The I-Spi Tool

Jean Goubault-Larrecq
LSV/UMR 8643 CNRS & ENS Cachan, INRIA Futurs projet SECSI
61 avenue du président-Wilson, F-94235 Cachan Cedex
goubault@lsv.ens-cachan.fr
Phone: +33-1 47 40 75 68   Fax: +33-1 47 40 75 21

December 6, 2005

Contents

1 Syntax of the Core Calculus **1**

2 Approximating Equality **4**
- 2.1 Finding an over-approximation of equality 5
- 2.2 Finding an over-approximation of disequality 7

3 The Semantics of the Core Calculus **12**
- 3.1 The Lean Semantics 12
- 3.2 The Light Semantics 16

4 The Full Language **22**

The I-Spi tool is meant to verify cryptographic protocols, in a notation close to that of the spi-calculus (Abadi and Gordon, 1997). In practice, the syntax of I-Spi processes is very close to that used in Blanchet's ProVerif tool (Blanchet, 2004).

1 Syntax of the Core Calculus

First, *expressions* are meant to denote messages:

$e, \ldots$	$::=$	X	variables
	$\mid$	$f(e_1, \ldots, e_n)$	application of constructor f
	$\mid$	$\{e_1\}_{e_2}$	symmetric encryption
	$\mid$	$[e_1]_{e_2}$	asymmetric encryption

Variables are always capitalized, as in Prolog. Other identifiers always start uncapitalized. Constructors may be any identifier. The constructors $__\mathtt{c}_\Xi$ are reserved, for each finite list

$\Xi = X_1, \ldots, X_k$ of process variables, and are used to represent value environments. E.g., the environment mapping $\mathtt{M}$ to t_1, $\mathtt{Na}$ to t_2, and $\mathtt{X}$ to t_3, is represented as $\mathtt{__c}_{\mathtt{M,Na,X}}(t_1, t_2, t_3)$.

The constructors $\mathtt{crypt}$ and $\mathtt{acrypt}$ are reserved, too. In fact, $\{e_1\}_{e_2}$ is synonymous with $\mathtt{crypt}(e_1, e_2)$, and $[e_1]_{e_2}$ is synonymous with $\mathtt{acrypt}(e_1, e_2)$. The first denotes symmetric encryption of e_1 using key e_2. The second denotes asymmetric encryption of e_1 with e_2. This is a simplification of the EVA model (Goubault-Larrecq, 2001). The idea is that, whether you use a symmetric, a public or a private key e_2, to decrypt $\{e_1\}_{e_2}$, you need to know e_2 (not its inverse in case it has one); and to decrypt $[e_1]_{e_2}$, you need to know the inverse of e_2 (and it needs to have one). In other words, there are two encryption algorithms in I-Spi, one symmetric, one asymmetric. You must use the same keys to encrypt and decrypt with the first one, and you must use inverse keys with the second one. In other models, the shape of the key determines whether you should take the same key or an inverse; this is hardly realistic. The EVA model allows for arbitrary many encryption algorithms, some symmetric, some asymmetric; this is simplified here by just having one of each.

The unary constructors $\mathtt{pub}$ and $\mathtt{prv}$ are also reserved to denote asymmetric keys. The inverse of $\mathtt{pub}(k)$ is $\mathtt{prv}(k)$, and the inverse of $\mathtt{prv}(k)$ is $\mathtt{pub}(k)$.

The ISpi language is comprised of a so-called *core language*, from which all other constructs of the language can be defined.

The *core processes* are described by the following grammar:

$P, Q, R, \ldots$	$::=$	$\mathtt{stop}$	stop
	$\mid$	$![X]P$	replication
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\mathtt{new}X; P$	fresh name creation
	$\mid$	$\mathtt{out}(e_1, e_2); P$	writing to a channel
	$\mid$	$\mathtt{in}(e_1, X); P$	reading from a channel
	$\mid$	$\mathtt{let}\ X = e\ \mathtt{in}\ P$	local definition
	$\mid$	$\mathtt{case}\ e_1\ \mathtt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \mathtt{else}\ Q$	constructor pattern-matching
	$\mid$	$\mathtt{case}\ e_1\ \mathtt{of}\ \{X\}_{e_2} \Rightarrow P\ \mathtt{else}\ Q$	symmetric decryption
	$\mid$	$\mathtt{case}\ e_1\ \mathtt{of}\ [X]_{e_2} \Rightarrow P\ \mathtt{else}\ Q$	asymmetric decryption
	$\mid$	$\mathtt{if}\ e_1 = e_2\ \mathtt{then}\ P\ \mathtt{else}\ Q$	equality test
	$\mid$	$f(e_1, \ldots, e_n)$	process call

Intuitively, $\mathtt{stop}$ just stops; $![X]P$ launches an unbounded number of copies of process P, each copy receiving a fresh natural number (its *pid*) in variable X; $P|Q$ launches P and Q in parallel; $\mathtt{new}X; P$ creates a fresh name (nonce, key, channel name), binds it to X, then executes P; $\mathtt{out}(e_1, e_2); P$ writes e_2 on channel e_1, then executes P; $\mathtt{in}(e_1, X); P$ blocks reading on channel e_1, binds any received value to variable X, then executes P; $\mathtt{let}\ X = e\ \mathtt{in}\ P$ binds X to the value of expression e before executing P; $\mathtt{case}\ e_1\ \mathtt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \mathtt{else}\ Q$ evaluates e_1, and if its value is of the form $f(t_1, \ldots, t_n)$, it binds each X_i to the corresponding t_i, then executes P, otherwise it executes Q; $\mathtt{case}\ e_1\ \mathtt{of}\ \{X\}_{e_2} \Rightarrow P\ \mathtt{else}\ Q$ decrypts e_1 using key e_2 and a symmetric decryption algorithm, and on success binds X to the plaintext and executes P, otherwise it executes Q; $\mathtt{case}\ e_1\ \mathtt{of}\ [X]_{e_2} \Rightarrow P\ \mathtt{else}\ Q$ does the same with asymmetric

decryption, where e_2 evaluates to the inverse of the key with which e_1 was obtained; $f(e_1, \ldots, e_n)$ is meant to call the process named f, passing it the n arguments $e_1, \ldots, e_n$.

A *program* is, syntactically, a sequence of declarations. Each declaration can be:

- $\texttt{proc}\ f_1(\overline{X}_1) = P_1\ \texttt{and}\ f_2(\overline{X}_2) = P_2\ \texttt{and}\ \ldots\ \texttt{and}\ f_k(\overline{X}_k) = P_k$ declaring k process names $f_1, \ldots, f_k$, parameterized by argument lists $\overline{X}_i$, $1 \leq i \leq k$, and defined as the respective processes P_i;

- [`private`] `data` f/n declares the identifier f as a constructor of arity n; initially, I-Spi starts as though we had written `data` $\texttt{0}/0$, `data` $\texttt{s}/1$, `data` $\texttt{__c}_\Xi/k$ for every list of process variables Ξ of length k; constructors can be used in constructor pattern-matching. The `private` keyword is optional. If present, it states that a Dolev-Yao intruder cannot use f either to build messages $f(M_1, \ldots, M_n)$ or to do pattern-matching on f. (The only influence `private` has is in the definition of processes P_{syn} and P_{anl} below.)

- [`private`] `fun` f/n declares the identifier f as a function of arity n; it cannot be used in constructor pattern-matching, but can be used to build new messages. The `private` keyword is optional. If present, it states that a Dolev-Yao intruder cannot use f either to build messages $f(M_1, \ldots, M_n)$. (The only influence `private` has is in the definition of process P_{syn} below.) The intruder, as well as any process, cannot do case analysis on f; if you want to do this, use `data` instead of `fun`.

We assume that these declarations have been checked for consistency (meaning that process names are only declared once, that arities match, etc.), and have been compiled to:

- A *signature* Σ, which is a map from identifiers f to natural numbers (the *arity* of f); Σ contains the map $\{\texttt{crypt} \mapsto 2, \texttt{acrypt} \mapsto 2, \texttt{__eq} \mapsto 1, \texttt{pub} \mapsto 1, \texttt{prv} \mapsto 1, \texttt{0} \mapsto 0, \texttt{s} \mapsto 1, \texttt{__nu} \mapsto 2\} \cup \{\texttt{__c}_\Xi | \Xi \text{ process variable list}\}$;

- A *constructor signature* K is a subset of $\text{dom}\ \Sigma$; K contains the set $\{\texttt{0}, \texttt{s},\} \cup \{\texttt{__c}_\Xi | \Xi \text{ process variable list}\}$.

- A *set of private functions* Prv, which is a subset of $\text{dom}\ \Sigma$; Prv contains the set $\{\texttt{__eq}, \texttt{__nu}\}$.

- A *process table* $Proc$, which is a finite map from identifiers to *process abstractions*. A process abstraction is an expression $(X_1, \ldots, X_n)P$, where $X_1, \ldots, X_n$ are n distinct variables, and P is a process. Typically, writing $\texttt{proc}\ f(X_1, \ldots, X_n) = P$ means adding a binding from f to $(X_1, \ldots, X_n)P$ to $Proc$.

