and which are non-simple.");
                         Clause(ax_simple);
                         Clause(ax_nonsimple)
                        ]

let ax_dep = aquantify ["e"; "x"; "t"; "s"; "s'"]
  (Leadsto(preand_of_list [PPredicate(envrel, varlist ["x"; "t"; "e"]);
                           PPredicate(okrel, varlist ["t"; "s'"]);
                           PPredicate(envdeprel, varlist ["s"; "e"])],
           CPredicate(deprel, varlist ["s"; "s'"])))

let ax_dep_trans = aquantify ["s1"; "s2"; "s3"]
  (Leadsto(preand_of_list [PPredicate(deprel, varlist ["s1"; "s2"]);
                           PPredicate(deprel, varlist ["s2"; "s3"]);],
           CPredicate(deprel, varlist ["s1"; "s3"])))

let axioms_dependency = [Comment("We find dependencies between effect variables.");
                         Clause(ax_dep);
                         Clause(ax_dep_trans)
                        ]

let ax_initial = aquantify ["s"; "e"]
  (Leadsto(PreAnd(PPredicate(simplerel, [Var("e")]),
                  PPredicate(envdeprel, varlist ["s"; "e"])),
           CPredicate(initrel, [Var("s")])))

let ax_noninitial = aquantify ["s"; "e"]
  (Leadsto(PreAnd(PPredicate(envdeprel, varlist ["s"; "e"]),
                  PPredicate(nonsimplerel, [Var("e")])),
           CPredicate(noninitrel, [Var("s")])
          ))

let axioms_initiality = [Comment("We find initial and non-initial effect variables.");
                         Clause(ax_initial);
                         Clause(ax_noninitial)
                        ]

let ax_admits = aquantify ["m"; "e"; "n"]
  (Leadsto(Or(NegPredicate(fnrel, varlist ["n"; "m"]),
              PPredicate(domrel, varlist ["n"; "e"])),
           CPredicate(admitsrel, varlist ["e"; "m"])))

let ax_effadmits = aquantify ["m"; "e"; "s"]
  (Leadsto(PreAnd(PPredicate(admitsrel, varlist ["e"; "m"]),
                  PPredicate(envdeprel, varlist ["s"; "e"])),
           CPredicate(effadmitsrel, varlist ["s"; "m"])))

let axioms_admissibility = [Comment("We find which messages are admitted by " ^
                                    "environments and effects");
                            Clause(ax_admits);
                            Clause(ax_effadmits)
                           ]

let ax_abs_ch = aquantify ["t1"; "t1'"; "t2"; "m"; "x"]
```

```
  (Leadsto(PreAnd(PPredicate(chrel, varlist ["t1"; "t2"]),
                  PPredicate(absrel, varlist ["t1'"; "t1"; "m"; "x"])),
           CPredicate(absrel, varlist ["t1'"; "t2"; "m"; "x"])))

let ax_abs_key = aquantify ["t1"; "t1'"; "t2"; "m"; "x"]
  (Leadsto(PreAnd(PPredicate(keyrel, varlist ["t1"; "t2"]),
                  PPredicate(absrel, varlist ["t1'"; "t1"; "m"; "x"])),
           CPredicate(absrel, varlist ["t1'"; "t2"; "m"; "x"])))

let ax_abs_pair = aquantify ["t1"; "t1'"; "t2"; "t3"; "t2'"; "m"; "x1"; "x2"]
  (Leadsto(PreAnd(PPredicate(pairrel, varlist ["t1"; "x1"; "t2"; "t3"]),
                  PPredicate(absrel, varlist ["t1'"; "t1"; "m"; "x2"])),
           ClAnd (CPredicate(absrel, varlist ["t1'"; "t2"; "m"; "x2"]),
                  CPredicate(absrel, varlist ["t1'"; "t3"; "m"; "x2"]))))

let ax_abs_ok = aquantify ["t1"; "t1'"; "s"; "m"; "x"]
  (Leadsto(PreAnd(PPredicate(okrel, varlist ["t1"; "s"]),
                  PPredicate(absrel, varlist ["t1'"; "t1"; "m"; "x"])),
           CPredicate(absrel, varlist ["t1'"; "s"; "m"; "x"])))

let axioms_abstraction = [Comment("Axioms for abstraction. " ^
                                  "Propagates abstraction into abstracted type.");
                          Clause(ax_abs_ch);
                          Clause(ax_abs_key);
                          Clause(ax_abs_pair);
                          Clause(ax_abs_ok)
                         ]

let ax_app_ch = aquantify ["t1"; "t2"; "t3"; "t4"; "m1"; "m2"; "x"]
  (Leadsto(preand_of_list [PPredicate(chrel, varlist ["t1"; "t2"]);
                           PPredicate(absrel, varlist ["t1"; "t3"; "m1"; "x"]);
                           PPredicate(apprel, varlist ["t4"; "t1"; "m2"])
                          ],
           ClAnd(CPredicate(chrel, varlist["t4"; "t2"]),
                 CPredicate(applyrel, varlist["t4"; "t4"; "m2"; "m1"]))))

