HoCL High level specification of dataflow graphs

Jocelyn Sérot

HoCL
High level specification of dataflow graphs

Jocelyn Sérot

Institut Pascal, UMR 6602 U. Clermont-Auvergne / CNRS
IETR, UMR 6164 I. Rennes 1 / CNRS

GdR ISIS 2020-11-18

Introduction

Question 1

Q : What are the three most things in programming ?

A :

1. abstraction
2. abstraction
3. abstraction

Question 2

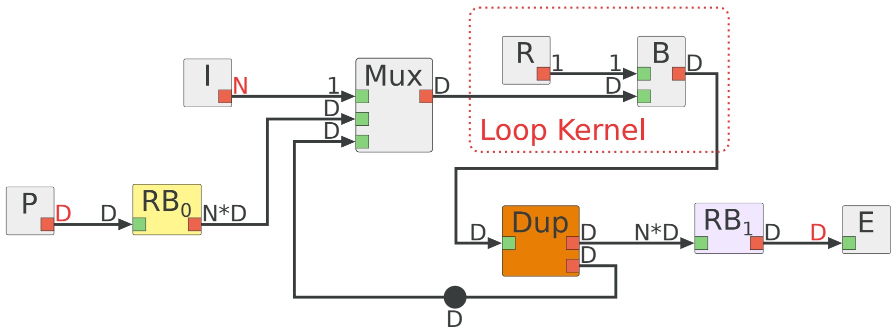
Q : Do dataflow models promote abstraction ?

A : Well, it depends…

EXAMPLE 1

Dataflow formulation of an iterative algorithm in Preesm

https://preesm.github.io

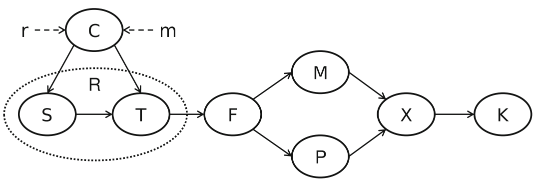



```
P();
for i=0 to N do B() done;
E();
```

Source : *Extensions and Analysis of Dataflow Models of Computation for Embedded Runtimes.*. PhD thesis. F. Arrestier, 2020.

EXAMPLE 2

Dataflow formulation of an RMOD application in DIF (Dataflow Interchange format)




```
CFDF RMOD {
  topology {
    nodes = C, S, T, F, M, P, X, K;
    edges = e1(C, S), e2(C, T), e3(S, T), e4(T, F),
      e5(F, M), e6(F, P), e7(M, X), e8(P, X), e9(X, K);
  }
  actor C {
    name = "mod_ctrl";
    out_r = e1; out_m = e2; /* Assign edges to ports */
  }
  actor S {
    name = "mod_src";
    in_ctrl = e1; out_data = e3;
    mode_count = 3;
  }
  actor T {
    name = "mod_lut";
    in_ctrl = e1; in_bits = e3; out_symbol = e4;
    mode_count = 4;
  }
  /* Other actor definitions */
  /* ... */
}
```

Source : *The DSPCAD Framework for Modeling and Synthesis of Signal Processing Systems.* Shuoxin Lin, Yanzhou Liu, Kyunghun Lee, Lin Li, William Plishker, and Shuvra S. Bhattacharyya, 2017.

Motivations

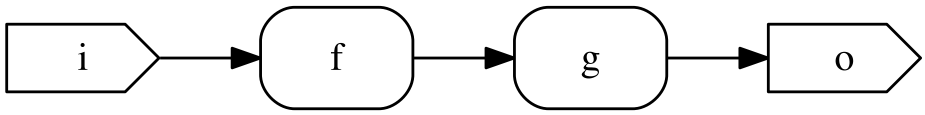
- Simplify the specification of large and complex dataflow graphs
- Independently of the underlying dataflow model of computation
  - pure coordination language (..CL = *Coordination Language*)
- Support for hierarchical and parameterized graphs
- Independently of the target implementation platform (software, hardware, mixed, ...)
- Support for mixed-style descriptions (structural or functional)

This presentation

- Informal presentation of the language by means of small examples
- Technical details such as typing, semantics, *etc.* deliberately omitted
  - https://github.com/jserot/hocl

