Investigations of Spatial Logic

By Martin Lundfall, Denis Erfurt

July 27, 2016

Contents

theory *RhoCalc*
imports *Main*
begin

1 Introduction

This is a formalization of a reflective, higher order process calculus known as the rho-calculus and its associated logic, namespace logic. The calculus is described by Greg Meredith in detail in the following resources:

- Namespace Logic
- Policy as Types

The main differentiating feature of the rho-calculus is that names have structure: they are built up out of quoted processes. Along with the ability to unquote or "drop" a name back to a process, this gives the calulus its higher order nature.

datatype P = *Null* $(\mathbf{0})$
| *Input n n P* $(\text{-}{\leftarrow}\text{-.-. } 80)$
| *Lift n P* $(\text{-}{\lhd}\text{-}{\rhd}\ 80)$
| *Drop n* $({}'\text{-}`\ 80)$
| *Par P P* $(\mathbf{infixl} \parallel 75)$
and n = *Quote P* $(`\text{-}')$

abbreviation *Output* :: $n \Rightarrow n \Rightarrow P$ (-[[-]])
where *Output x y* $\equiv x \lhd {}'y` \rhd$

value $a[[b]]$
value $zero \leftarrow zero.\mathbf{0}.$
value $\mathbf{0}\|\mathbf{0}$
value $'zero`$
value $zero\lhd \mathbf{0} \rhd$
value $(c{\leftarrow}d.(e).)$

fun *newName* :: *nat* $\Rightarrow n$
where *newName 0* = $`\mathbf{0}'$
|*newName* (*Suc n*) = $`$(*Output* (*newName n*) (*newName n*))$'$

abbreviation *zero* :: n
where *zero* $\equiv `\mathbf{0}'$
abbreviation *one* :: n
where *one* $\equiv$ *newName 1*
abbreviation *two* :: n
where *two* $\equiv$ *newName 2*
abbreviation *three* :: n
where *three* $\equiv$ *newName 3*
abbreviation *four* :: n
where *four* $\equiv$ *newName 4*

abbreviation *Zero* :: P
where *Zero* $\equiv$ *Null*

```
abbreviation One :: P
  where One ≡ ´one‘
abbreviation Two :: P
  where Two ≡ ´two‘
abbreviation Three :: P
  where Three ≡ ´three‘
abbreviation Four :: P
  where Four ≡ ´four‘
```

2 Structural congruence

The smallest congruence we require is structural congruence, which expresses the fact that parallelisation is commutative and associative, and running a process parallell with the Null process is the same as running that process on its own. In Meredith's paper, this is captured by the equations:

$$P|Null \equiv_C P \equiv_C Null|P \tag{1}$$

$$P|Q \equiv_C Q|P \tag{2}$$

$$(P|Q)|R \equiv_C P(Q|R) \tag{3}$$

While this seems fairly straight forward, trying to formalise this notion turns out to be quite verbose. In order to do this, we create a list of processes running in parallel and then create the structural congruence relation as a mutually recursive function comparing lists and "atomic" processes.

```
fun getList :: P ⇒ P list
where
  getList 0 = []
 |getList (a‖b) = ((getList a)@(getList b))
 |getList a = [a]
```

```
function congru :: P ⇒ P ⇒ bool (infixl =C 42)
    and eq2 :: P list ⇒ P list ⇒ P list ⇒ bool
where
   congru (a‖b) (c‖d)        = (eq2 (getList (a‖b)) (getList (c‖d)) [])
  |congru (x←y.(a).) (xx←yy.(b).) = ((a =C b) ∧ (x=xx) ∧ (y = yy))
  |congru (x◁ a ▷) (y◁ b ▷)       = ((a =C b) ∧ (x = y))
  |congru (´a‘) (´b‘)        = (a = b)
  |congru Null Null          = True
  |congru (a‖b) Null         = ((a =C Null)∧(b =C Null))
  |congru Null (a‖b)         = ((a =C Null)∧(b =C Null))
  |congru (a‖b) (c←d.(e).)  = (eq2 (getList (a‖b)) ((c←d.(e).)#[]) [])
  |congru (c←d.(e).) (a‖b)  = (eq2 (getList (a‖b)) ((c←d.(e).)#[]) [])
  |congru (a‖b) (c◁ d ▷)       = (eq2 (getList (a‖b)) ((c◁ d ▷)#[]) [])
```

