Towards Automatization of Framed Bisimilarity in Coq

M. Miculan   I. Scagnetto

Dipartimento di Matematica e Informatica
Università di Udine

TYPES Annual Workshop, April 2006

Background.

Processes algebras and cryptographic protocols: the spi-calculus.

- The study of reactive systems requires to consider both the steps taken by the system and those taken by its environment.
- The spi-calculus is an extension of the π-calculus designed for reasoning about cryptographic protocols. In particular terms exchanged during communications can be encrypted with a shared-key scheme:

$$c.(x)P \mid \overline{c}.\langle\{M\}_K\rangle Q \xrightarrow{\tau} P[\{M\}_K/x] \mid Q$$

- The environment may be hostile and little can be assumed about its behaviour.
- As a consequence, representing the environment as a nondeterministic process is hard, so bisimulation techniques are often used.

Background.

Processes algebras and cryptographic protocols: the spi-calculus.

- The study of reactive systems requires to consider both the steps taken by the system and those taken by its environment.
- The spi-calculus is an extension of the π-calculus designed for reasoning about cryptographic protocols. In particular terms exchanged during communications can be encrypted with a shared-key scheme:

$$c.(x)P \mid \overline{c}.\langle\{M\}_K\rangle Q \xrightarrow{\tau} P[\{M\}_K/x] \mid Q$$

- The environment may be hostile and little can be assumed about its behaviour.
- As a consequence, representing the environment as a nondeterministic process is hard, so bisimulation techniques are often used.

Background.

Processes algebras and cryptographic protocols: the spi-calculus.

- The study of reactive systems requires to consider both the steps taken by the system and those taken by its environment.
- The spi-calculus is an extension of the π-calculus designed for reasoning about cryptographic protocols. In particular terms exchanged during communications can be encrypted with a shared-key scheme:

$$c.(x)P \mid \overline{c}.\langle\{M\}_K\rangle Q \xrightarrow{\tau} P[\{M\}_K/x] \mid Q$$

- The environment may be hostile and little can be assumed about its behaviour.
- As a consequence, representing the environment as a nondeterministic process is hard, so bisimulation techniques are often used.

Background.

Processes algebras and cryptographic protocols: the spi-calculus.

- The study of reactive systems requires to consider both the steps taken by the system and those taken by its environment.
- The spi-calculus is an extension of the π-calculus designed for reasoning about cryptographic protocols. In particular terms exchanged during communications can be encrypted with a shared-key scheme:

$$c.(x)P \mid \overline{c}.\langle\{M\}_K\rangle Q \xrightarrow{\tau} P[\{M\}_K/x] \mid Q$$

- The environment may be hostile and little can be assumed about its behaviour.
- As a consequence, representing the environment as a nondeterministic process is hard, so bisimulation techniques are often used.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.
Testing equivalence

- Usually, testing equivalence ($\sim$) is used in order to reason about processes.
- Intended meaning of $P \sim Q$:
  - P is the implementation of a protocol,
  - Q is the specification of the protocol.

  If the equivalence holds, the implementation of the protocol meets the corresponding specification.
- This approach is applied for verifying many protocols.
- Another interesting application: PCA (PCC for security purposes):
  - P is the mobile code received from the producer,
  - Q is the security policy specified by the consumer,
  - "$d : P \sim Q$" (proof that P complies to Q): provided by the producer and checked by the consumer.

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.
Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Background.

Indistinguishable terms and Framed Bisimilarity.

- Verifying testing equivalences is difficult.
- Moreover, when reasoning about cryptographic protocols new challenges arise:
  - two cleartexts M and N are encrypted under a session key, yielding two cyphertexts $P(M)$ and $P(N)$,
  - in order to express preservation of secrecy, an attacker should not be able to distinguish between $P(M)$ and $P(N)$,
  - standard notions of bisimulations do not allow that; hence it is necessary to relax the usual definition in order to introduce indistinguishable messages.
- Framed Bisimulation address both problems and is more tractable; moreover, we have: $P \sim_f Q \Rightarrow P \sim Q$
- Framed Bisimulation is decidable is we consider a suitable finite fragment of the spi-calculus and there exists a decision algorithm provided by Hüttel in [2].

Our idea.

- Our work in progress focus on the integration of proof-assistants and automatic decision procedures.
- We aim to provide a Coq-signature such that the user can specify its protocol and the goal-equivalence $P \sim Q$.
- The proof can then proceed interactively, as usual, but with the possibility of invoking an ad-hoc tactic to automatically verify finite subgoals.
- Eventually, the tactic could not terminate or fail if a depth limit is imposed.