let ax_app_key = aquantify ["t1"; "t2"; "t3"; "t4"; "m1"; "m2"; "x"]
  (Leadsto(preand_of_list [PPredicate(keyrel, varlist ["t1"; "t2"]);
                           PPredicate(absrel, varlist ["t1"; "t3"; "m1"; "x"]);
                           PPredicate(apprel, varlist ["t4"; "t1"; "m2"])
                          ],
           ClAnd(CPredicate(keyrel, varlist["t4"; "t2"]),
                 CPredicate(applyrel, varlist["t4"; "t4"; "m2"; "m1"]))))

let ax_app_pair = aquantify ["t1"; "t2"; "t3"; "t4"; "t5"; "m1"; "m2"; "x"; "x1"]
  (Leadsto(preand_of_list [PPredicate(pairrel, varlist ["t1"; "x1"; "t2"; "t5"]);
                           PPredicate(absrel, varlist ["t1"; "t3"; "m1"; "x"]);
                           PPredicate(apprel, varlist ["t4"; "t1"; "m2"])
                          ],
           ClAnd(CPredicate(pairrel, varlist["t4"; "x1"; "t2"; "t5"]),
                 CPredicate(applyrel, varlist["t4"; "t4"; "m2"; "m1"]))))

let ax_app_ok = aquantify ["t1"; "s"; "t3"; "t4"; "m1"; "m2"; "x"]
  (Leadsto(preand_of_list [PPredicate(okrel, varlist ["t1"; "s"]);
                           PPredicate(absrel, varlist ["t1"; "t3"; "m1"; "x"]);
                           PPredicate(apprel, varlist ["t4"; "t1"; "m2"])
                          ],
           ClAnd(CPredicate(okrel, varlist["t4"; "s"]),
                 CPredicate(applyrel, varlist["t4"; "t4"; "m2"; "m1"]))))

let axioms_application = [Comment("Axioms for application. " ^
                                  "Indicates application of a type abstraction.");
                          Clause(ax_app_ch);
                          Clause(ax_app_key);
                          Clause(ax_app_pair);
                          Clause(ax_app_ok)
                         ]

let ax_apply_ch = aquantify ["t"; "t1"; "t2"; "m1"; "m2"]
  (Leadsto(PreAnd(PPredicate(chrel, varlist ["t1"; "t2"]),
                  PPredicate(applyrel, varlist ["t"; "t1"; "m2"; "m1"])),
           CPredicate(applyrel, varlist ["t"; "t2"; "m2"; "m1"])))

let ax_apply_key = aquantify ["t"; "t1"; "t2"; "m1"; "m2"]
```

```
  (Leadsto(PreAnd(PPredicate(keyrel, varlist ["t1"; "t2"]),
                  PPredicate(applyrel, varlist ["t"; "t1"; "m2"; "m1"])),
           CPredicate(applyrel, varlist ["t"; "t2"; "m2"; "m1"])))

let ax_apply_pair = aquantify ["t"; "t1"; "t2"; "t3"; "m1"; "m2"; "x"]
  (Leadsto(PreAnd(PPredicate(pairrel, varlist ["t1"; "x"; "t2"; "t3"]),
                  PPredicate(applyrel, varlist ["t"; "t1"; "m2"; "m1"])),
           ClAnd(CPredicate(applyrel, varlist ["t"; "t2"; "m2"; "m1"]),
                 CPredicate(applyrel, varlist ["t"; "t3"; "m2"; "m1"]))))

let ax_apply_ok = aquantify ["t"; "t1"; "s"; "m1"; "m2"]
  (Leadsto(PreAnd(PPredicate(okrel, varlist ["t1"; "s"]),
                  PPredicate(applyrel, varlist ["t"; "t1"; "m2"; "m1"])),
           CPredicate(applyrel, varlist ["t"; "s"; "m2"; "m1"])))

let axioms_apply = [Comment("Axioms for apply propagation. " ^
                            "Propagates an application to inner types.");
                    Clause(ax_apply_ch);
                    Clause(ax_apply_key);
                    Clause(ax_apply_pair);
                    Clause(ax_apply_ok)
                   ]