|*congru* (*c*◁ *d* ▷) (*a*‖*b*) = (*eq2* (*getList* (*a*‖*b*)) ((*c*◁ *d* ▷)#[]) [])
|*congru* (*a*‖*b*) (´*c*‘) = (*eq2* (*getList* (*a*‖*b*)) ((´*c*‘)#[]) [])
|*congru* (´*c*‘) (*a*‖*b*) = (*eq2* (*getList* (*a*‖*b*)) ((´*c*‘)#[]) [])

|*congru* (´*a*‘) *Null* = *False*
|*congru* *Null* (´*b*‘) = *False*
|*congru* (´*a*‘) (-←-.-.) = *False*
|*congru* (-←-.-.) (´*b*‘) = *False*
|*congru* (´*a*‘) (-◁ - ▷) = *False*
|*congru* (-◁ - ▷) (´*b*‘) = *False*
|*congru* (-◁ - ▷) *Null* = *False*
|*congru* *Null* (-◁ - ▷) = *False*
|*congru* (-◁ - ▷) (-←-.-.) = *False*
|*congru* (-←-.-.) (-◁ - ▷) = *False*
|*congru* (-←-.-.) *Null* = *False*
|*congru* *Null* (-←-.-.) = *False*

|*eq2* [] [] [] = *True*
|*eq2* (*x*#*xs*) [] [] = *False*
|*eq2* [] [] (*a*#*as*) = *False*
|*eq2* (*x*#*xs*) [] (*b*#*bs*) = *False*
|*eq2* [] (*b*#*bs*) - = *False*
|*eq2* (*x*#*xs*) (*y*#*ys*) *zs* = (*if* (*x* =*C* *y*) *then* (*eq2* *xs* (*zs*@*ys*) []) *else* (*eq2* (*x*#*xs*) *ys* (*y*#*zs*)))
by (*auto*, *pat-completeness*)

2.1 Attempts to prove termination

We spent quite a lot of time trying to prove the termination of the above function. What follows are some of our efforts. Although termination is ultimately not proven, the function computes and by looking at some non-trivial examples we find that it does what it is supposed to.

abbreviation *maxl* **where** *maxl* ≡ λ*l*. *foldl* (λ*a*.λ*b*. (*max a b*)) *0 l*

fun *sdepth* :: *P* ⇒ *nat* **where**
sdepth Null = *0*
| *sdepth* (*x* ← *y* . *P*.) = *1* + *sdepth P*
| *sdepth* (*x* ◁ *P* ▷) = *1* + *sdepth P*
| *sdepth* (*P* ‖ *Q*) = *1* + (*max* (*sdepth P*) (*sdepth Q*))
| *sdepth* (´*x*‘) = *0*

fun *di*::*nat* ⇒ *nat* **where**
di 0 = *0*
| *di* (*Suc*(*Suc v*)) = (*Suc* (*di v*))

abbreviation *llength* **where** *llength* ≡ λ*x*::*P list*. (*di* ((*length x*)∗(*length x* + *1*)))

function
  depth :: *P* ⇒ *nat*
where
  depth **0** = *0*
  | *depth* ($x \leftarrow y$. *P*.) = *1* + *depth P*
  | *depth* ($x \lhd\ P\ \rhd$) = *1* + *depth P*
  | *depth* ($P \parallel Q$) = *1* + (*maxl* (*map depth* (*getList* ($P \| Q$)))) + *llength* (*getList* ($P \| Q$))
  | *depth* (´*x*‘) = *0*
apply *auto*
apply *pat-completeness*
done
termination
apply (*relation measure* (λx.(*sdepth x*)))
apply *auto*

sorry

termination *congru*
apply (*relation measure* (
λx. *case x of*
  Inl (*a*,*b*)
    ⇒ (*Max*({*depth a*, *depth b*}))
| *Inr* (*a*,*b*,*c*)
    ⇒ *max* (*maxl* (*map depth* (*a*))) (*maxl* (*map depth* (*b*@*c*)))))
apply *auto*
sorry