  $Proc$ contains at least the entries $\texttt{dy_synthesizer} \mapsto (\texttt{c_in}, \texttt{c_out})P_{syn}$ and $\texttt{dy_analyzer} \mapsto (\texttt{c_in}, \texttt{c_out})P_{anl}$. These are meant to describe capabilities of a Dolev-Yao intruder.

  P_{syn} is a process that is the parallel composition of processes of the form

$$\texttt{in}(\texttt{c_in}, M_1); \ldots; \texttt{in}(\texttt{c_in}, M_n); \texttt{out}(\texttt{c_out}, f(M_1, \ldots, M_n)); \texttt{stop} \tag{1}$$

DRAFT

for each $f \mapsto n$ in Σ such that $f \notin Prv$; this expresses the fact that a Dolev-Yao intruder can build new messages using the functions in Σ that are not private.

P_{anl} is the parallel composition of

$$\texttt{in}(\texttt{c_in}, M); \texttt{in}(\texttt{c_in}, M_2); \texttt{case}\ M\ \texttt{of}\ \{M_1\}_{M_2} \Rightarrow \texttt{out}(\texttt{c_out}, M_1)\ \texttt{else}\ \texttt{stop} \quad (2)$$

$$\texttt{in}(\texttt{c_in}, M); \texttt{in}(\texttt{c_in}, M_2); \texttt{case}\ M\ \texttt{of}\ [M_1]_{M_2} \Rightarrow \texttt{out}(\texttt{c_out}, M_1)\ \texttt{else}\ \texttt{stop} \quad (3)$$

$$\begin{array}{c}\texttt{in}(\texttt{c_in}, M); \texttt{case}\ M\ \texttt{of}\ f(M_1, \ldots, M_n) \Rightarrow \\ (\texttt{out}(\texttt{c_out}, M_1); \texttt{stop} | \ldots | \texttt{out}(\texttt{c_out}, M_n); \texttt{stop})\ \texttt{else}\ \texttt{stop}\end{array} \quad (4)$$

where, in the last process, f ranges over $K \setminus Prv$, and $\Sigma(f) = n$. The parallel composition $(\texttt{out}(\texttt{c_out}, M_1); \texttt{stop} | \ldots | \texttt{out}(\texttt{c_out}, M_n); \texttt{stop})$ denotes $\texttt{stop}$ if $n = 0$, and $\texttt{out}(\texttt{c_out}, M_1); \texttt{stop}$ if $n = 1$.

A typical encoding of the Dolev-Yao intruder is then:

$$\begin{array}{lll}\texttt{proc dolev_yao_intruder}(c, init) = \texttt{new}\ id; & !(& \texttt{out}(c, init); \texttt{stop} \\ & | & \texttt{out}(c, id); \texttt{stop} \\ & | & \texttt{in}(c, M); \texttt{out}(c, M); \texttt{out}(c, M); \texttt{stop} \\ & | & \texttt{new}N; \texttt{out}(c, N); \texttt{stop} \\ & | & \texttt{dy_synthesizer}(c, c) \\ & | & \texttt{dy_analyzer}(c, c) \\ &) & \end{array}$$

- A *main process*, i.e., a process name describing the whole system to be analyzed. The main process is always called `main`, and must be in $\mathrm{dom}\ Proc$. This main process is meant to describe the whole system, including all honest agents, as well as all intruders.

For the purpose of verification, we also add to Σ the set of all I-Spi processes, seen as constants (i.e., we add $P \mapsto 0$, for every process P, to Σ, considering that no process is equal to any identifier in $\mathrm{dom}\ \Sigma$). The binary symbol `__nu` is used to represent fresh names: see the semantic rules for the $\texttt{new}X; P$ construct.

2 Approximating Equality

In the various approximate semantics of the core calculus, it will be necessary to deal with equality of values. For example, decrypting a message of the form $\{t\}_u$ with some key v only works provided that $u = v$. Conversely, if $u \neq v$ then attempting to decrypt $\{t\}_u$ with v must fail.

We wish to encode both the equality relation $=$ and the disequality relation $\neq$ as binary predicates in logic. On the free term algebra, encoding the equality relation is easy: just write the clause

$$X = X$$

DRAFT

However, this escapes the $\mathcal{H}_1$ class (Nielson et al., 2002), a particularly nice decidable and broad fragment of Horn logic. This is not in $\mathcal{H}_1$, technically, because the head $X = X$ of this clause is not *linear*. A linear head is one where every variable occurs at most once.

In fact, any extension of $\mathcal{H}_1$ that encodes equality is undecidable (Goubault-Larrecq, 2005). In particular, it is also futile to try to replace the clause $X = X$ above by clauses encoding a recursive characterization of equality on ground terms such as

$$f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \quad \Leftarrow \quad X_1 = Y_1, \ldots, X_n = Y_n$$

for each $f \in \Sigma$, $\Sigma(f) = n$, $n \in \mathbb{N}$. However, it is possible to define approximate relations $\approx$ (possibly equal) and $\not\approx$ (possibly distinct).

Additionally, we wish to take some fixed equational theory E into account. That is, we wish to approximate $=_E$ (equality modulo E) and $\neq_E$ (disequality modulo E) by two relations $\approx_E$ (possibly equal modulo E) and $\not\approx_E$ (possibly distinct modulo E). We assume E is presented through a finite set of equations, which we write again E.

In the following subsections, we describe an automated way to compute some sets of clauses that achieve this purpose.

Imagine we have some *positive examples* $s_1 =_E t_1, \ldots, s_m =_E t_m$ of equations that do hold modulo E, and some *negative examples* $s_{m+1} \neq_E t_{m+1}, \ldots, s_{m+n} \neq t_{m+n}$ of disequations that do also hold modulo E. (All these are implicitly universally quantified.) This can be checked when E is presented by a convergent set of rewrite rules modulo AC, for example: rewrite s_i and t_i to their E-normal form modulo AC, $1 \leq i \leq m$, and check that the resulting normal forms are equal modulo AC; for $m+1 \leq i \leq m+n$, *narrow* s_i and t_i instead of rewriting them, and check that no common term is obtained from s_i and t_i. (Narrowing will in general fail to terminate, unless negative examples are ground, in which case narrowing will coincide with rewriting and therefore will terminate.)

Our approximations will be separated in finding an over-approximation $\approx_E$ of $=_E$ (Section 2.1) and in finding an over-approximation $\not\approx_E$ of $\neq_E$ (Section 2.2).

2.1 Finding an over-approximation of equality

Then, let $S_\approx$ be the set of clauses consisting of: 1. all equations of E, i.e., all clauses $\texttt{equal}(s, t)$, where $s = t$ is an equation in E, 2. all negative examples, i.e., the clauses $\neg\texttt{equal}(s_{m+j}, t_{m+j})$, $1 \leq j \leq n$. (`equal` is the equality predicate in TPTP notation, the input format for `h1` and for Paradox.)

Find a finite model $\mathcal{M}$ of $S_\approx$ plus the theory of equality using a finite-model finder such as Paradox (Claessen and Sörensson, 2003). E.g., consider the equational theory E given as follows:

$$X + (Y + Z) = (X + Y) + Z \quad X + Y = Y + X \quad X + 0 = X$$

and take as negative example

$$\{X + Y\}_Z \neq_E \{X\}_Z + \{Y\}Z$$

DRAFT

stating that symmetric encryption never commutes with sums. The resulting set $S_{\approx}$ of clauses is, in TPTP notation:

```
input_clause(assoc_plus,axiom,
             [++equal(plus(X,plus(Y,Z)),plus(plus(X,Y),Z))]).
input_clause(comm_plus,axiom,
             [++equal(plus(X,Y),plus(Y,X))]).
input_clause(plus_zero,axiom,
             [++equal(plus(X,zero),X)]).

input_clause(non_homo_1,conjecture,
             [--equal(crypt(plus(X,Y),Z),plus(crypt(X,Z),crypt(Y,Z)))]).
```

Say the above clauses are in some file named `ex1.p`. Launch Paradox on $S_{\approx}$,

```
paradox --model ex1.mod ex1.p
```

and look at the resulting model, here of size 2, in file `ex1.mod`:

```
crypt('1,'1) = '2
crypt('2,'1) = '2

plus('1,'1) = '1
plus('1,'2) = '2
plus('2,'1) = '2
plus('2,'2) = '1

zero = '1
```

(Of course, such a model may fail to exist, typically if the examples contradict E, or if there are only infinite models.)

