# A gentle introduction to the HoCL language

v 1.2 - May 2020

J. Sérot (jocelyn.serot@uca.fr)

github.com/jserot/hocl

# Introduction

# Motivations

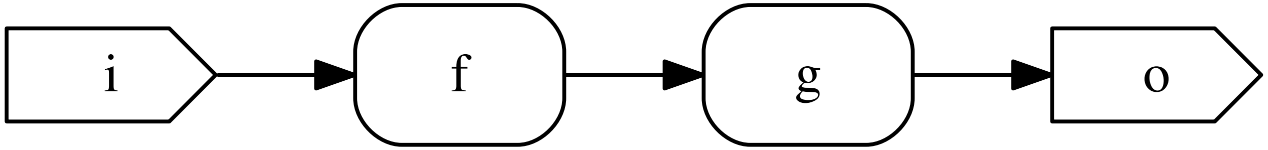
- HoCL = Higher-order Coordination Language
- Language for describing dataflow process networks
- Hierarchical and/or parameterized graphs
- Multi-style descriptions (structural or functional)
- Support of data flow variants (SDF, PSDF, …) by means of annotations
- Independant of the target implementation platform (software, hardware, mixed, ...)
  - targeting done using dedicated backends (Preesm, DIF, XDF, SystemC, …)

# This document

- Informal presentation of the main language features
  - by means of small examples
- Introduce the main existing backends
  - DOT
  - SystemC
  - PREESM

# Basic examples

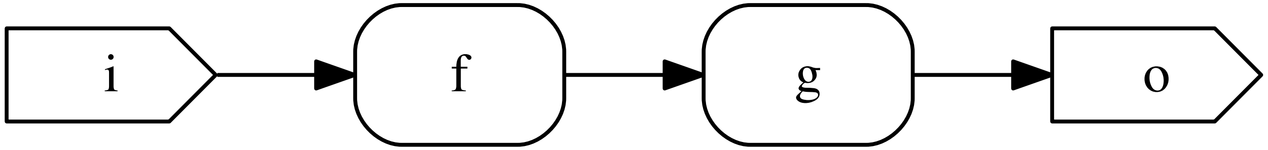
# Example 1




```
node f
  in (i: int) out(o: int);

node g
  in (i: int) out(o: int);

graph top
  in (i: int)
 out (o: int)
struct
  wire w: int
  box n1: f(i)(w)
  box n2: g(w)(o)
end;
```

- This defines a graph *top*, with input *i* and output *o*.
- This graph is built from two boxes, *n1* and *n2*, linked by a wire *w*
- Boxes and wires are *typed*
- Each box is an *instance* of a node (*f* and *g* resp.)
- Nodes *f* and *g* are here defined as opaque actors (black boxes)
- The graph *top* is here defined structurally

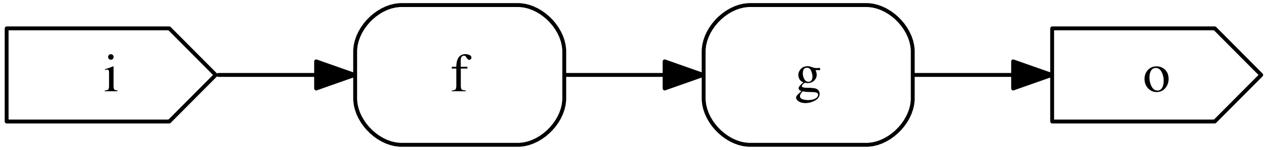
# Example 1




```
node f
  in (i: int)
 out(o: int);

node g
  in (i: int)
 out(o: int);

graph top
  in (i: int)
 out (o: int)
fun
  val o = g (f i)
end;
```

- This is an alternative description of graph top using a functional style
- Nodes are interpreted as *functions* and the graph is described using function application
  - applying function `f` to value `x` (here denoted as `f x)` builds a node by instantiating actor *f* and connecting the wire representing the value *x* to its input
  - Function composition here corresponds to actor *chaining*

# Example 1



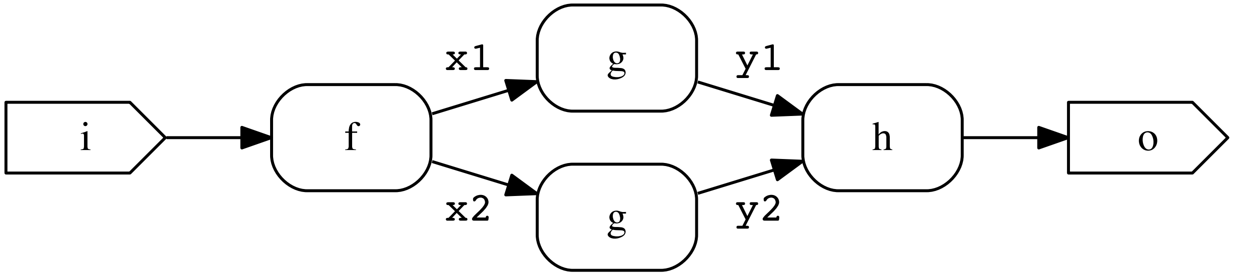

```
node f
  in (i: int)
 out(o: int);

node g
  in (i: int)
 out(o: int);

graph top
  in (i: int)
 out (o: int)
fun
  val o = i |> f |> g
end;
```

- Another functional formulation using the *reverse application operator* |> :

```
x |> f = f x
```