2.2 Examples of structural congruence

value ((*One*‖*Two*)‖(*Three*‖*Four*))‖*Null* =*C* ((*One*‖*Three*)‖(*Two*‖*Four*))
value (((*zero*←*zero*.**0**.) ‖ **0**) ‖ (*zero*←*zero*.**0**.)) =*C* (((*zero*←*zero*.**0**.) ‖ (*zero*←*zero*.**0**.)) ‖ **0**)
value ((**0** ‖ ($n_1 \lhd$ **0** ‖ **0** $\rhd$)) =*C* ($n_1 \lhd$ **0** $\rhd$))

theorem *example*:
shows (((*zero*←*zero*.**0**.) ‖ **0**) ‖ (*zero*←*zero*.**0**.)) =*C* (((*zero*←*zero*.**0**.) ‖ (*zero*←*zero*.**0**.)) ‖ **0**)
by *simp*

value *getList* ((*a* ‖ *b*) ‖ *c*)

declare [[*smt-timeout* = *20*]]

2.3 Structural congruence is an equivalence relation

Proofs unfortunately not completeted

abbreviation *reflexive*
  where $reflexive \equiv \lambda\ R.\ \forall\ r.\ R\ r\ r$
abbreviation *transitive*
  where $transitive \equiv \lambda\ R.\ \forall\ x.\ \forall\ y.\ \forall\ z.\ R\ x\ y \wedge R\ y\ z \longrightarrow R\ x\ z$
abbreviation *symmetric*
  where $symmetric \equiv \lambda\ R.\ \forall\ x.\ \forall\ y.\ R\ x\ y \longrightarrow R\ y\ x$

theorem *congruReflexive*:
  shows *reflexive congru*
sorry

theorem *congruTransitive*:
  shows *transitive congru*
sorry

theorem *congruSymmetric*:
  shows *symmetric congru*
sorry

theorem *eqReflexive*:
  shows $\forall\ a\ b.\ (eq2\ a\ a\ [])$
sorry

theorem *eqSymmetric*:
  shows $\forall\ a\ b.\ (eq2\ a\ b\ [] \longrightarrow eq2\ b\ a\ [])$
sorry

theorem *eqTransitive*:
  shows $\forall\ a\ b\ c.\ (((eq2\ a\ b\ []) \wedge (eq2\ b\ c\ [])) \longrightarrow (eq2\ a\ c\ []))$
sorry

theorem *parAssoc*:
  shows $((a \parallel b) \| c) =C\ (a \| (b \| c))$
sorry

theorem *parCommutative*:
  shows $a \| b =C\ b \| a$
sorry

theorem *zeroLeft*:
  shows $(Null \parallel a) =C\ a$
sorry

theorem *zeroRight*:
  shows $(a \parallel Null) =C\ a$
sorry

end

theory *NameEquiv*
imports *RhoCalc*
begin

3 Name equivalence

Similarly to structural congruence, we build up an equivalence of names: To quote an unquoted name n gives n back:

$$n \equiv_N \ulcorner\urcorner n \ulcorner\urcorner . \tag{4}$$

Furtermore, the quotations of two structurally equivalent processes are name equivalent

```
fun name-equivalence :: n ⇒ n ⇒ bool (infix =N 52)
  where
    zero =N ‘Input x y P´ = False
  | zero =N ‘Lift - -´ = False
  | zero =N ‘Par P Q´ = (Null =C (P‖Q))
  | zero =N ‘´a‘´ = (zero =N a)
  | zero =N zero = True

  | ((‘(Input x y P)´) =N ‘Input p q Q´) = ((Input x y P) =C (Input p q Q))
  | ((‘(Input x y P)´) =N ‘Lift - -´) = False
  | ((‘(Input x y P)´) =N ‘Par - -´) = False
  | ((‘(Input x y P)´) =N zero) = False
  | ((‘(Input x y P)´) =N ‘´a‘´) = (‘(Input x y P)´ =N a)

  | ((‘Lift x P´) =N ‘Lift y Q´) = (Lift x P =C Lift y Q)
  | ((‘Lift x P´) =N ‘Input - - -´) = False
  | ((‘Lift x P´) =N ‘Par - -´) = False
  | ((‘Lift x P´) =N zero) = False
  | ((‘Lift x P´) =N ‘´a‘´) = (‘(Lift x P)´ =N a)

  | ((‘P‖Q´) =N (‘A‖B´)) = ((P‖Q) =C (A‖B))
  | ((‘P‖Q´) =N (‘Lift - -´)) = False
  | ((‘P‖Q´) =N (‘Input - - -´)) = False
  | ((‘P‖Q´) =N zero) = ((P‖Q) =C Null)
  | ((‘P‖Q´) =N (‘´a‘´)) = (‘(P‖Q)´ =N a)

  | ‘´a‘´ =N ‘Input x y P´ = (a =N ‘Input x y P´)
  | ‘´a‘´ =N ‘Lift x P´ = (a =N ‘Lift x P´)
  | ‘´a‘´ =N ‘Par P Q´ = (a =N ‘Par P Q´)
  | ‘´a‘´ =N ‘´b‘´ = (a =N b)
  | ‘´a‘´ =N zero = (a =N zero)
```

Some examples

value *‘´zero‘´ =N zero*
value *‘´‘´zero‘´‘´ =N zero*

value *newName 3 =N three*
value *newName 3 =N zero*
value *three =N newName 3*
value ‘((*One*‖*Two*)‖(*Three*‖*Four*))‖*Null*´ =*N* ‘((*One*‖*Three*)‖(*Two*‖*Four*))´
value ‘*p* ‖ (**0** ‖ *q*)´ =*N* ‘*q* ‖ *p*´
theorem *testNameEQ1*:
  shows ‘*p* ‖ (**0** ‖ *q*)´ =*N* ‘*q* ‖ *p*´
using *parCommutative* **by** *auto*

3.1 Proof of equivalence relation

Reflexivity and symmetry is proven: only transitivity is unfinished. (It also relies on unfinished proofs of structural congruence)

theorem *name-equivalence-reflexive*:
  shows *reflexive name-equivalence*
apply *auto*
proof –
  fix *r*
  show *r =N r*
  using *congruReflexive* **by** (*induction r rule*: *name-equivalence.induct*, *auto*)
qed

theorem *name-equivalence-symmetric*:
  assumes *x =N y*
  shows *y =N x*
using *assms congruSymmetric eqSymmetric* **by** (*induct x rule*: *name-equivalence.induct*, *auto*)

theorem *name-equivalence-transitive*:
  assumes *a =N b* **and** *b =N c*
  shows *a =N c*

sorry

end
theory *Substitution*
imports *RhoCalc NameEquiv*
begin

4 Substitution

In the rho-calculus, we deal with two different notions of substitution, a syntactical and a semantic one, differing in the way which we deal with dropped names. One can think of the semantic substitution as a way of making sure that the process about to be run will be executed in the correct context.