Since a finite model is just a complete deterministic automaton (Goubault-Larrecq, 2004), it can be expressed as $\mathcal{H}_1$ clauses. The example model above, for instance, gives rise to the following clauses, in Prolog notation:

$$\begin{array}{rcl}
q_2(\texttt{crypt}(X,Y)) & \Leftarrow & q_1(X), q_1(Y) \\
q_2(\texttt{crypt}(X,Y)) & \Leftarrow & q_2(X), q_1(Y) \\
q_1(\texttt{plus}(X,Y)) & \Leftarrow & q_1(X), q_1(Y) \\
q_2(\texttt{plus}(X,Y)) & \Leftarrow & q_1(X), q_2(Y) \\
q_2(\texttt{plus}(X,Y)) & \Leftarrow & q_2(X), q_1(Y) \\
q_1(\texttt{plus}(X,Y)) & \Leftarrow & q_2(X), q_2(Y)
\end{array}$$

For all unspecified cases, fix some value, e.g., decide that `crypt('1,'2) = '1` and `crypt('2,'2) = '1`:

$$\begin{array}{rcl}
q_1(\texttt{crypt}(X,Y)) & \Leftarrow & q_1(X), q_2(Y) \\
q_1(\texttt{crypt}(X,Y)) & \Leftarrow & q_2(X), q_2(Y)
\end{array}$$

DRAFT

Formally, this is done as follows. Each value v of the model $\mathcal{M}$ is translated to a fresh predicate q_v. For every $f \in \Sigma$, $\Sigma(f) = n$, for every n-tuple of values $v_1, \ldots, v_n$ in $\mathcal{M}$, write down the clause

$$q_v(f(X_1, \ldots, X_n)) \ \Leftarrow \ q_{v_1}(X_1), \ldots, q_{v_n}(X_n)$$

where v is the value of f applied to $v_1, \ldots, v_n$ in $\mathcal{M}$. Let $\mathcal{A}$ be the resulting set of clauses.

If M is equal to N modulo E, then M and N will have the same value in $\mathcal{M}$, so for every value v, both $q_v(M)$ and $q_v(N)$ will be consequences of the clauses in $\mathcal{A}$. Alternatively, defining the predicate $\approx_E$ by the set of clauses

$$X_1 \approx_E X_2 \ \Leftarrow \ q_v(X_1), q_v(X_2) \tag{5}$$

when v ranges over the values in $\mathcal{M}$, describes $\approx_E$ as an over-approximation of equality modulo E.

Additionally, the automaton $\mathcal{A}$ has the property that, for every ground term t, $q_v(t)$ is derivable from $\mathcal{A}$ if and only if the value of t in $\mathcal{M}$ is v. Since $\mathcal{M}$ satisfies all clauses $\neg\mathtt{equal}(s_{m+j}, t_{m+j})$, $1 \leq j \leq n$, there is no pair of ground instances $s_{m+j}\sigma$, $t_{m+j}\sigma$ that are made equal in $\mathcal{M}$. It follows that there is no value v in $\mathcal{M}$ such that $q_v(s_{m+j}\sigma)$ and $q_v(t_{m+j}\sigma)$ are both derivable from $\mathcal{A}$. In other words, no instance of $s_{m+j} \approx_E t_{m+j}$ is derivable from $\mathcal{A}$ and the clauses (5): we have found an over-approximation of equality modulo E that correctly recognizes all negative examples.

2.2 Finding an over-approximation of disequality

To over-approximate disequality, under-approximate equality. A trivial way is to enumerate finitely many ground instances of the equational theory E and of the positive examples $s_1 =_E t_1$, $\ldots$, $s_m =_E t_m$. This provides a finite set of pairs. Any finite set of terms is regular, and the complement of any regular set is regular. The resulting complement is then an over-approximation of disequality.

Let us give an example. Imagine that, in fact, encryption is homomorphic: $\{s + t\}_u =_E \{s\}_u + \{t\}_u$ for every terms s, t, u. Imagine we have one positive example, which is a particular ground instance of this, say $\{\mathtt{a} + \mathtt{b}\}_{\mathtt{pub(c)}} =_E \{\mathtt{a}\}_{\mathtt{pub(c)}} + \{\mathtt{b}\}_{\mathtt{pub(c)}}$. Write this pair of equal terms as a Prolog clause:

```
eq(pair(crypt(plus(a,b),pub(c)),plus(crypt(a,pub(c)),crypt(b,pub(c)))))
```

The `h1` tool suite understands this syntax without modification. Add all clauses that describe properties of equality and which are $\mathcal{H}_1$ clauses. These are:

$$\begin{array}{rcl} \mathtt{eq}(\mathtt{pair}(X, Y)) & \Leftarrow & \mathtt{eq}(\mathtt{pair}(Y, X)) \\ \mathtt{eq}(\mathtt{pair}(X, Y)) & \Leftarrow & \mathtt{eq}(\mathtt{pair}(X, Z)), \mathtt{eq}(\mathtt{pair}(Z, Y)) \\ \mathtt{eq}(\mathtt{pair}(c, c)) & & \text{for every constant } c \in \Sigma \end{array}$$

Put these clauses into some file `eq1.pl`, and run

```
pl2tptp eq1.pl | h1 -no-trim -model -
mv h1out.model.pl eq1.nd.pl
```

and you'll get the corresponding automaton, as Prolog clauses, in file `eq1.nd.pl`:

```
__def_4(pub(X)) :- __def_7(X).
eq(pair(X1,X2)) :- __def_2(X1), __def_1(X2).
eq(pair(X1,X2)) :- __def_2(X1), __def_2(X2).
eq(pair(X1,X2)) :- __def_6(X1), __def_6(X2).
eq(pair(X1,X2)) :- __def_1(X1), __def_2(X2).
eq(pair(X1,X2)) :- __def_5(X1), __def_5(X2).
eq(pair(X1,X2)) :- __def_7(X1), __def_7(X2).
eq(pair(X1,X2)) :- __def_1(X1), __def_1(X2).
__def_3(plus(X1,X2)) :- __def_5(X1), __def_6(X2).
__def_1(crypt(X1,X2)) :- __def_3(X1), __def_4(X2).
__def_2(plus(X1,X2)) :- __def_8(X1), __def_9(X2).
__def_6(b).
__def_9(crypt(X1,X2)) :- __def_6(X1), __def_4(X2).
__def_7(c).
__def_8(crypt(X1,X2)) :- __def_5(X1), __def_4(X2).
__def_5(a).
```

Here `__def_1`, ..., `__def_9` are auxiliary states of the automaton.

Next, complement this automaton. You first need to add clauses so as to describe the signature. Let's say, for the purpose of illustration, that we have a constant symbol `zero` and a unary symbol `minus` in our signature Σ, in addition to `crypt`, `plus`, `a`, `b`, `c` already mentioned above. Add the clauses:

```
sig(zero).
sig(minus(X)) :- sig(X).
sig(crypt(X,Y)) :- sig(X), sig(Y).
sig(plus(X,Y)) :- sig(X), sig(Y).
sig(a).
sig(b).
sig(c).
```

to `eq1.nd.pl`, by writing

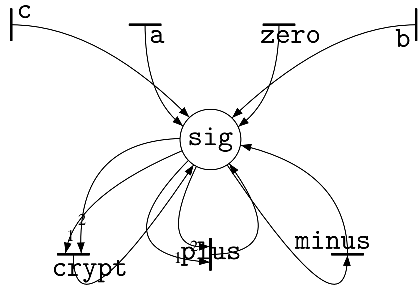
```
cat sig1.pl >>eq1.nd.pl
```

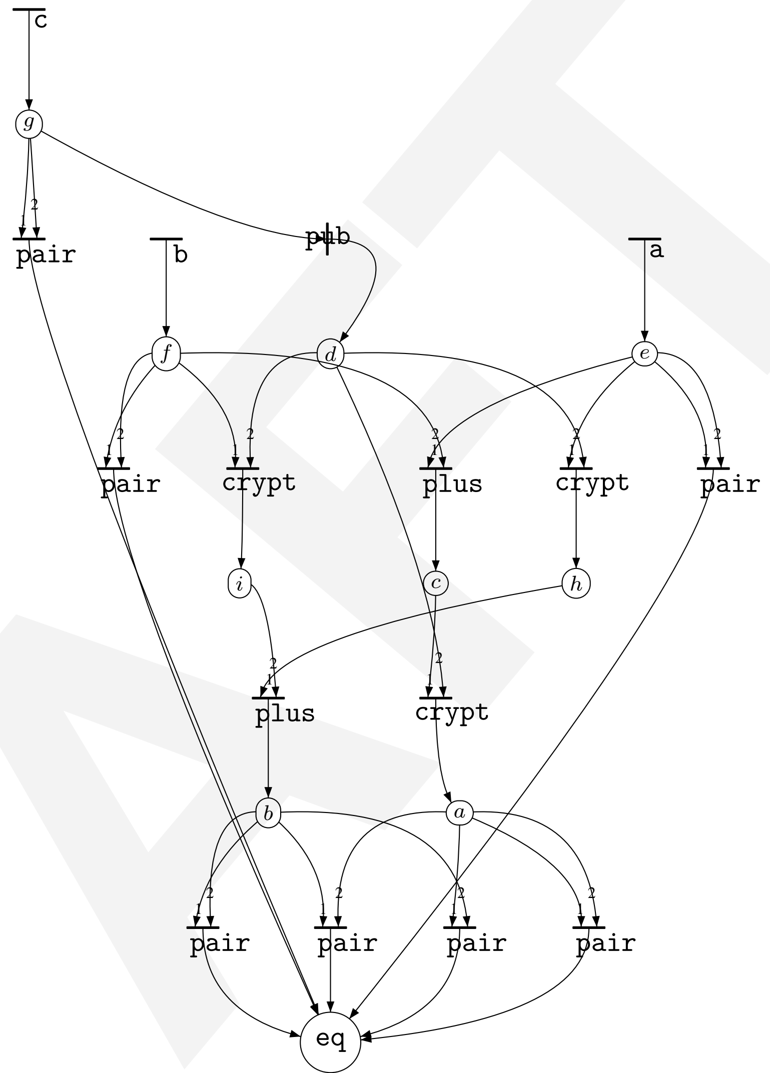
(We assume the above clauses were written in a file `sig1.pl`.) Note that `pair` is *not* a symbol in the signature.