(* Unification of isolated variables *)
let ax_uni_ch = aquantify ["t1"; "t1'"; "t2"]
  (Leadsto(PreAnd(PPredicate(chrel, varlist ["t1"; "t2"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(chrel, varlist ["t1'"; "t2"])))

let ax_uni_key = aquantify ["t1"; "t1'"; "t2"]
  (Leadsto(PreAnd(PPredicate(keyrel, varlist ["t1"; "t2"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(keyrel, varlist ["t1'"; "t2"])))

let ax_uni_pair = aquantify ["t1"; "t1'"; "t2"; "t3"; "x"]
  (Leadsto(PreAnd(PPredicate(pairrel, varlist ["t1"; "x"; "t2"; "t3"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(pairrel, varlist ["t1'"; "x"; "t2"; "t3"])))

let ax_uni_ok = aquantify ["t1"; "t1'"; "t2"]
  (Leadsto(PreAnd(PPredicate(okrel, varlist ["t1"; "t2"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(okrel, varlist ["t1'"; "t2"])))

let ax_uni_app = aquantify ["t1"; "t1'"; "t2"; "m"]
  (Leadsto(PreAnd(PPredicate(apprel, varlist ["t1"; "t2"; "m"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(apprel, varlist ["t1'"; "t2"; "m"])))

let ax_uni_abs = aquantify ["t1"; "t1'"; "t2"; "m"; "x"]
  (Leadsto(PreAnd(PPredicate(absrel, varlist ["t1"; "t2"; "m"; "x"]),
                  PPredicate(teqrel, varlist ["t1"; "t1'"])),
           CPredicate(absrel, varlist ["t1'"; "t2"; "m"; "x"])))

let axioms_unification = [Comment("Unification axioms. This is also a way of assigning " ^
                                  "types to isolated type variables.");
                          Clause(ax_uni_ch);
                          Clause(ax_uni_key);
                          Clause(ax_uni_pair);
                          Clause(ax_uni_ok);
                          Clause(ax_uni_app);
                          Clause(ax_uni_abs)
                         ]

(* Proposition 1 *)
let ax_initial_effects = aquantify ["s"; "s1"; "e"; "m"]
  (Leadsto(preand_of_list [PPredicate(effadmitsrel, varlist ["s1"; "m"]);
                           PPredicate(deprefrel, varlist ["s"; "s1"]);
                           PPredicate(initrel, [Var("s")]);
                           PPredicate(envrel, varlist ["l"; "m"; "e"])
                          ],
           CPredicate(effectrel, varlist ["l"; "m"; "s1"])))

(* Proposition 4 *)
```

```
(*let ax_noninitial_effects = *)

(* Proposition 5 *)
(*let ax_semiinitial_effects =*)

let axioms = List.concat [axioms_eqneq;
                          axioms_gen;
                          axioms_teq;
                          axioms_genteq;
                          axioms_subs;
                          axioms_effsubs;
                          (*axioms_typsubs;*)
                          axioms_fn;
                          axioms_typfn;
                          axioms_may;
                          axioms_eff;
                          axioms_simplicity;
                          axioms_dependency;
                          axioms_initiality;
                          axioms_admissibility;
                          axioms_abstraction;
                          axioms_application;
                          axioms_apply;
                          axioms_unification
                         ]

(* Conjunction of axioms *)
let axiom_conjunction =
  let filtered_list = List.fold_right (fun element accum_list ->
                                         match element with
                                           | Comment(_) -> accum_list
                                           | Clause(cl) -> cl::accum_list
                                      ) axioms []
  in
  let ax_hd, ax_tl = (List.hd filtered_list, List.tl filtered_list) in
    List.fold_left (fun ax_accum cl ->
                      ClAnd(ax_accum, cl)
                    ) ax_hd ax_tl
```

E.8 ssolver/alfp.ml

Modification of `ssolver/alfp.ml` from D.6.

```
module SMap = Map.Make(String);;
module SSet = Set.Make(String);;

type term = Const of string
          | Var of string
          | FuncApp of string*(term list);;
type pre = PPredicate of string*(term list)
         | NegPredicate of string*(term list)
         | PreAnd of pre*pre
         | Or of pre*pre
         | Exists of string*pre
         | Equal of term*term
         | NEqual of term*term;;
type cl = CPredicate of string*(term list)
        | Truth
        | ClAnd of cl*cl
        | CommentClAnd of string*cl*cl
        | Leadsto of pre*cl
        | Forall of string*cl;;

let freevar_from_term ?(bound = SSet.empty) term =
  match term with
    | Var(x) ->
        if (SSet.mem x bound) then
          SSet.empty
        else
          SSet.add x SSet.empty
    | _ -> SSet.empty;;

let freevars_in_termlist ?(bound = SSet.empty) termlist =
```

```
  List.fold_left (fun varset term ->
                    let freevar = (freevar_from_term ~bound:bound term) in
                      SSet.union freevar varset
                  ) SSet.empty termlist;;

let rec freevars_in_pre ?(bound = SSet.empty) pre =
  match pre with
    | (PPredicate(_, termlist) | NegPredicate(_, termlist)) ->
        freevars_in_termlist ~bound:bound termlist
    | (PreAnd(pre1, pre2) | Or(pre1, pre2)) ->
        SSet.union (freevars_in_pre ~bound:bound pre1) (freevars_in_pre ~bound:bound pre2)
    | Exists(x, pre) ->
        freevars_in_pre ~bound:(SSet.add x bound) pre
    | (Equal(term1, term2) | NEqual(term1, term2)) ->
        SSet.union
          (freevar_from_term ~bound:bound term1)
          (freevar_from_term ~bound:bound term2);;