4.1 Free and bound names

fun *free* :: $P \Rightarrow n\ set$ **where**
  free **0** = {}
  | *free* $(x \leftarrow y\ .\ P.) = \{x\} \cup (free(P) - \{y\})$
  | *free* $(x \lhd\ P\ \rhd) = \{x\} \cup free\ P$
  | *free* $(P \parallel Q) = free\ P \cup free\ Q$
  | *free* (´*x*‘) = $\{x\}$

fun *bound* :: $P \Rightarrow n\ set$ **where**
  bound **0** = {}
  | *bound* $(x \leftarrow y\ .\ P.) = \{y\} \cup bound(P)$
  | *bound* $(x \lhd\ P\ \rhd) = bound\ P$
  | *bound* $(P \parallel Q) = bound\ P \cup bound\ Q$
  | *bound* (´*x*‘) = {}

fun *names* :: $P \Rightarrow n\ set$
  where *names* $P = free(P) \cup bound(P)$

function *n-depth* :: $n \Rightarrow nat$ (# *60*)
  and *P-depth* :: $P \Rightarrow nat$ (# *60*)
  where
  n-depth ‘*P*´ = *1* + (*P-depth P*)
  | *P-depth P* = (*if* (*names P* $\neq$ {}) *then Max*({ (*n-depth x*) | *x*. *x* $\in$ (*names P*)})
else 0)
  apply *pat-completeness*
  apply *blast*
  apply *simp*
  by *blast*
termination
  sorry

value *P-depth* (´*zero*‘)

4.2 Syntactic substitution

The base case is a substitution of names if they are name equivalent:

abbreviation *sn* :: $n \Rightarrow n \Rightarrow n \Rightarrow n$ **where**
  sn x q p $\equiv$ (*if* (*x* =*N p*) *then q else x*)

value (*sn zero zero zero*)
value *newName* (*Max* ({(*n-depth zero*), *0*::*nat*}))

Generate new name not used in the relevant processes

abbreviation *genz*

where *genz* ≡ λ *q*::*n*. λ *p*::*n*. λ *R*::*P*. *newName* (*Max*({(*n-depth*(*q*)), (*P-depth*(*R*)), (*n-depth*(*p*)) }))

Syntactic substitution can now be given by the following function:

function *s* :: *P* ⇒ *n* ⇒ *n* ⇒ *P* ((-) {-∖-} *52*)
where (**0**){-∖-} = **0**
| (*R* ∥ *S*){*q*∖*p*} = ((*R*){*q*∖*p*}) ∥ ((*S*){*q*∖*p*})
| (*x* ← *y* . *R*.){*q*∖*p*} = ((*sn x q p*) ← (*genz q p R*) . ((*R* {(*genz q p R*)∖*y*}){*q*∖*p*}).)
| (*x* ◁ *R* ▷) {*q*∖*p*} = ((*sn x q p*) ◁ *R*{*q*∖*p*} ▷)
| (´*x*‘){*q*∖*p*} = (*if* (*x* =*N p*) *then* ´*q*‘ *else* ´*x*‘)
apply *pat-completeness* **by** *auto*
termination
apply (*relation measure* (λ(*p*,*x*,*y*). (*P-depth p*)), *auto*)

sorry

4.3 Semantic substitution

function *ss* :: *P* ⇒ *n* ⇒ *n* ⇒ *P* ((-) *s*{-∖-} *52*)
where (**0**)*s*{-∖-} = **0**
| (*R* ∥ *S*) *s*{*q*∖*p*} = ((*R*) *s*{*q*∖*p*}) ∥ ((*S*) *s*{*q*∖*p*})
| (*x* ← *y* . *R*.) *s*{*q*∖*p*} = ((*sn x q p*) ← (*genz q p R*) . ((*R* {(*genz q p R*)∖*y*}) *s*{*q*∖*p*}).)
| (*x* ◁ *R* ▷) *s*{*q*∖*p*} = ((*sn x q p*) ◁ *R* *s*{*q*∖*p*} ▷)
| (´*x*‘) *s*{‘*q*´∖*p*} = (*if* (*x* =*N p*) *then q else* ´*x*‘)
apply *pat-completeness* **by** *auto*
termination
apply (*relation measure* (λ(*p*,*x*,*y*). (*P-depth p*)), *auto*)