# Example 2

## A slightly more complex graph




```
node f in (i: int) out (o1: int, o2:int);
node g in (i: int) out (o: int);
node h in (i1: int, i2:int) out (o:int);
```

### Structural description

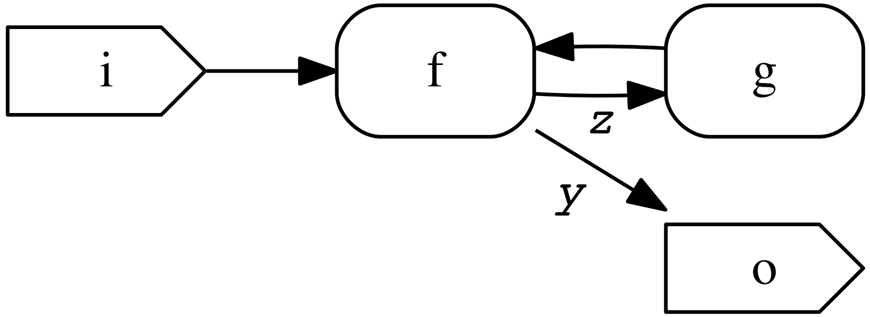
```
graph top
  in (i: int)
 out (o: int)
struct
  wire x1,x2: int
  wire y1,y2: int
  box n1: f(i)(x1,x2)
  box n2: g(x1)(y1)
  box n3: g(x2)(y2)
  box n4: h(y1,y2)(o)
end;
```

### Functional description

```
graph top
  in (i: int)
 out (o: int)
fun
  val (x1,x2) = f i
  val o = h (g x1) (g x2)
end;
```

Note : `f x y` really means `f(x,y)`
(*curried* notation for function application)

# Cycles and recursive wiring




## Functional description

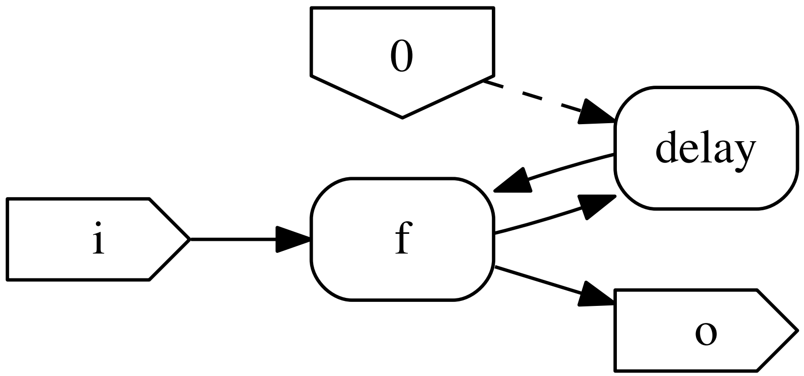
```
graph top
  in (i: int) out (o: int)
fun
  val rec (o,z) = f i (g z)
end;
```

## Structural description

```
graph top
  in (i: int) out (o: int)
struct
  wire w1, w2: int
  box n1: f(i,w1)(o,w2)
  box n2: g(w2)(w1)
end;
```

- Cycles in the graph are created using **recursive definitions**

# Delayed cycles

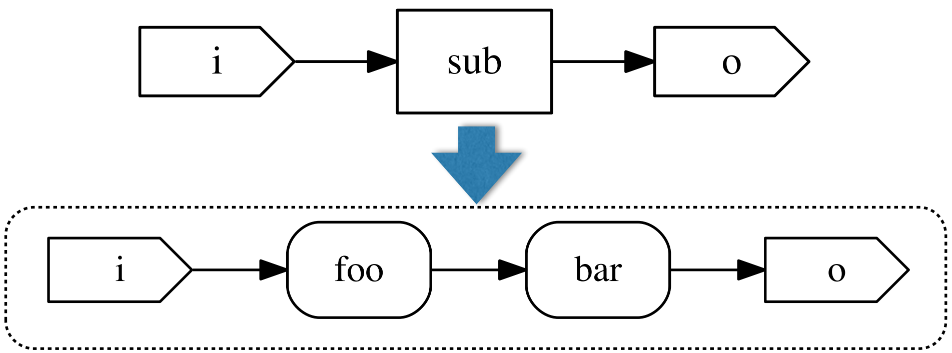



```
graph top
  in (i: int) out (o: int)
fun
 val rec (o,z) = f (i, delay ‘0’ z)
end;
```

```
graph top
  in (i: int) out (o: int)
struct
  wire w1, w2: int
  box n1: f(i,w1)(o,w2)
  box n2: delay(‘0’,w2)(w1)
end;
```

- Delays are required to avoid deadlock when **simulating** the graph (they provide the initial token(s) on the feedback edge(s)
- The special actor *delay* is predefined (and interpreted specifically by the various backends)
  - the actor *parameter* (`‘0’`, here) specifies the initial value)
- Using type or application specific delay actors is also possible

