A gentle introduction to the HoCL language

v 0.1 - May 2019

J. Sérot (jocelyn.serot@uca.fr)

github.com/jserot/hocl

Introduction

Motivations

- HoCL = Higher-order Coordination Language (temporary name...)
- Main goal : a language for describing dataflow process networks
- Should support a large class of dataflow variants (SDF, PSDF, DDF, ...)
- Should be - AFAP - independant of the target implementation platform (software, hardware, mixed, ...)
  - targeting is done by dedicated backends
- Relying on concepts drawn from functional programming languages (Haskell, Caml, ...) to allow the description of DFPNs in a concise and abstract manner
  - polymorphic type system
  - automatic type inference and checking
  - functions for top-down descriptions
  - higher-order functions for encapsulation graph patterns

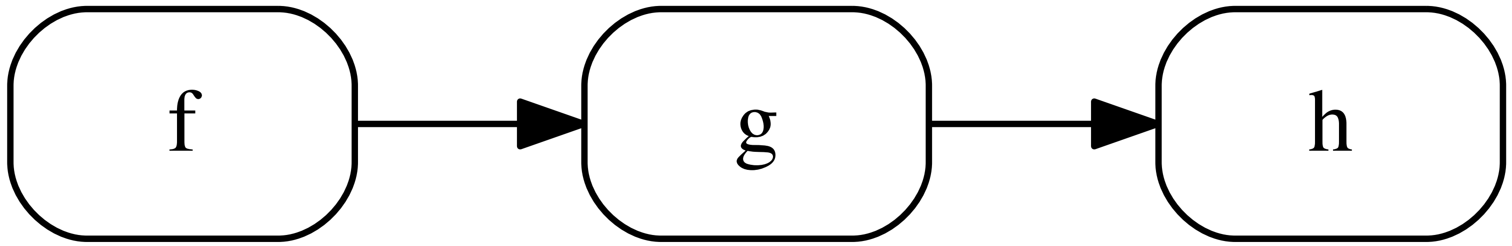
This document

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

Examples

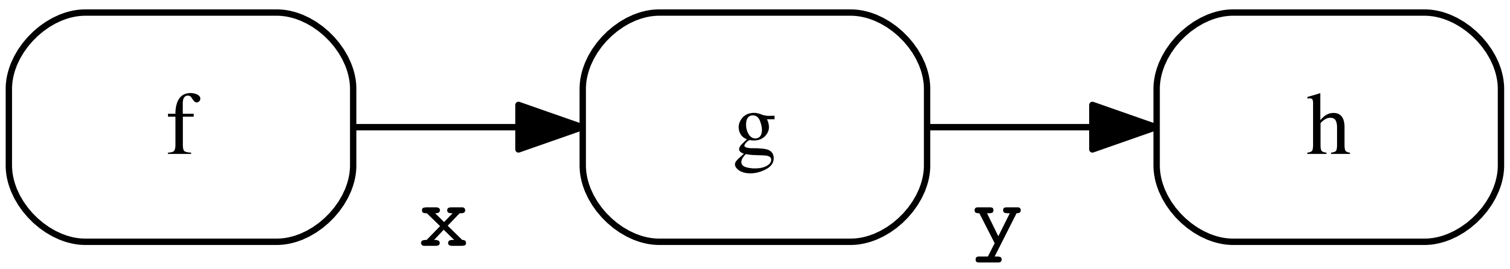
Example 1

Dataflow graphs are functional programs



Example 1

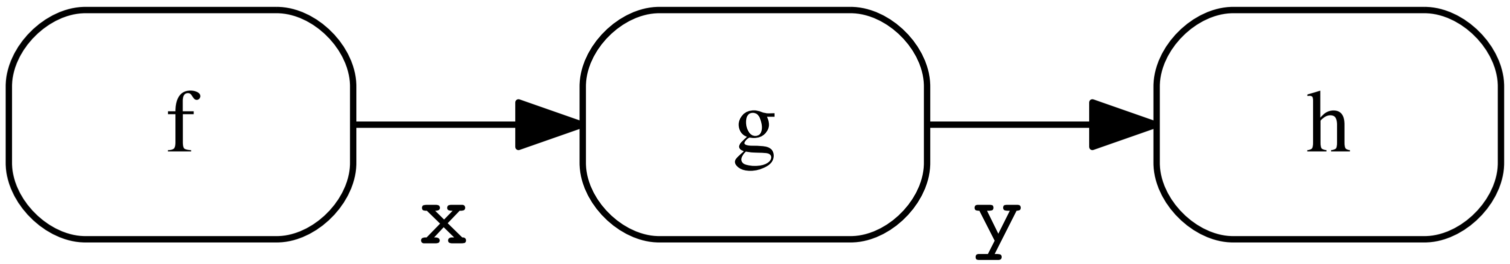
Dataflow graphs are functional programs




```
let x = f ();
let y = g x;
let _ = h y;
```

Example 1

Dataflow graphs are functional programs

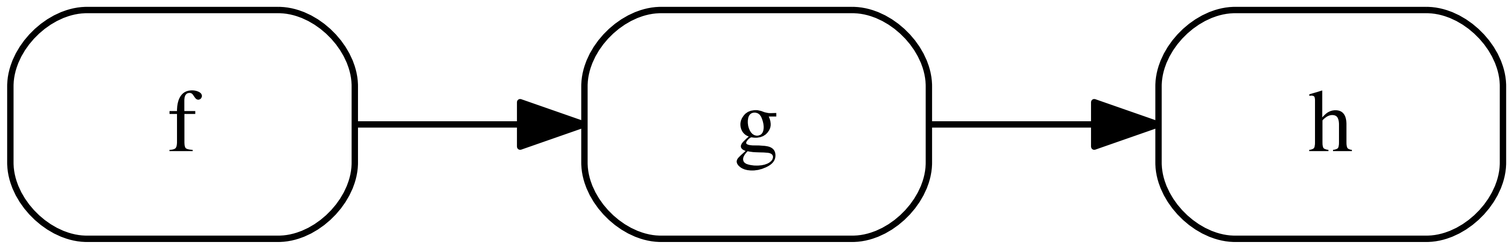



```
let x = f ();
let y = g x;
let _ = h y;
```

- Actors are interpreted as *functions*
- Function *applications* build nodes
  - application is here denoted without parens : `f x` really means `f(x)`
  - `()` means *unit* (*void*)
  - _ means *dont care*
- Wiring is (here) explicited by *names*

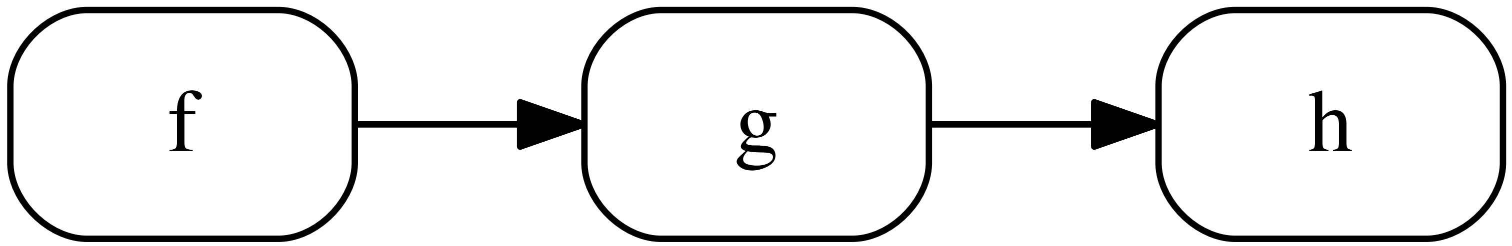
Example 1

Another formulation



Example 1

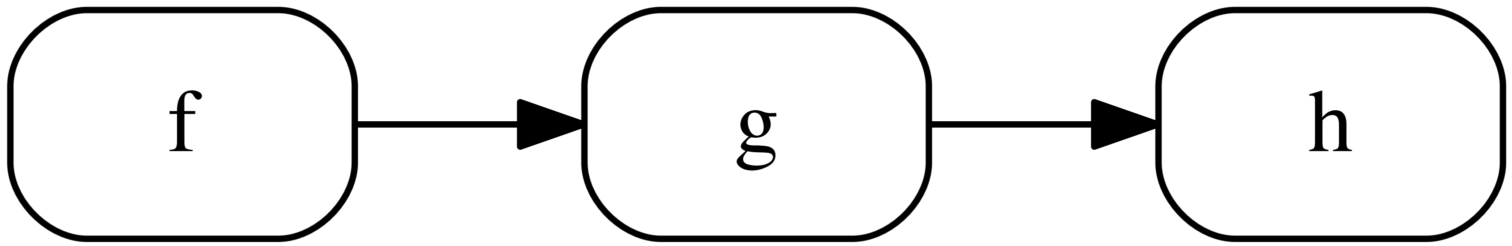
Another formulation




```
let _ = h (g (f ()));
```

Example 1

Another formulation

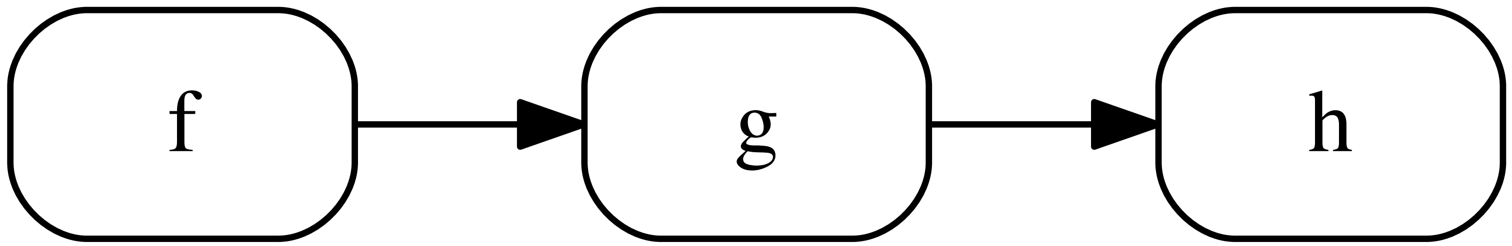



```
let _ = h (g (f ()));
```

- Names are not necessary
- The graph structure is here explicited by means of *function composition*

Example 1

Another formulation




```
let _ = h (g (f ()));
```

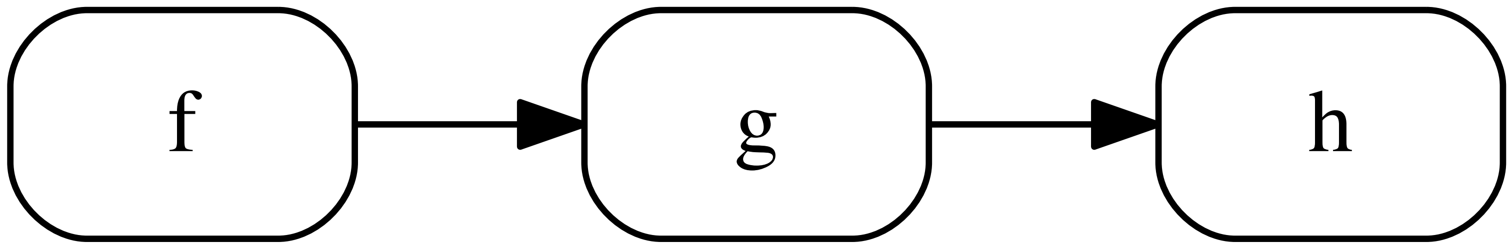
- Names are not necessary
- The graph structure is here explicited by means of *function composition*

☹ parenthesis tend to accumulate

☹ functions appear in reversed order *wrt.* the graph

Example 1

A more friendly syntax

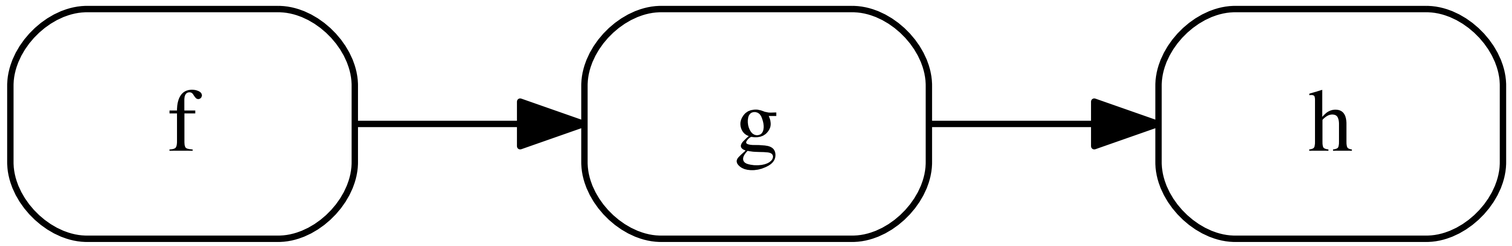



```
let _ = () >> f >> g >> h;
```

- ... thanks to the *reverse application operator* >> :   `x >> f = f x`

Example 1

... even more friendly



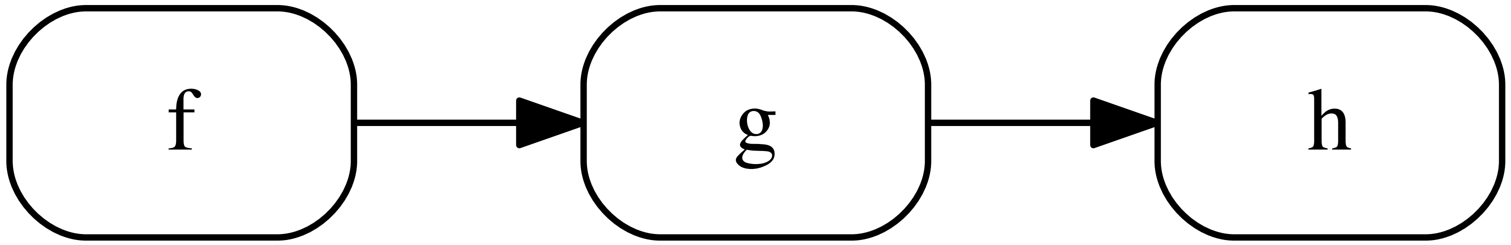

```
let _ = f |> g >> h;
```

- ... thanks the *reverse unit application operator* |> :

```
     f |> g
=    () >> f >> g
=    g (f ())
```