By the way, here is a graphical version of this automaton, rendered using

```
pl2gastex -layout dot eq1.nd.pl >eq1.tex
```


DRAFT





Then determinize the augmented automaton using `pldet`. Using `auto2pl` with the `-negate` option on the output of `pldet` will produce the complemented automaton:

```
pldet eq1.nd.pl | auto2pl -defs -negate eq - >neq1.pl
echo "distinct(X,Y) :- __not_eq(pair(X,Y)), sig(X), sig(Y)." >>neq1.pl
```

The last line above serves to define those pairs (X, Y) such that not only the pair $\texttt{pair}(X, Y)$ is recognized at `__not_eq`, but also X and Y are terms in the signature—complementation produces lots of strange terms recognized by `__not_eq`. We can then massage the output `neq1.pl` by filtering out all those predicates that do not serve to define the `distinct` predicate (using `plpurge`), and by calling `h1` to eliminate the `__not_eq` predicate entirely:

```
plpurge -final distinct neq1.pl | pl2tptp - | h1 -no-trim -model -no-alternation -
plpurge -final distinct h1out.model.pl >distinct1.pl
```

Here this yields a 130 transition automaton for distinctness. The `distinct` predicate is the desired over-approximation $\not\approx_E$ of $\neq_E$.

DRAFT

3 The Semantics of the Core Calculus

The core calculus of I-Spi could be given a semantics looking like the one of the spi-calculus (Abadi and Gordon, 1997). In practice, it is more useful to consider approximate semantics, i.e., semantics that are guaranteed to simulate at least all valid execution traces of I-Spi programs, but which may simulate a bit more. In practice this means that it may be that a given cryptographic protocol has no actual attack, but that one will exist in the approximate semantics.

This is done in the name of computability, and in some cases even efficiency. The point is that is no attack is found in any of the approximate semantics, then the protocol is indeed, definitely, secure. But if some purported attack is found, there may actually be no attack at all.

In the sequel, we assume a signature Σ, a constructor signature and a set of private functions $K, Prv \subseteq \mathrm{dom}\ \Sigma$, a process table $Proc$ and a main process $\mathtt{main} \in \mathrm{dom}\ Proc$ to be given once and for all.

We give several such semantics: from most approximate to most precise, the *lean* semantics (Section 3.1), the *light* semantics (Section 3.2).

3.1 The Lean Semantics

The coarsest semantics of I-Spi is the *lean semantics*, and is inspired from Nielson et al. (2002). (I.e., this is really close to the semantics in the cited paper, except you may fail to recognize it.) It is given as a set of rules allowing one to derive one of the following forms of judgments:

- $\vdash e \simeq t$ states that the I-Spi expression e may evaluate to the value t; values are represented as terms, just like expressions, but we keep the denominations "expression" ("e") and "term" ("t") distinct so as to emphasize the difference. Note that, contrarily to what the notation suggests, $\simeq$ is not meant to be particularly reflexive, symmetric, or transitive here.

- $\vdash P : t \triangleright u$ states that message u may have traveled on channel t, and sent at program point P; i.e., P may have sent u on channel t. Here t and u are terms, that is, values, not I-Spi expressions.

- $\vdash P : t \triangleright u : Q$ states that P may have sent message u on channel t, and that u was then read by Q.

- $\Xi \vdash E \to P$ states that execution may reach the program point P, and that the last program point before P was E. (Sub-processes of a given process are basically equivalent of program points in usual programming languages.) To be precise, E may either be a program point Q, or the special symbol $\bullet$ (no event yet, the program has just started). Ξ is a list of variables; they are the variables bound in message read instructions, in replications, and formal parameters above P.

Additional, auxiliary judgments will be introduced on a case by case basis. They are $t \in \mathbb{N}$ (meaning that t is a natural number), $\vdash e^{-1} \simeq t$ (meaning that the value of e is the inverse of key t).

DRAFT

The first rule states that we may start running the whole system, starting from `main`:

$$\frac{}{\vdash \bullet \to \mathtt{main}}\ (Start)$$

The semantics of replication $![X]P$ states that, when you have reached $![X]P$, you can immediately proceed to evaluate P, and X is any natural number.

$$\frac{\Xi \vdash E \to ![X]P}{X, \Xi \vdash ![X]P \to P}\ (!) \quad \frac{\Xi \vdash E \to ![X]P \quad t \in \mathbb{N}}{\vdash X \simeq t}\ (!\simeq) \quad \frac{}{\mathtt{0} \in \mathbb{N}}\ (\mathtt{0}) \quad \frac{t \in \mathbb{N}}{\mathtt{s}(t) \in \mathbb{N}}\ (\mathtt{s})$$

The second rule states that X may be any natural number t. Natural numbers are defined through the auxiliary judgment $t \in \mathbb{N}$, which can only be derived used the last two rules. In other words, the natural number n is the term $\mathsf{S}^n(\mathtt{0})$.

There is some loss of precision in the rules above, and in similar rules to follow: rule $(!)$ states that we may reach P, and rule $(!\simeq)$ states that X may be bound to t, but these rules are independent. These rules do not enforce that X should be bound to t only when we reach P.

To execute the parallel composition $P|Q$, you need to execute both P and Q:

$$\frac{\Xi \vdash E \to P|Q}{\Xi \vdash P|Q \to P}\ (|_1) \quad \frac{\Xi \vdash E \to P|Q}{\Xi \vdash P|Q \to Q}\ (|_2)$$

The semantics of fresh name creation involves the auxiliary function symbol $\mathtt{_nu}$. In executing $\mathtt{new}X; P$, we bind X to the term $\mathtt{_nu}(\mathtt{new}X; P, \mathtt{_c}_\Xi(t_1, \ldots, t_k))$, where t_1, …, t_k are the values of the variables X_1, …, X_k in the list Ξ to the left of $\vdash$. This way of representing fresh names is a variant on one due to Blanchet (2001).

$$\frac{\Xi \vdash E \to \mathtt{new}X; P}{\Xi \vdash \mathtt{new}X; P \to P}\ (\mathtt{new}) \quad \frac{X_1, \ldots, X_k \vdash E \to \mathtt{new}X; P \quad \vdash X_1 \simeq t_1 \ldots \vdash X_k \simeq t_k}{\vdash X \simeq \mathtt{_nu}(\mathtt{new}X; P, \mathtt{_c}_{X_1,\ldots,X_k}(t_1, \ldots, t_k))}\ (\mathtt{new}\simeq)$$

Note that, in rule ($\mathtt{new}$), we do *not* add X to the set Ξ in the conclusion of the rule.

There is some slight loss of information here, too. Normally, executing $\mathtt{new}X; P$ creates a fresh name each time, and stores it into X. The rules above may create twice the same name, namely when we execute the same process $\mathtt{new}X; P$ twice, with the same values t_1, …, t_k of the variables X_1, …, X_k.

Given that we know the values of variables, we may infer the value of more complex expressions. This is given by the following rule, which is parameterized by the function symbol f, of arity k.

$$\frac{f \in \Sigma \quad \Sigma(f) = k \quad \vdash e_1 \simeq t_1 \quad \ldots \quad \vdash e_k \simeq t_k}{\vdash f(e_1, \ldots, e_k) \simeq f(t_1, \ldots, t_k)}\ (f\simeq)$$

The $(f\simeq)$ rule does not lose any information by itself, but propagates, and aggravates any loss of information. E.g., if there are two possible values t_1 for e_1, …, two possible values t_k for e_k, then there are 2^k possible values for $f(e_1, \ldots, e_k)$.