Our idea.

- Our work in progress focus on the integration of proof-assistants and automatic decision procedures.
- We aim to provide a Coq-signature such that the user can specify its protocol and the goal-equivalence $P \sim Q$.
- The proof can then proceed interactively, as usual, but with the possibility of invoking an ad-hoc tactic to automatically verify finite subgoals.
- Eventually, the tactic could not terminate or fail if a depth limit is imposed.

Our idea.

- Our work in progress focus on the integration of proof-assistants and automatic decision procedures.
- We aim to provide a Coq-signature such that the user can specify its protocol and the goal-equivalence $P \sim Q$.
- The proof can then proceed interactively, as usual, but with the possibility of invoking an ad-hoc tactic to automatically verify finite subgoals.
- Eventually, the tactic could not terminate or fail if a depth limit is imposed.

Our idea.

- Our work in progress focus on the integration of proof-assistants and automatic decision procedures.
- We aim to provide a Coq-signature such that the user can specify its protocol and the goal-equivalence $P \sim Q$.
- The proof can then proceed interactively, as usual, but with the possibility of invoking an ad-hoc tactic to automatically verify finite subgoals.
- Eventually, the tactic could not terminate or fail if a depth limit is imposed.

Problems.

- In general it is not sufficient to have an “oracle” able to say “yes/no” (which amounts to introduce a new axiom for the related case) when invoked on a goal $P \sim_f Q$, since it can be bugged.
- Moreover, this approach is not acceptable in PCA.
- Hence, we need a tactic which can provide an effective witness.
- Thus, eventual bugs in the algorithm/implementation can be easily spotted (and the size of TCB decreases).

Problems.

- In general it is not sufficient to have an “oracle” able to say “yes/no” (which amounts to introduce a new axiom for the related case) when invoked on a goal $P \sim_f Q$, since it can be bugged.
- Moreover, this approach is not acceptable in PCA.
- Hence, we need a tactic which can provide an effective witness.
- Thus, eventual bugs in the algorithm/implementation can be easily spotted (and the size of TCB decreases).

Problems.

- In general it is not sufficient to have an “oracle” able to say “yes/no” (which amounts to introduce a new axiom for the related case) when invoked on a goal $P \sim_f Q$, since it can be bugged.
- Moreover, this approach is not acceptable in PCA.
- Hence, we need a tactic which can provide an effective witness.
- Thus, eventual bugs in the algorithm/implementation can be easily spotted (and the size of TCB decreases).

Problems.

- In general it is not sufficient to have an “oracle” able to say “yes/no” (which amounts to introduce a new axiom for the related case) when invoked on a goal $P \sim_f Q$, since it can be bugged.
- Moreover, this approach is not acceptable in PCA.
- Hence, we need a tactic which can provide an effective witness.
- Thus, eventual bugs in the algorithm/implementation can be easily spotted (and the size of TCB decreases).

Status of the work.

- Implementation in Coq: done (using weak-HOAS, coinductive types, multiple judgments, capitalizing on similar experience with π-calculus, ambients, ...).
- Testing of the implementation, by manual verification of some example equivalence: done.
- Implementation of the tactic for finite processes: to do
  - modification of existing algorithms to produce witnesses of equivalences,
  - implementation as `Ltac`.

Status of the work.

- Implementation in Coq: done (using weak-HOAS, coinductive types, multiple judgments, capitalizing on similar experience with π-calculus, ambients, . . .).
- Testing of the implementation, by manual verification of some example equivalence: done.
- Implementation of the tactic for finite processes: to do
  - modification of existing algorithms to produce witnesses of equivalences,
  - implementation as `Ltac`.

Status of the work.

- Implementation in Coq: done (using weak-HOAS, coinductive types, multiple judgments, capitalizing on similar experience with π-calculus, ambients, ...).
- Testing of the implementation, by manual verification of some example equivalence: done.
- Implementation of the tactic for finite processes: to do
  - modification of existing algorithms to produce witnesses of equivalences,
  - implementation as `Ltac`.

Status of the work.

- Implementation in Coq: done (using weak-HOAS, coinductive types, multiple judgments, capitalizing on similar experience with π-calculus, ambients, . . .).
- Testing of the implementation, by manual verification of some example equivalence: done.
- Implementation of the tactic for finite processes: to do
  - modification of existing algorithms to produce witnesses of equivalences,
  - implementation as `Ltac`.

Names, Variables and Terms.