Example 1

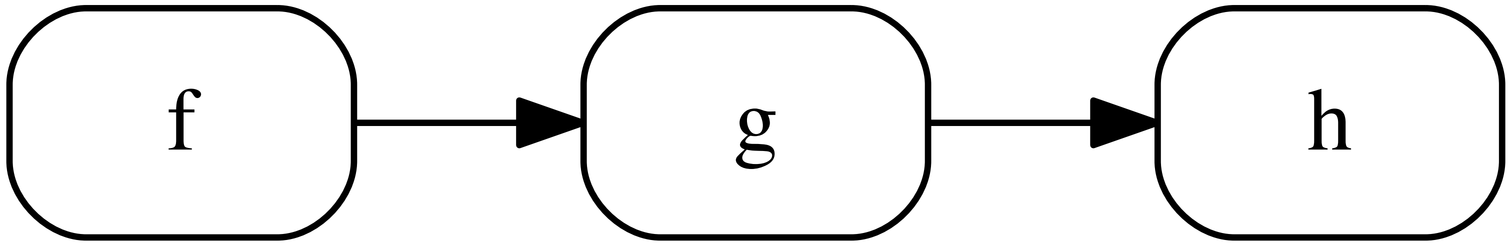
And what about actors ?




```
actor f in () out (o: t);

actor g in (i: t) out (o: t');

actor h in (i: t') out ();
```

Example 1

And what about actors ?




```
actor f in () out (o: t);

actor g in (i: t) out (o: t');

actor h in (i: t') out ();
```

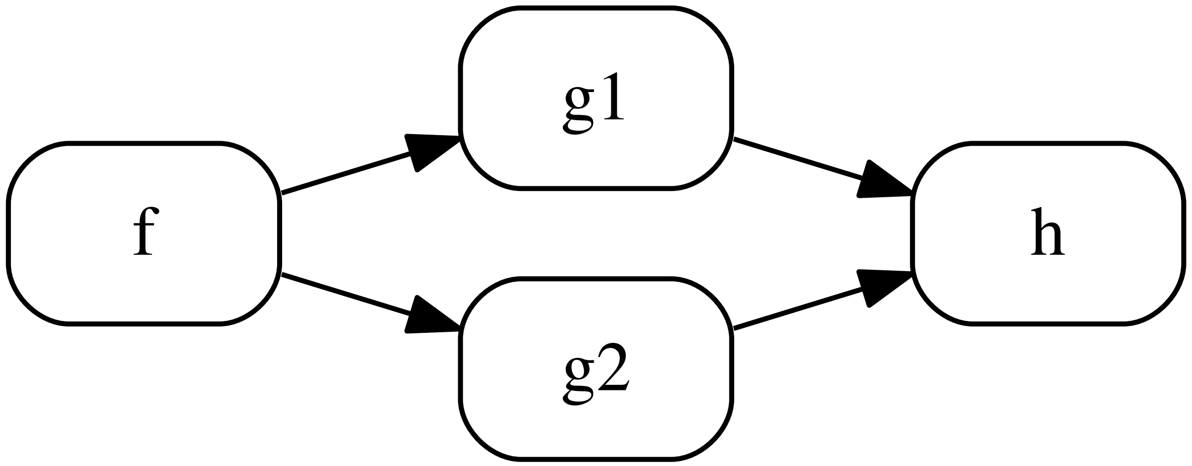
- This gives the following types to the *functions* representing the actors :
- ... where `t1 → t2`

  is the type of a function taking an *argument* with type `t1` and returning a result with type `t2`

```
f : unit → t
g : t → t'
h : t' → unit
```

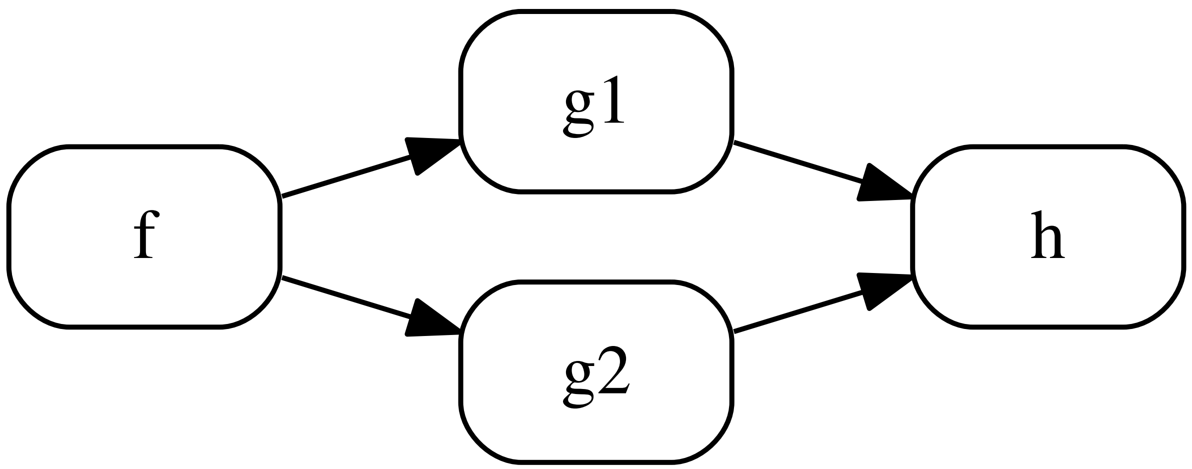
Example 2

Introducing tuples



Example 2

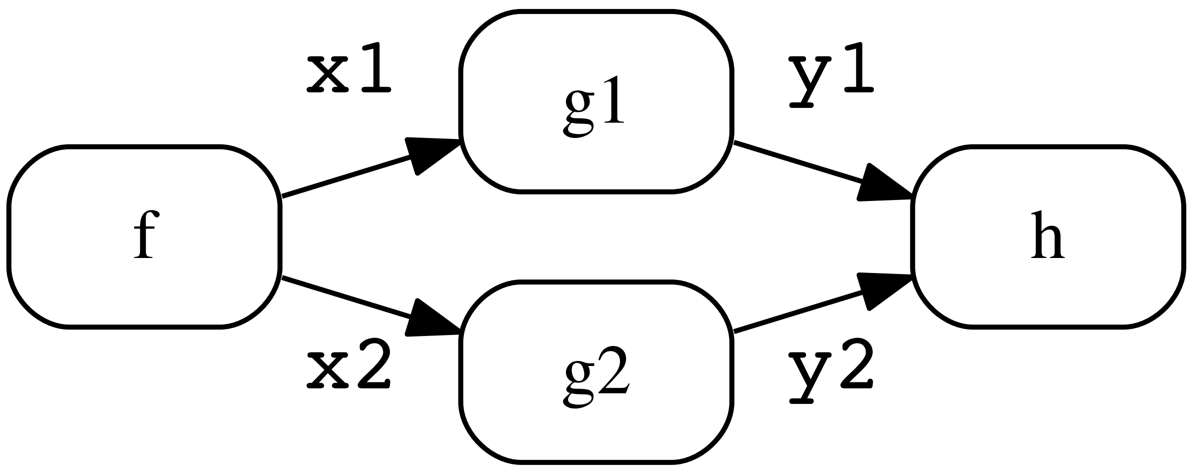
Introducing tuples



- Actor f now has two outputs
- Actor h now has two inputs

Example 2

Introducing tuples

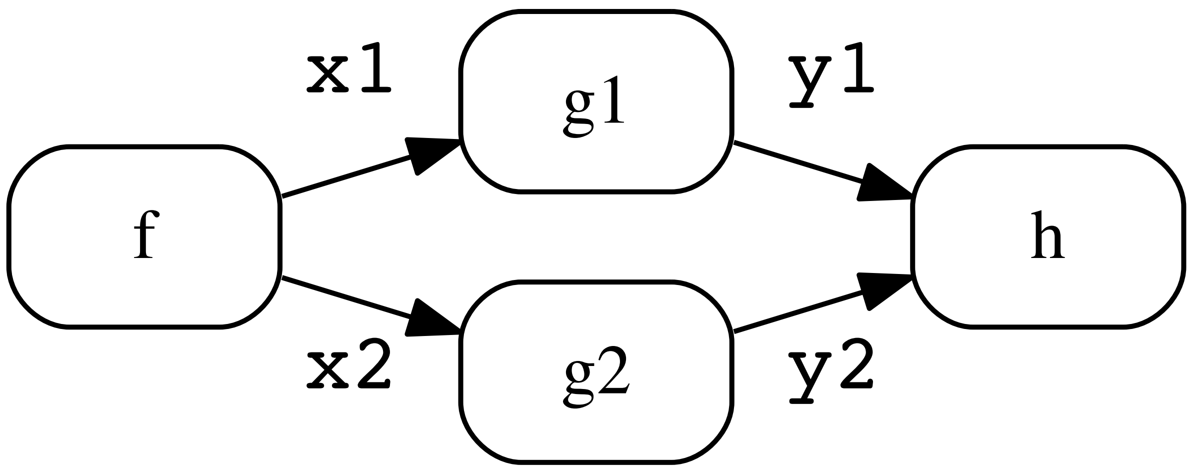


- Actor f now has two outputs
- Actor h now has two inputs

```
let (x1,x2) = f ();
let y1 = g1 x1;
let y2 = g2 x2;
let _ = h (y1,y2);
```

Example 2

Introducing tuples



- Actor f now has two outputs
- Actor h now has two inputs

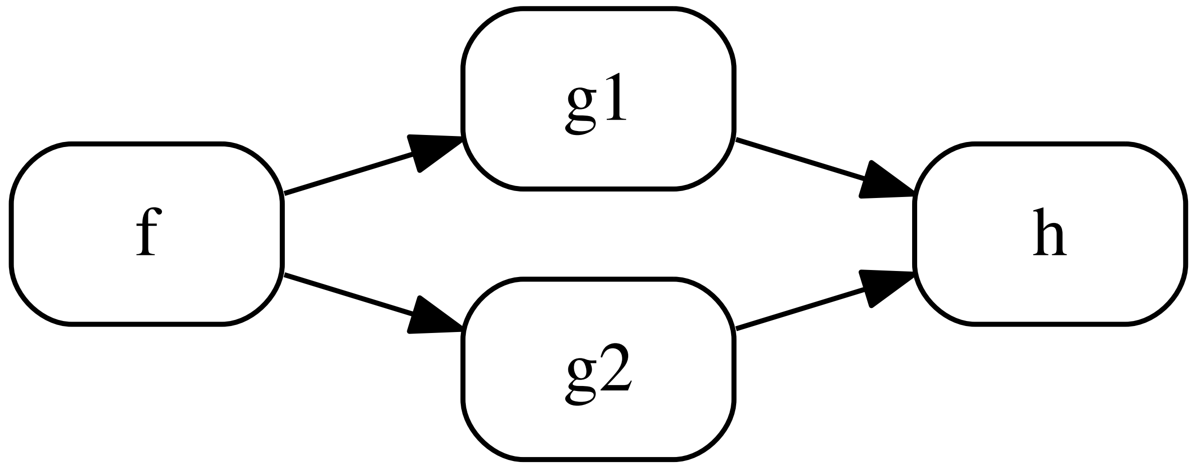
```
let (x1,x2) = f ();
let y1 = g1 x1;
let y2 = g2 x2;
let _ = h (y1,y2);
```

- Sets of arguments (resp. results) are represented by *tuples*
- Naming the components of a *tuples* allows wires to be distinguished
- The `f`, `g1`, `g2` and `h` functions resp. have type (e.g.) :

```
f  : unit → t1 * t2
g1 : t1 → t1'
g2 : t2 → t2'
h  : t1' * t2' → unit
```

Example 2

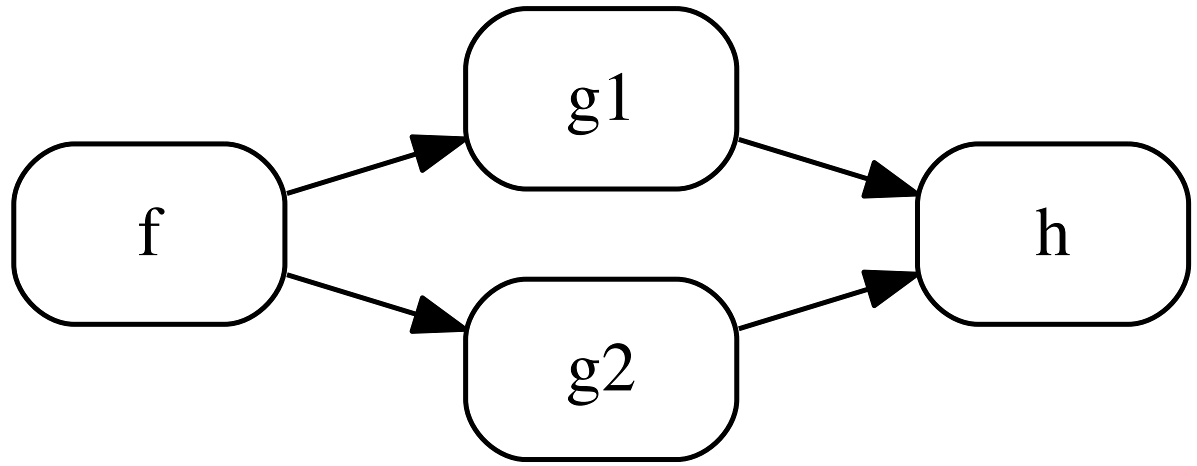
A slightly less verbose reformulation




```
let (x1,x2) = f ();
let _ = h (g1 x1, g2 x2);
```

Example 2

A slightly less verbose reformulation

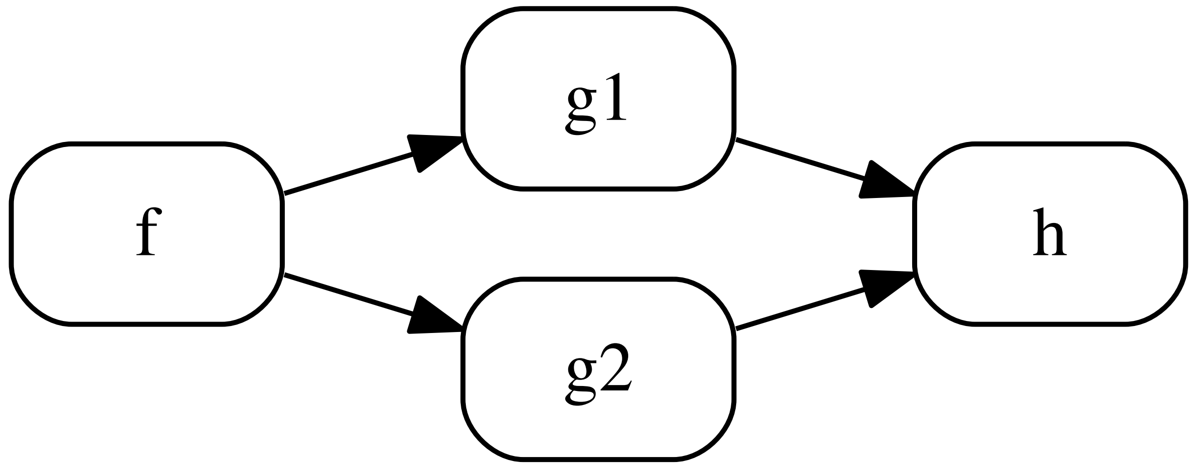



```
let (x1,x2) = f ();
let _ = h (g1 x1, g2 x2);
```