let rec string_of_term term =
  match term with
    | Const(c) -> "\"" ^ c ^ "\""
    | Var(x) -> "\"" ^ x ^ "\""
    | FuncApp(func, termlist) ->
        string_of_predicate func termlist

and string_of_predicate rel termlist =
  let (term, trest) = ((List.hd termlist), (List.tl termlist)) in
    rel ^ "(" ^ (string_of_term term) ^
      (List.fold_left (
         fun termstring term ->
           termstring ^ ", " ^ (string_of_term term)) "" trest)
    ^ ")";;

let rec string_of_pre pred =
  match pred with
    | PPredicate(rel, termlist) ->
        string_of_predicate rel termlist
    | NegPredicate(rel, termlist) ->
        "!" ^ (string_of_predicate rel termlist)
    | PreAnd(pre1, pre2) ->
        Printf.sprintf "%s & %s" (string_of_pre pre1) (string_of_pre pre2)
    | Or(pre1, pre2) ->
        Printf.sprintf "(%s | %s)" (string_of_pre pre1) (string_of_pre pre2)
    | Exists(str, pre) ->
        Printf.sprintf "(E %s.%s)" str (string_of_pre pre)
    | Equal(term1, term2) ->
        Printf.sprintf "%s = %s" (string_of_term term1) (string_of_term term2)
    | NEqual(term1, term2) ->
        Printf.sprintf "%s != %s" (string_of_term term1) (string_of_term term2);;

let rec string_of_cl ?(commented = false) clause =
  match clause with
    | CPredicate(rel, termlist) ->
        string_of_predicate rel termlist
    | Truth -> "1"
    | ClAnd(cl1, cl2) ->
        Printf.sprintf "%s & %s" (string_of_cl ~commented:commented cl1)
                                 (string_of_cl ~commented:commented cl2)
    | CommentClAnd(com, cl1, cl2) ->
        if commented = false then
          Printf.sprintf "%s & %s" (string_of_cl cl1) (string_of_cl cl2)
        else
          Printf.sprintf "\n\n\n#%s\n\n%s & %s" com (string_of_cl ~commented:commented cl1)
                                                   (string_of_cl ~commented:commented cl2)
    | Leadsto(pre, cl) ->
        Printf.sprintf "((%s) => %s)" (string_of_pre pre) (string_of_cl ~commented:commented cl)
    | Forall(str, cl) ->
        Printf.sprintf "(A %s.%s)" str (string_of_cl ~commented:commented cl);;


let rec latex_of_term term =
  match term with
    | Const(c) -> "\\Const{" ^ c ^ "}"
    | Var(x) -> "\\Var{" ^ x ^ "}"
```

```
    | FuncApp(func, termlist) ->
        latex_of_predicate func termlist

and latex_of_predicate rel termlist =
  let (term, trest) = ((List.hd termlist), (List.tl termlist)) in
    "\\Rel{" ^ rel ^ "}_(" ^ (latex_of_term term) ^
      (List.fold_left (
         fun termstring term ->
           termstring ^ ",_" ^ (latex_of_term term)) "" trest)
    ^ ")";;

let rec latex_of_pre pred =
  match pred with
    | PPredicate(rel, termlist) ->
        latex_of_predicate rel termlist
    | NegPredicate(rel, termlist) ->
        "\\neg" ^ (latex_of_predicate rel termlist)
    | PreAnd(pre1, pre2) ->
        Printf.sprintf "%s_\\land_%s" (latex_of_pre pre1) (latex_of_pre pre2)
    | Or(pre1, pre2) ->
        Printf.sprintf "(%s_\\lor_%s)" (latex_of_pre pre1) (latex_of_pre pre2)
    | Exists(str, pre) ->
        Printf.sprintf "\\exists_\\Const{%s}.%s" str (latex_of_pre pre)
    | Equal(term1, term2) ->
        Printf.sprintf "%s_=_%s" (latex_of_term term1) (latex_of_term term2)
    | NEqual(term1, term2) ->
        Printf.sprintf "%s_\\not=_%s" (latex_of_term term1) (latex_of_term term2);;

let rec latex_of_cl clause =
  match clause with
    | CPredicate(rel, termlist) ->
        latex_of_predicate rel termlist
    | Truth -> "\\Const{1}"
    | ClAnd(cl1, cl2) ->
        Printf.sprintf "%s_\\land_%s" (latex_of_cl cl1) (latex_of_cl cl2)
    | CommentClAnd(com, cl1, cl2) ->
        Printf.sprintf "\n\n%s\n\n%s_&_%s" com (latex_of_cl cl1) (latex_of_cl cl2)
    | Leadsto(pre, cl) ->
        Printf.sprintf "(%s)_\\Ra_%s" (latex_of_pre pre) (latex_of_cl cl)
    | Forall(str, cl) ->
        Printf.sprintf "\\forall_\\Const{%s}.%s" str (latex_of_cl cl);;
```

E.9 ssolver/outputparser.mly

Identical to `ssolver/outputparser.mly` from D.7.

E.10 ssolver/outputlexer.mll

Identical to `ssolver/outputlexer.mll` from D.8.

E.11 alfpgen.ml