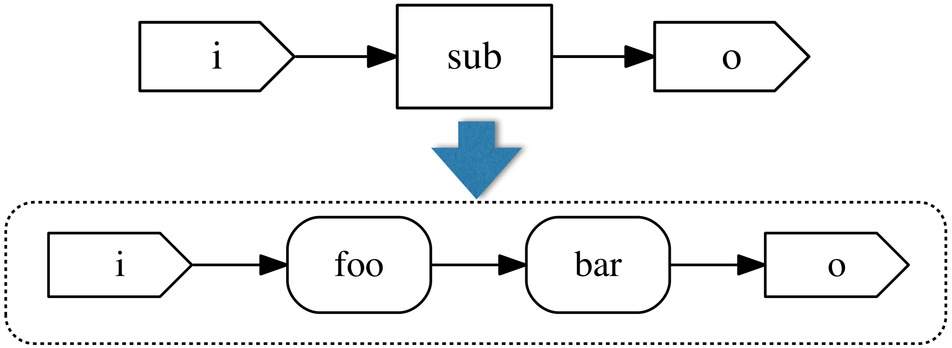
# Hierarchical graphs




```
node foo in (i: t) out (o: t);
node bar in (e: t) out (s: t);

node sub in (i: t) out (o: t)
fun
  val o = i |> foo |> bar
end;

graph top in (i: t) out (o: t)
fun
  val o = i|> sub
end;
```

- Nodes can be described as (sub)graphs (either structurally or functionally), giving rise to **hierarchical** graphs
- Node with no description are interpreted as opaque actors (« blackboxes »)
- **Toplevel** graphs are identified with the *graph* keyword

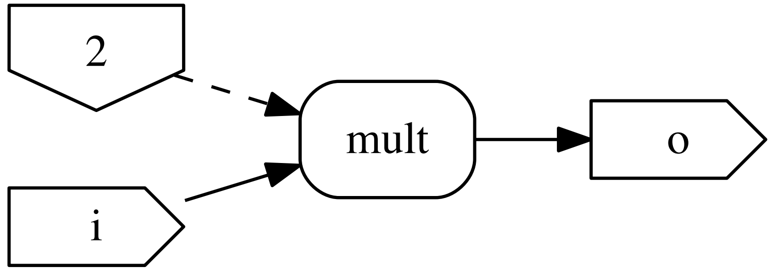
# Hierarchical graphs




```
node foo in (i: t) out (o: t);
node bar in (e: t) out (s: t);

node sub in (i: t) out (o: t)
struct
  wire w1, w2: t
  box n1: foo(i)(w1)
  box n2: bar(w1)(o)
end;

graph top in (i: t) out (o: t)
fun
  val o = i|> sub
end;
```

- Within hierarchical descriptions, structural and functional definitions can be mixed freely

# Parameters



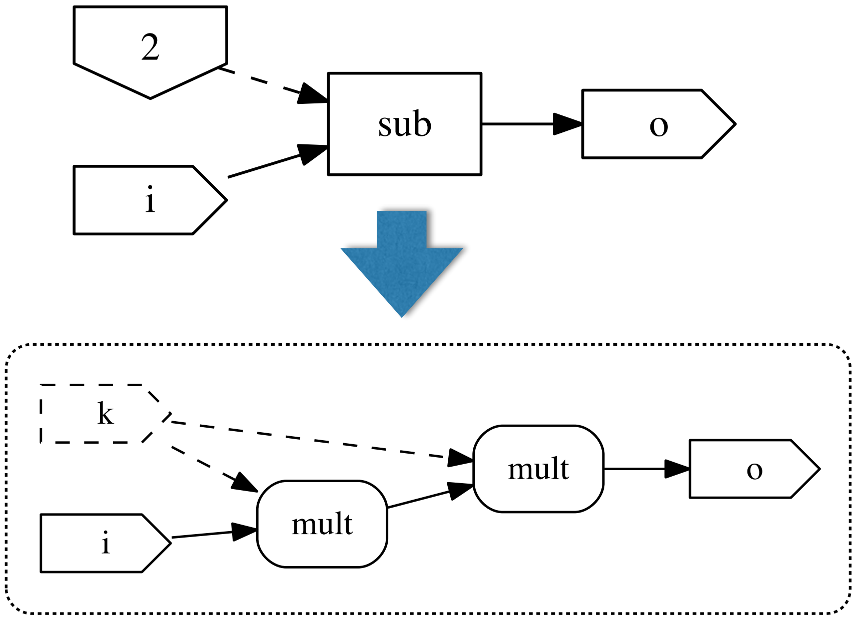

```
node mult
 in (k: int param, i: int)
out (o: int);

graph top
  in (i: int) out (o: int)
fun
  val o = i |> mult '2'
end;
```

```
...

graph top
  in (i: int) out (o: int)
struct
  box n: mult('2',i)(o)
end;
```

- Parameters are used to *configure* (*specialize*) nodes
- Parameters are distinguished from data by their type :
  - `t param` is the type of a parameter having itself type `t`
  - the `'...'` notation turns a value of type `t` into a value of type `t param`
- Parameters normally appear in first position(s) in the list of node arguments
- In functional descriptions, this allows specifying their value using partial application of the corresponding function
  - here `mult '2'` denotes the specialized version of the `mult` node obtained by setting `k=2`

