Effect system for BoCa: a Prolog Implementation

U. de'Liguoro, G. Falzetta

Università di Torino, Corso Svizzera 185, 10149 Torino (Italy)
deligu@di.unito.it, falzetta@di.unito.it

The paper [BBDCS03a] (of which [BBDCS03b] is a preliminary version) focuses on space control and develops an analysis of space usage in the context of an ambient-like calculus with bounded capacities and weighed processes, where migration and activation require space.

Beside the main type system, the paper contains a system to derive sets of constraints that are satisfiable if the effect related part of types stay within bounds declared in the context ([BBDCS03a] section 2.4). The latter system, which is simpler than the full typing system, is implemented by the present Prolog program: given a well formed context Γ and a pre-term P representing a process with capacities, the program replies with a type A and a set of constraints Δ if the judgment $\Gamma \vdash P : A \Downarrow \Delta$ is derivable in the effect system.

Appendix A is the full program code.

1 The effect system for BoCa

In this section we recall what is necessary form [BBDCS03a]. In the meanwhile, owing to the limited aim of the implementation, we redefine the term and type syntax in the restricted form as it is actually employed in the Effect system: term syntax does not allow for message sending and receiving primitives; type syntax has a simplified form of ambient and process types.

Definition 1.1 Pre-terms *are defined by the following grammar:*

$$\begin{array}{lll} \textit{Processes}\ P & ::= & \blacksquare \mid \mathbf{0} \mid \pi.P \mid P|P \mid M^{\mathsf{k}}[\,P\,] \mid !\pi.P \mid (\nu a : W)P \\ \textit{Messages}\ M & ::= & a \mid \mathbf{in}\ M \mid \mathbf{out}\ M \mid \mathbf{open}\ M \mid \overline{\mathbf{open}} \mid \mathbf{get}\ M \mid \mathbf{get}^{\uparrow} \mid \mathbf{put} \mid \mathbf{put}^{\downarrow} \mid M.M \\ \textit{Prefixes}\ \pi & ::= & M \mid \mathsf{k} \triangleright \end{array}$$

where a ranges over a denumerable set of names, k *is any integer* ≥ 0*.*

Terms are pre-terms P such that $w(P) \neq \perp$, where the weight function w is defined as follows:

$$
\begin{array}{rcl}
w(\_) & = & 1 \\
w(\mathbf{0}) & = & 0 \\
w(P \mid Q) & = & w(P) + w(Q) \\
w((\boldsymbol{\nu} a : W)P) & = & w(P) \\
w(M.P) & = & w(P) \\
w(a^{\mathsf{k}}[\,P\,]) & = & \text{if } w(P) = \mathsf{k} \text{ then } \mathsf{k} \text{ else } \bot \\
w(\mathsf{k} \triangleright P) & = & \text{if } w(P) = \mathsf{k} \text{ then } 0 \text{ else } \bot \\
w(!\pi.P) & = & \text{if } w(\pi.P) = 0 \text{ then } 0 \text{ else } \bot
\end{array}
$$

writing simply $\mathsf{k} \triangleright P$ for $\mathsf{k} \triangleright .P$

Definition 1.2 *The type syntax is defined by the following productions:*

$$
\begin{array}{lll}
\textit{Message Types} & W & ::= Amb^{-}\langle \iota \rangle \mid Cap\langle \phi \rangle \\
\textit{Process Types} & \Pi & ::= Proc\langle \varepsilon \rangle
\end{array}
$$

where

$$EXP ::= d_a \mid i_a \mid \mathsf{n} \in \mathcal{Z} \mid EXP + EXP \mid EXP - EXP \mid \mathsf{min}(EXP, EXP) \mid \mathsf{max}(EXP, EXP)$$