DRAFT

The following rules state that you may always proceed to execute P from $\texttt{out}(e_1, e_2); P$, and that the event $(\texttt{out}(e_1, e_2); P) : t_1 \triangleright t_2$ is derivable.

$$\frac{\Xi \vdash E \to \texttt{out}(e_1, e_2); P \quad \vdash e_1 \simeq t_1 \quad \vdash e_2 \simeq t_2}{\Xi \vdash \texttt{out}(e_1, e_2); P \to P}\ (\texttt{out})$$

$$\frac{\Xi \vdash E \to \texttt{out}(e_1, e_2); P \quad \vdash e_1 \simeq t_1 \quad \vdash e_2 \simeq t_2}{\vdash (\texttt{out}(e_1, e_2); P) : t_1 \triangleright t_2}\ (\texttt{out}\triangleright)$$

To read a message, executing $\texttt{in}(e, X); Q$, we evaluate e to a value t_1, then look for some derivable event $(\texttt{out}(e_1, e_2); P) : t_1 \triangleright t_2$, with the same value t_1 for the channel on which t_2 was sent. Then we bind X to the message t_2.

$$\frac{\Xi \vdash E \to \texttt{in}(e, X); Q \quad \vdash P' : t_1 \triangleright t_2 \quad \vdash e \simeq t_1}{X, \Xi \vdash \texttt{in}(e, X); Q \to Q}\ (\texttt{in})$$

$$\frac{\Xi \vdash E \to \texttt{in}(e, X); Q \quad \vdash P' : t_1 \triangleright t_2 \quad \vdash e \simeq t_1}{\vdash X \simeq t_2}\ (\texttt{in}\simeq)$$

$$\frac{\Xi \vdash E \to \texttt{in}(e, X); Q \quad \vdash P' : t_1 \triangleright t_2 \quad \vdash e \simeq t_1}{\vdash P' : t_1 \triangleright t_2 : \texttt{in}(e, X); Q}\ (\texttt{in}\triangleright)$$

This is the set of rules that most likely loses the most information. Essentially, they state that you can read some message t_2 on channel t_1 as soon as some process P' has sent, *or will send*, *or would have sent* t_2 on t_1 if some other execution branch had been taken. "Will send": the semantics above forces the process $\texttt{in}(c, X); \texttt{out}(c, e); P$ to accept the value t of e (sent by $\texttt{out}(c, e)$) as possible value of X (received by $\texttt{in}(c, X)$), as soon as some (arbitrary) message has been sent on c. "Would have sent": the process $\texttt{if}\ e_1 = e_2\ \texttt{then}\ \texttt{in}(c, X); P\ \texttt{else}\ \texttt{out}(c, e); Q$ will be dealt with as though P (in the `then` branch) is reachable with X bound to the value of e (as given in the `else` branch), as soon as both branches are reachable, although not both branches can be reached.

$$\frac{\Xi \vdash E \to \texttt{let}\ X = e\ \texttt{in}\ P \quad \vdash e \simeq t}{X, \Xi \vdash \texttt{let}\ X = e\ \texttt{in}\ P \to P}\ (\texttt{let}) \qquad \frac{\Xi \vdash E \to \texttt{let}\ X = e\ \texttt{in}\ P \quad \vdash e \simeq t}{\vdash X \simeq t}\ (\texttt{let}\simeq)$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq f(t_1, \ldots, t_n)}{X_1, \ldots, X_n, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \to P}\ (\texttt{case}f)$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq f(t_1, \ldots, t_n)}{\vdash X_i \simeq t_i \quad (1 \leq i \leq n)}\ (\texttt{case}f\simeq)$$

$$\frac{\begin{array}{c}\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \\ \vdash e_1 \simeq g(t_1, \ldots, t_m) \quad g \in \Sigma,\ \Sigma(g) = m,\ g \neq f\end{array}}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \to Q}\ (\texttt{case}f \neq g)$$

DRAFT

There are as many instances of rule $(\texttt{case}f \neq g)$ as there are symbols g other than f. This says that you may execute the else branch as soon as e_1 may have a value that does not match the pattern $f(X_1, \ldots, X_n)$.

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq \{t\}_u \quad \vdash e_2 \simeq u}{X, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \to P}\ (\texttt{case}\{\})$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq \{t\}_u \quad \vdash e_2 \simeq u}{\vdash X \simeq t}\ (\texttt{case}\{\} \simeq)$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq t}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \to Q}\ (\texttt{case}\{\} \neq)$$

The most notable feature of the rules above is that we estimate that decryption may always fail: rule $(\texttt{case}\{\} \neq)$ indeed says that, as soon as e_1 has at least one value, you can always follow the `else` branch. We shall produce a more precise rule in the light semantics (Section 3.2).

Asymmetric decryption is very similar, except that we now need an auxiliary judgment $\vdash e^{-1} \simeq t$ to say when e denotes an asymmetric key k, and k is the inverse of key t.

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq [t]_u \quad \vdash e_2^{-1} \simeq u}{X, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \to P}\ (\texttt{case}[])$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq [t]_u \quad \vdash e_2^{-1} \simeq u}{\vdash X \simeq t}\ (\texttt{case}[] \simeq)$$

$$\frac{\Xi \vdash E \to \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \quad \vdash e_1 \simeq t}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \to Q}\ (\texttt{case}[] \neq)$$

The judgment $\vdash e^{-1} \simeq t$ is defined by the following rules:

$$\frac{\vdash e \simeq \texttt{pub}(t)}{\vdash e^{-1} \simeq \texttt{prv}(t)}\ (\texttt{pub}^{-1}) \qquad \frac{\vdash e \simeq \texttt{prv}(t)}{\vdash e^{-1} \simeq \texttt{pub}(t)}\ (\texttt{prv}^{-1})$$

$$\frac{\Xi \vdash E \to \texttt{if}\ e_1 = e_2\ \texttt{then}\ P\ \texttt{else}\ Q \quad \vdash e_1 \simeq t \quad \vdash e_2 \simeq t}{\Xi \vdash \texttt{if}\ e_1 = e_2\ \texttt{then}\ P\ \texttt{else}\ Q \to P}\ (\texttt{if} \simeq)$$

$$\frac{\Xi \vdash E \to \texttt{if}\ e_1 = e_2\ \texttt{then}\ P\ \texttt{else}\ Q \quad \vdash e_1 \simeq t \quad \vdash e_2 \simeq u}{\Xi \vdash \texttt{if}\ e_1 = e_2\ \texttt{then}\ P\ \texttt{else}\ Q \to P}\ (\texttt{if} \neq)$$

Note that e_1 and e_2 are required to have the same value t in rule $(\texttt{if} \simeq)$. In $(\texttt{if} \neq)$, we only check that e_1 and e_2 both have a value, not necessarily distinct.

DRAFT

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow f(e_1, \ldots, e_n) \\ f \in \mathrm{dom}\, Proc \quad Proc(f) = (X_1, \ldots, X_n)P \\ \vdash e_1 \simeq t_1 \ldots \vdash e_n \simeq t_n\end{array}}{X_1, \ldots, X_n \vdash f(e_1, \ldots, e_n) \rightarrow P}\ (Call)$$

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow f(e_1, \ldots, e_n) \\ f \in \mathrm{dom}\, Proc \quad Proc(f) = (X_1, \ldots, X_n)P \\ \vdash e_1 \simeq t_1 \ldots \vdash e_n \simeq t_n\end{array}}{\vdash X_i \simeq t_i \quad (1 \leq i \leq n)}\ (Call \simeq)$$

3.2 The Light Semantics

The idea behind the light semantics, which is more precise than the lean semantics, is to keep more relations between the values of the variables in the given scope Ξ. In the lean semantics, if you have two runs of the same process that lead respectively to give X the value 0 or to give X the value 1, then you bind Y to X, the lean semantics will estimate that X can be 0 or 1, and that Y can be 0 or 1, but will forget that the values of X and Y are the same. The purpose of the light semantics is to try and keep such relations as much as possible.

The new judgments of the light semantics are:

- $P \vdash e \simeq t$ states that expression e may have value t when program point P is reached. Additionally, processes P will not be limited to be subprocesses of some initial process. In general, P will be obtained by replacing variables by terms in subprocesses. E.g., $\mathtt{new}X; P$ will involve instantiating X by terms of the form $__\mathtt{nu}(\mathtt{new}X; P, __\mathtt{c}_\Xi(t_1, \ldots, t_k))$ in all contexts where X_1 has value t_1, ..., X_k has value t_k.

- $\vdash P\langle\langle\rho\rangle\rangle : t \triangleright u$ states that message u may have traveled on channel t, and sent at program point P under context ρ. A *context* is a finite mapping from variables to values. That is, it states which values the variables of P had when P sent u on t. We choose to represent ρ as a term of the form $__\mathtt{c}_\Xi(t_1, \ldots, t_k)$, where Ξ is a list of exactly k variables X_1, ..., X_k; this represents $\{X_1 \mapsto t_1, \ldots, X_k \mapsto t_k\}$.

- $\vdash P\langle\langle\rho\rangle\rangle : t \triangleright u : Q\langle\langle\rho'\rangle\rangle$ states that P (under context ρ) may have sent u on t, and u was received by Q (under context ρ').

- $\Xi \vdash E \rightarrow P\langle\langle\rho\rangle\rangle$ states that execution may reach P, with context ρ, and that the last program point before P was E. This judgment will always obey the invariant that Ξ, as a set, is exactly the set of free variables of P. Also, if Ξ is $X_1, \ldots, X_k$, then ρ is a term of the form $__\mathtt{c}_\Xi(t_1, \ldots, t_k)$.

In the sequel, we assume that all bound variables are renamed apart, so that no alpha-renaming is ever needed.