- The >> and |> operators dont help a lot here because we need to distinguish between the two results (resp. arguments) produced by `f` (resp. given to `h`)

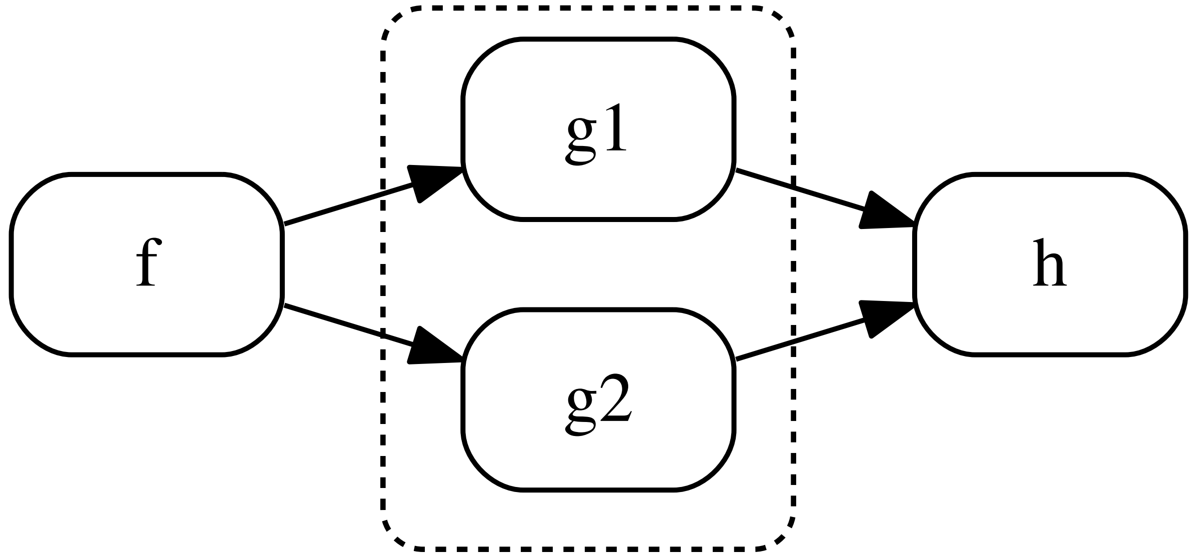
Example 3

Wiring functions



Example 3

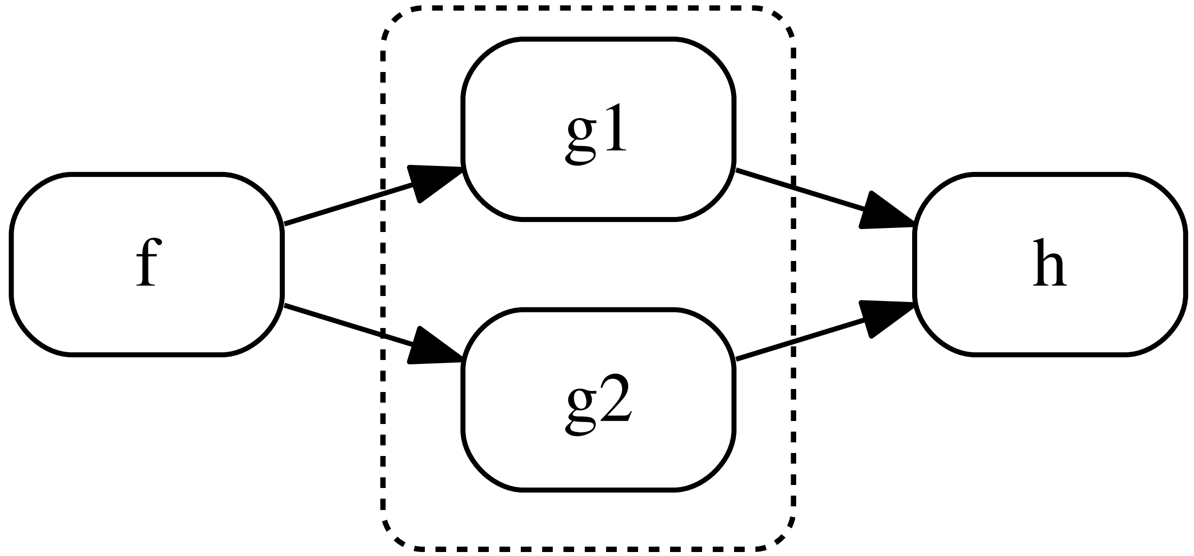
Wiring functions




```
let m (x,y) = g1 x, g2 y;
let _ = f |> m >> h;
```

Example 3

Wiring functions



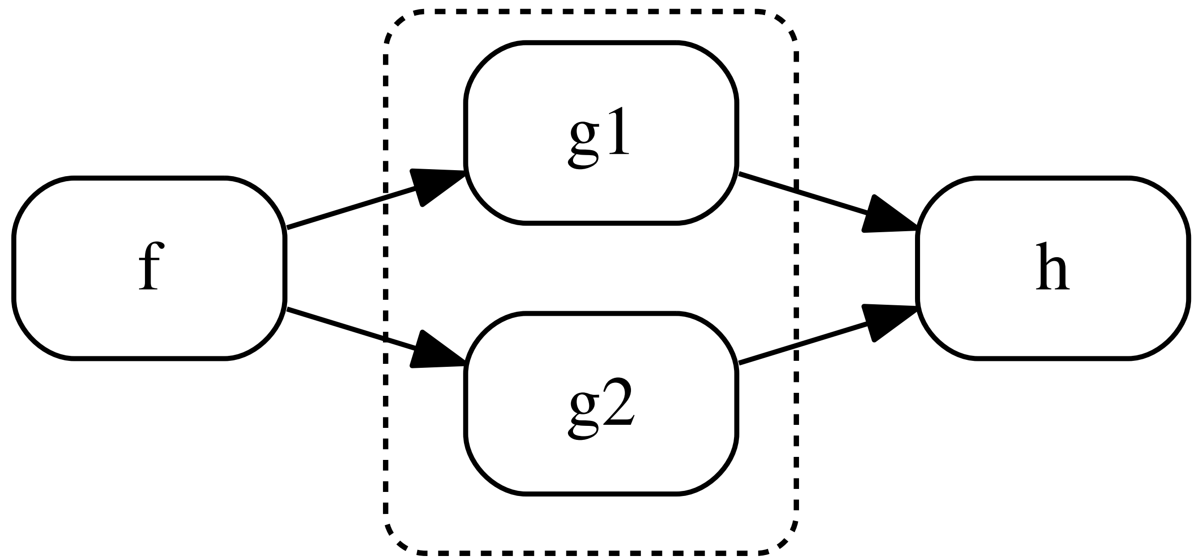

```
let m (x,y) = g1 x, g2 y;
let _ = f |> m >> h;
```

- The first lines defines `m` as a *function*
  - taking a pair of arguments and returning a pair of results
- The `m` function is a *wiring function*
- Its type is :

```
m  : t1 * t2 → t1’ * t2’
```