$$
\begin{array}{lll}
\textit{Intervals} & \iota \in \mathfrak{I} & \triangleq \{[\mathsf{n}, \mathsf{N}] \mid \mathsf{n}, \mathsf{N} \in \mathcal{Z}^{+},\ \mathsf{n} \leq \mathsf{N}\} \\
\textit{Effects} & \varepsilon \in \mathcal{E} & \triangleq \{(e, e') \mid e, e' \in EXP\} \\
\textit{Thread Effects} & \phi \in \Phi & \subseteq \mathcal{E} \to \mathcal{E}
\end{array}
$$

and Φ is generated by closing under functional composition the following set of functions:

$$
\begin{array}{rcl}
\mathsf{Id} & = & \lambda(e_1, e_2).(e_1, e_2) \\
\mathsf{Put} & = & \lambda(e_1, e_2).(e_1 - 1, \mathsf{max}(0, e_2 - 1)) \\
\mathsf{Get} & = & \lambda(e_1, e_2).(\mathsf{min}(0, e_1 + 1), e_2 + 1) \\
\mathsf{Open}(e_1, e_2) & = & \lambda(e_1', e_2').(e_1 + e_1', e_2 + e_2')
\end{array}
$$

The effect system splits into two parts: the first one, called Good Messages in [BBDCS03a], assigns types to capibilities. Restricting the system to the present syntax we obtain the system in figure 1 to type messages, and in figure 2 to derive process type and (sets of) formal constraints.

2 Prolog syntax and implementation

Term and type syntax translate into Prolog terms as follows:

Figure 1 Good Messages

$$\frac{}{\Gamma, a : Amb^{-}\langle\iota\rangle \vdash a : Amb^{-}\langle\iota\rangle}\ (axiom) \qquad \frac{\Gamma \vdash M : Cap\langle\phi\rangle \quad \Gamma \vdash M' : Cap\langle\phi'\rangle}{\Gamma \vdash M.M' : Cap\langle\phi \circ \phi'\rangle}\ (path)$$

$$\frac{\Gamma \vdash M : Amb^{-}\langle\iota\rangle}{\Gamma \vdash \mathbf{get}\ M : Cap\langle\mathsf{Get}\rangle}\ (\mathbf{get}\ M) \qquad \frac{}{\Gamma \vdash \mathbf{put} : Cap\langle\mathsf{Put}\rangle}\ (\mathbf{put})$$

$$\frac{}{\Gamma \vdash \mathbf{get}^{\uparrow} : Cap\langle\mathsf{Get}\rangle}\ (\mathbf{get}^{\uparrow}) \qquad \frac{}{\Gamma \vdash \mathbf{put}^{\downarrow} : Cap\langle\mathsf{Id}\rangle}\ (\mathbf{put}^{\downarrow})$$

$$\frac{\Gamma \vdash M : Amb^{-}\langle\iota\rangle}{\Gamma \vdash \mathbf{in}\ M : Cap\langle\mathsf{Id}\rangle}\ (\mathbf{in}\ M) \qquad \frac{\Gamma \vdash M : Amb^{-}\langle\iota\rangle}{\Gamma \vdash \mathbf{out}\ M : Cap\langle\mathsf{Id}\rangle}\ (\mathbf{out}\ M)$$

$$\frac{\Gamma \vdash M : Amb^{-}\langle\iota\rangle}{\Gamma \vdash \mathbf{open}\ M : Cap\langle\mathsf{Open}(\mathtt{d}_a, \mathtt{i}_a)\rangle}\ (\mathbf{open}\ M) \qquad \frac{}{\Gamma \vdash \overline{\mathbf{open}} : Cap\langle\mathsf{Id}\rangle}\ (\overline{\mathbf{open}})$$

```
Processes     P    ::  = _ | zero | Pi pref P | P|P
                   | spamb(M, K, P) | bang (Pi, P)
                   | nu(A : W, P)
Capabilities  C    ::  = in(M) | out(M) | open(M) | open_bar
                   | get(M) | get_up | put | put_down
Message       M    ::= A | C | M dot M
Prefix        Pi   ::= M | sp(K)
```

where `A` is any alphanumeric string beginning by `a`, `K` is any positive integer, and `W` is a message type as defined below:

```
T      ::=  proc(EXP,EXP)
W      ::=  amb(Iota) | cap(Phi)
Phi    ::=  id | put | get | open(EXP,EXP) | comp(Phi,Phi)
EXP    ::=  d(A) | i(A) | Z | EXP + EXP | EXP - EXP
            | min_(EXP, EXP) | max_(EXP,EXP)
Z      ::=  any integer
Iota   ::=  sqb(N1,N2)
```

where `N1, N2` are positive integers such that `0` $\leq$ `N1` $\leq$ `N2`.