(Names) $\mathcal{N}$ $\rightsquigarrow$ `Parameter Name : Set.`
`forall m n:Name, m = n + m <> n.`

(Variables) $\mathcal{V}$ $\rightsquigarrow$ `Parameter Var : Set.`

Terms are encoded by means of an inductive type:

```
Inductive Term : Set :=
  name : Name -> Term                        (name)
| var   : Var -> Term                        (variable)
| zero : Term                                (zero)
| suc   : Term -> Term                       (successor)
| pair : Term -> Term -> Term                (pair)
| sk_enc : Term -> Term -> Term.             (shared-key encryption)
```

Names, Variables and Terms.

(Names) $\mathcal{N}$ $\rightsquigarrow$ `Parameter Name : Set.`

`forall m n:Name, m = n + m <> n.`

(Variables) $\mathcal{V}$ $\rightsquigarrow$ `Parameter Var : Set.`

Terms are encoded by means of an inductive type:

```
Inductive Term : Set :=
  name : Name -> Term                    (name)
| var  : Var -> Term                     (variable)
| zero : Term                            (zero)
| suc  : Term -> Term                    (successor)
| pair : Term -> Term -> Term            (pair)
| sk_enc : Term -> Term -> Term.         (shared-key encryption)
```

Names, Variables and Terms.

(Names) $\mathcal{N}$ $\rightsquigarrow$ `Parameter Name : Set.`

`forall m n:Name, m = n + m <> n.`

(Variables) $\mathcal{V}$ $\rightsquigarrow$ `Parameter Var : Set.`

Terms are encoded by means of an inductive type:

```
Inductive Term : Set :=
  name : Name -> Term                    (name)
| var  : Var -> Term                     (variable)
| zero : Term                            (zero)
| suc  : Term -> Term                    (successor)
| pair : Term -> Term -> Term            (pair)
| sk_enc : Term -> Term -> Term.         (shared-key encryption)
```

Names, Variables and Terms.

(Names) $\mathcal{N}$ $\rightsquigarrow$ `Parameter Name : Set.`

`forall m n:Name, m = n + m <> n.`

(Variables) $\mathcal{V}$ $\rightsquigarrow$ `Parameter Var : Set.`

Terms are encoded by means of an inductive type:

```
Inductive Term : Set :=
  name : Name -> Term                  (name)
| var  : Var -> Term                   (variable)
| zero : Term                          (zero)
| suc  : Term -> Term                  (successor)
| pair : Term -> Term -> Term          (pair)
| sk_enc : Term -> Term -> Term.       (shared-key encryption)
```

Names, Variables and Terms.

(Names) $\mathcal{N}$ $\rightsquigarrow$ `Parameter Name : Set.`
`forall m n:Name, m = n + m <> n.`
(Variables) $\mathcal{V}$ $\rightsquigarrow$ `Parameter Var : Set.`

Terms are encoded by means of an inductive type:

```
Inductive Term : Set :=
  name : Name -> Term                      (name)
| var  : Var -> Term                       (variable)
| zero : Term                              (zero)
| suc  : Term -> Term                      (successor)
| pair : Term -> Term -> Term              (pair)
| sk_enc : Term -> Term -> Term.           (shared-key encryption)
```

Processes.

Processes are also encoded by means of an inductive type:

```
Inductive Proc : Set :=
```

plain, i.e., first order constructors:

```
  out_barb : Term -> Term -> Proc -> Proc
| par : Proc -> Proc -> Proc
...
| nil : Proc
```

(output) (parallel composition) (null process)

binders, i.e., higher order constructors:

```
| in_barb  : Term -> (Var-> Proc) -> Proc
...
| nu  : (Name -> Proc) -> Proc.
```

(input) (restriction)

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Processes.

Processes are also encoded by means of an inductive type:

```
Inductive Proc : Set :=
```

plain, i.e., first order constructors:

```
  out_barb : Term -> Term -> Proc -> Proc (output)
| par : Proc -> Proc -> Proc (parallel composition)
...
| nil : Proc (null process)
```

binders, i.e., higher order constructors:

```
| in_barb  : Term -> (Var-> Proc) -> Proc (input)
...
| nu  : (Name -> Proc) -> Proc. (restriction)
```

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Processes.

Processes are also encoded by means of an inductive type:

```
Inductive Proc : Set :=
```

plain, i.e., first order constructors:

```
  out_barb : Term -> Term -> Proc -> Proc
```
(output)

```
| par : Proc -> Proc -> Proc
```
(parallel composition)

```
...
| nil : Proc
```
(null process)

binders, i.e., higher order constructors:

```
| in_barb  : Term -> (Var-> Proc) -> Proc
```
(input)

```
...
| nu   : (Name -> Proc) -> Proc.
```
(restriction)

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Processes.