Example 4

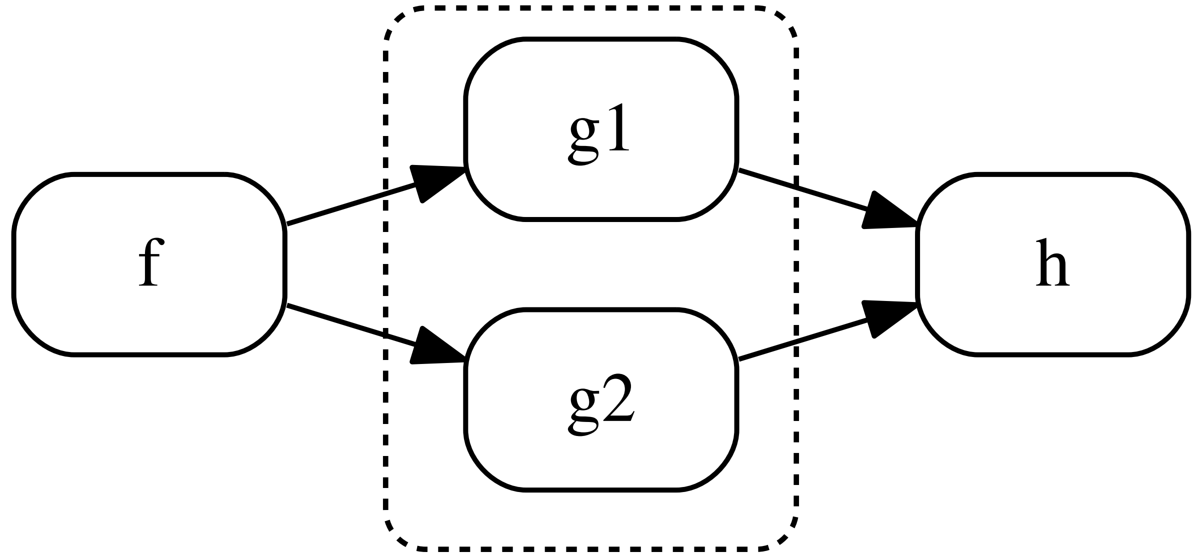
Pushing abstraction a little bit further : HOWF




```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

Example 4

Pushing abstraction a little bit further : HOWF

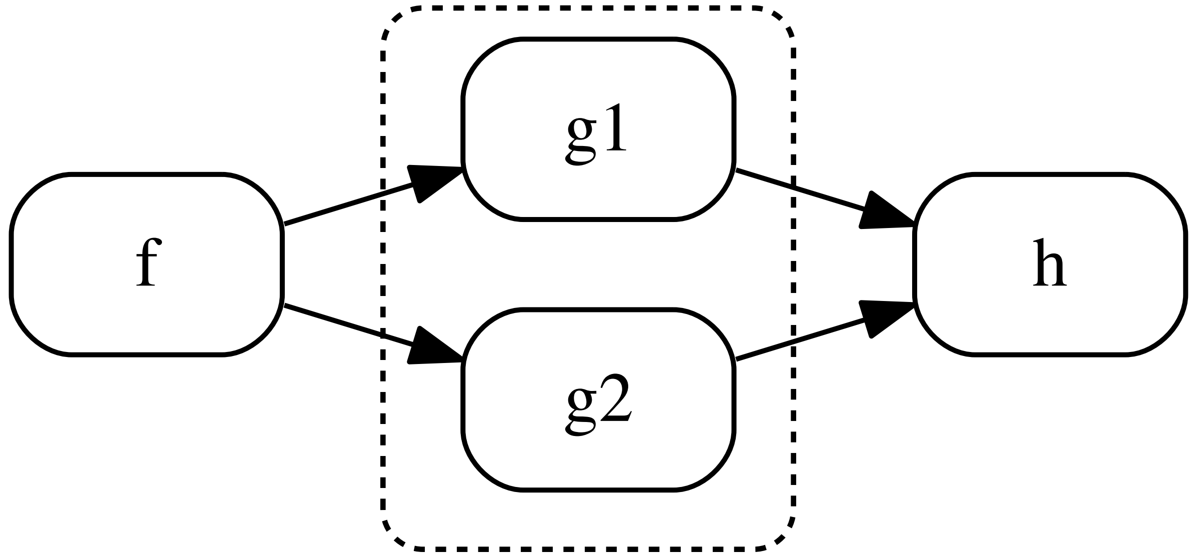



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :

Example 4

Pushing abstraction a little bit further : HOWF

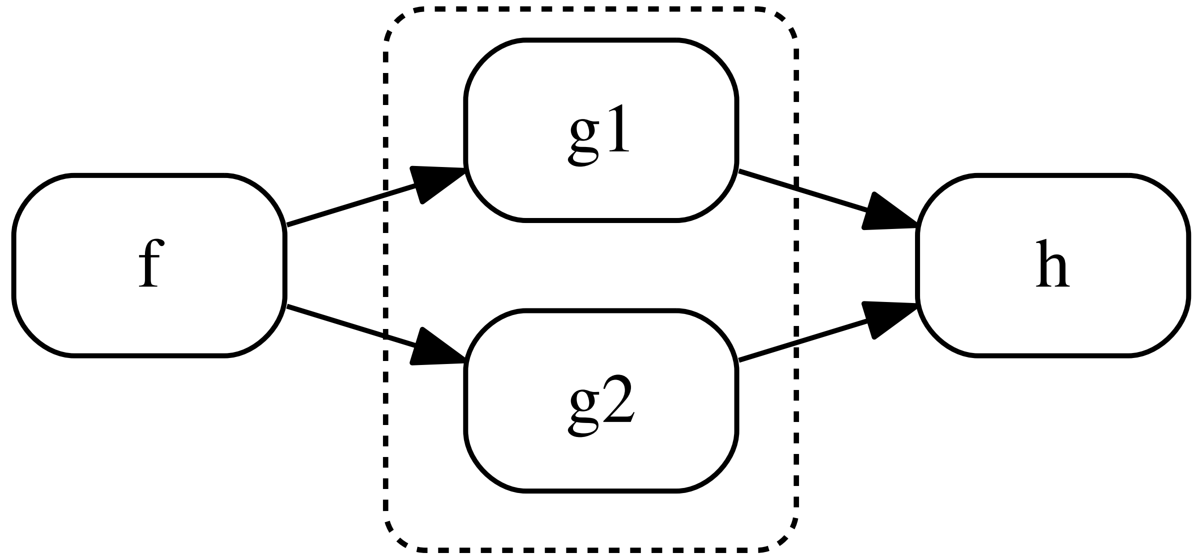



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions* : `u` and `d`

Example 4

Pushing abstraction a little bit further : HOWF

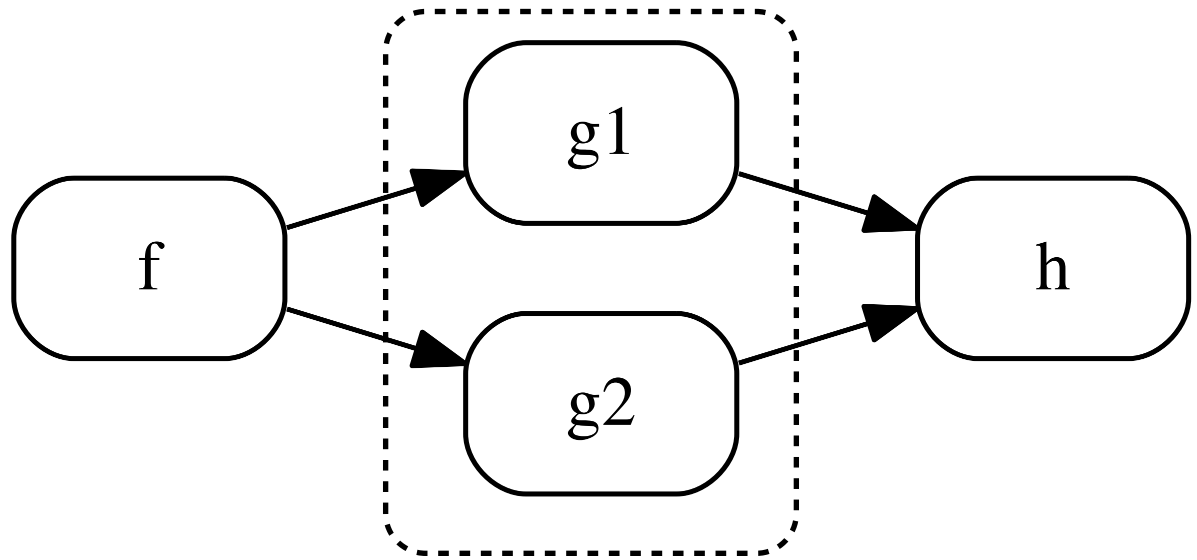



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions :* `u` and `d`
  - a pair of values `(x,y)`

Example 4

Pushing abstraction a little bit further : HOWF

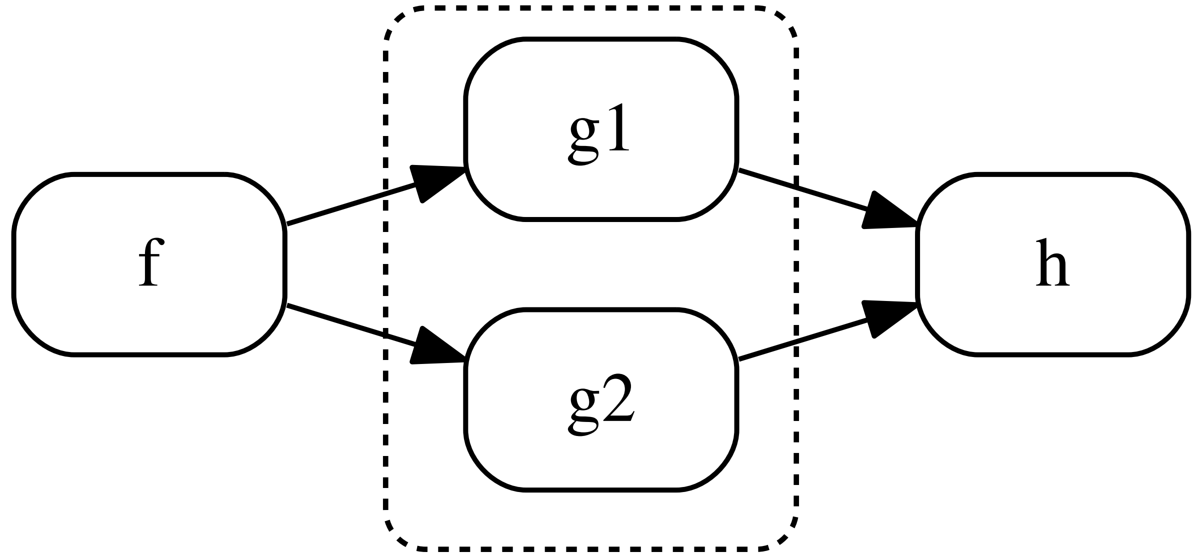



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions* : `u` and `d`
  - a pair of values `(x,y)`
- The main graph is obtained by *applying* `m` to `g1` and `g2`

Example 4

Pushing abstraction a little bit further : HOWF

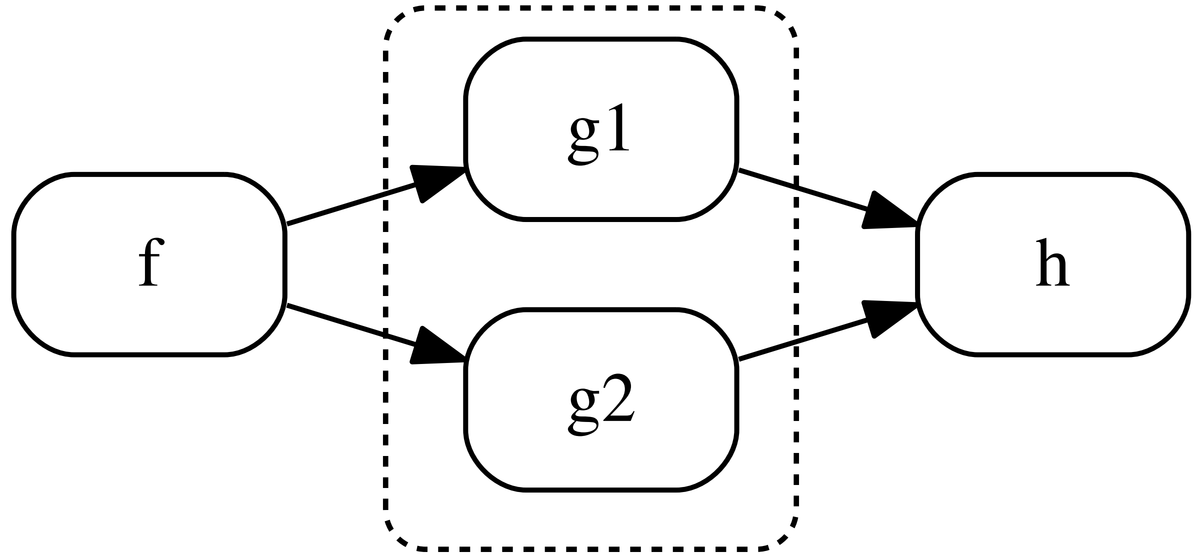



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions :* `u` and `d`
  - a pair of values `(x,y)`
- The main graph is obtained by *applying* `m` to `g1` and `g2`
  - again, application is denoted without parens (*curried* form)

Example 4

Pushing abstraction a little bit further : HOWF

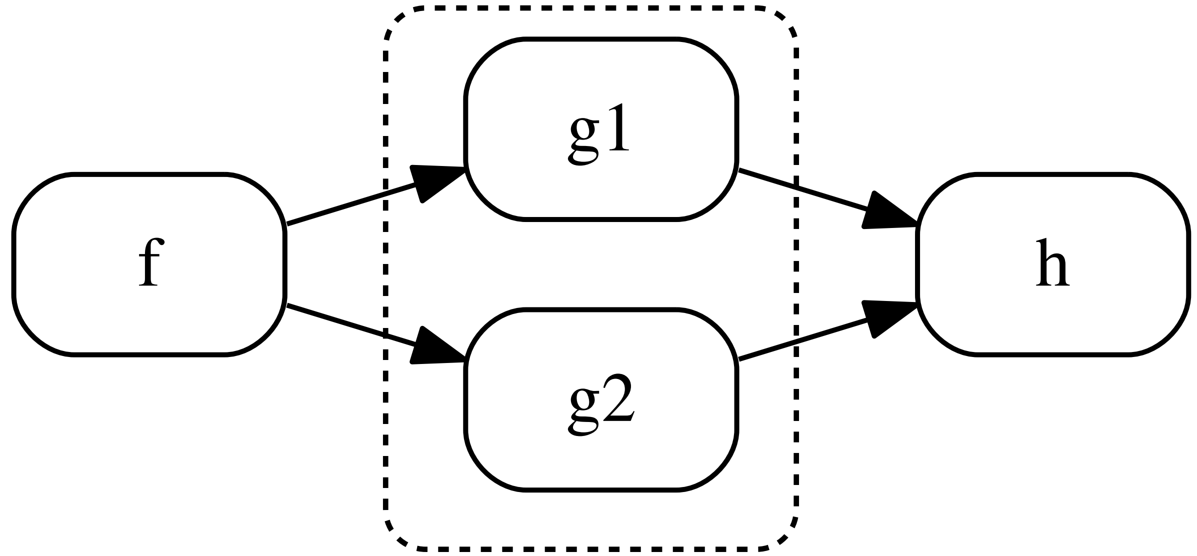



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions :* `u` and `d`
  - a pair of values `(x,y)`
- The main graph is obtained by *applying* `m` to `g1` and `g2`
  - again, application is denoted without parens (*curried* form)
- `m` is a *higher-order wiring function* (*HOWF)*

Example 4

Pushing abstraction a little bit further : HOWF

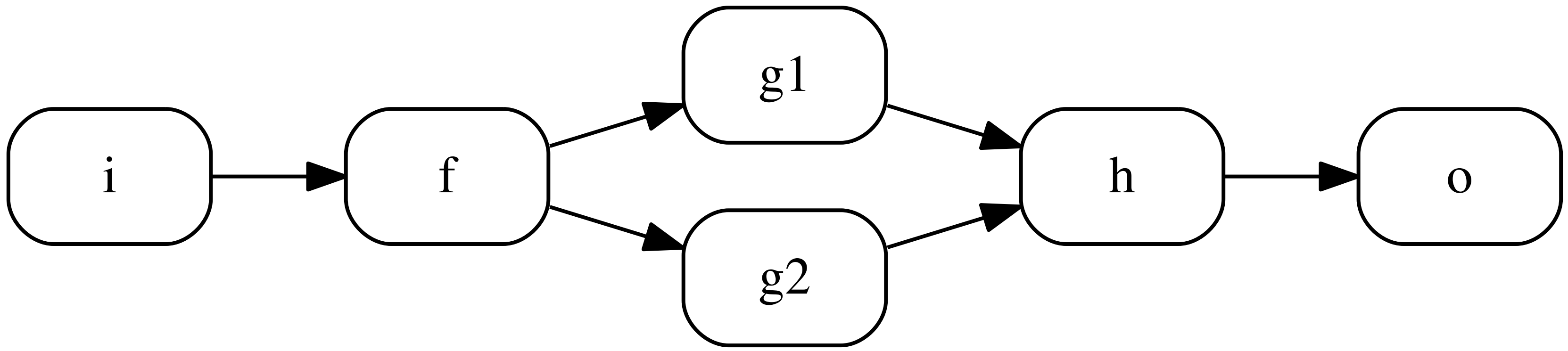



```
let m u d (x,y) = u x, d y;
let _ = f |> m g1 g2 >> h;
```

- The `m` function now takes three arguments :
  - two *functions :* `u` and `d`
  - a pair of values `(x,y)`
- The main graph is obtained by *applying* `m` to `g1` and `g2`
  - again, application is denoted without parens (*curried* form)
- `m` is a *higher-order wiring function (HOWF)*
  - HOWFs promote abstraction, i.e. the ability to encapsulate *graph patterns*

Example 5

HOWFs at work



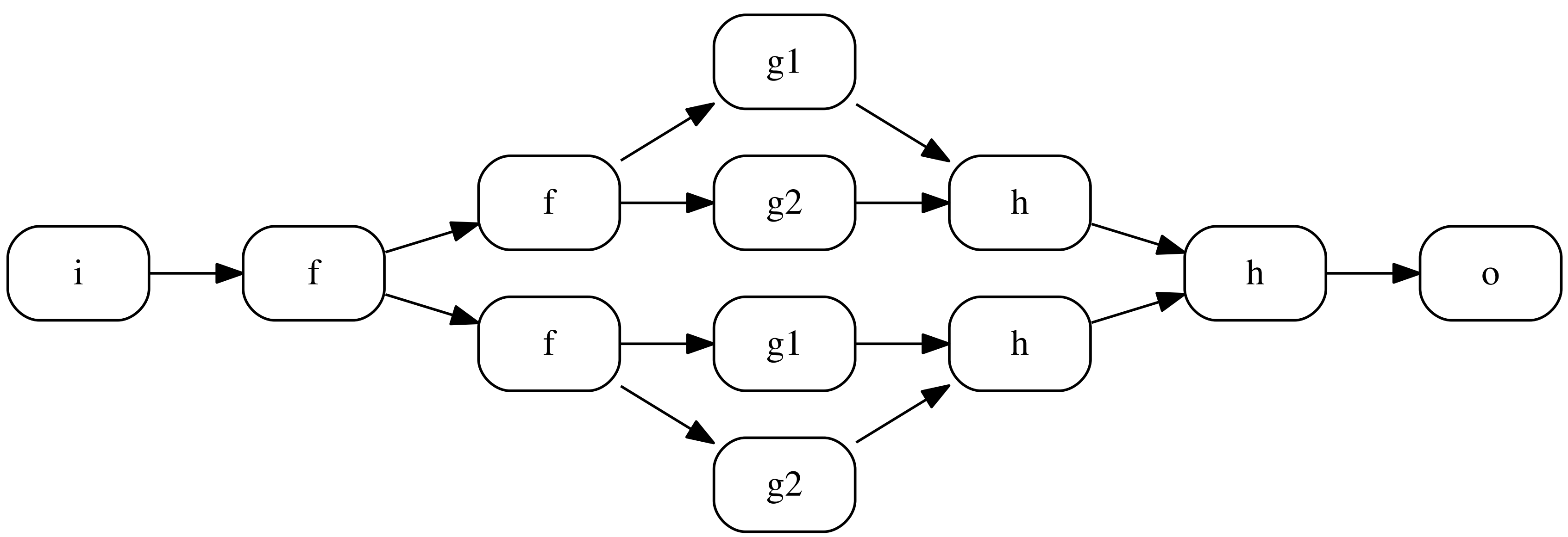

```
let diamond l u d r v =
  let (x,y) = l v in
  r (u x, d y);

let _ = i |> diamond f g1 g2 h >> o;
```

- A slight variation on the previous example
- The `diamond` *higher-order wiring function* uses a local let-definition

Example 5

HOWFs at work