# Parameter passing

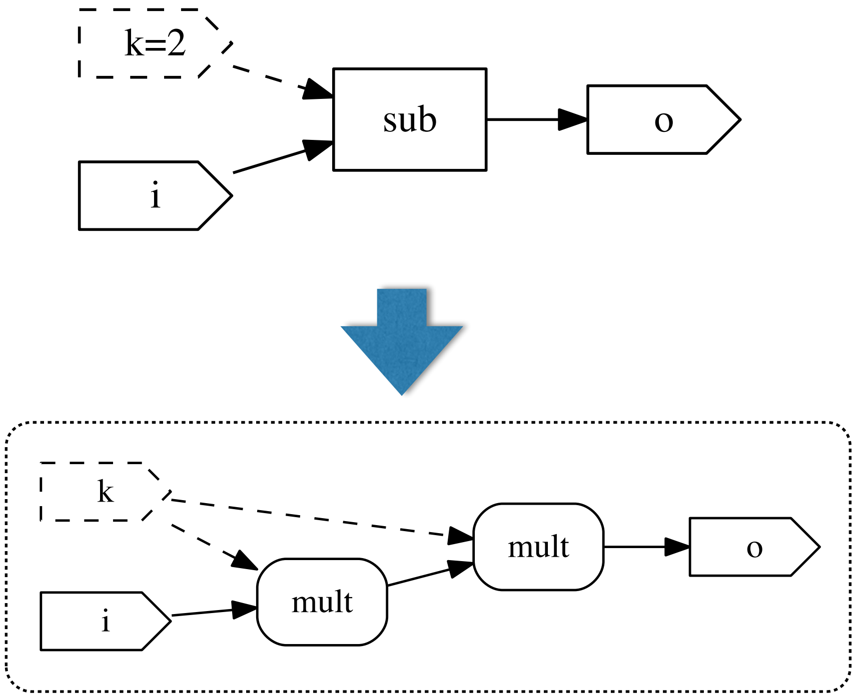
```
node sub
 in (k: int param, i: int)
out (o: int)
fun
  val o =
    i |> mult k |> mult k
end;

graph top
  in (i: int) out (o: int)
fun
  val o = i |> sub ‘2‘
end;
```




- Parameters can be passed from one hierarchy level to a nested one

# Parameter passing

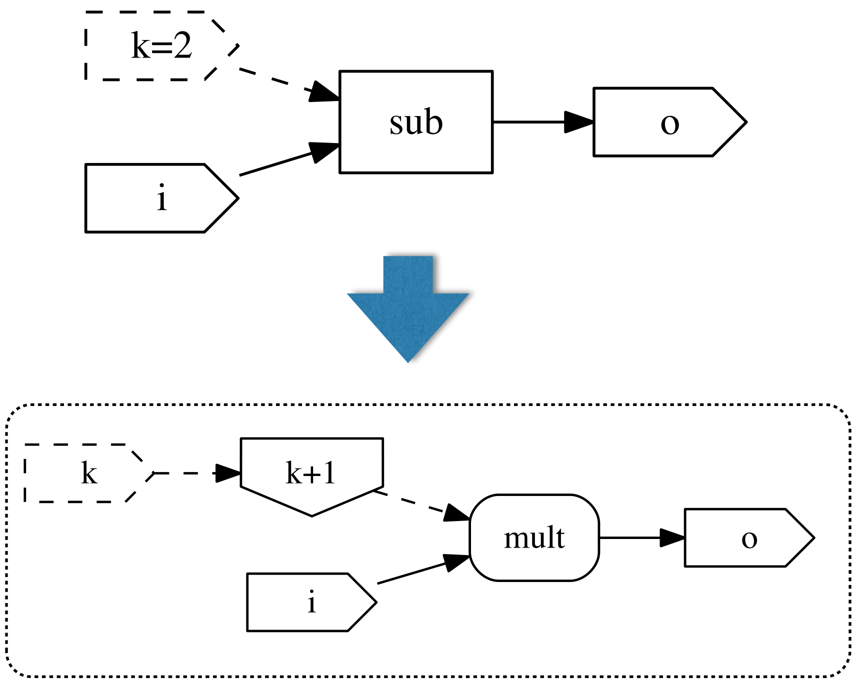
```
node sub
    in (k: int param, i: int)
   out (o: int)
fun
   val o =
     i |> mult k |> mult k
end;

graph top
  in (k: int param=2, i: int)
out (o: int)
fun
   val o = i |> sub k
end;
```




- The value of the toplevel parameters can be defined in the corresponding graph interface

# Parameter dependencies

```
node sub
   in (k: int param, i: int)
  out (o: int)
fun
   val o =
     i |> mult 'k+1'
end;

graph top
   in (k: int param=2, i: int)
  out (o: int)
fun
   val o = i |> sub k
end;
```



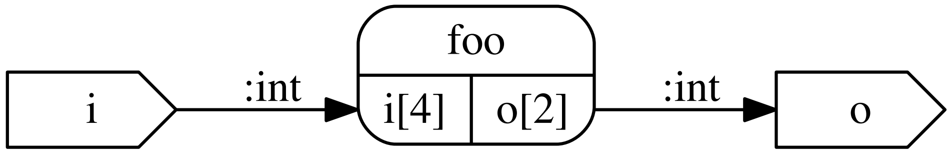

- The value of some parameters can depend on that of other parameters, defined at the same or at higher level(s) in the graph hierarchy
- Dependencies between parameter values create a *tree* in graph, which is "orthogonal" to the data flow

# Dataflow modeling

- HoCL is *a priori* independent of the target execution model (SDF, DDF, …)
  - because the **topology** of the graph does not depend on this model
- Model-specific informations are passed to the backends using **annotations**
- Default annotations refer to production/consumption rates in (P)SDF

## Example

```
node foo
  in (i: int[4])
 out (o: int[2]);

graph top
  in (i: int[4])
 out (o: int[2])
fun
  val o = i |> foo
end;
```