Processes are also encoded by means of an inductive type:

```
 Inductive Proc : Set :=
```

plain, i.e., first order constructors:

```
  out_barb : Term -> Term -> Proc -> Proc (output)
| par : Proc -> Proc -> Proc (parallel composition)
...
| nil : Proc (null process)
```

binders, i.e., higher order constructors:

```
| in_barb  : Term -> (Var-> Proc) -> Proc (input)
...
| nu   : (Name -> Proc) -> Proc. (restriction)
```

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Processes.

Processes are also encoded by means of an inductive type:

```
 Inductive Proc : Set :=
```

plain, i.e., first order constructors:

```
  out_barb : Term -> Term -> Proc -> Proc
```
(output)

```
| par : Proc -> Proc -> Proc
```
(parallel composition)

```
...
| nil : Proc
```
(null process)

binders, i.e., higher order constructors:

```
| in_barb  : Term -> (Var-> Proc) -> Proc
```
(input)

```
...
| nu  : (Name -> Proc) -> Proc.
```
(restriction)

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Processes.

Processes are also encoded by means of an inductive type:

```
 Inductive Proc : Set :=
```
plain, i.e., first order constructors:
```
  out_barb : Term -> Term -> Proc -> Proc
```
(output)
```
| par : Proc -> Proc -> Proc
```
(parallel composition)
```
...
| nil : Proc
```
(null process)

binders, i.e., higher order constructors:
```
| in_barb  : Term -> (Var-> Proc) -> Proc
```
(input)
```
...
| nu  : (Name -> Proc) -> Proc.
```
(restriction)

As usual, the weak-HOAS encoding approach allows to delegate α-conversion and fresh renaming to the metalanguage.

Judgments

- Commitment relation $P \xrightarrow{a} A$ (modeling the dynamic behaviour of processes):

```
Inductive commit :
Proc -> Barb -> Agent -> Prop := ...
```

- Equivalence between "undistinguishable" terms $(fr, th) \vdash M \leftrightarrow N$:

```
Inductive eqTerm (fr:Frame) (th:Theory) :
Term -> Term -> Prop := ...
```

- Framed Bisimilarity $(fr, th) \vdash P \sim_f Q$:

```
CoInductive fBisim :
Frame -> Theory -> Proc -> Proc -> Prop := ...
```

Abstractions and concretions.

- Abstractions are monadic, so they can be representend in a straightforward way by functional terms over `Var`:
  `Definition Abs := Var -> Proc.`
- Concretions instead can exhibit a prefix of restrictions of arbitrary length:

$$(\nu\vec{n})\langle M\rangle Q$$

- In order to correctly render the notion of pseudo-application $(x)P@(\nu\vec{n})\langle M\rangle Q = (\nu\vec{n})(P[M/x] \mid Q)$, we need to "decompose" the prefix before carrying out the communication:

```
Inductive interactl : Abs -> Agent -> Proc -> Prop :=
 interactl_base : forall A:Abs, forall M:Term, forall P Q:Proc,
  (substProc M A P) -> (interactl A (conc_base M Q) (par P Q))
| interactl_bind : forall A:Abs, forall C:Name->Agent, forall P:Name->Proc,
  (forall n:Name, interactl A (C n) (P n)) ->

  interactl A (nu_ag C) (nu P).
```

Abstractions and concretions.

- Abstractions are monadic, so they can be representend in a straightforward way by functional terms over `Var`:
  `Definition Abs := Var -> Proc.`
- Concretions instead can exhibit a prefix of restrictions of arbitrary length:

$$(\nu\vec{n})\langle M\rangle Q$$

- In order to correctly render the notion of pseudo-application $(x)P@(\nu\vec{n})\langle M\rangle Q = (\nu\vec{n})(P[M/x] \mid Q)$, we need to "decompose" the prefix before carrying out the communication:

```
Inductive interactl : Abs -> Agent -> Proc -> Prop :=
 interactl_base : forall A:Abs, forall M:Term, forall P Q:Proc,
  (substProc M A P) -> (interactl A (conc_base M Q) (par P Q))
| interactl_bind : forall A:Abs, forall C:Name->Agent, forall P:Name->Proc,
  (forall n:Name, interactl A (C n) (P n)) ->

  interactl A (nu_ag C) (nu P).
```

Abstractions and concretions.