```
let diamond l u d r v =
  let (x,y) = l v in
  r (u x, d y);

let inner = diamond f g1 g2 h;
let _ = i |> diamond f inner inner h >> o;
```

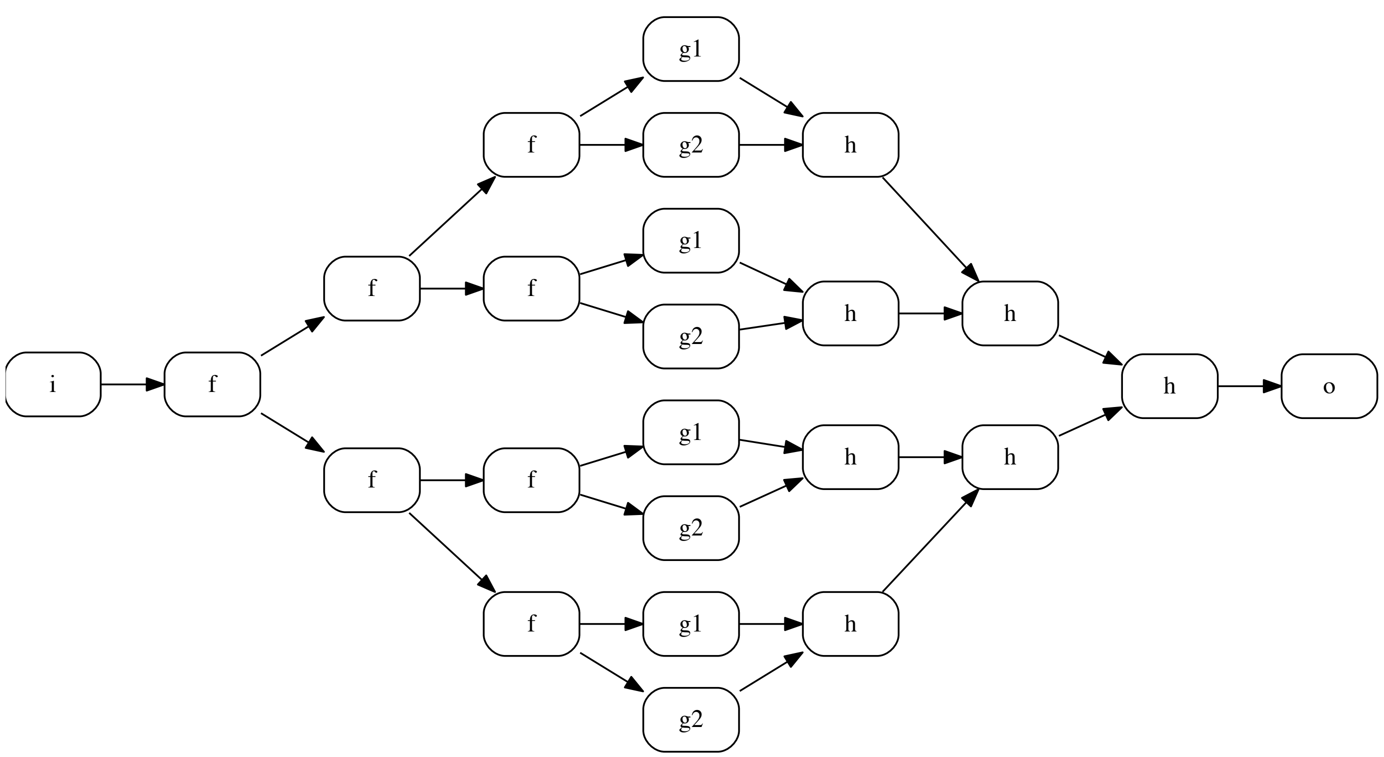


- The `diamond` *higher-order wiring function* is here instanciated at two levels

Example 5

HOWFs at work

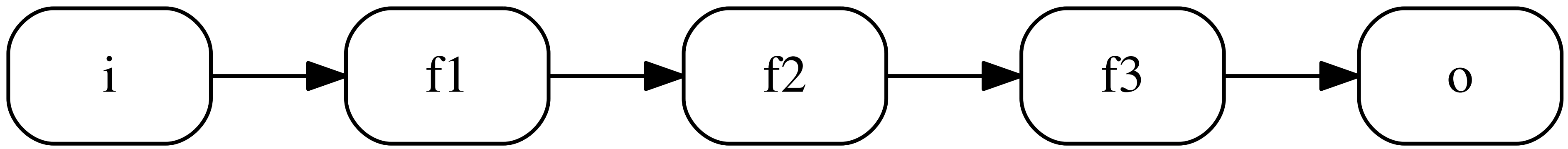
```
let diamond l u d r v =
   let (x,y) = l v in
   r (u x, d y);

let deeper = diamond f g1 g2 h;
let inner = diamond f deeper deeper h;
let _ = i |> diamond f inner inner h >> o;
```



- The `diamond` *higher-order wiring function* is here instanciated at three levels
- Describing *textually* such a graph would be very tedious

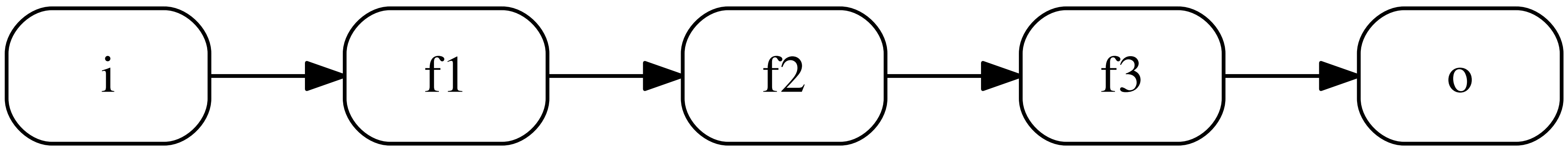
Classic patterns

Pipe



Classic patterns

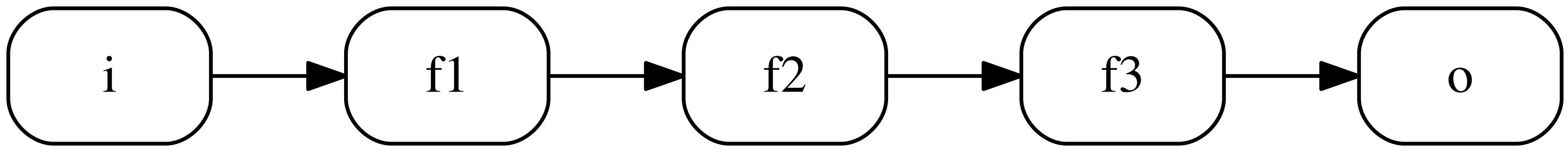
Pipe




```
let _ = i |> pipe [f1,f2,f3] >> o;
```

Classic patterns

Pipe




```
let _ = i |> pipe [f1,f2,f3] >> o;
```

- The `pipe` *higher-order wiring function* is defined in the standard library as :

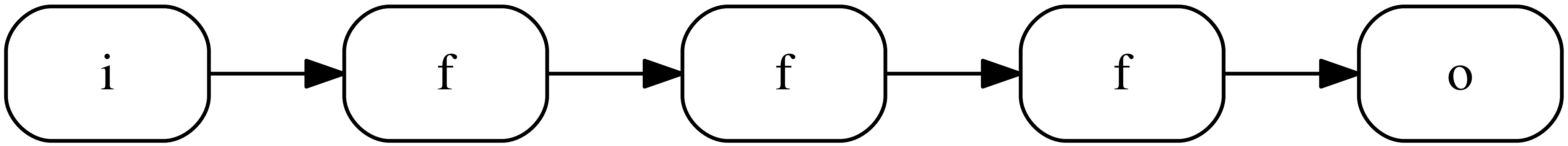
```
let rec pipe fs x = match fs with
      [] -> x
    | f::fs' -> pipe fs' (f x);
```

In other words :

$$\texttt{pipe } [f_1,f_2,\ldots f_n] \; x = f_n \; (\ldots \; f_2 \; (f_1 \; x)) \; \ldots)$$

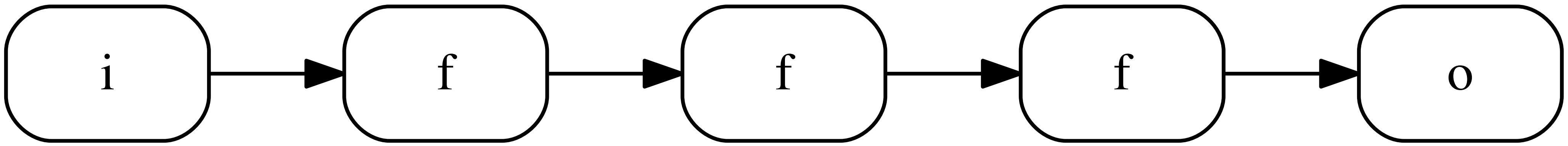
Classic patterns

Iterate



Classic patterns

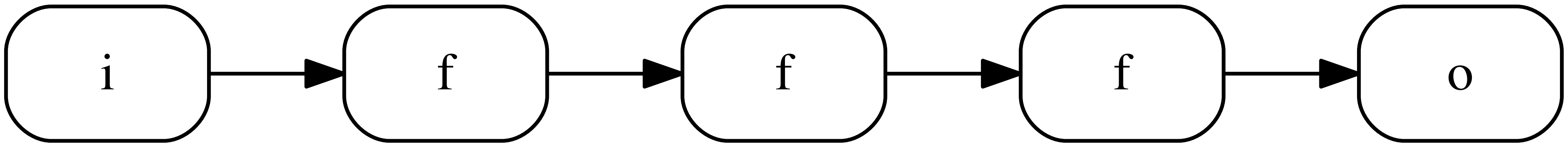
Iterate




```
let _ = i |> iter 3 f >> o;
```

Classic patterns

Iterate




```
let _ = i |> iter 3 f >> o;
```

- The `iter` *higher-order wiring function* is defined in the standard library as :

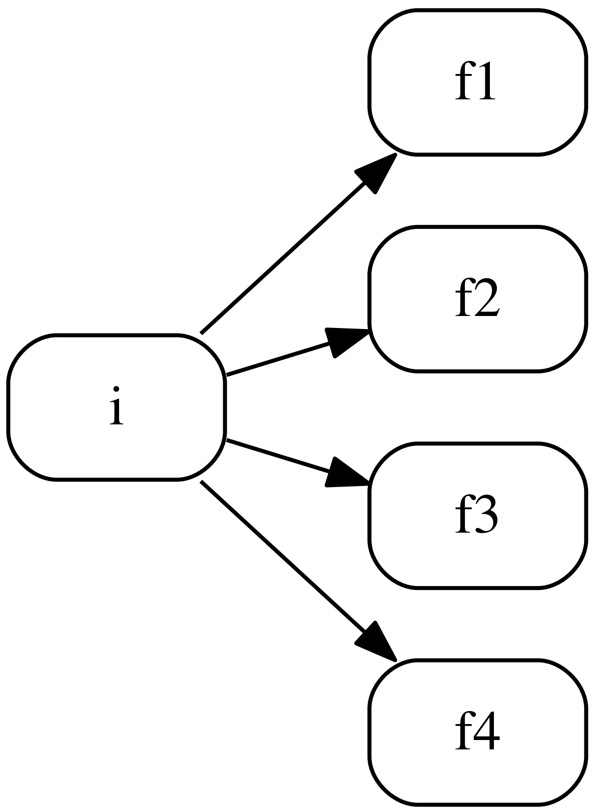
```
iter n f x = pipe (repl n f) x
```

where

$$\texttt{repl n x = }\underbrace{\texttt{[x, ..., x]}}_{n\ \textit{times}}$$

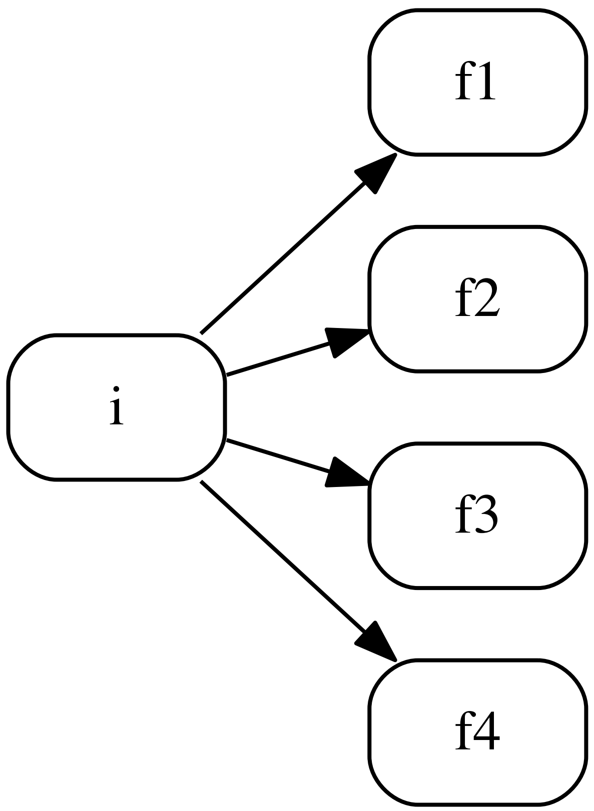
Classic patterns

Mapf



Classic patterns

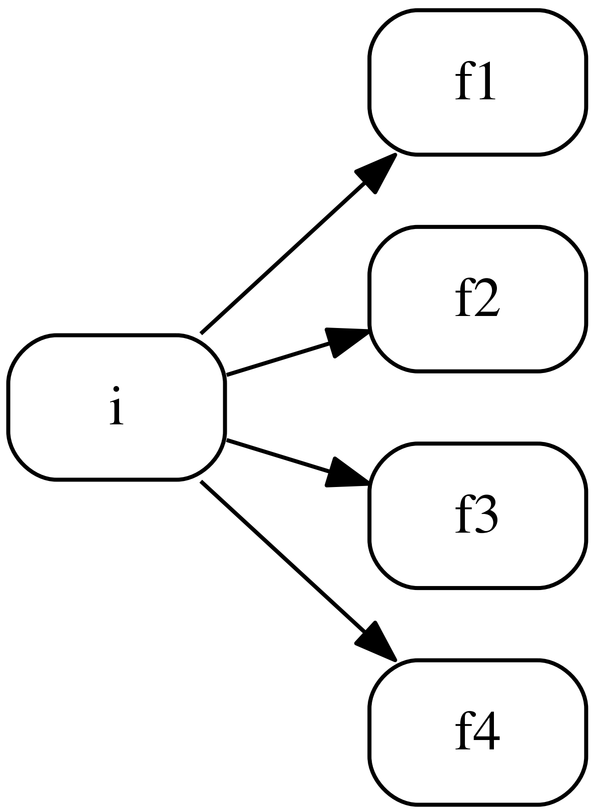
Mapf