```
#load "spitree.cmo";;
#load "aconv.cmo";;
#load "spiparser.cmo";;
#load "spilexer.cmo";;
#load "axioms.cmo";;
#load "constraints.cmo";;
#load "constraintgen.cmo";;
#load "ssolver.cma";;

open Alfp
open Axioms;;
```

```
module SMap = Map.Make(String);;

let propername_of_msg msg =
  Spitree.string_of_msg msg

let propername_of_env env =
  Constraints.string_of_env env
  (*Spitree.string_of_env env*)

let formula_of_env env =
  let e = propername_of_env env in
  let rec assemble env =
    match env with
      | Constraints.Type(x, t)::restenv ->
        ClAnd(CPredicate(envrel, constlist [x; t; e]),
              assemble restenv)
      | Constraints.UnType(x)::restenv -> assemble restenv
      | Constraints.Effect(reso)::restenv ->
        let rec formula_of_reso reso =
          (match reso with
             | Constraints.Reso(l, m) ->
                 CPredicate(envrel, constlist [l; propername_of_msg m; e])
             | Constraints.RPair(r1, r2) -> ClAnd(formula_of_reso r1,
                                                  formula_of_reso r2)
             | Constraints.Empty -> Truth)
        in
          ClAnd(formula_of_reso reso, assemble restenv)
      | [] -> Truth
  in
    assemble env

let rec formula_of_envlist envlist =
  match envlist with
    | env::restenvs ->
      ClAnd(formula_of_env env, formula_of_envlist restenvs)
    | [] -> Truth

let listdom env =
  List.fold_left (fun dlist typ ->
                    match typ with
                      | Constraints.Type(x, _) -> x::dlist
                      | Constraints.UnType(x) -> x::dlist
                      | _ -> dlist
                 ) [] env

let domformula envlist =
  let domains =
    List.map (fun env ->
                listdom env
             ) envlist
  in
  let formula_list =
    List.concat
      (List.map2 (fun env domain ->
                    List.map (fun name ->
                                CPredicate(domrel, constlist
                                                     [name; Constraints.string_of_env env])
                             ) domain
                 ) envlist domains)
  in
    List.fold_left (fun accum_formula formula ->
                      ClAnd(accum_formula, formula)
                   ) (List.hd formula_list) (List.tl formula_list)

(* Create a map with the above ranks *)
let rank = List.fold_left (fun map_accum reltuple ->
                             let relation, rank = reltuple in
                               SMap.add relation rank map_accum
                          ) SMap.empty ranklist

(* Stratify alfp formulae according to specified ranks *)
exception UnmatchedRank of string;;
let stratify alfp_cl =
  let rec rank_of_cl cl =
```

```
    match cl with
      | CPredicate(rel, _) ->
          print_endline rel;
          SMap.find rel rank
      | Truth -> 0    (* Not a true/false value! 0 is used to indicate to the rank comparison below
                         * that truth statements have no rank. *)
      | (ClAnd(cl1, cl2) | CommentClAnd(_, cl1, cl2)) ->
          let rank1, rank2 = (rank_of_cl cl1), (rank_of_cl cl2) in
            (match (rank1, rank2) with
               | (0, _) -> rank2
               | (_, 0) -> rank1
               | (_, _) ->
                   if (rank1 = rank2) || (rank1 = 0) || (rank2 = 0) then
                     rank1
                   else
                     raise (UnmatchedRank ("␣" ^ (string_of_cl cl1) ^ "␣␣␣" ^ (string_of_cl cl2) ^
                                           "␣␣␣Ranks␣of␣the␣consequence␣of␣a␣Horn␣clause␣must␣match")))
      | Leadsto(_, cl) ->
          rank_of_cl cl
      | Forall(_, cl) ->
          rank_of_cl cl
  in
  let rec annotate clause =
    match clause with
      | CPredicate(rel, _) ->
          [(rank_of_cl clause, clause)]
      | Truth -> [(rank_of_cl Truth, Truth)]
      | (ClAnd(cl1, cl2) | CommentClAnd(_, cl1, cl2)) ->
          List.append (annotate cl1) (annotate cl2)
      | Leadsto(_, cl) ->
          [(rank_of_cl cl, clause)]
      | Forall(_, cl) ->
          let (rank, _) = List.hd (annotate cl) in
            [(rank, clause)]
  in
  let annotated_list = annotate alfp_cl in
  let sorted_annotation = List.sort (fun el1 el2 ->
                                       let (rank1, _) = el1 in
                                       let (rank2, _) = el2 in
                                         compare rank1 rank2
                                    ) annotated_list
  in
  let (_, stratified_list) = List.split sorted_annotation in
    List.fold_left (fun accum_cl cl ->
                      ClAnd(accum_cl, cl)
                   ) Truth stratified_list

let print_commented_conjunction conj =
  List.iter (fun element ->
               match element with
                 | Clause(cl) ->
                     print_endline ((string_of_cl cl) ^ "\n")
                 | Comment(str) ->
                     print_endline ("\n" ^ str ^ "\n")
            ) conj

