c -- public channel

Constructs party's name from his key shared with S

$A \rightarrow B : A, N_A$
$B \rightarrow S : B, N_B, \{A,N_A\}_{Kbs}$
$S \rightarrow A : N_B, \{B,K_{ab},N_A\}_{Kas}, \{A,K_{ab},N_B\}_{Kbs}$
$A \rightarrow B : \{A,K_{ab}, N_B\}_{Kbs}, \{N_B\}_{Kab}$

```
free c.

fun host/1.
private reduc getkey(host(x)) = x.
```

Destructor getkey gives party's key from his name
private – adversary cannot use

```
(* Shared key cryptography *)

fun encrypt/2.
reduc decrypt(encrypt(x,y),y) = x.
```

public constructor and public destructor

```
(* Secrecy assumptions *)

not kas.
not kbs.
```

Hints for ProVerif
May speed up verification
Semantically insignificant

```
private free secretA, secretB.
```

new names
adversary does not see them

secrecy queries

```
query attacker:secretA; attacker:secretB.

query evinj:endAparam(x) ==> evinj:beginAparam(x).
```

endAparam and beginAparam are events
This is a correspondence query
We are interested in injective agreement
for non-inj. replace evinj with ev
ProVerif can check more complex properties about sequences of events

```
query evinj:endBparam(x) ==> evinj:beginBparam(x).
query evinj:endBkey(x,y,z,t) ==> evinj:beginBkey(x,y,z,t).
```

Identifiers can be bound to processes
These identifiers can be used where a process is normally expected

```
let processA =
1.  new Na;
    out(c, (host(kas), Na));

    in(c, (nb, m1, m2));
    let (b, kab, na2) = decrypt(m1, kas) in
    event beginBparam(b);
3.  if na2 = Na then
```

Input a triple;
Let the result of decryption be a triple.
Standard way of parsing tuples in ProVerif.

Syntactic sugar.
Could be represented with:
reduc testeq(x,x) = x.
...
let dummy = testeq(na2,Na) in ...

```
    event beginBkey(b, host(kas), nb, kab);
4.  out(c, (m2, encrypt(nb, kab)));
    (* OK *)
secret
    if b = host(kbs) then
    event endAparam(host(kas));
    out(c, encrypt(secretA, kab)).
```

One round of
•Alice acting as initiator
•Bob acting as responder

```
let processB =
    in(c, (a, na));
    event beginAparam(a);
    new Nb;
    out(c, (host(kbs), Nb, encrypt((a,na), kbs)));
    in(c, (m3, m4));
    let (=a, kab, =Nb) = decrypt(m3, kbs) in
```

Syntactic sugar.
Equivalent to
let (x1,kab,x2)=decrypt (m3,kbs) in
if x1=a & x2=Nb then

```
    if Nb = decrypt(m4, kab) then
    (* OK *)
    if a = host(kas) then
    event endBparam(host(kbs));
    event endBkey(host(kbs), a, Nb, kab);
    out(c, encrypt(secretB, kab)).

let processS =
    in(c, (b, nb, m5));
    let kbs2 = getkey(b) in
    let (a, na) = decrypt(m5,kbs2) in
    let kas2 = getkey(a) in
    new kab;
    out(c, (nb, encrypt((b, kab, na), kas2),
            encrypt((a, kab, nb), kbs2))).
```

Equivalent to
declaration
private free kas, kbs.

Alice acting only as initiator
Bob acting only as responder
(unbounded number of sessions)

```
process
    new kas; new kbs;
    ((!processA) | (!processB) | (!processS))
```

processA, processB, processS are replaced with their declarations.
NB! This is textual replacement only.
Similar to C preprocessor replacing `#define`-s