```
let _ = i |> mapf [f1,f2,f3,f4];
```

Classic patterns

Mapf




```
let _ = i |> mapf [f1,f2,f3,f4];
```

- The `mapf` *higher-order wiring function* is defined in the standard library as :

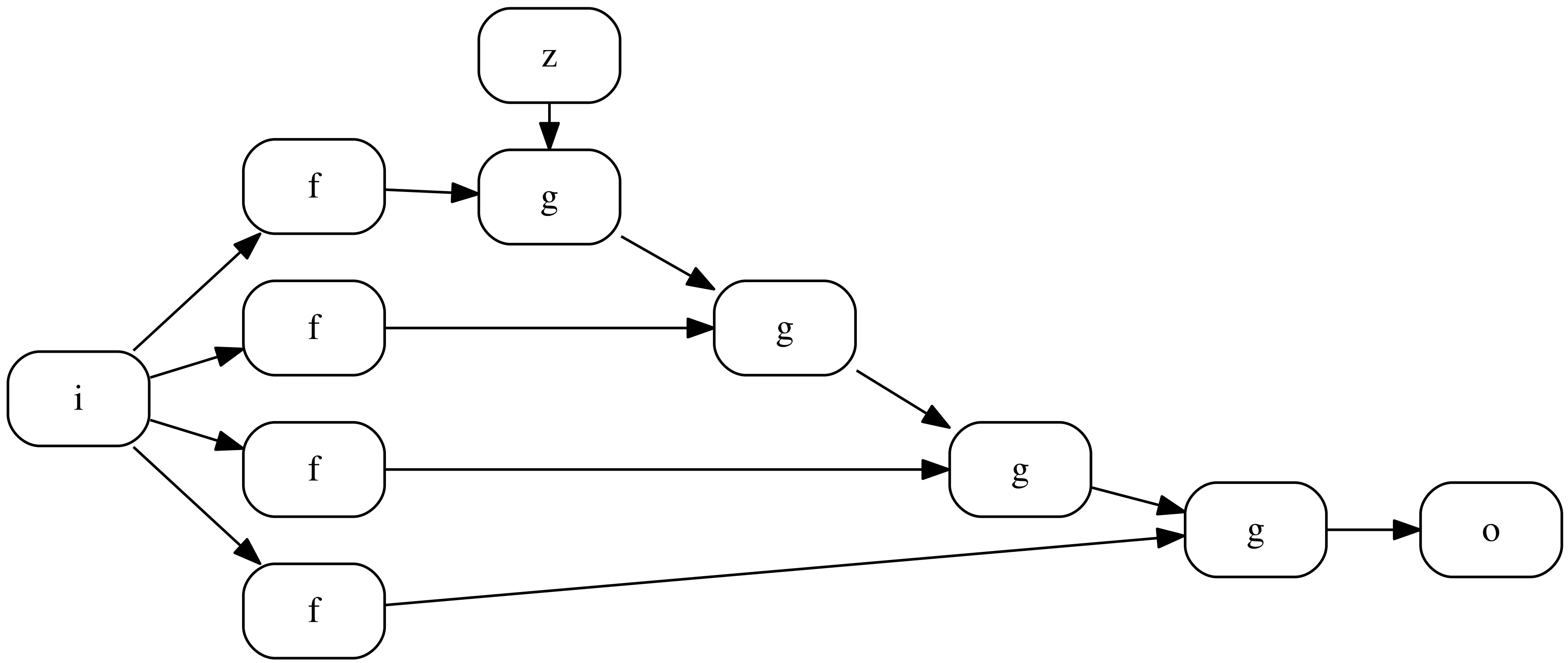
```
let rec mapf fs x = match fs with
    [] -> []
  | f::fs' ->  f x :: mapf fs' x;
```

In other words :

$\texttt{mapf [}f_1, f_2, \ldots f_n\texttt{] x = [}f_1\ \texttt{x}, \ldots, f_n\ \texttt{x]}$

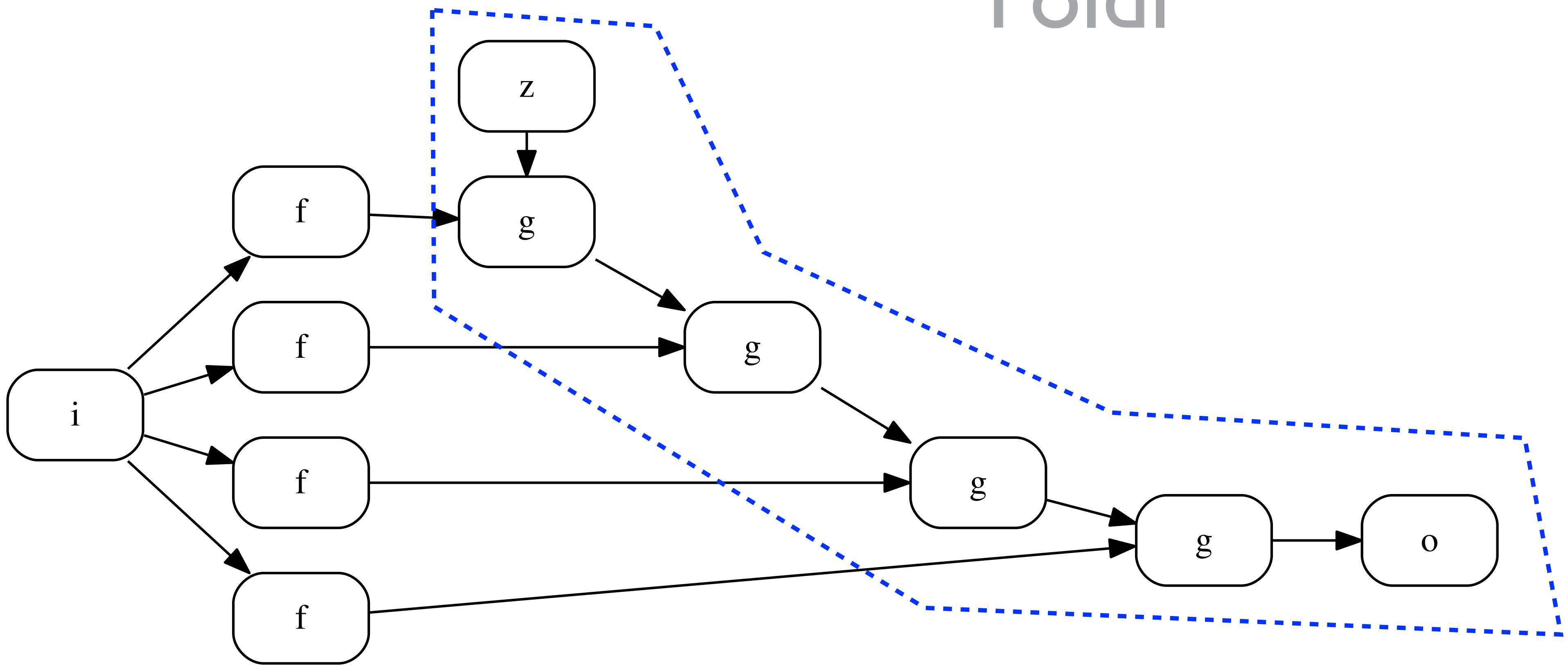
Classic patterns

Foldl



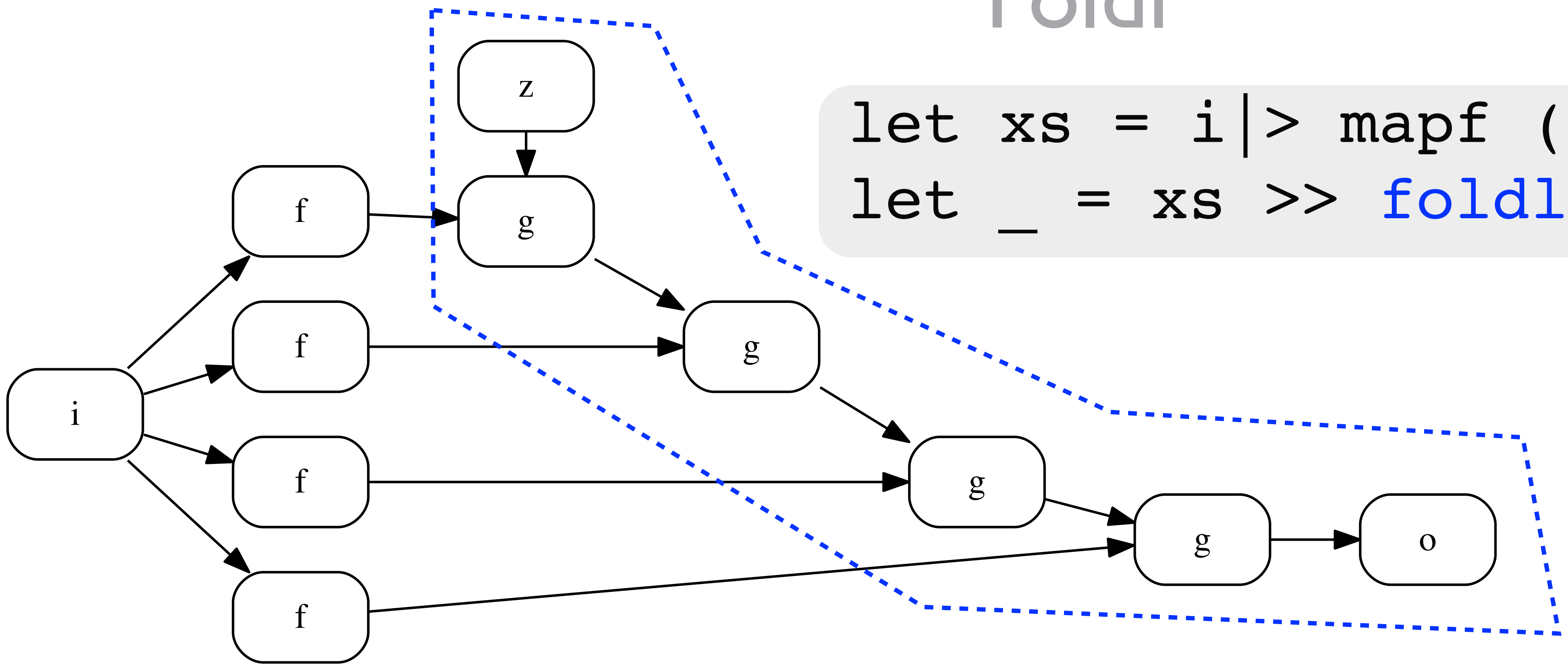
Classic patterns

Foldl



Classic patterns

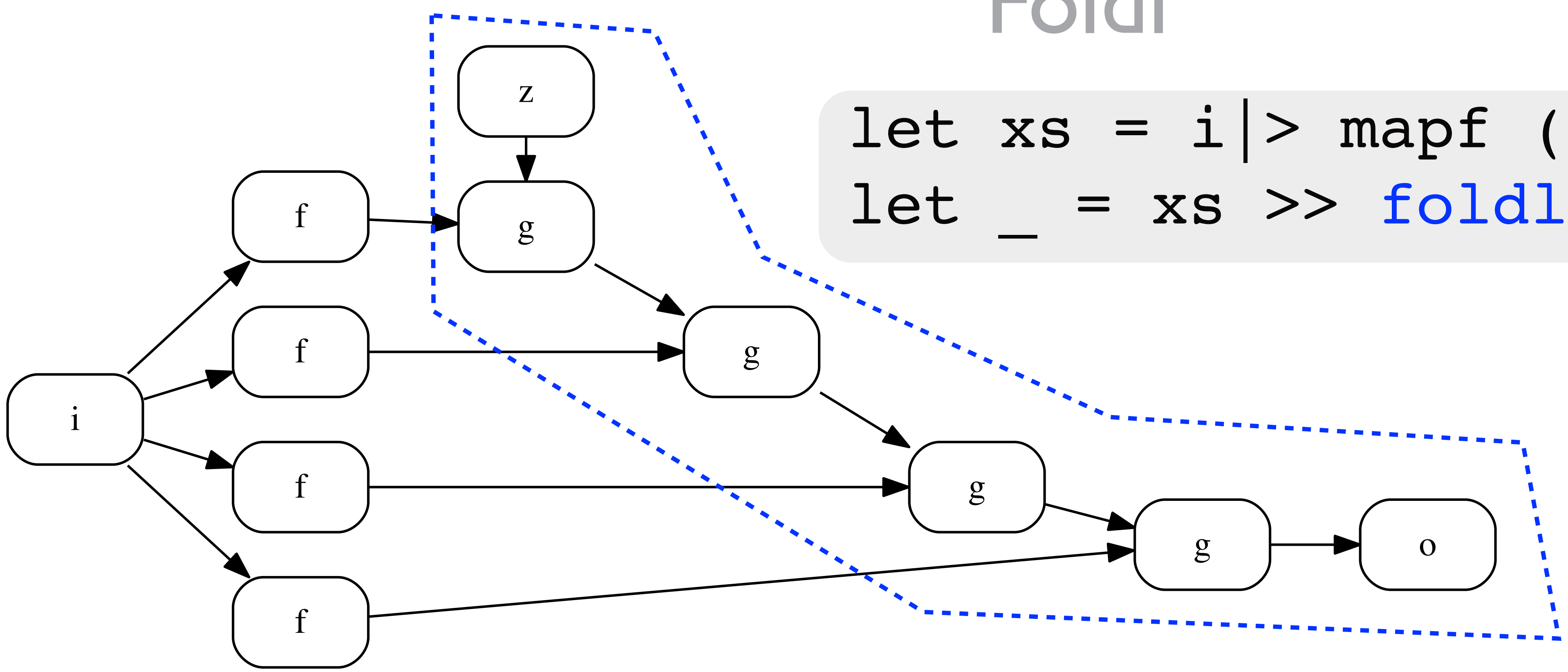
Foldl




```
let xs = i|> mapf (repl 4 f);
let _ = xs >> foldl g (z()) >> o;
```

Classic patterns

Foldl