To run the program one is expected to write:

```
:- boca(G,P,T,D).
```

Figure 2 Effect Inference

$$\frac{}{\Gamma \vdash \blacksquare : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \emptyset}\ \widehat{(\blacksquare)} \qquad \frac{}{\Gamma \vdash \mathbf{0} : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \emptyset}\ \widehat{(\mathbf{0})}$$

$$\frac{\Gamma \vdash M : Cap\langle \phi \rangle \quad \Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta}{\Gamma \vdash M.P : Proc\langle \phi(\varepsilon) \rangle \Downarrow \Delta}\ \widehat{(prefix)}$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta \quad \Gamma \vdash Q : Proc\langle \varepsilon' \rangle \Downarrow \Delta'}{\Gamma \vdash P \mid Q : Proc\langle \varepsilon + \varepsilon' \rangle \Downarrow \Delta \cup \Delta'}\ \widehat{(par)}$$

$$\frac{\Gamma, a : Amb^{-}\langle \iota \rangle \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta}{\Gamma \vdash (\boldsymbol{\nu} a : Amb^{-}\langle \iota \rangle)P : Proc\langle \varepsilon \rangle \Downarrow \Delta}\ \widehat{(new)}$$

$$\frac{\Gamma \vdash a : Amb^{-}\langle [\mathsf{n}, \mathsf{N}] \rangle \quad \Gamma \vdash P : Proc\langle (e, e') \rangle \Downarrow \Delta \quad w(P) = \mathsf{k}}{\Gamma \vdash a^{\mathsf{k}}[P] : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta \cup \left\{ \begin{array}{l} \mathsf{n} \leq \max(\mathsf{k} + e, 0),\ \mathsf{k} + e' \leq \mathsf{N}, \\ \mathsf{d}_a \leq e - e',\ \min(\mathsf{N} - \mathsf{n}, e' - e) \leq \mathtt{i}_a \end{array} \right\}}\ \widehat{(amb)}$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta \quad w(P) = \mathsf{k}}{\Gamma \vdash \mathsf{k} \triangleright P : Proc\langle \varepsilon \rangle \Downarrow \Delta}\ \widehat{(spawn)} \qquad \frac{\Gamma \vdash \pi.P : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta \quad w(\pi.P) = 0}{\Gamma \vdash\ !\pi.P : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta}\ \widehat{(bang)}$$

This predicate accepts as input a pair `G` and `P`, such that `G = [A : W, ..., A : W]` is a list of assumptions with distinct subjects (the `A`'s), whose meaning is $\Gamma = \{a_1 : Amb^{-}\langle \iota_1 \rangle, \ldots a_n : Amb^{-}\langle \iota_n \rangle\}$, and `P` is a well formed process, such that all bound names `nu(A : W, _)` are distinct and do not clash with free names (the present version of the program does not implement renaming of bound names). It returns a pair `T, D` consisting in a type and a list of formal inequalities of the shape `EXP leq EXP` meaning $e_1 \leq e_2$.