- Abstractions are monadic, so they can be representend in a straightforward way by functional terms over `Var`:
  `Definition Abs := Var -> Proc.`
- Concretions instead can exhibit a prefix of restrictions of arbitrary length:

$$(\nu\vec{n})\langle M\rangle Q$$

- In order to correctly render the notion of pseudo-application $(x)P@(\nu\vec{n})\langle M\rangle Q = (\nu\vec{n})(P[M/x] \mid Q)$, we need to "decompose" the prefix before carrying out the communication:

```
Inductive interactl : Abs -> Agent -> Proc -> Prop :=
 interactl_base : forall A:Abs, forall M:Term, forall P Q:Proc,
  (substProc M A P) -> (interactl A (conc_base M Q) (par P Q))
| interactl_bind : forall A:Abs, forall C:Name->Agent, forall P:Name->Proc,
  (forall n:Name, interactl A (C n) (P n)) ->

  interactl A (nu_ag C) (nu P).
```

Example.

- The processes

$$(\nu K)\overline{c}\langle\{M\}_K\rangle \text{ and } (\nu K)\overline{c}\langle\{M'\}_K\rangle$$

are in a framed bisimulation according to Example 1 of [1].

- Intuitively, this means that the abovementioned processes do not reveal M and M', respectively.
- This can be rendered in Coq as follows:

```
Lemma Example1: forall M M':Term, forall c:Name,
(closedTerm M) -> (closedTerm M') ->
exists th:Theory,
(ok (frame_add c (empty_set Name)) th) /\
(fBisim (frame_add c (empty_set Name))
        th
        (nu (fun K:Name => (out_barb (name c) (sk_enc M (name K)) nil)))
        (nu (fun K':Name => (out_barb (name c) (sk_enc M' (name K')) nil)))

).
```

- The previous lemma can be proved mimicking the proof made with “pencil and paper”.

Example.

- The processes

$$(\nu K)\overline{c}\langle\{M\}_K\rangle \text{ and } (\nu K)\overline{c}\langle\{M'\}_K\rangle$$

  are in a framed bisimulation according to Example 1 of [1].
- Intuitively, this means that the abovementioned processes do not reveal M and M', respectively.
- This can be rendered in Coq as follows:

```
Lemma Example1: forall M M':Term, forall c:Name,
(closedTerm M) -> (closedTerm M') ->
exists th:Theory,
(ok (frame_add c (empty_set Name)) th) /\
(fBisim (frame_add c (empty_set Name))
        th
        (nu (fun K:Name => (out_barb (name c) (sk_enc M (name K)) nil)))
        (nu (fun K':Name => (out_barb (name c) (sk_enc M' (name K')) nil)))

).
```

- The previous lemma can be proved mimicking the proof made with “pencil and paper”.

Example.

- The processes

$$(\nu K)\overline{c}\langle\{M\}_K\rangle \text{ and } (\nu K)\overline{c}\langle\{M'\}_K\rangle$$

  are in a framed bisimulation according to Example 1 of [1].
- Intuitively, this means that the abovementioned processes do not reveal M and M', respectively.
- This can be rendered in Coq as follows:

```
Lemma Example1: forall M M':Term, forall c:Name,
(closedTerm M) -> (closedTerm M') ->
exists th:Theory,
(ok (frame_add c (empty_set Name)) th) /\
(fBisim (frame_add c (empty_set Name))
        th
        (nu (fun K:Name => (out_barb (name c) (sk_enc M (name K)) nil)))
        (nu (fun K':Name => (out_barb (name c) (sk_enc M' (name K')) nil)))

).
```

- The previous lemma can be proved mimicking the proof made with “pencil and paper”.

Example.

- The processes

$$(\nu K)\overline{c}\langle\{M\}_K\rangle \text{ and } (\nu K)\overline{c}\langle\{M'\}_K\rangle$$

are in a framed bisimulation according to Example 1 of [1].
- Intuitively, this means that the abovementioned processes do not reveal M and M', respectively.
- This can be rendered in Coq as follows:

```
Lemma Example1: forall M M':Term, forall c:Name,
(closedTerm M) -> (closedTerm M') ->
exists th:Theory,
(ok (frame_add c (empty_set Name)) th) /\
(fBisim (frame_add c (empty_set Name))
        th
        (nu (fun K:Name => (out_barb (name c) (sk_enc M (name K)) nil)))
        (nu (fun K':Name => (out_barb (name c) (sk_enc M' (name K')) nil)))

).
```

- The previous lemma can be proved mimicking the proof made with “pencil and paper”.

References I

M. Abadi and A.D. Gordon
A Bisimulation Method for Cryptographic Protocols.
Nordic Journal of Computing, 1998.

H. Hüttel.
Deciding Framed Bisimilarity.
Pre-Proceedings of Infinity'02, June 2002.