```
let xs = i|> mapf (repl 4 f);
let _ = xs >> foldl g (z()) >> o;
```

- The `foldl` *higher-order wiring function* is used to describe *dyadic reduction* patterns. *It* is defined in the standard library as :

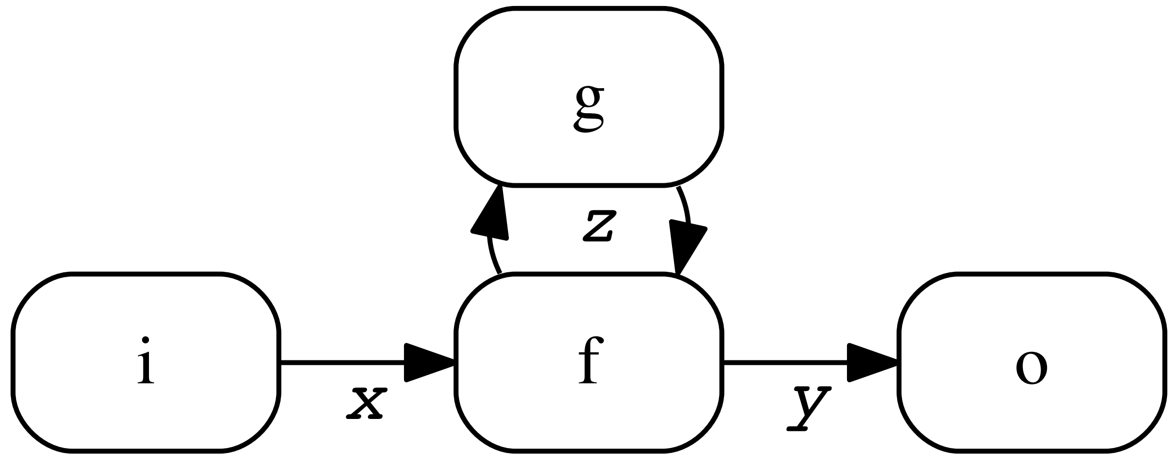
```
let rec foldl f z xs =
  match xs with
  [] -> z
| x::xs' -> foldl f (f (z,x)) xs';
```

In other words : `foldl f z [x₁, ..., xₙ]`

`= f (... (f (z,x₁), x₂), ..., xn)`

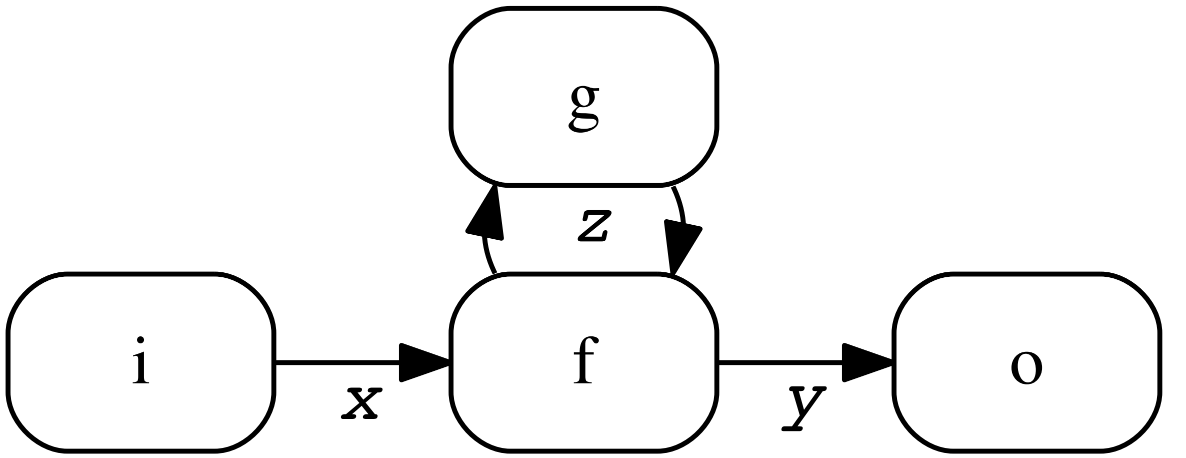
What about cycles ?

Recursive wiring



What about cycles ?

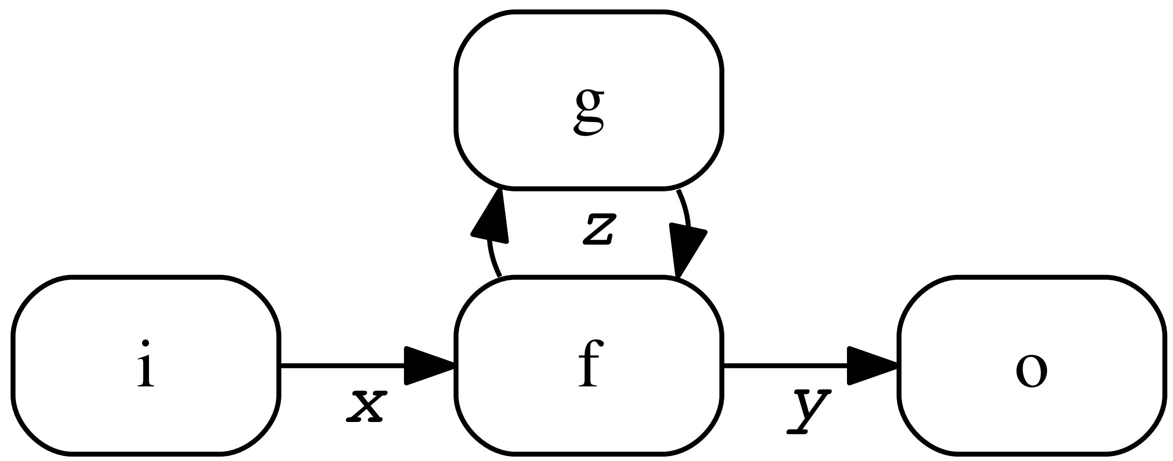
Recursive wiring




```
let main x =
  let rec (y,z) = f (x,g z) in
  y;

let _ = i |> main >> o;
```

What about cycles ?

Recursive wiring

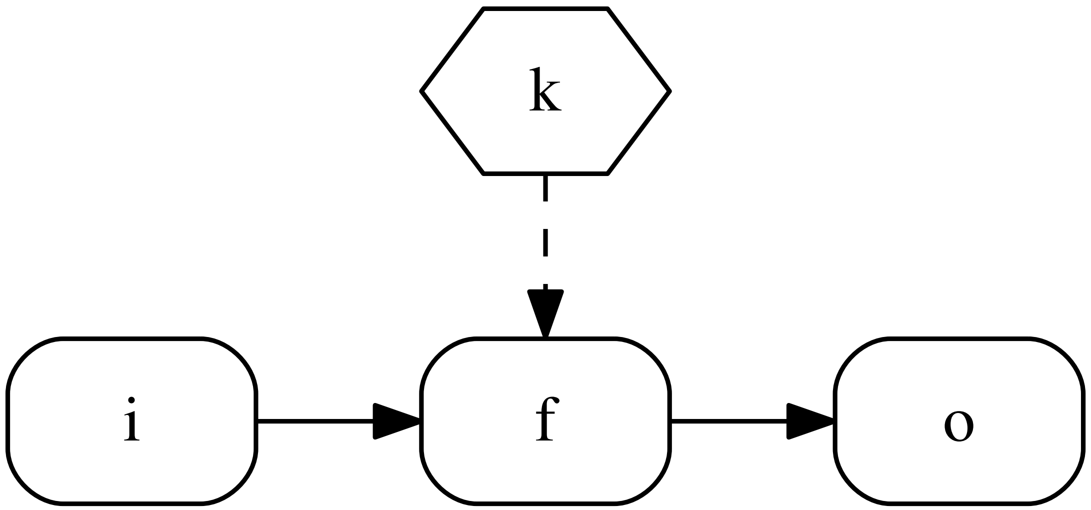



```
let main x =
  let rec (y,z) = f (x,g z) in
  y;

let _ = i |> main >> o;
```

- The recursive definition in `main` creates a loop in the graph since `z` is used both as an input and an output of the `f` function
- Because this definition is local, the recursion does not escape the scope of `main` here

Parameters

```
parameter k: nat = 2;

actor i in () out(o: t);
actor f param (k: nat)
        in (i: t) out (o: t);
actor o in (i: t) out ();

let _ = i |> f k >> o;
```

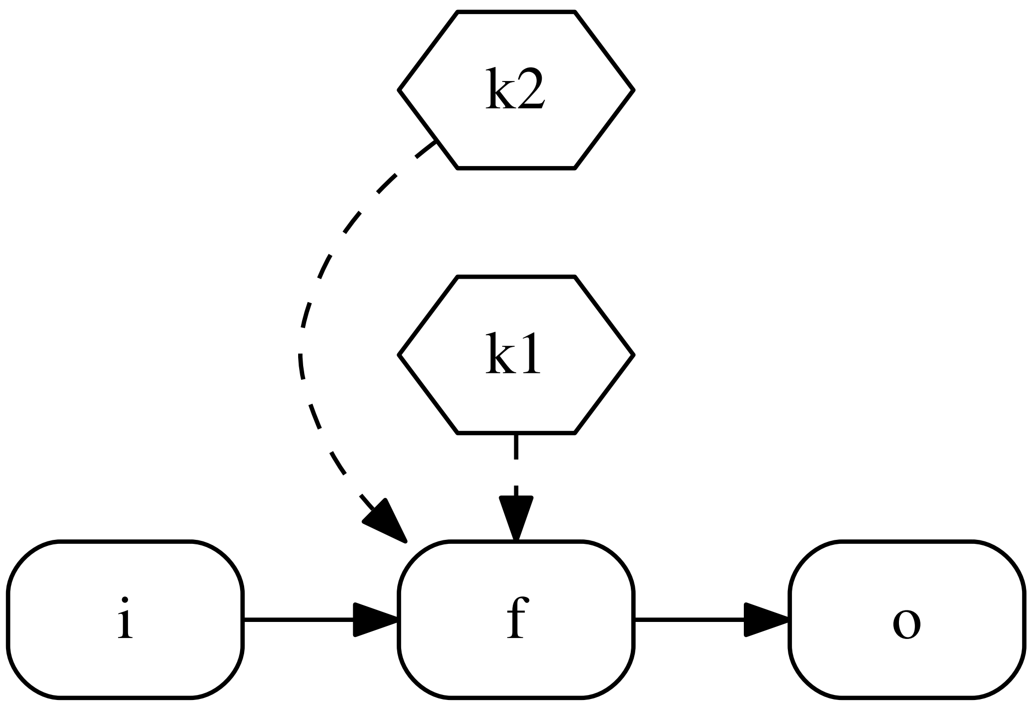


- Parameters are used to *configure* actor nodes
- They are viewed as an (extra) argument to the function representating the actor
  - e.g., function `f` has type : `nat → t → t`
  - the *partial application* `(f k)` gives the configured actor
- They must be given an initial value (2 here)
- Types of parameters are (now) limited to `nat` (natural numbers) and `bool`

Parameters

Multiple parameters

```
parameter k1: nat = 2;
parameter k2: nat = 3;

actor i in () out(o: t);
actor f param (k1: nat, k2: nat)
        in (i: t) out (o: t);
actor o in (i: t) out ();

let _ = i |> f (k1,k2) >> o;
```

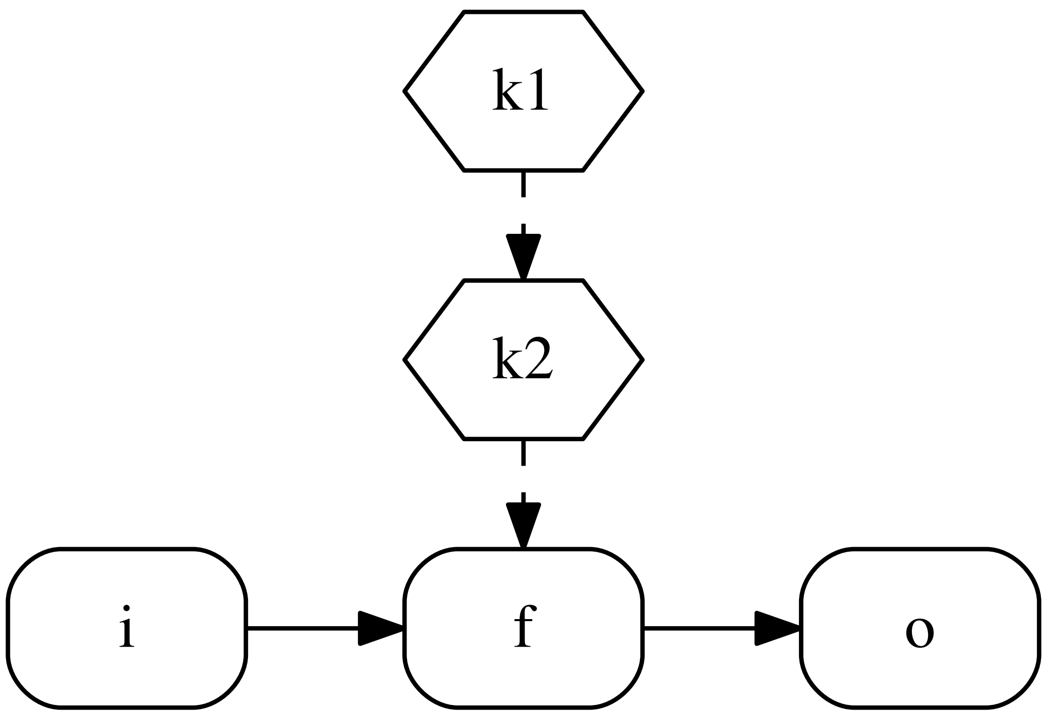


- When an actor takes several parameters, they are wrapped within a tuple
- The type of the actor is then : `(t₁ * ... tₙ) → t → t'` where
  - t_1, ..., t_n are the types of the parameters
  - `t` (resp. `t'`) is the type of the input (resp. output) data flow

Parameters

Dependent parameters

```
parameter k1: nat = 2;
parameter k2: nat = k1+1;

actor i in () out(o: t);
actor f param (k: nat)
        in (i: t) out (o: t);
actor o in (i: t) out ();

let _ = i |> f k2 >> o;
```



- Data dependencies between parameter values create a *tree* in graph
- This tree is "orthogonal" to the data flow
- Parameter dependencies are (now) resolved statically

Using the compiler

1. Generating and visualizing .dot graphs

```
type t;

actor i
  in ()
  out (o: t);
actor f
  in (i: t)
  out (o1: t, o2: t);
actor g
  in (i: t)
  out (o: t);
actor h
  in (i1: t, i2: t)
  out (o: t);
actor o
  in (i: t)
  out ();

let m (x,y) = h (g x, g y);
let _ = i |> f >> m >> o;
```

main.hcl

1. Generating and visualizing .dot graphs