The predicate `boca` depends on the predicates:

```
% effect(G,P,T,D) succeeds if it is derivable
% G |- P:T Downarrow D

% typing(G, M, T) succeeds in case G is a given context
% and M is a given message such that G |- M:T
```

implementing Effect and Message systems respectively. They rely on a reduction system simplifying the combinators `Phi` and evaluating their application to pairs of formal expressions. Reduction is the compatible closure of the rules:

```
% reduce1(phi, psi) succeeds in case phi -> psi
% where -> is the one step, compatible reduction relation
% over phi-combinators generated by the following rules:
% open(0,0) -> id
% comp(id,phi) -> phi
% comp(phi,id) -> phi
% comp(comp(phi1,phi2),phi3) -> comp(phi1,comp(phi2,phi3))
% reduce(phi,psi) succeeds in case phi ->^* psi
% where ->^* is the reflexive and transitive closure of ->
```

Eventually the combinator `Phi` in normal form is evaluated against the pair of expressions `E1, E2`:

```
% apply(Phi, E1, E2, E3, E4) succeeds if Phi(E1,E2) = (E3,E4)
```

The present version does not support further symbolic simplification of expressions: one might obtain e.g. `0 + min_(0,1) leq 3` as a constraint in the list `D` representing Δ .

References

[BBDCS03a] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A calculus of bounded capacities. Submitted to a journal. Extendend version of [BBDCS03b], December 2003.

[BBDCS03b] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A calculus of bounded capacities. In *ASIAN'03*, number 2896 in LNCS, pages 205–223. Springer-Verlag, December 2003.