(* - Encodings - *)

(* Message terms *)
let rec alfp_of_msg msg =
  let n = propername_of_msg msg in
    (n,
     match msg with
       | Spitree.MPair(msg1, msg2) ->
           let m1, psi1 = alfp_of_msg msg1 in
           let m2, psi2 = alfp_of_msg msg2 in
             ClAnd(CPredicate(tpairrel, [Const(n); Const(m1); Const(m2)]),
                   ClAnd(psi1, psi2))
       | Spitree.Encr(msg1, msg2) ->
           let m1, psi1 = alfp_of_msg msg1 in
           let m2, psi2 = alfp_of_msg msg2 in
             ClAnd(CPredicate(encrel, [Const(n); Const(m1); Const(m2)]),
                   ClAnd(psi1, psi2))
       | Spitree.Name(_) ->
```

```
            CPredicate(namerel, [Const(n)])
        | Spitree.Ok ->
            CPredicate(tokrel, [Const(n)])
    )

(* let msg_equality m n =
 * Hang on. Where on earth are we using message equality? *)

(* Type constraints *)
exception UnsupportedByALFP of string;;
exception NotImplemented of string;;
let rec clause_of_typeconstraint typeconstraint =
  match typeconstraint with
    | Constraints.Equal(t, basetype) ->
        (match basetype with
            | Constraints.Key(t1) -> CPredicate(keyrel, [Const(t); Const(t1)])
            | Constraints.Pair(x, t1, t2) ->
                CPredicate(pairrel, [Const(t); Const(x); Const(t1); Const(t2)])
            | Constraints.Ch(t1) -> CPredicate(chrel, [Const(t); Const(t1)])
            | Constraints.Un -> CPredicate(unrel, [Const(t)])
            | Constraints.Ok(r) -> CPredicate(okrel, [Const(t); Const(r)])
            | Constraints.Abstraction(u, m, x) ->
                ClAnd(CPredicate(teqrel, constlist [t; u]),
                      CPredicate(absrel, constlist [t; u; propername_of_msg m; x]))
            | Constraints.Application(u, n) ->
                CPredicate(apprel, constlist [t; u; propername_of_msg n])
        )
    | Constraints.MayEqual(t, basetype) ->
        (match basetype with
            | Constraints.Key(t1) -> CPredicate(keyrel ^ maypf, [Const(t); Const(t1)])
            | Constraints.Pair(x, t1, t2) ->
                CPredicate(pairrel ^ maypf, [Const(t); Const(x); Const(t1); Const(t2)])
            | Constraints.Ch(t1) -> CPredicate(chrel ^ maypf, [Const(t); Const(t1)])
            | Constraints.Un -> CPredicate(unrel ^ maypf, [Const(t)])
            | Constraints.Ok(r) -> CPredicate(okrel ^ maypf, [Const(t); Const(r)])
            | Constraints.Abstraction(u, m, x) ->
                ClAnd(CPredicate(teqrel, constlist [t; u]),
                      CPredicate(absrel ^ maypf, constlist [t; u; propername_of_msg m; x]))
            | Constraints.Application(u, n) ->
                CPredicate(apprel ^ maypf, constlist [t; u; propername_of_msg n])
        )
    | Constraints.Generative(t) -> CPredicate(genrel, [Const(t)])
    | Constraints.NotGenerative(t) ->
        raise (UnsupportedByALFP "Negated_predicates_can_only_occur_in_preconditions")
    | Constraints.HasType(x, t) -> CPredicate(typrel, constlist [x; t])
    | Constraints.Fail -> CPredicate(failrel, [Const("Something_failed!")])
(*    | Constraints.NotFN(_, _) ->
        raise (UnsupportedByALFP "Negated predicates can only occur in preconditions")*)
    | Constraints.NotUn(_) ->
        raise (UnsupportedByALFP "Negated_predicates_can_only_occur_in_preconditions")
;;