sorry

4.4 Examples

value **0**{*zero*∖*zero*}
value (*zero* ← *zero* . *Zero*.){*two* ∖ *zero*}
value (**0** ∥ (´*zero*‘)) { (*newName 2*) ∖ *zero* }

value *zero*◁ *zero*[[*three*]] ▷{*two*∖*three*}
value (*zero*◁ *zero*[[*three*]] ▷{*two*∖*three*}) = (*zero*◁ *zero*[[*two*]] ▷)

value *zero*[[‘*zero*[[*three*]]´]]{*two*∖*three*}
value *zero*[[‘*zero*[[*three*]]´]]{*two*∖*three*} = (*zero*[[‘*zero*[[*three*]]´]])

5 Alpha equivalence

Alpha equivalence equates processes that only differ by their bound variables. In our calculus, the bound variables are the names to which we bound input values. As an example we would want the following terms to be alpha-equal:

```
fun alphaEq :: P ⇒ P ⇒ bool (infix ≡α 52)
  where Null ≡α P = (Null =C P)
    | ((a←b. P.) ≡α (c←d. Q.)) =  ((b =N d) ∧ (P ≡α (Q{a↖c})))
    | - ≡α - = True

theorem alphaEq:
shows zero ← zero . Zero. ≡α one ← zero . Zero.
sorry

end
theory Dynamics
imports RhoCalc NameEquiv Substitution
begin
```

6 Operational Semantics

The following processes gives a set of reduction rules which corresponds to the dynamics of the rho-calculus. Essentially, the main way in which a process can reduce is by synchronization: If two processes P and Q run in parallel, where P is listening on a channel (name equivalent to) y, and Q is ready to output a process R on the channel y, then P and Q will synchronize. The name to which P writes the input will be substituted throughout the rest of P to 'R'. This reduction is called the communication rule:

$$\frac{x_0 \equiv_N x_1}{x_0 \lhd Q \rhd | x_1(y).P \to P' Q' \setminus y} \; COMM \tag{5}$$

```
fun toPar:: P list ⇒ P where
  toPar []   = Null
 |toPar (x#[]) = x
 |toPar (x#y#xs) = (x ∥ (toPar (y#xs)))
```

```
fun syncable:: P ⇒ P ⇒ bool where
  syncable (Lift x Q) (Input y z P)    = (x =N z)
 |syncable (Input y z P) (Lift x Q)    = (x =N z)
 |syncable (Lift y P) (Lift x Q)       = False
 |syncable (Input y z P) (Input a b Q) = False
 |syncable - (vc ∥ vd)                 = False
 |syncable   (vc ∥ vd) -               = False
```

|*syncable* (´*v*‘) - = *False*
|*syncable* - (´*v*‘) = *False*
|*syncable* *Null* - = *False*
|*syncable* - *Null* = *False*

fun *sync*:: *P* ⇒ *P* ⇒ *P* **where**
sync (*Lift x Q*) (*Input y z P*) = (*P s*{‘*Q*´∖*y*})
|*sync* (*Input y z P*) (*Lift x Q*) = (*P s*{‘*Q*´∖*y*})

function *fineRun*:: *P list* ⇒ *P list* ⇒ *P list* ⇒ *P list* **where**
fineRun [] [] [] = []
|*fineRun* (*x*#[]) [] [] = (*x*#[])
|*fineRun* (*x*#*y*#*xs*) [] *zs* = (*fineRun* (*y*#*xs*) (*x*#[]) *zs*)
|*fineRun* [] (*y*#*ys*) (*z*#[]) = (*z*#*y*#*ys*)
|*fineRun* [] (*y*#*ys*) (*z*#*zs*) = (*fineRun zs* (*z*#*y*#*ys*) [])
|*fineRun* (*x*#*xs*) (*y*#*ys*) *zs* = (*if* (*syncable x y*) *then* ((*sync x y*)#*xs*@*ys*@*zs*)
else (*fineRun* (*xs*) (*y*#*ys*) (*x*#*zs*)))
apply *auto*
sorry
termination
sorry

fun *step*:: *P* ⇒ *P* **where**
step P = *toPar*(*fineRun* (*getList P*) [] [])