A Program Code

```
/* TYPE INFERENCE FOR BOUNDED CAPACITIES FOR SWI-PROLOG
(Multi-threaded, Version 5.4.6) AUTHORS:

prof. Ugo de’Liguoro Dipartimento di Informatica Universita‘ di
Torino, Corso Svizzera 185, 10149, Torino, Italy home page:
www.di.unito.it/~deligu e-mail:    deligu@di.unito.it

and

Ph.D student Giuseppe Falzetta Dipartimento di Informatica
Universita‘ di Torino, Corso Svizzera 185, 10149, Torino, Italy
home page: www.di.unito.it/~falzetta e-mail:
falzetta@di.unito.it

*/


/*

Preterms and terms syntax:

Processes      P  :: = _ | zero | Pi pref P |  P|P  | spamb(M, K,
P)
                       M pref P | bang (Pi, P)  | nu(A : W, P)

Capabilities   C  :: = in(M) | out(M) | open(M) | open_bar |
get(M) |
                       get_up | put | put_down

Messages       M  :: = A | C | M dot M

Prefixes       Pi :: = M | sp(K)

K ::= positive integer

A ::= alphanumeric string beginnig by 'a' (ambinet names)


Type syntax

T ::= proc(EXP,EXP)

W ::= amb(Iota) | cap(Phi)

Phi ::= id | put | get | open(EXP,EXP) | comp(Phi,Phi)

EXP ::= d(A) | i(A) | Z | EXP + EXP | EXP - EXP |
        min_(EXP, EXP) | max_(EXP,EXP)

Z ::= any integer

Iota ::= sqb(N1,N2)  where 0 =< N1 =< N2 are integers
```

```
*/

% Operators declaration

:- op(110,xfx,dot). :- op(130,xfx,pref). :- op(200,xfx,leq).


boca(G,_,_,_) :-
    not(goodG(G)), !,
    write('Error in: '), write('-->'), write(G), write('<--'), fail.

boca(_,P,_,_) :-
    not(goodProcess(P)), !,
    write('Error in: '), write('-->'), write(P), write('<--'), fail.

boca(G,P,T,D) :-
    effect(G,P,T,D).

goodProcess(P) :-
    goodP(P).
goodProcess(P) :-
    goodM(P).
goodProcess(P) :-
    goodPi(P).
goodProcess(P) :-
    goodT(P).
goodProcess(P) :-
    goodW(P).
goodProcess(P) :-
    goodPhi(P).
goodProcess(P) :-
    goodExp(P).


goodG(G) :-
    is_Gamma(G).

is_Gamma([]). is_Gamma([A ':' amb(Iota)|G]) :-
    is_AmbName(A),
    is_iota(Iota),
    is_Gamma(G).

goodS(String) :-
   goodP(String).

goodS(String) :-
   goodM(String).

% Patterms and terms syntax:

% Good proces

goodP('_'). goodP(zero). goodP(P1 '|' P2) :-
    goodP(P1),
    goodP(P2).
goodP(nu(A ':' W, P)) :-
    is_AmbName(A),
```

```
    goodW(W),
    goodP(P).
goodP(spawn(K, P)) :-
    integer(K),
    K >= 0,
    goodP(P).
goodP(M pref P) :-
    goodM(M),
    goodP(P).
goodP(bang(Pi, P)) :-
    goodP(Pi),
    goodP(P).
goodP(spamb(M, K, P)) :-
    is_AmbName(M),
    K >= 0,
    goodP(P).

% Good message

goodM(A) :-
    is_AmbName(A).
goodM(open_bar). goodM(get_up). goodM(put). goodM(put_down).
goodM(in(M)) :-
     goodM(M).
goodM(out(M)):-
     goodM(M).
goodM(open(M)):-
     goodM(M).
goodM(get(M)) :-
     goodM(M).
goodM(M1 dot M2) :-
     goodM(M1),
     goodM(M2).

% Good Pi

goodPi(sp(K)) :-
    integer(K),
    K >= 0.
goodPi(Pi) :-
    goodG(Pi).

% Type syntax

goodT(proc(Exp1,Exp2)) :-
    goodExp(Exp1),
    goodExp(Exp2).
goodT(agent(Exp1,Exp2)) :-
    goodExp(Exp1),
    goodExp(Exp2).

goodW(amb(Iota)) :-
    is_iota(Iota).
goodW(cap(Phi)) :-
    goodPhi(Phi).

goodPhi(id). goodPhi(put). goodPhi(get). goodPhi(open(Exp1,Exp2))
```

```
:-
    goodExp(Exp1),
    goodExp(Exp2).
goodPhi(comp(Phi1,Phi2)) :-
    goodPhi(Phi1),
    goodPhi(Phi2).

goodExp(d(A)) :-
    is_AmbName(A).
goodExp(i(A)) :-
    is_AmbName(A).
goodExp(Z) :-
  integer(Z).
goodExp(Exp1 + Exp2) :-
    goodExp(Exp1),
    goodExp(Exp2).