let rec precondition_of_typeconstraint typeconstraint =
  match typeconstraint with
    | Constraints.Equal(t, basetype) ->
        (match basetype with
            | Constraints.Key(t1) -> PPredicate(keyrel, [Const(t); Const(t1)])
            | Constraints.Pair(x, t1, t2) ->
                PPredicate(pairrel, [Const(t); Const(x); Const(t1); Const(t2)])
            | Constraints.Ch(t1) -> PPredicate(chrel, [Const(t); Const(t1)])
            | Constraints.Un -> PPredicate(unrel, [Const(t)])
            | Constraints.Ok(r) -> PPredicate(okrel, [Const(t); Const(r)])
            | Constraints.Abstraction(u, m, x) ->
                PPredicate(absrel, constlist [t; u; propername_of_msg m; x])
            | Constraints.Application(u, n) ->
                PPredicate(apprel, constlist [t; u; propername_of_msg n])
        )
    | Constraints.MayEqual(t, basetype) ->
        (match basetype with
            | Constraints.Key(t1) -> PPredicate(keyrel ^ maypf, [Const(t); Const(t1)])
            | Constraints.Pair(x, t1, t2) ->
                PPredicate(pairrel ^ maypf, [Const(t); Const(x); Const(t1); Const(t2)])
            | Constraints.Ch(t1) -> PPredicate(chrel ^ maypf, [Const(t); Const(t1)])
            | Constraints.Un -> PPredicate(unrel ^ maypf, [Const(t)])
```

```
            | Constraints.Ok(r) -> PPredicate(okrel ^ maypf, [Const(t); Const(r)])
            | Constraints.Abstraction(u, m, x) ->
                PPredicate(absrel ^ maypf, constlist [t; u; propername_of_msg m; x])
            | Constraints.Application(u, n) ->
                PPredicate(apprel ^ maypf, constlist [t; u; propername_of_msg n])
        )
    | Constraints.Generative(t) -> PPredicate(genrel, [Const(t)])
    | Constraints.NotGenerative(t) -> NegPredicate(genrel, [Const(t)])
    | Constraints.HasType(x, t) -> PPredicate(typrel, constlist [x; t])
    | Constraints.Fail ->
        raise (NotImplemented "We only use Fail as the final relation in the final stratum. " ^
                              "We never use it in a precondition")
(*  | Constraints.NotFN(x, p) ->
        raise (NotImplemented "We need to somehow determine the free names of a process.")*)
    | Constraints.NotUn(t) ->
        NegPredicate(teqrel, constlist [t; "Un"]) (* Might be wrong *)
;;

let clause_of_envconstraint envconstraint =
  let namesindom propername env =
    let e = propername_of_env env in
      Leadsto(Exists("n",
                     PreAnd(PPredicate(fnrel, [Var("n"); Const(propername)]),
                            NegPredicate(domrel, [Var("n"); Const(e)]))),
              CPredicate(failrel, [Const("Free names of " ^ propername ^
                                         " not in environment " ^ e)])
             )
  in
    match envconstraint with
      | Constraints.NotInDom(name, env) ->
          Leadsto(PPredicate(domrel, constlist [name; propername_of_env env]),
                  CPredicate(failrel, [Const("Restricted name must not be in " ^
                                             "the environment of the restriction")]))
      | Constraints.NamesInDom(msg, env) ->
          let m = propername_of_msg msg in
            namesindom m env
      | Constraints.WellFormed(env) ->
          let rec wf env =
            (match env with
               | [] -> Truth
               | Constraints.Type(x, tvar)::restenv ->
                   let name_restenv = propername_of_env restenv in
                     ClAnd(wf restenv,
                           ClAnd(Leadsto(PPredicate(domrel, constlist [x; name_restenv]),
                                         CPredicate(failrel,
                                                    [Const("Environment is not well-formed: " ^
                                                           (Constraints.string_of_env env))])),
                                 namesindom tvar restenv))
               | Constraints.UnType(x)::restenv ->
                   let name_restenv = propername_of_env restenv in
                     ClAnd(wf restenv,
                           Leadsto(PPredicate(domrel, constlist [x; name_restenv]),
                                   CPredicate(failrel,
                                              [Const("Environment is not well-formed: " ^
                                                     (Constraints.string_of_env env))])))
                         )
               | Constraints.Effect(s)::restenv ->
                   ClAnd(wf restenv,
                         namesindom (Constraints.string_of_reso s) restenv)
            )
          in
            wf env

(* The next two functions are the main functions for translating
 * constraints to ALFP formulae. The functions illustrate a certain
 * loss of expressive power compared to the original constraints.
 * But if all goes well, we should not have overstepped these boundaries
 * during constraint generation or the ALFP encoding. *)
let rec precondition_of_formula formula =
  match formula with
    | Constraints.TypeC(constr) -> precondition_of_typeconstraint constr
    | Constraints.EnvC(constr) ->
        raise (UnsupportedByALFP ("Environment constraints are assertions " ^
                                  "and should not be in a precondition."))
```