DRAFT

We may start running the whole system, starting from `main`, with no variable bound to any term:

$$\frac{}{\epsilon \vdash \bullet \rightarrow \mathtt{main}\langle\!\langle \mathtt{__c}_{\epsilon}() \rangle\!\rangle} \; (Start')$$

We write ϵ the empty sequence of variables.

Replication, as expected, creates a fresh process identifier and stores it into X:

$$\frac{\Xi \vdash E \rightarrow ![X]P\langle\!\langle \mathtt{__c}_{\Xi}(t_1, \ldots, t_k) \rangle\!\rangle \quad t \in \mathbb{N}}{X, \Xi \vdash ![X]P \rightarrow P\langle\!\langle \mathtt{__c}_{X,\Xi}(t, t_1, \ldots, t_k) \rangle\!\rangle} \; (!')$$

The judgment $t \in \mathbb{N}$ was defined in Section 3.1.

Rule $(!')$ does not suffer from the same loss of precision problem than $(!)$ and $(!\simeq)$ (Section 3.1). The dependencies between the values of all variables are indeed kept.

Parallel composition does not modify any context:

$$\frac{\Xi \vdash E \rightarrow P|Q\langle\!\langle \rho \rangle\!\rangle}{\Xi \vdash P|Q \rightarrow P\langle\!\langle \rho \rangle\!\rangle} \; (|'_1) \qquad \frac{\Xi \vdash E \rightarrow (P|Q)\langle\!\langle \rho \rangle\!\rangle}{\Xi \vdash P|Q \rightarrow Q\langle\!\langle \rho \rangle\!\rangle (P|Q)\langle\!\langle \rho \rangle\!\rangle} \; (|'_2)$$

Evaluating an expression of the form $\mathtt{new}X; P$ involves substituting the term $\mathtt{__nu}(\mathtt{new}X; P, \mathtt{__c}_{\Xi}(X_1, \ldots, X_k))$ for X, where $X_1, \ldots, X_k$ are the variables in Ξ. We assume that substitution $P[X := e]$ of X for e in P is defined in the usual, capture-avoiding way. What we would like to write is: let Ξ be $X_1, \ldots, X_k$, and assume that $\Xi \vdash E \rightarrow \mathtt{new}X; P\langle\!\langle \rho \rangle\!\rangle$, i.e., we can reach program point $\mathtt{new}X; P$ with environment ρ; then we can reach P in the environment ρ augmented with the binding from X to $\mathtt{__nu}(\mathtt{new}X; P, \rho)$. To gain some precision, we directly replace X by $\mathtt{__nu}(\mathtt{new}X; P, \mathtt{__c}_{\Xi}(X_1, \ldots, X_k))$, knowing that ρ will give the correct values to $X_1, \ldots, X_k$.

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \\ \Xi \vdash E \rightarrow \mathtt{new}X; P\langle\!\langle \rho \rangle\!\rangle\end{array}}{\Xi \vdash \mathtt{new}X; P \rightarrow P[X := \mathtt{__nu}(\mathtt{new}X; P, \mathtt{__c}_{\Xi}(X_1, \ldots, X_k))]\langle\!\langle \rho \rangle\!\rangle} \; (\mathtt{new}')$$

Note that we do not add X to Ξ on the left-hand side of $\vdash$. This is because X just got replaced by the term $\mathtt{__c}_{\Xi}(X_1, \ldots, X_k)$. The context ρ does not change either. The first argument to `__nu`, which is the process $\mathtt{new}X; P$ itself, is a constant.

As in the lean semantics, the rules for inferring the values of expressions e aggravate any preexisting loss of information. This is done so as to produce judgments that can be easily translated into the $\mathcal{H}_1$ class. The main source of information loss is the fact that we forget about environments in judgments $P \vdash e \simeq t$. In other words, we really ought to write judgments of the form $P\langle\!\langle \rho \rangle\!\rangle \vdash e \simeq t$, which would say that if we reach program point P under context ρ, then the

DRAFT

value of e is t. Instead, we just say that this is so whenever P can be reached, under any context.

$$\frac{\Xi = X_1, \ldots, X_k \quad \Xi_i = X_i, \ldots, X_k \quad \text{(for some } 1 \leq i \leq k+1) \qquad \Xi \vdash E \to P\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle}{P \vdash __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k) \simeq __\mathtt{c}_{\Xi_i}(t_i, \ldots, t_k)} \ (__\mathtt{c}_{X_i}\simeq)$$

$$\frac{\Xi = X_1, \ldots, X_k \quad 1 \leq i \leq k \qquad \Xi \vdash E \to P\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle}{P \vdash X_i \simeq t_i} \ (Var\simeq)$$

$$\frac{\Xi = X_1, \ldots, X_k \quad 1 \leq i_1, \ldots, i_n \leq k \quad i_1, \ldots, i_n \text{ pairwise distinct} \qquad \Xi \vdash E \to P\langle\langle __\mathtt{c}_\Xi(t_1, , t_k)\rangle\rangle}{P \vdash f(X_{i_1}, \ldots, X_{i_n}) \simeq f(t_{i_1}, \ldots, t_{i_n})} \ (Flat\simeq)$$

$$\frac{P \vdash e_1 \simeq t_1 \quad \ldots \quad P \vdash e_n \simeq t_n}{P \vdash f(e_1, \ldots, e_n) \simeq f(t_1, \ldots, t_n)} \ (f'\simeq)$$

In rule $(Flat\simeq)$, we require $f \in \Sigma$, $\Sigma(f) = n$, but we may think this is implicit. In rule $(f'\simeq)$, we require $f \in \Sigma$, $\Sigma(f) = n$, and either some e_i is not a variable or all are variables but at least two are identical. In other words, $(f'\simeq)$ only applies if none of $(__\mathtt{c}_{X_i}\simeq)$, $(Var\simeq)$, $(Flat\simeq)$ apply. Rule $(Var\simeq)$ applies only if there is no super-expression on which $(__\mathtt{c}_{X_i}\simeq)$ or $(Flat\simeq)$ applies.

To emit a message, we do essentially as in the lean semantics, adding contexts to processes. We could content ourselves with writing analogous rules, i.e., if $\Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\langle\rho\rangle\rangle$ (we can reach $\mathtt{out}(e_1, e_2); P$ with context ρ), if $\mathtt{out}(e_1, e_2); P \vdash e_1 \simeq t_1$ (e_1's value may be t_1), and if $\mathtt{out}(e_1, e_2); P \vdash e_2 \simeq t_2$ (e_2's value may be t_2), then, first, $\Xi \vdash \mathtt{out}(e_1, e_2); P \to P\langle\langle\rho\rangle\rangle$ (we can reach P with an unmodified context, ρ), and second, $(\mathtt{out}(e_1, e_2); P)\langle\langle\rho\rangle\rangle : t_1 \triangleright t_2$ ($\mathtt{out}(e_1, e_2); P$ send t_2 on channel t_1).

However, there are interesting special cases that deserve to be considered in a special way, so as to lose as little information possible, namely when e_1 is a variable, and the case where e_2 is a name $__\mathtt{nu}(Q, v)$. We consider that, as in the pi-calculus or the spi-calculus, communication can only take place on channels, and that only names can be values of channels. This can only happen if e_1 is a variable (e.g., a formal parameter, or something gotten from some other communication channel using `in`), or if e_1 is of the form $__\mathtt{nu}(Q, v)$ (i.e., when the current process wants to output on a channel it itself created earlier).

Let's look at the case where e_1 is a name.

$$\frac{e_1 = __\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k)) \qquad \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle \quad \mathtt{out}(e_1, e_2); P \vdash e_2 \simeq u \qquad \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\langle\rho\rangle\rangle}{\vdash (\mathtt{out}(e_1, e_2); P)\langle\langle\rho\rangle\rangle : __\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(t_i, \ldots, t_k)) \triangleright u} \ (__\mathtt{c}_{X_i}\mathtt{out}'\triangleright)$$

This is a bit tricky, as the two premises $\Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle$ and $\Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\langle\rho\rangle\rangle$ may seem redundant (we really mean that $\rho = __\mathtt{c}_\Xi(t_1, \ldots, t_k)$). However,

DRAFT

dispensing with the second one and replacing ρ by $__\mathtt{c}_{\Xi}(t_1, \ldots, t_k)$ in the conclusion would make this rule a non-$\mathcal{H}_1$ clause. Call this the *premise duplication trick*.

When e_1 is a variable, we check that its value is a name.

$$\frac{\begin{array}{l} \qquad\qquad \Xi = X_1, \ldots, X_k \quad e_1 = X_i \quad 1 \leq i \leq k \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle __\mathtt{c}_{\Xi}(t_1, \ldots, t_{i-1}, __\mathtt{nu}(s,t), t_{i+1}, \ldots, t_k)\rangle\!\rangle \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle __\mathtt{c}_{\Xi}(t_1, \ldots, t_k)\rangle\!\rangle \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle \rho \rangle\!\rangle \qquad\qquad \mathtt{out}(e_1, e_2); P \vdash e_2 \simeq u \end{array}}{\vdash (\mathtt{out}(e_1, e_2); P)\langle\!\langle \rho \rangle\!\rangle : t_i \rhd u} \; (Var\mathtt{out}'\rhd)$$

We again use the premise duplication trick; here we make three copies of what should essentially be the same premise.