goodExp(Exp1 - Exp2) :-
    goodExp(Exp1),
    goodExp(Exp2).
goodExp(min_(Exp1,Exp2)) :-
    goodExp(Exp1),
    goodExp(Exp2).
goodExp(max_(Exp1,Exp2)) :-
    goodExp(Exp1),
    goodExp(Exp2).


weight('_', 1). weight(zero, 0). weight(P '|' Q, K) :-
    weight(P, K1),
    weight(Q, K2),
    K is K1 + K2.
weight(nu(_,P), K) :-
    weight(P, K).
weight(_ pref P, K) :-
    weight(P, K).
weight(spamb(_,K,P), K) :-
    !, weight(P, K).
weight(spawn(K,P), 0) :-
    !, weight(P, K).
weight(bang(_,P), 0) :-
    !, weight(P, 0).


% Utilities

member(X, [X]). member(X, [X|_]). member(X, [_|Y]) :- member(X,Y).


% Syntactical categories

is_iota(sqb(0,0)). is_iota(sqb(M,N)) :-
    0 =< M, M =< N.


is_Var(X) :-
    atom(X),
    name(X,[120|_]).
```

```
is_AmbName(X) :-
    atom(X),
    name(X,[97|_]).

% Combinators and reduction systems for representing
% and evaluating phi:E -> E functions, where E = EXP x EXP
% (below called phi-combinators)

% reduce1(phi, psi) succeeds in case phi --> psi
% where --> is the one step, compatible reduction relation
% over phi-combinators generated by the following rules:

% open(0,0) --> id
% comp(id,phi) --> phi
% comp(phi,id) --> phi
% comp(comp(phi1,phi2),phi3) --> comp(phi1,comp(phi2,phi3))

reduce1(open(0,0),id). reduce1(comp(id,Phi),Phi).
reduce1(comp(Phi,id),Phi).
reduce1(comp(comp(Phi1,Phi2),Phi3),comp(Phi1,comp(Phi2,Phi3))).
reduce1(comp(Phi1,Phi2),comp(Psi1,Phi2)) :-
    reduce1(Phi1,Psi1).
reduce1(comp(Phi1,Phi2),comp(Phi1,Psi2)) :-
    reduce1(Phi2,Psi2).

% reduce(phi,psi) succeeds in case phi -->^* psi
% where -->^* is the reflexive and transitive closure
% of -->

reduce(Phi,Psi) :-
    reduce1(Phi,Phi1),!,
    reduce(Phi1,Psi).
reduce(Phi,Phi).



% apply(Phi, E1, E2, E3, E4) succeeds if Phi(E1,E2) = (E3,E4)

apply(id, E1, E2, E1, E2).

apply(Phi, E1, E2, E3, E4) :-
    reduce1(Phi,_),!,
    reduce(Phi,Phi1),
    apply(Phi1, E1, E2, E3, E4).

apply(open(D,I), E1, E2, E3, E4) :-
    integer(D), integer(E1), integer(I), integer(E2),
    E3 is D + E1,
    E4 is I + E2.

apply(open(D,I), E1, E2, E3, I + E2) :-
    integer(D), integer(E1),
    E3 is D + E1.

apply(open(D,I), E1, E2,  D + E1, E4) :-
    integer(I), integer(E2),
```

```
    E4 is I + E2.

apply(open(D,I), E1, E2, D + E1, I + E2).

apply(put, E1, E2, E3, E4):-
    integer(E1), integer(E2),
    E3 is E1 - 1,
    E4 is max(0, E2 - 1).

apply(put, E1, E2, E3, max_(0, E2 - 1)):-
    integer(E1),
    E3 is E1 - 1.

apply(put, E1, E2, E1 - 1, E4):-
    integer(E2),
    E4 is max(0, E2 - 1).

apply(put, E1, E2, E1 - 1, max_(0, E2 - 1)).

apply(get, E1, E2, E3, E4):-
    integer(E1), integer(E2),
    E3 is min(0, E1 + 1),
    E4 is E2 + 1.

apply(get, E1, E2, E3, E2 + 1):-
    integer(E1),
    E3 is min(0, E1 + 1).

apply(get, E1, E2, min_(0, E1 + 1), E4):-
    integer(E2),
    E4 is E2 + 1.

apply(get, E1, E2, min_(0, E1 + 1), E2 + 1).

apply(comp(Phi1,Phi2), E1, E2, E3, E4) :-
    apply(Phi2, E1, E2, E5, E6),
    apply(Phi1, E5, E6, E3, E4).


% Context manipulation and generation

% unify(G1,G2,G3) where G1,G2,G3 are contexts of the shape
% [M ':' W , ... ], succeeds in case G3 is the union of G1 G2
% and is a legal conetxt (no subject has more than one type)

unify([],G,G). unify([M ':' W|G1], G2, G3) :-
    member(M ':' W, G2),
    unify(G1,G2,G3).
unify([M ':' _|_], G, _) :-
    inFV(M,G), ! , fail.
unify([M ':' W|G1], G2, [M ':' W|G3]) :-
    unify(G1,G2,G3).