```
type t;

actor i
  in ()
  out (o: t);
actor f
  in (i: t)
  out (o1: t, o2: t);
actor g
  in (i: t)
  out (o: t);
actor h
  in (i1: t, i2: t)
  out (o: t);
actor o
  in (i: t)
  out ();

let m (x,y) = h (g x, g y);
let _ = i |> f >> m >> o;
```

main.hcl

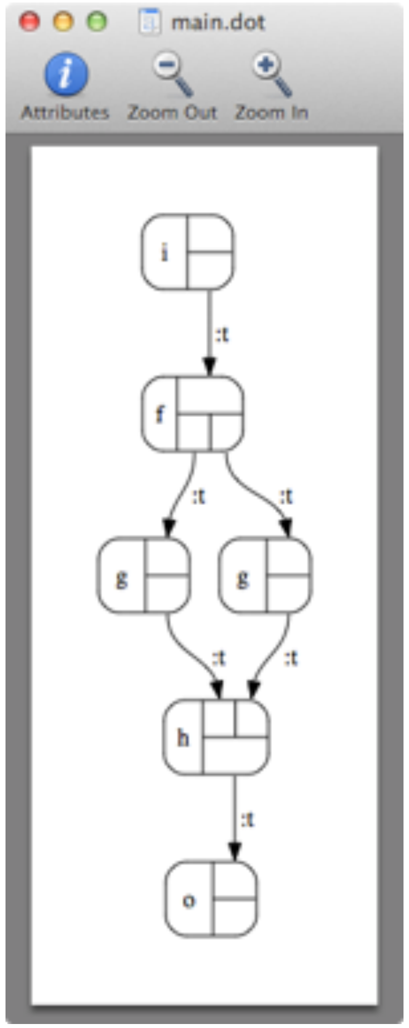
```
bash> hoclc -dot main.hcl
# Wrote file ./main.dot
bash> graphviz main.dot
```

1. Generating and visualizing .dot graphs

```
type t;

actor i
  in ()
  out (o: t);
actor f
  in (i: t)
  out (o1: t, o2: t);
actor g
  in (i: t)
  out (o: t);
actor h
  in (i1: t, i2: t)
  out (o: t);
actor o
  in (i: t)
  out ();

let m (x,y) = h (g x, g y);
let _ = i |> f >> m >> o;
```

main.hcl

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



2. Interfacing to PREESM

```
type t;

parameter k: nat = 100;
parameter p: nat = 2;

#pragma code("inp",
  "./include/input.h",
   "input", "inputInit")
#pragma code("outp",
  "./include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "./include/foo.h",
  "foo")

actor inp
 in () out(o: t "1");
actor foo
  param (k: nat, p: nat)
  in (i: t "p*2")
  out (o: t "p");
actor outp
  param (p: nat)
  in (i: t "p") out ();

let _ =
  inp |> foo (k,p) >> outp p;
```

main.hcl

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

foo.h

```
#include "preesm.h"
void inputInit(void);
void input(OUT int *o);
```

input.h

```
#include "preesm.h"
void output(IN int *i);
void outputInit(void);
```

output.h

2. Interfacing to PREESM

```
type t;

parameter k: nat = 100;
parameter p: nat = 2;

#pragma code("inp",
  "./include/input.h",
   "input", "inputInit")
#pragma code("outp",
  "./include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "./include/foo.h",
  "foo")

actor inp
 in () out(o: t "1");
actor foo
  param (k: nat, p: nat)
  in (i: t "p*2")
  out (o: t "p");
actor outp
  param (p: nat)
  in (i: t "p") out ();

let _ =
  inp |> foo (k,p) >> outp p;
```

main.hcl

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

foo.h

```
#include "preesm.h"
void inputInit(void);
void input(OUT int *o);
```

input.h

```
#include "preesm.h"
void output(IN int *i);
void outputInit(void);
```

output.h

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

2. Interfacing to PREESM

```
type t;

parameter k: nat = 100;
parameter p: nat = 2;

#pragma code("inp",
  "./include/input.h",
   "input", "inputInit")
#pragma code("outp",
  "./include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "./include/foo.h",
  "foo")

actor inp
 in () out(o: t "1");
actor foo
  param (k: nat, p: nat)
  in (i: t "p*2")
  out (o: t "p");
actor outp
  param (p: nat)
  in (i: t "p") out ();

let _ =
  inp |> foo (k,p) >> outp p;
```

main.hcl

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

foo.h

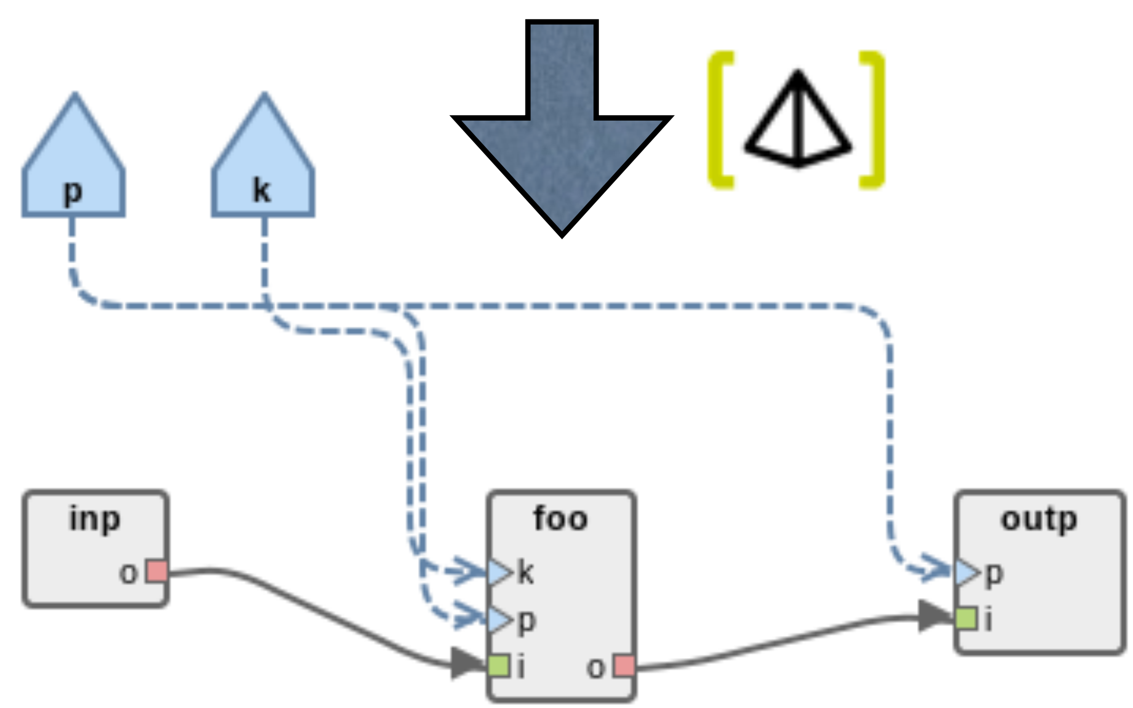
```
#include "preesm.h"
void inputInit(void);
void input(OUT int *o);
```

input.h

```
#include "preesm.h"
void output(IN int *i);
void outputInit(void);
```

output.h

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



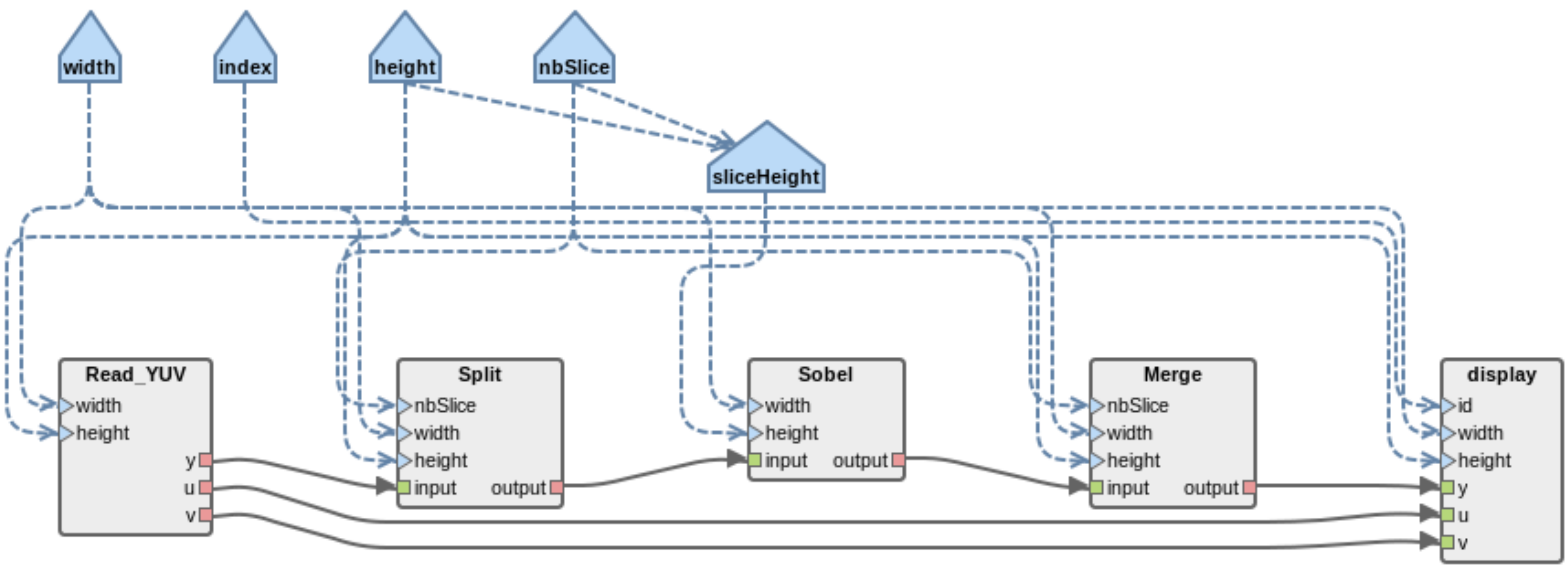
2. Interfacing to PREESM - A classical example

```
type uchar;

parameter width: nat = 352;
parameter height: nat = 288;
parameter index: nat = 0;
parameter nbSlice: nat = 8;
parameter sliceHeight: nat
  = (height/nbSlice)+2;

#pragma code("Read_YUV",
  "include/yuvRead.h",
  "readYUV",
   "initReadYUV")
#pragma code("Merge",
  "include/splitMerge.h",
  "merge")
#pragma code("Sobel",
  "include/sobel.h",
  "sobel")
#pragma code("Split",
  "include/splitMerge.h",
  "split")
#pragma code("display",
  "include/yuvDisplay.h",
  "yuvDisplay",
  "yuvDisplayInit")
```

```
actor Read_YUV
  param (width: nat, height: nat)
  in ()
  out (y: uchar "height*width",
       u: uchar "height/2*width/2",
       v: uchar "height/2*width/2")
;

...

actor Sobel
  param (width: nat, height: nat)
  in (input: uchar "height*width")
  out (output: uchar "height*width")
;
```

```
let (yi,u,v) = Read_YUV(width, height) ();
let yo = yi
       >> Split (nbSlice, width, height)
       >> Sobel (width, sliceHeight)
       >> Merge (nbSlice, width, height);
let _ = display
          (index, width, height)
          (yo,u,v);
```

2. Interfacing to PREESM - A classical example



https://github.com/jserot/hocl/tree/master/examples/working/preesm/sobel

3. Generating and running SystemC code

```
type int;

#pragma code("inp",
 "../code/include/input.h",
 "input", "inputInit")
#pragma code("outp",
  "../code/include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "../code/include/foo.h",
  "foo")

actor inp
 in () out(o: int);
actor foo
  in (i: int) out (o: int);
actor outp
  in (i: int) out ();

let _ = inp |> foo >> outp;
```

main.hcl

input.c

```
#include "input.h"
static int cnt = 0;
void inputInit(void) { cnt=0; }
void input(OUT int *o) { *o = cnt++; }
```

foo.c

```
#include "foo.h"
void foo(IN int *i, OUT int *o)
 { *o = *i * 2; }
```

output.c

```
#include "output.h"
void outputInit(void) { }
void output(IN int *i) {
  printf("output: got %d\n", *i);}
```

3. Generating and running SystemC code

```
type int;

#pragma code("inp",
 "../code/include/input.h",
 "input", "inputInit")
#pragma code("outp",
  "../code/include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "../code/include/foo.h",
  "foo")

actor inp
 in () out(o: int);
actor foo
  in (i: int) out (o: int);
actor outp
  in (i: int) out ();

let _ = inp |> foo >> outp;
```

main.hcl

input.c

```
#include "input.h"
static int cnt = 0;
void inputInit(void) { cnt=0; }
void input(OUT int *o) { *o = cnt++; }
```

foo.c

```
#include "foo.h"
void foo(IN int *i, OUT int *o)
 { *o = *i * 2; }
```

output.c

```
#include "output.h"
void outputInit(void) { }
void output(IN int *i) {
  printf("output: got %d\n", *i);}
```

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

3. Generating and running SystemC code

```
type int;

#pragma code("inp",
 "../code/include/input.h",
 "input", "inputInit")
#pragma code("outp",
  "../code/include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "../code/include/foo.h",
  "foo")

actor inp
 in () out(o: int);
actor foo
  in (i: int) out (o: int);
actor outp
  in (i: int) out ();

let _ = inp |> foo >> outp;
```

main.hcl

input.c

```
#include "input.h"
static int cnt = 0;
void inputInit(void) { cnt=0; }
void input(OUT int *o) { *o = cnt++; }
```

foo.c

```
#include "foo.h"
void foo(IN int *i, OUT int *o)
 { *o = *i * 2; }
```

output.c

```
#include "output.h"
void outputInit(void) { }
void output(IN int *i) {
  printf("output: got %d\n", *i);}
```

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

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

3. Generating and running SystemC code

```
type int;

#pragma code("inp",
 "../code/include/input.h",
 "input", "inputInit")
#pragma code("outp",
  "../code/include/output.h",
  "output", "outputInit")
#pragma code("foo",
  "../code/include/foo.h",
  "foo")

actor inp
 in () out(o: int);
actor foo
  in (i: int) out (o: int);
actor outp
  in (i: int) out ();

let _ = inp |> foo >> outp;
```

main.hcl

input.c

```
#include "input.h"
static int cnt = 0;
void inputInit(void) { cnt=0; }
void input(OUT int *o) { *o = cnt++; }
```

foo.c

```
#include "foo.h"
void foo(IN int *i, OUT int *o)
 { *o = *i * 2; }
```

output.c

```
#include "output.h"
void outputInit(void) { }
void output(IN int *i) {
  printf("output: got %d\n", *i);}
```

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

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

```
bash> ./main_sc
# output: got 0
# output: got 2
# output: got 4
# ...
```