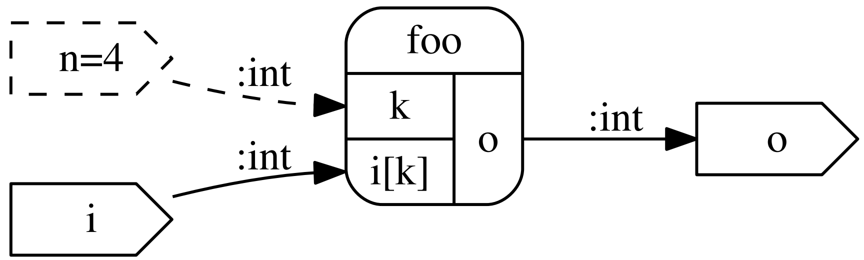

- The actor *foo* consumes 4 tokens (ints) and produces 2 tokens per activation

# Dataflow modeling

- PSDF models are described/implemented by binding C/P rates to parameter values

## Example

```
node foo
 in (k: int param,
     i: int[k])
out (o: int[1]);

graph top
 in (n: int param=3,
     i: int[n])
 out (o: int[1])
fun
  val o = i |> foo n
end;
```

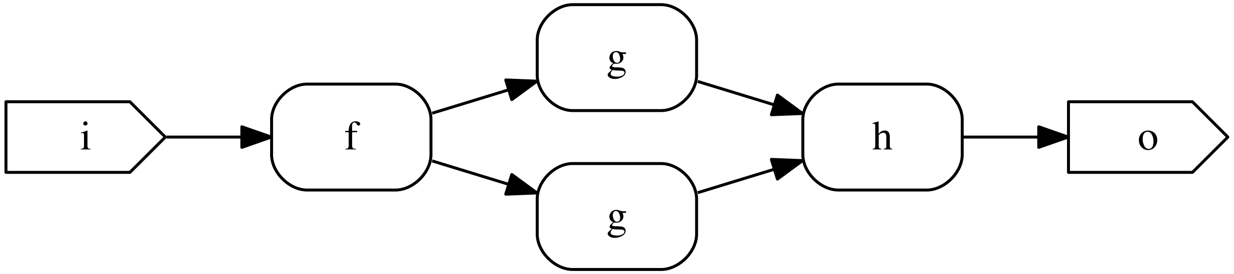



- The actor *foo* here consumes 3 tokens (ints) and produces 1 token per activation
- This can simply be changed by adjusting the value of the toplevel parameter *n*

# Higher order features

... hence the name

# Wiring functions

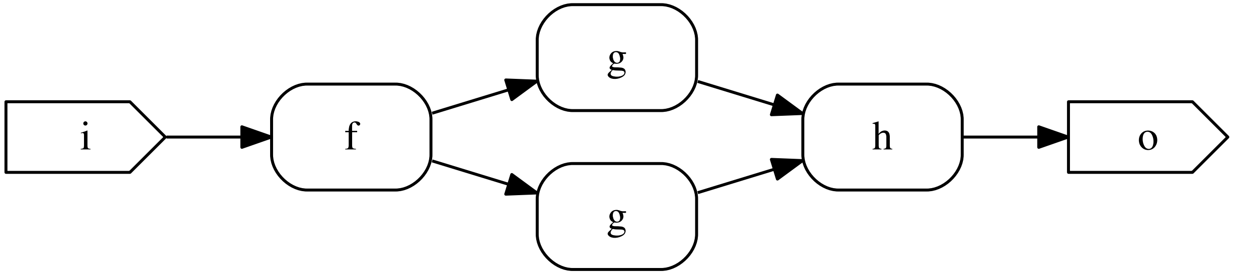



Another formulation :

```
graph top
  in (i: int)
 out (o: int)
fun
  val body x =
   let (x1,x2) = f x in
   h (g x1) (g x2)
  val o = body i
end;
```

- body is a **wiring function** : it encapsulates the wiring pattern of the encoded graph
- The definition of body makes use of a local definition (*let .. in*)
- The top graph is built by simply applying this function
- Wiring functions can be defined within a (sub)graph (local scope) or globally

# Higher order wiring functions

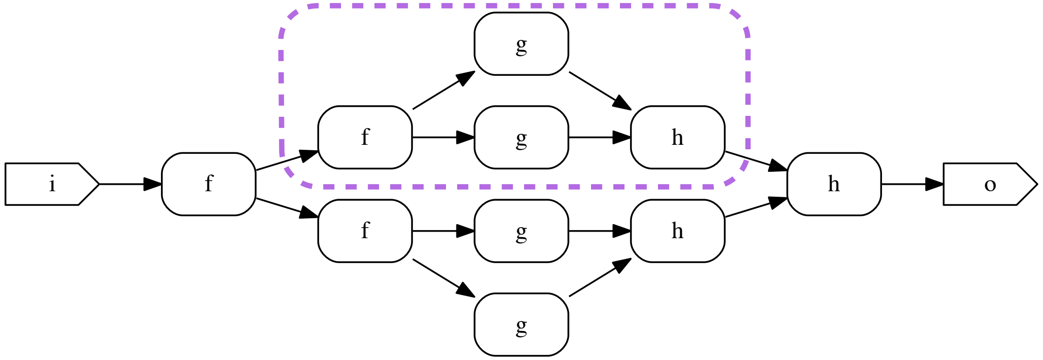



Pushing the abstraction a bit further :

```
graph top
  in (i: int)
 out (o: int)
fun
  val diamond left middle right x =
   let (x1,x2) = left x in
   right (middle x1) (middle x2)
  val o = diamond f g h i
end;
```

- The *diamond* function abstracts further the definition of *body*, by taking as parameters the actors to be instantiated to build the defined graph
- The graph *top* is built by supplying the actual actors (*f*, *g* and *h*) as arguments to *diamond*.
- *diamond* is an **higher-order wiring function** (HOWF)