inFV(M, [M ':' _|_]). inFV(M, [_|G]) :-
    inFV(M,G).

% Union of set of constraints
```

```
union([],D,D). union([C|D1],D2,D3) :-
    member(C,D2), !,
    union(D1,D2,D3).
union([C|D1],D2,[C|D3]) :-
    union(D1,D2,D3).


% Good messages (cap e amb)
% restricted version of system in Fig. 3
% where the type syntax is given by their use
% in the Effect system (see above): Amb^- written
% symply amb

% typing(G, M, T) succeeds in case G is a given context
% and M is a give message such that G |- M:T

% put
typing([], put, cap(put)). typing( _, put, cap(put)).


% put_down
typing([], put_down, cap(id)). typing( _, put_down, cap(id)).


% get_up
typing([],get_up, cap(get)). typing( _,get_up, cap(get)).

% open_bar
typing([],open_bar, cap(id)). typing( _,open_bar, cap(id)).


% get M
typing(G, get(M), cap(get)) :-
    is_iota(Iota),
    typing(G, M, amb(Iota)).

% in M
typing(G, in(M), cap(id)) :-
    is_iota(Iota),
    typing(G, M, amb(Iota)).

% out M
typing(G, out(M), cap(id)) :-
    is_iota(Iota),
    typing(G, M, amb(Iota)).


% open M

typing(G, open(A), cap(open(d(A),i(A)))) :-
    is_AmbName(A),
    typing(G, A, amb(_)).

% path

typing(G, M1 dot M2, cap(comp(Phi1,Phi2))) :-
```

```
    typing(G1, M1, cap(Phi1)),
    typing(G2, M2, cap(Phi2)),
    unify(G1, G2, G).

% axiom

typing(G,M,W) :-
    is_AmbName(M),
    member(M ':' W, G).


% Effect inference
% Fig. 5

% effect(G,P,T,D) succeeds if it is derivable
% G |- P:T Downarrow D

% hat{underscore}

effect([], '_', proc(0,0), []). effect( _, '_', proc(0,0), []).

% hat{zero}

effect([], zero, proc(0,0), []). effect( _, zero, proc(0,0), []).

% hat{spawn}

effect(G, sp(K) pref P, proc(E1,E2), D) :- !,
    weight(P, K),
    effect(G, P, proc(E1,E2), D).

% hat{prefix}

effect(G, M pref P, proc(E1,E2), D) :-
    typing(G, M, cap(Phi)),
    effect(G, P, proc(E3,E4), D),
    apply(Phi,E3,E4,E1,E2).

% hat{par}

effect(G, P '|' Q, proc(E5, E6), D) :-
    effect(G, P, proc(E1,E3), D1),
    integer(E1),
    integer(E3),
    effect(G, Q, proc(E2,E4), D2),
    integer(E2),
    integer(E4),
        E5 is E1 + E2,
        E6 is E3 + E4,
    union(D1,D2,D).

effect(G, P '|' Q, proc(E1 + E2, E3 + E4), D) :-
    effect(G, P, proc(E1,E3), D1),
    effect(G, Q, proc(E2,E4), D2),
    union(D1,D2,D).

% hat{new}
```

```
effect(G, nu(A ':' amb(Iota), P), proc(E1,E2), D) :-
    is_AmbName(A), !,
    effect([A ':' amb(Iota)|G], P, proc(E1,E2), D).

% hat{amb}

effect(G, spamb(A, K, P), proc(0,0), D) :-
    is_AmbName(A), %!,
    weight(P, K),
    typing(G, A, amb(sqb(_n,N))),
    effect(G, P, proc(E1,E2), D1),
            integer(K),
            integer(E1),
            integer(N),
            integer(_n),
            integer(E2),
    D2 = [_n leq E3,
          E4 leq N,
          d(A) leq E5,
          E6 leq i(A)],
              E3 is max(K + E1, 0),
              E4 is K + E2,
              E5 is E1 - E2,
              E6 is min(N - _n, E2 - E1),
    union(D1, D2, D).

effect(G, spamb(A, K, P), proc(0,0), D) :-
    is_AmbName(A), !,
    weight(P, K),
    typing(G, A, amb(sqb(_n,N))),
    effect(G, P, proc(E1,E2), D1),
    D2 = [_n leq max_(K + E1, 0),
          K + E2 leq N,
          d(A) leq E1 - E2,
          min_(N - _n, E2 - E1) leq i(A)],
    union(D1, D2, D).

% hat{bang}

effect(G, bang(Pi, P), proc(0,0), D) :-
    weight(Pi pref P, 0),
    effect(G, Pi pref P, proc(0,0), D).

% EOF
```