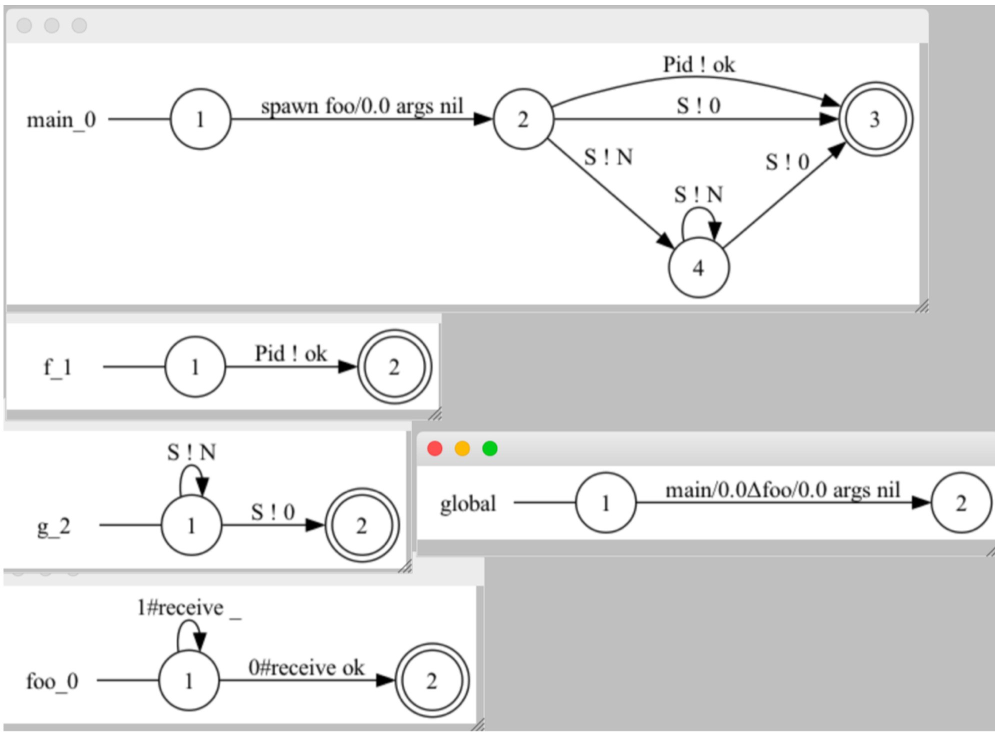
Some chorer test examples

Example foo9b:

```
-module(foo9b).
-export([main/0,f/1,g/2,foo/0]).

main() ->
   S = spawn(foo9b,foo,[]),
   f(g(S,3)).

g(S,N) when N>0 -> S ! N, g(S,N-1);
g(S,0) -> S ! 0, S.

f(Pid) -> Pid ! ok.

foo() -> receive
            ok -> done;
            _ -> foo()
         end.
```

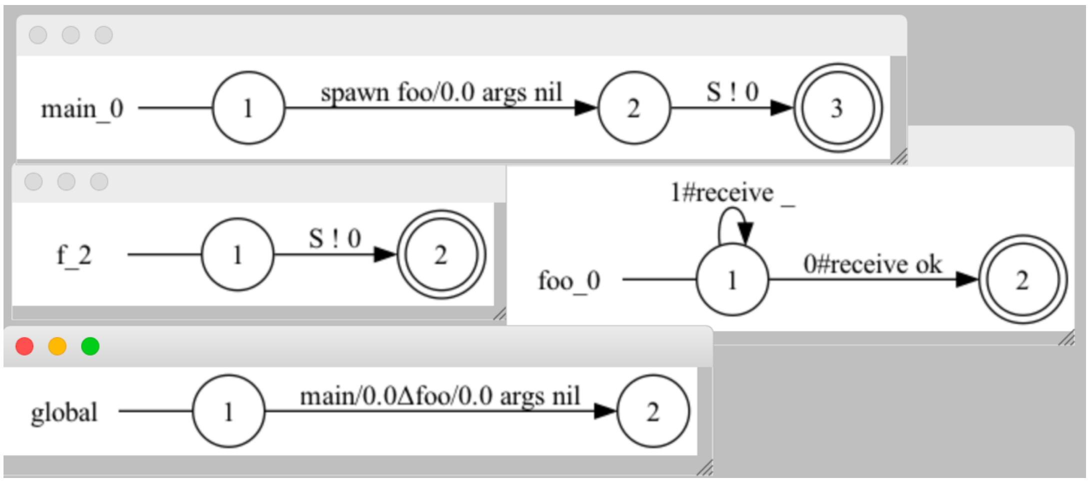


Comments:

- main's local view should not have Pid ! ok and the other messages in parallel, but in a sequence (otherwise, we cannot do a sequence like S ! 0, Pid ! ok)
- global view is not correct

Example foo9d:

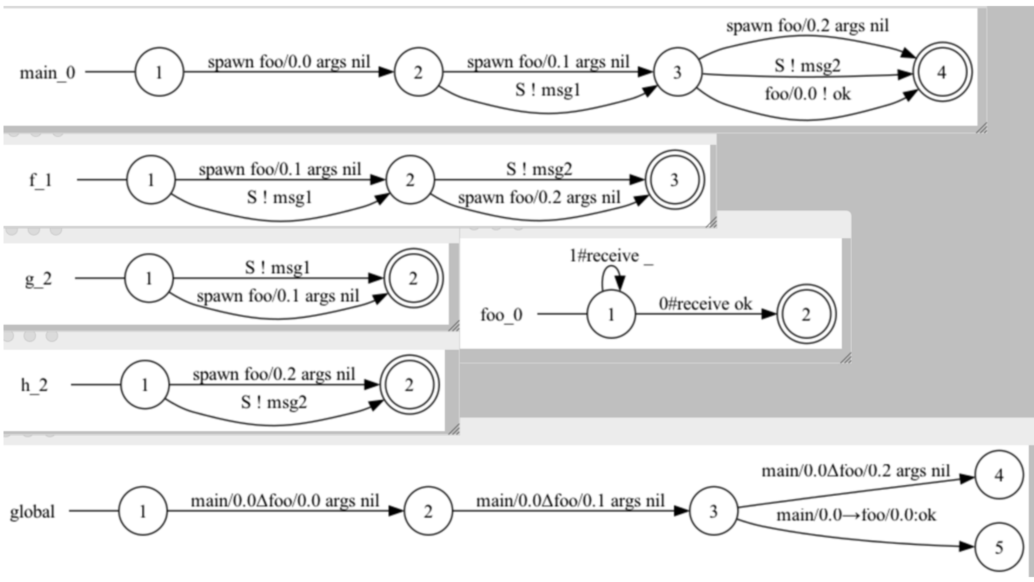
```
-module(foo9d).
-export([main/0,f/2,foo/0]).

main() ->
    S = spawn(foo9d,foo,[]),
    f(S,3).

f(S,N) -> case N of
            0 -> S ! 0;
            M when M>0 ->
                  f(S,N-1), S ! M
          end.

foo() -> receive
            ok -> done;
            _ -> foo()
         end.
```



Coments:
- the local view of function f is not correct, since it ignores the code to the right of the recursive call
- same problem in the local view of main
- global view not correct

Example foo9e:

```
-module(foo9e).
-export([main/0,f/1,g/2,h/2,foo/0]).

main() ->
    S = spawn(foo9e,foo,[]),
    f(S),
    S ! ok.

f(S) -> g(S,1), h(S,0).

g(S,N) -> case N of
            0 -> spawn(foo9e,foo,[]);
            1 -> S ! msg1
          end.

h(S,N) -> case N of
            0 -> S ! msg2;
            1 -> spawn(foo9e,foo,[])
          end.

foo() -> receive
            ok -> done;
            _ -> foo()
         end.
```

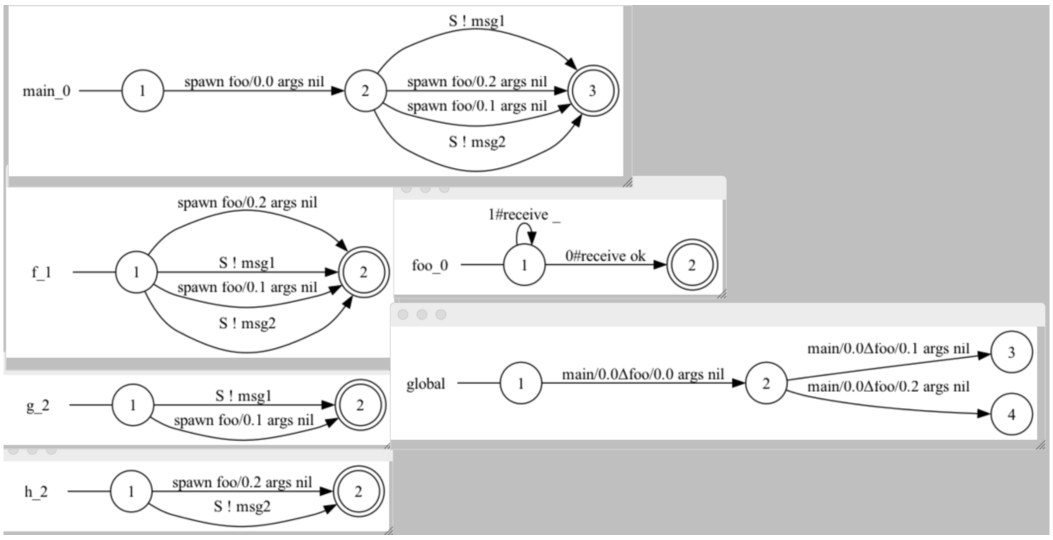


Coments:

- local views ok except for having foo ! ok in parallel with the other two actions in the local view of function main
- nevertheless, I wonder why the graphs of functions g and h have only one final node (in a CFG for imperative programs, we would have two final nodes); I guess it's a correct overapproximation though
- global view not correct

Example foo9f:

```
-module(foo9f).
-export([main/0,f/1,g/2,h/2,foo/0]).

main() ->
   S = spawn(foo9f,foo,[]),
   f(S),
   S ! ok.

f(S) -> h(g(S,1),0).

g(S,N) -> case N of
         0 -> spawn(foo9f,foo,[]);
         1 -> S ! msg1
       end,
       S.

h(S,N) -> case N of
         0 -> S ! msg2;
         1 -> spawn(foo9f,foo,[])
       end.

foo() -> receive
         ok -> done;
         _ -> foo()
      end.
```

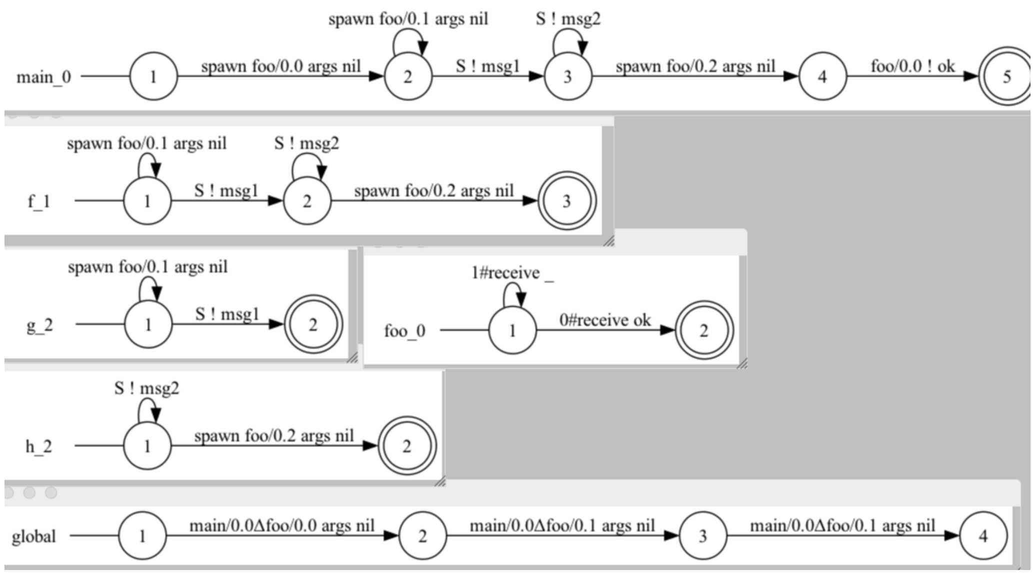


Coments:

- same example as before, but now all actions in functions g and h are shown in parallel in the local view of function f (rather than sequentially, first the actions of g then those of h)
- global view not correct

Example foo9g:

```
-module(foo9g).
-export([main/0,f/1,g/2,h/2,foo/0]).

main() ->
    S = spawn(foo9g,foo,[]),
    f(S),
    S ! ok.

f(S) -> g(S,1),h(S,1).

g(S,N) -> case N of
            0 -> spawn(foo9g,foo,[]),
                   g(S,1),
                   S ! msg3;
            1 -> S ! msg1
          end,
          S.

h(S,N) -> case N of
            0 -> S ! msg2,
                   h(S,1),
                   S ! msg4;
            1 -> spawn(foo9g,foo,[])
          end.

foo() -> receive
            ok -> done;
            _ -> foo()
         end.
```



Coments:

- the actions of functions f and h after the recursive call are lost in their local views
- global view not correct

General comments:

1. Why the branches of a case expression take us to the same state? In the CFG of an imperative program, if_then_else take us to two different states. I guess it's a correct overapproximation, but I don't see a reason to do that.
2. Why the different behaviour between f(g(N)) and M=g(N),f(M)? In principle, under an eager semantics, they should produce the same local view. Currently, f(g(N)) performs all actions in f and in g in parallel, while M=g(N),f(M) first shows the actions of g and then those of f.
3. Functions with recursive calls incorrectly ignore the code after the recursive call. In principle, these recursive calls may enter the base case so that the expressions after the recursive call are evaluated afterwards.