The above rules say what messages are sent on the relevant channel. The following states that $\mathtt{out}(e_1, e_2); P$ may proceed to execute P, in the same cases.

$$\frac{\begin{array}{c} e_1 = __\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k)) \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle \rho \rangle\!\rangle \quad \mathtt{out}(e_1, e_2); P \vdash e_2 \simeq u \end{array}}{\Xi \vdash \mathtt{out}(e_1, e_2); P \to P\langle\!\langle \rho \rangle\!\rangle} \; (__\mathtt{c}_{X_i}\mathtt{out}')$$

$$\frac{\begin{array}{l} \qquad\qquad \Xi = X_1, \ldots, X_k \quad e_1 = X_i \quad 1 \leq i \leq k \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle __\mathtt{c}_{\Xi}(t_1, \ldots, t_{i-1}, __\mathtt{nu}(s,t), t_{i+1}, \ldots, t_k)\rangle\!\rangle \\ \Xi \vdash E \to \mathtt{out}(e_1, e_2); P\langle\!\langle \rho \rangle\!\rangle \qquad \mathtt{out}(e_1, e_2); P \vdash e_2 \simeq u \end{array}}{\vdash \Xi \vdash \mathtt{out}(e_1, e_2); P \to P\langle\!\langle \rho \rangle\!\rangle} \; (Var\mathtt{out}')$$

Let's now read messages. The idea is that, if $\Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\!\langle __\mathtt{c}_{\Xi}(t_1, \ldots, t_k)\rangle\!\rangle$ (we can reach $\mathtt{in}(e, X); Q$), $\vdash P'\langle\!\langle \rho' \rangle\!\rangle : u_1 \rhd u_2$ (some process P' sent a message u_2 on channel u_1), and $\mathtt{in}(e, X); Q \vdash e \simeq u_1$ (Q actually expects some input on channel u_1), then $X, \Xi \vdash \mathtt{in}(e, X); Q \to Q\langle\!\langle __\mathtt{c}_{X,\Xi}(u_2, t_1, \ldots, t_k)\rangle\!\rangle$ on the one hand (you can reach Q with X bound to u_2), and $P'\langle\!\langle \rho' \rangle\!\rangle : t_1 \rhd t_2 : (\mathtt{in}(e, X); Q)\langle\!\langle \rho \rangle\!\rangle$ on the other hand (Q received its message from P'). However, just writing $\mathtt{in}(e, X); Q \vdash e \simeq u_1$ as a premise incurs some loss of precision, in that we require that the value of e be u_1, disregarding the context $__\mathtt{c}_{\Xi}(t_1, \ldots, t_k)$ that we knew from the other premise $\Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\!\langle __\mathtt{c}_{\Xi}(t_1, \ldots, t_k)\rangle\!\rangle$. In some cases, i.e., when e is a variable (as in rule $(Var\simeq)$), or when e is a term of the form $__\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k))$ with Ξ_i a suffix of Ξ (as in rule $(__\mathtt{c}_{X_i}\simeq)$), or when e is a flat term $f(X_{i_1}, \ldots, X_{i_n})$ (as in rule $(Flat\simeq)$), we can actually relate the value of e to the context $__\mathtt{c}_{\Xi}(t_1, \ldots, t_k)$, while remaining inside the $\mathcal{H}_1$ class.

In fact, we only have to deal with the cases where e is a variable X_i or a name $__\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k))$. This is because only names can be used as communication channels, as in the pi-calculus or the spi-calculus.

Let's deal with the case where e is of the form $__\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k))$ first. This is the case where Q expects some data from a fresh channel (created by `new` earlier in the same

DRAFT

process).

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad \Xi_i = X_i, \ldots, X_k \quad \text{(for some } 1 \leq i \leq k+1\text{)} \\ e = _\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k)) \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle \\ \vdash P'\langle\langle \rho' \rangle\rangle : _\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(t_i, \ldots, t_k)) \triangleright u\end{array}}{X, \Xi \vdash \mathtt{in}(e, X); Q \to Q\langle\langle __\mathtt{c}_{X,\Xi}(u, t_1, \ldots, t_k)\rangle\rangle} \; (__\mathtt{c}_{X_i}\mathtt{in}')$$

Rule $(__\mathtt{c}_{X_i}\mathtt{in}')$ is the only one, until now, where we extend the context $\rho = __\mathtt{c}_\Xi(t_1, \ldots, t_k)$ inside $\langle\langle _ \rangle\rangle$, to $__\mathtt{c}_{X,\Xi}(u, t_1, \ldots, t_k)$. Another will be given by rules on `case` expressions below.

In this case, we write the following analogue of $(\mathtt{in}\triangleright)$:

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad \Xi_i = X_i, \ldots, X_k \quad \text{(for some } 1 \leq i \leq k+1\text{)} \\ e = _\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(X_i, \ldots, X_k)) \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle \quad \vdash P'\langle\langle \rho' \rangle\rangle : _\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(t_i, \ldots, t_k)) \triangleright u \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle \rho \rangle\rangle \quad \vdash P'\langle\langle \rho' \rangle\rangle : s \triangleright u\end{array}}{\vdash P'\langle\langle \rho' \rangle\rangle : s \triangleright u : (\mathtt{in}(e, X); Q)\langle\langle \rho \rangle\rangle} \; (__\mathtt{c}_{X_i}\mathtt{in}'\triangleright)$$

We again use the premise duplication trick here. The two premises $\Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle$ and $\Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle \rho \rangle\rangle$ may seem redundant (we really mean that $\rho = __\mathtt{c}_\Xi(t_1, \ldots, t_k)$). However, dispensing with the second one and replacing ρ by $__\mathtt{c}_\Xi(t_1, \ldots, t_k)$ in the conclusion would make this rule a non-$\mathcal{H}_1$ clause. Similarly with the two premises $\vdash P'\langle\langle \rho' \rangle\rangle : _\mathtt{nu}(Q_1, __\mathtt{c}_{\Xi_i}(t_i, \ldots, t_k)) \triangleright u$ and $\vdash P'\langle\langle \rho' \rangle\rangle : s \triangleright u$.

There is no need for a rule such as $(\mathtt{in}\simeq)$, which allowed one to conclude that u was a possible value for X after executing $\mathtt{in}(e, X)$, in the lean semantics. Indeed, this is done here by applying rule $(__\mathtt{c}_X\simeq)$, profiting from the fact that all values of variables are recorded in the $\langle\langle _ \rangle\rangle$ part of the conclusion of rule $(__\mathtt{c}_{X_i}\mathtt{in}')$.

Second, the case where e is a variable X_i is the most common. It is the case where Q expects data from a channel that was passed as a parameter to it, or on yet another channel. Note that the value of the channel is the term t_i, which should be restricted to be a name, i.e., a term of the form $_\mathtt{nu}(s, t)$. But doing so would produce a clause outside of $\mathcal{H}_1$. We use the premise duplication trick again.

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad e = X_i \quad 1 \leq i \leq k \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle \qquad \vdash P'\langle\langle \rho' \rangle\rangle : t_i \triangleright u \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_{i-1}, _\mathtt{nu}(u, v), t_{i+1}, \ldots, t_k)\rangle\rangle\end{array}}{X, \Xi \vdash \mathtt{in}(e, X); Q \to Q\langle\langle __\mathtt{c}_{X,\Xi}(u, t_1, \ldots, t_k)\rangle\rangle} \; (Var\mathtt{in}')$$

Again, the corresponding analogue of $(\mathtt{in}\triangleright)$ uses the premise duplication trick.