# Higher order wiring functions



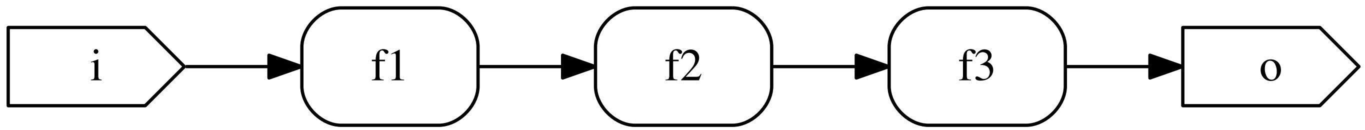

```
graph top
  in (i: int) out (o: int)
fun
  val diamond l m r x = …
  val sub = diamond f g h
  val o = diamond f sub h i
end;
```

```
graph top
  in (i: int) out (o: int)
struct
  wire w1,w2,w3,w4,
       w5,w6,w7,w8,
       w9,w10,w11,w12:int
  box f1: f(i)(w1,w2)
  box f2: f(w1)(w3,w4)
  box f3: f(w2)(w5,w6)
  box g1: g(w3)(w7)
  box g2: g(w4)(w8)
  box g3: g(w5)(w9)
  box g4: g(w6)(w10)
  box h1: h(w7,w8)(w11)
  box h2: h(w9,w10)(w12)
  box h3: h(w11,w12)(o)
end;
```

- The *diamond* function is here instantiated at two levels :
  - within the *sub* function, to describe the « inner » diamond structure
  - within the definition of the output *o*, to build the toplevel graph structure

# « Classic » higher order wiring functions

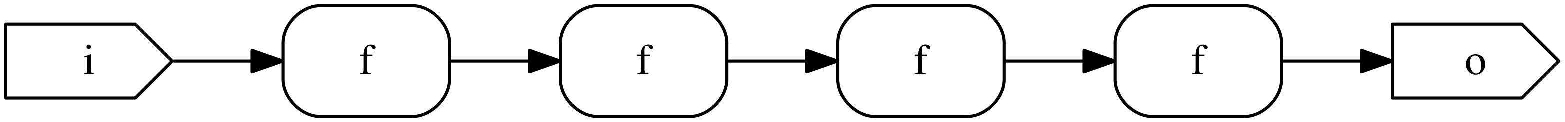
- Many recurrent graph patterns can be encapsulated using higher-order wiring functions
- Several of these functions are given in the HoCL standard library




```
graph top
   in (i: int)
  out (o: int)
fun
   val o = i |> pipe [f1;f2;f3]
end;
```

# Classic higher order wiring functions

- Many recurrent graph patterns can be encapsulated using higher-order wiring functions
- Several of these functions are given in the HoCL standard library

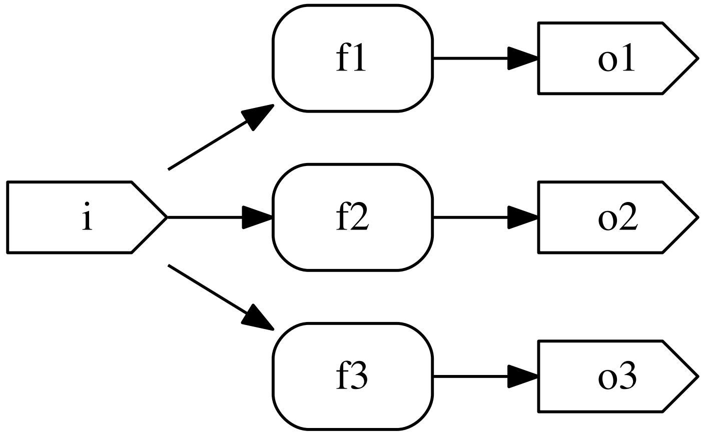



```
graph top
  in (i: int)
 out (o: int)
fun
  val o = i |> iter 4 f
end;
```

# Classic higher order wiring functions

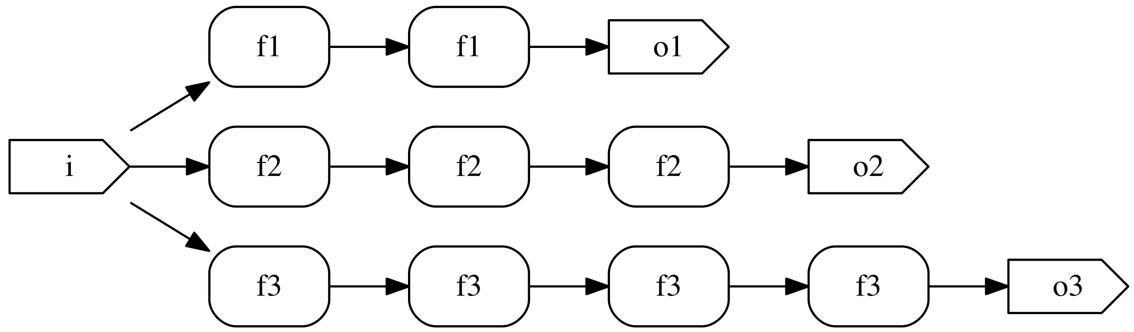
- Many recurrent graph patterns can be encapsulated using higher-order wiring functions
- Several of these functions are given in the HoCL standard library

```
graph top
  in (i:int)
 out (o1:int, o2:int, o3:int)
fun
  val (o1,o2,o3) =
    i |> mapf [f1;f2;f3]
end;
```