6.1 Example: replication

In traditional process calculae, there is usually a specific construction for

abbreviation *replication* **where** *replication* ≡ (λ*y*.λ(*P*,*x*).(*x*◁((*y*←*x*.(*x*[[*y*]]‖´*y*‘).)‖*P*)▷‖(*y* ← *x*.(*x*[[*y*]]‖´*y*‘).)))(‘*zero*[[*zero*]]´)
abbreviation *xx* **where** *xx* ≡ *zero*
abbreviation *yy* **where** *yy* ≡ ‘*xx*[[*xx*]]´

value (*replication* (*Two*, *zero*))
value *step* (*replication* (*Two*, *zero*))
value *step*(*step* (*replication* (*Two*, *zero*)))
value *step*(*step*(*step* (*replication* (*Two*, *zero*))))
value *step*(*step*(*step*(*step* (*replication* (*Two*, *zero*)))))

end
theory *NamespaceLogic*
imports *RhoCalc NameEquiv Substitution Dynamics*
begin

7 Namespace logic

The logic of the rho-calculus closely mimics the structure of our processes, in order to be able to express things such as: this process only takes input over these channels throughout its lifetime. Since names are simply quoted processes, this logic is called namespace logic. We begin by the datatype of the constructible formulae of namespace logic:

datatype F = *true*
 | *false*
 | *negation* F (¬-)
 | *conjunction* F F (-&- *80*)
 | *separation* F F (-‖- *80*)
 | *disclosure* a (´-‘ *80*)
 | *dissemination* a F (-!(-) *80*)
 | *reception* a n F (⟨-?-⟩ *80*)
 | *greatestFixPoint* F F (*rec*-.- *80*)
 | *quantification* n F F (∀ -:-.-)
 and a = *indication* F (‘-´)
 | n

8 Semantics

Instead of evaluating formulae to truth values, we ask for which processes or names satisfy the given formula. In other words, to evaluate a formula we give it a candidate set of processes, and then we are returned the set of processes 'witnessing' the formula.

abbreviation *toNames* :: P *set* ⇒ n *set*
 where *toNames* A ≡ {‘x´ | x. x ∈ A}

abbreviation *toProc* :: n *set* ⇒ P *set*
 where *toProc* A ≡ {´x‘ | x. x ∈ A}

fun *evalF* :: P *set* ⇒ F ⇒ P *set*
 and *evalA* :: n *set* ⇒ a ⇒ n *set*
 where *evalF* A *true* = A
 | *evalF* - *false* = {*Null*}
 | *evalF* A (*negation* φ) = A − (*evalF* A φ)
 | *evalF* A (φ & ψ) = *evalF* A φ ∩ *evalF* A ψ
 | *evalF* A (φ ‖ ψ) = {p‖q | p q. p‖q ∈ A ∧ (
  (p ∈ (*evalF* A φ) ∧ q ∈ (*evalF* A ψ))
  ∨ (p ∈ (*evalF* A ψ) ∧ q ∈ (*evalF* A ψ)))}
 | *evalF* A (*disclosure* a) = {P | P x. (P ≡α ´x‘) ∧ P ∈ A ∧ x ∈ (*evalA* (*toNames* A) a)}
 | *evalF* A (*dissemination* a P) = {P | P Q x. (P ≡α (x◁ Q ▷)) ∧ (P ∈ A) ∧ (Q ∈ A) ∧ (x ∈ *evalA* (*toNames* A) a)}
 | *evalA* N ‘φ´ = {x | x P. (x =N ‘P´) ∧ (P ∈ (*evalF* (*toProc* N) φ)) ∧ (x ∈ N)}

end