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad e = X_i \quad 1 \leq i \leq k \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_k)\rangle\rangle \qquad \vdash P'\langle\langle \rho' \rangle\rangle : t_i \triangleright u \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle \rho \rangle\rangle \\ \Xi \vdash E \to (\mathtt{in}(e, X); Q)\langle\langle __\mathtt{c}_\Xi(t_1, \ldots, t_{i-1}, _\mathtt{nu}(u, v), t_{i+1}, \ldots, t_k)\rangle\rangle\end{array}}{\vdash P'\langle\langle \rho' \rangle\rangle : t_i \triangleright u : (\mathtt{in}(e, X); Q)\langle\langle \rho \rangle\rangle} \; (Var\mathtt{in}'\triangleright)$$

DRAFT

Just as for $(\texttt{new}')$, in expressions of the form $\texttt{let}\ X = e\ \texttt{in}\ P$ we have the choice between recording the value of the bound variable X inside the environment (the $\langle\!\langle \rho \rangle\!\rangle$ part in the judgment $\Xi \vdash E \to (\texttt{let}\ X = e\ \texttt{in}\ P)\langle\!\langle \rho \rangle\!\rangle$), or to replace X by e in P. The latter is more precise, so we choose this way of expressing the semantics of $\texttt{let}$ expressions.

$$\frac{\Xi \vdash E \to (\texttt{let}\ X = e\ \texttt{in}\ P)\langle\!\langle \rho \rangle\!\rangle}{\Xi \vdash \texttt{let}\ X = e\ \texttt{in}\ P \to P[X := e]\langle\!\langle \rho \rangle\!\rangle}\ (\texttt{let}')$$

Case expressions are then evaluated in the obvious way.

$$\frac{\begin{array}{c}\Xi \vdash E \to (\texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle __\texttt{c}_\Xi(u_1, \ldots, u_k) \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq f(t_1, \ldots, t_n)\end{array}}{\begin{array}{c}X_1, \ldots, X_n, \Xi \vdash \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \to P\langle\!\langle __\texttt{c}_{X_1,\ldots,X_n,\Xi}(t_1, \ldots, t_n, u_1, \ldots, u_k) \rangle\!\rangle\end{array}}\ (\texttt{case}f)$$

$$\frac{\begin{array}{c}\Xi \vdash E \to (\texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle \rho \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq g(t_1, \ldots, t_m) \\ g \in \Sigma,\ \Sigma(g) = m,\ g \neq f\end{array}}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \to Q\langle\!\langle \rho \rangle\!\rangle}\ (\texttt{case}f \neq g)$$

As in the lean semantics, there are as many instances of rule $(\texttt{case}f \neq g)$ as there are symbols g other than f. This says that you may execute the else branch as soon as e_1 may have a value that does not match the pattern $f(X_1, \ldots, X_n)$.

As promised in Section 3.1, we improve on the lean semantics for symmetric decryption. Look at expression $\texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q$. Instead of checking that e_1 has some value $\{t\}_u$ such that u is also a value of e_2, we shall impose that e_1 has a value of the form $\{t\}_{e_2\sigma}$, where σ is some substitution of the free variables of e_2 for terms. Additionally, the most interesting case is when e_1, and deserves a special rule of its own.

Accordingly, we have two rules replacing $(\texttt{case}\{\})$. When e_1 is a variable X_i,

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad 1 \leq i \leq k \\ \text{The free variables of } e_2 \text{ are } X_{i_1}, \ldots, X_{i_m}, 1 \leq i_1 < \ldots < i_m \leq k \\ P' = \texttt{case}\ X_i\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \\ \Xi \vdash E \to P'\langle\!\langle __\texttt{c}_\Xi(t_1, \ldots, t_k) \rangle\!\rangle \\ \Xi \vdash E \to P'\langle\!\langle __\texttt{c}_\Xi(t_1, \ldots, t_{i-1}, \{t\}_{e_2[X_{i_1} := t_{i_1}, \ldots, X_{i_m} := t_{i_m}]}, t_{i+1}, \ldots, t_k) \rangle\!\rangle\end{array}}{X, \Xi \vdash P' \to P\langle\!\langle __\texttt{c}_{X,\Xi}(t, t_1, \ldots, t_k) \rangle\!\rangle}\ (Var\texttt{case}'\{\})$$

Again, we are using the premise duplication trick in order to approximate an equality, here $t_i = \{t\}_{e_2[X_{i_1} := t_{i_1}, \ldots, X_{i_m} := t_{i_m}]}$. Otherwise,

$$\frac{\begin{array}{c}\Xi = X_1, \ldots, X_k \quad e_1 \text{ not a variable} \\ \text{The free variables of } e_2 \text{ are } X_{i_1}, \ldots, X_{i_m}, 1 \leq i_1 < \ldots < i_m \leq k \\ \Xi \vdash E \to (\texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle __\texttt{c}_\Xi(t_1, \ldots, t_k) \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq \{t\}_{e_2[X_{i_1} := t_{i_1}, \ldots, X_{i_m} := t_{i_m}]}\end{array}}{X, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \to P\langle\!\langle __\texttt{c}_{X,\Xi}(t, t_1, \ldots, t_k) \rangle\!\rangle}\ (\texttt{case}'\{\})$$

DRAFT

Then, we have a rule saying that decryption may fail. This may happen because the value of e_1 is not a symmetric encryption:

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow (\texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle \rho \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq g(t_1, \ldots, t_m) \\ g \in \Sigma,\ \Sigma(g) = m,\ g \neq \texttt{crypt}\end{array}}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \rightarrow Q\langle\!\langle \rho \rangle\!\rangle}\ (\texttt{case}'\{\} \neq g)$$

This may happen also because the value of e_1 is a symmetric encryption with the wrong key.

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow (\texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle \rho \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq \{t\}_u \\ \texttt{case}\ e_1\ \texttt{of}\ f(X_1, \ldots, X_n) \Rightarrow P\ \texttt{else}\ Q \vdash e_2 \not\simeq u\end{array}}{\Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ \{X\}_{e_2} \Rightarrow P\ \texttt{else}\ Q \rightarrow Q\langle\!\langle \rho \rangle\!\rangle}\ (\texttt{case}'\{\} \neq)$$

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow (\texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle \rho \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq [t]_u \\ \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \vdash e_2^{-1} \simeq u\end{array}}{X, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \rightarrow P\langle\!\langle __\mathsf{c}_X(t, \rho) \rangle\!\rangle}\ (\texttt{case}'[])$$

$$\frac{\begin{array}{c}\Xi \vdash E \rightarrow (\texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q)\langle\!\langle \rho \rangle\!\rangle \\ \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \vdash e_1 \simeq t\end{array}}{X, \Xi \vdash \texttt{case}\ e_1\ \texttt{of}\ [X]_{e_2} \Rightarrow P\ \texttt{else}\ Q \rightarrow Q}\ (\texttt{case}[] \neq)$$

4 The Full Language

In the full language, *patterns* are meant to denote expressions matching only certain terms. For example, the pattern X (a variable) matches every term. The pattern $\texttt{cons2}(X, Y)$ matches every term of the form $\texttt{cons2}(u, v)$, if $\texttt{cons}$ is declared as a constructor of arity 2.

$pat, \ldots$	$::=$	X	variables
	$\mid$	$f(pat_1, \ldots, pat_n)$	application of constructor f
	$\mid$	$\{pat_1\}_{e_2}$	symmetric decryption
	$\mid$	$[pat_1]_{e_2}$	asymmetric decryption
	$\mid$	$= e$	equality check

The encryption patterns $\{pat_1\}_{e_2}$ and $[pat_1]_{e_2}$ use terms, not patterns, in key positions. This is intended. Matching a term u against $\{pat_1\}_{e_2}$ means checking that u is a symmetric encryption, using key e_2, of something matching pat_1. Matching a term u against $[pat_1]_{e_2}$ means checking that u is an asymmetric encryption, using the *inverse* of key e_2, of something matching pat_1.

Patterns are restricted to be *linear* and *non-ambiguous*. Linearity means that no pattern binds the same variable twice; e.g., $\texttt{cons}(X, X)$ is forbidden as a pattern because X occurs twice in it. Non-ambiguity means that no variable free in any subexpression (such as e_2 or e above) is also

DRAFT

bound in the pattern. For example, $\{X\}_X$ is forbidden as a pattern. Intuitively, matching against it would mean decrypting with the current value of X, getting the plaintext and binding it to the same variable X, but this is hard to read.

The $= e$ pattern only matches the term e, and is concrete semantics for `__eq`(e). This is useful for nonce verification, for instance.

In the core language, patterns are restricted to so-called *core patterns*, defined by the grammar:

$epat, \ldots$	$::=$	X	variables
	$\mid$	$f(X_1, \ldots, X_n)$	application of constructor f to n distinct variables
	$\mid$	$\{X\}_{e_2}$	symmetric decryption
	$\mid$	$[X]_{e_2}$	asymmetric decryption
	$\mid$	$= e$	equality check

References

Abadi, M. and Gordon, A. D. (1997). A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security (CCS)*, pages 36–47.

Blanchet, B. (2001). An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society Press.

Blanchet, B. (2004). *Cryptographic Protocol Verifier User Manual*. Ecole Normale Supérieure and Max-Planck-Institut für Informatik. `http://www.di.ens.fr/~blanchet/`.

Claessen, K. and Sörensson, N. (2003). New techniques that improve MACE-style finite model building. `http://www.uni-koblenz.de/~peter/models03/final/soerensson/main.ps`.

Goubault-Larrecq, J. (2001). Les syntaxes et la sémantique du langage de spécification EVA. Rapport numéro 3 du projet RNTL EVA. 32 pages.

Goubault-Larrecq, J. (2004). Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA 2004), Sainte-Marie-de-Ré, France, Jan. 2004*, pages 1–40. INRIA, collection didactique.

Goubault-Larrecq, J. (2005). Deciding $\mathcal{H}_1$ by resolution. *Information Processing Letters*, 95(3):401–408.

Nielson, F., Nielson, H. R., and Seidl, H. (2002). Normalizable Horn clauses, strongly recognizable relations and Spi. In *9th Static Analysis Symposium (SAS)*. Springer Verlag LNCS 2477.

DRAFT