# Classic higher order wiring functions

- HOWFs can be combined to describe complex structured graph patterns




```
graph top
   in (i:int)
  out (o1:int, o2:int, o3:int)
fun
   val (o1,o2,o3) =
     i |> mapf
      [iter 2 f1;
       iter 3 f2;
       iter 4 f3]
end;
```

# Backends

# Dot

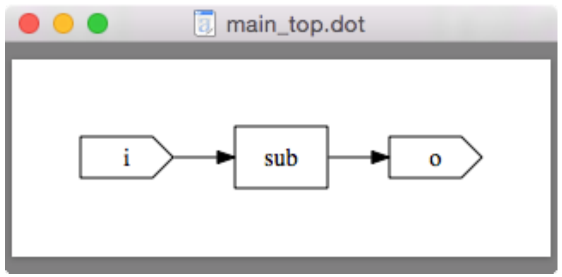
- Used to generate graphical representations of the described DPNs
- One graph per toplevel entity (*graph*) and sub-network (*node … struct/fun*)
- Many options to customize aspects
- No actor implementation required at this level

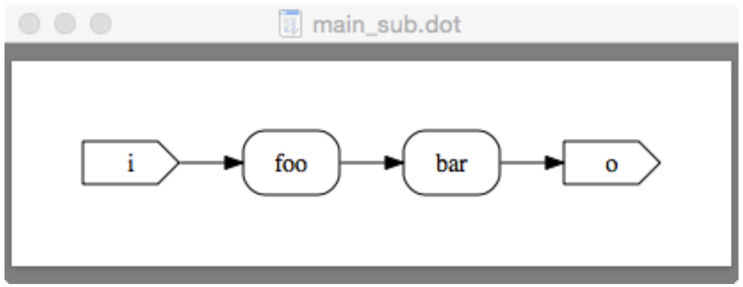
## Example

```
node foo  in (i: t) out (o: t);
node bar in (e: t) out (s: t);

node sub in (i: t) out (o: t)
fun
  val o = i |> foo |> bar
end;

graph top in (i: t) out (o: t)
fun
  val o = i|> sub
end;
```

*main.hcl*

```
bash> hoclc -dot main.hcl
# Wrote file ./main_top.dot
# Wrote file ./main_sub.dot
bash> graphviz *.dot
```



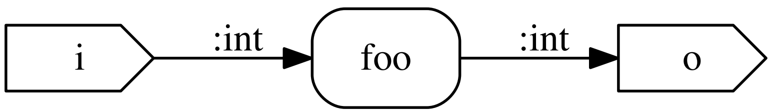
# SystemC

- Used to simulate the described DPNs
- Initialisation and per-activation code provided as external C functions
- Automatic generation of FIFOs, delay, broadcast and IO nodes (reading/writing files)

## Example

```
node foo
 in (i: int) out (o: int)
actor
  systemc(
    loop_fn="foo",
    incl_file="foo.h",
    src_file="foo.cpp")
end;

graph top
  in (i: int) out (o: int)
fun
  val o = i |> foo
end;
```

*main.hcl*




```
void foo(IN int *i, OUT int *o);
```

*foo.h*

```
void foo(IN int *i, OUT int *o)
  { *o = *i * 2; }
```

*foo.c*

```
1 2 3 4 …
```

*top_i.dat*

```
bash> hoclc -systemc main.hcl
# Wrote file systemc/main_top.cpp
# Wrote file systemc/top_gph.h
# Wrote file systemc/foo_act.h
# Wrote file systemc/foo_act.cpp
```

```
bash> cd ./systemc; make
```

```
2 4 6 8 …
```

*top_o.dat*

# Preesm

*main.hcl*

```
node inp in () out(o: int)
actor
 preesm(loop_fn="inp",
        init_fn="inpInit",
        incl_file="input.h",
        src_file="input.cpp")
end;

node foo
in (k: int param, p: int param,
    i: int)
out (o: int)
actor
  preesm(loop_fn="foo",
         incl_file="foo.h",
         src_file="foo.cpp")
end;

node outp
 in (p: int param, i: int) out ()
actor
  preesm(loop_fn="outp",
         init_fn="outpInit",
         incl_file="output.h",
         src_file="output.cpp")
end;

graph top
  in (k:int param=2, p:int param=2)
 out ()
fun
  val _ = inp |-> foo k p |> outp p
end;
```

*foo.h*

```
#include "preesm.h"
void foo(int k, IN int *i, OUT int *o);
```

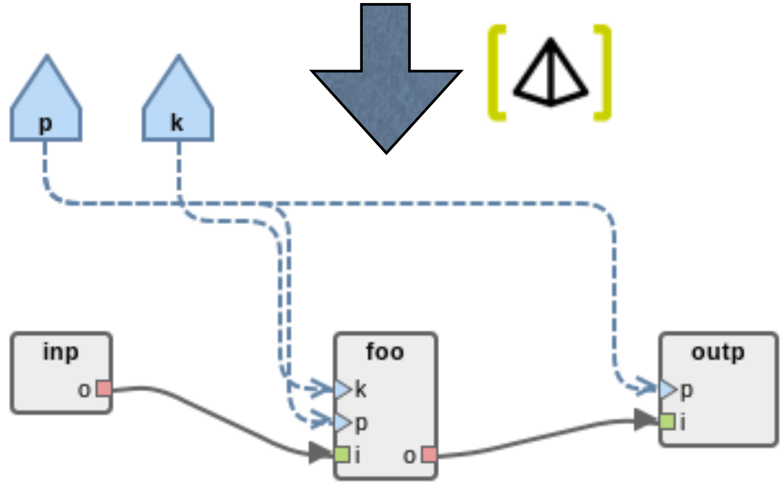
*input.h*

```
#include "preesm.h"
void inpInit(void);
void inp(OUT int *o);
```