```
    | Constraints.EffectC(constr) ->
        raise (UnsupportedByALFP "ALFP effect constraint generation not implemented yet.")
    | Constraints.And(pre1, pre2) ->
        PreAnd(precondition_of_formula pre1, precondition_of_formula pre2)
    | Constraints.Conjunction(pre::[]) -> precondition_of_formula pre
    | Constraints.Conjunction(pre::rest) ->
        PreAnd(precondition_of_formula pre,
               precondition_of_formula (Constraints.Conjunction(rest)))
    | Constraints.Conjunction([]) ->
        raise (UnsupportedByALFP "Precondition cannot be empty")
    | Constraints.CommentConjunction(_, _) ->
        raise (NotImplemented "We do not give comments in preconditions")
    | Constraints.Or(pre1, pre2) ->
        Or(precondition_of_formula pre1, precondition_of_formula pre2)
    | Constraints.Leadsto(pre, cl) ->
        raise (UnsupportedByALFP "Horn clauses are not allowed in ALFP preconditions")
    | Constraints.Forall(x, cl) ->
        raise (UnsupportedByALFP ("Quantification should happen over whole clauses, " ^
                                  "not just preconditions."))
    | Constraints.True ->
        raise (UnsupportedByALFP "Truth statements may only occur as a clause")

let rec clause_of_formula formula =
  match formula with
    | Constraints.TypeC(constr) -> clause_of_typeconstraint constr
    | Constraints.EnvC(constr) -> clause_of_envconstraint constr
    | Constraints.EffectC(constr) ->
        CPredicate(failrel, [Const("ALFP effect constraint generation " ^
                                   "not implemented yet.")])
    | Constraints.And(cl1, cl2) -> ClAnd(clause_of_formula cl1, clause_of_formula cl2)
    | Constraints.Conjunction(cl::[]) -> clause_of_formula cl
    | Constraints.Conjunction(cl::rest) ->
        ClAnd(clause_of_formula cl,
              clause_of_formula (Constraints.Conjunction(rest)))
    | Constraints.Conjunction([]) -> Truth
    | Constraints.CommentConjunction(com, cl::rest) ->
        CommentClAnd(com, clause_of_formula cl,
                          clause_of_formula (Constraints.Conjunction(rest)))
    | Constraints.CommentConjunction(com, []) -> Truth
    | Constraints.Or(cl1, cl2) ->
        raise (UnsupportedByALFP "Disjunction is only possible in preconditions!")
    | Constraints.Leadsto(pre, cl) ->
        Leadsto(precondition_of_formula pre, clause_of_formula cl)
    | Constraints.Forall(x, cl) -> Forall(x, clause_of_formula cl)
    | Constraints.True -> Truth
;;

let infile = "spisamples/match.spi" in
let filechan = open_in infile in
let lexbuf = Lexing.from_channel filechan in
let ast = Spiparser.process Spilexer.spi_tokens lexbuf in
let (ast_substituted, accum, substMap) = Aconv.proc_subst ast 0 Aconv.SMap.empty in
let freevars =
  Aconv.SMap.fold (fun _ name list ->
                     Constraints.UnType(name) :: list) substMap [] in

  print_endline "Alpha converted process:";
  print_endline ((Spitree.string_of_proc ast_substituted) ^ "\n");
  print_endline "Free names:";
  Aconv.SMap.iter (Printf.printf "%s: %s\n") substMap;
  print_endline "\n\nAxioms:";
  print_commented_conjunction axioms;

  let typeconstraints = Constraintgen.constraint_of_proc freevars ast_substituted in
  let messagesformula =
(*    print_endline "Beskeder:\n";
    List.iter (fun element ->
                 print_endline (Spitree.string_of_msg element)
              ) Constraintgen.messages#get;*)
    let messages = Constraintgen.messages#get in
    let hd, tl = List.hd messages, List.tl messages in
    let firstname, firstformula = alfp_of_msg hd in
      List.fold_left (fun accum_cl element ->
                        let propername, formula = alfp_of_msg element in
```

```
                        ClAnd(accum_cl, formula)
                    ) firstformula tl
  in
  let dom_conjunction = domformula Constraintgen.environments#get in
  let encodings = clause_of_formula typeconstraints in
  let envformula = formula_of_envlist Constraintgen.environments#get in
  let finalconjunction = (stratify (ClAnd(envformula,
                                    ClAnd(dom_conjunction,
                                    (ClAnd(axiom_conjunction,
                                     ClAnd(messagesformula, encodings)))))))) in
    print_endline (string_of_cl ~commented:true encodings);
    print_endline ("Noncommented_formula:\n\n" ^ (string_of_cl finalconjunction) ^
                   "\n\nStop_noncommented_formula");
    Ssolver.deducible_facts finalconjunction "ssolver/heap"
```

E.12 latex_axioms.ml

```
#use "axioms.ml";;
#load "alfp.cmo";;
open Alfp;;

let latex_format =
  "\\begin{itemize}" ^
  List.fold_left (fun latex_accum element ->
                    match element with
                      | Comment(str) -> latex_accum ^ "\n\n\\item " ^ str
                      | Clause(cl) -> latex_accum ^ "\\\\$" ^ (latex_of_cl cl) ^ "$"
                  ) "" axioms
  ^ "\\end{itemize}"
in
let outfile = "axioms.tex" in
let filechan = open_out outfile in
  output_string filechan latex_format
```