*output.h*

```
#include "preesm.h"
void outp(int p, IN int *i);
void outpInit(void);
```

```
bash> hoclc -preesm main.hcl
# Wrote file ./main_top.pi
```

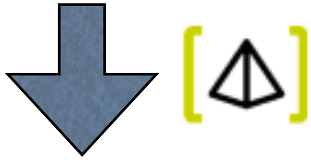


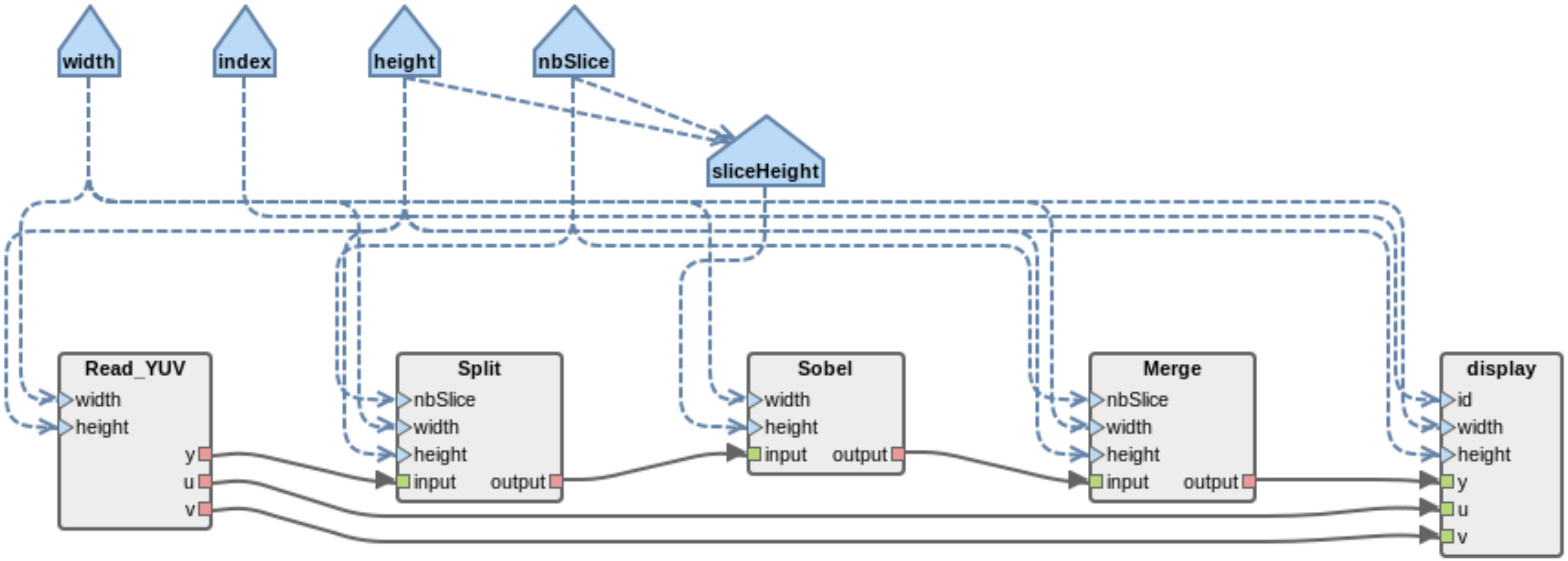
# PREESM backend - A classical example

```
type uchar;

node ReadYUV
  param (width: int, height: int)
  in ()
  out (y: uchar[height*width],
       u: uchar[height/2*width/2],
       v: uchar[height/2*width/2])
actor
  systemc(loop_fn="yuvRead",
    init_fn="yuvReadInit",
    incl_file="yuvRead.h",
    src_file="yuvRead.c")
end;

node DisplayYUV
  param (id:int, width:int,
         height: int)
  in (y: uchar[height*width],
      u: uchar[height/2*width/2],
      v: uchar[height/2*width/2])
  out ()
actor
  systemc(loop_fn="yuvDisplay",
   init_fn="yuvDisplayInit",
   incl_file="yuvDisplay.h",
   src_file="yuvDisplay.c")
end;
```

```
node Sobel
  param (width:int, height:int)
  in (input: uchar[height*width])
  out (output: uchar[height*width])
actor
  systemc(loop_fn="sobel",
   incl_file="sobel.h",
   src_file="sobel.c")
end;

graph main
  param (width:int=352, height:int=288,
         index:int=0)
  in () out ()
fun
  val (yi,u,v) =
    ReadYUV(width,height) ()
  val yo =
    yi |> Sobel(width,height)
  val _ =
    DisplayYUV(index,width,height) (yo,u,v)
end;
```

# PREESM backend - A classical example






https://github.com/jserot/hocl/tree/master/examples/working/apps/sobel1

# Other backends

- DIF (Dataflow Interchange Format)
  - for interfacing to external dataflow analysis tools / writing specific code generators
- XDF
  - for interfacing to CAL-based design flows / writing specific code generators

✓ Open source and documented API (OCaml) for writing dedicated backends « from the inside »

- gaining advantage of the compiler synthetized informations (types, annotations, …)