Core features

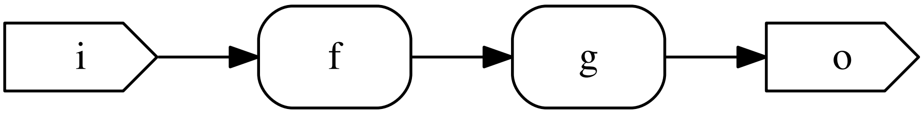
Example 1




```
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

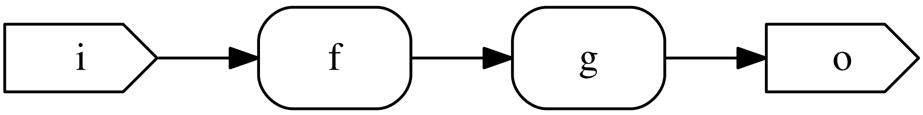



```
node f
  in (i: int)
 out(o: int);

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
- An actor with *m* inputs $e_1{:}t_1, \ldots, e_m{:}t_m$ and *n* inputs $s_1{:}t'_1, \ldots, s_n{:}t'_m$ is interpreted as a (curried) function of type

$$e_1{:}t_1 \rightarrow \ldots \rightarrow e_m{:}t_m \rightarrow t'_1 * \ldots * t'_m$$

Example 1



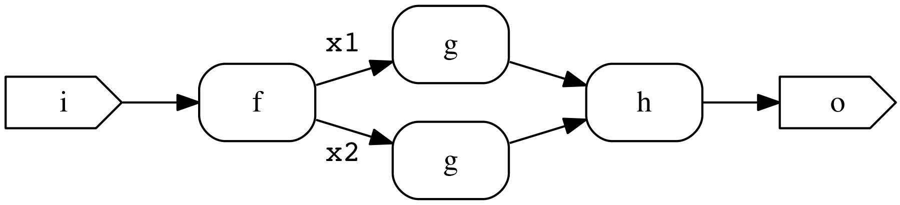

```
node f
  in (i: int)
 out(o: int);

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

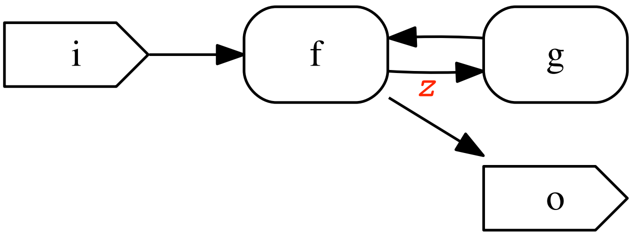
A slightly more complex graph




```
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);
```

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

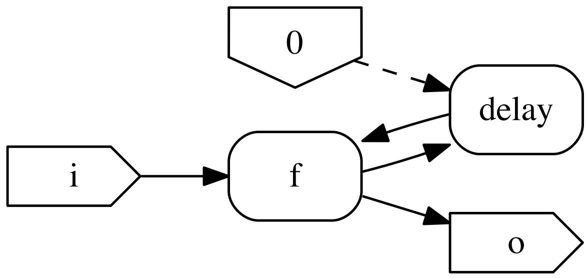
Cycles and recursive wiring




```
node f in (i1: t1, i2: t2) out (o1: t4, o2: t3);
node g in (i: t3) out (o: t2);
```

```
graph top
  in (i: t1) out (o: t4)
fun
  val rec (o,z) = f i (g z)
end;
```

Delayed cycles



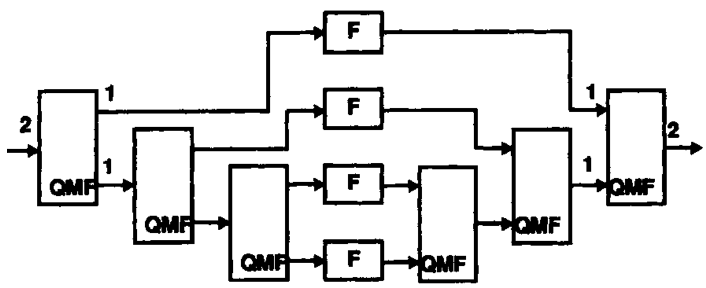

```
graph top
  in (i: int) out (o: int)
fun
 val rec (o,z) = f (i, delay ‘0’ z)
end;
```

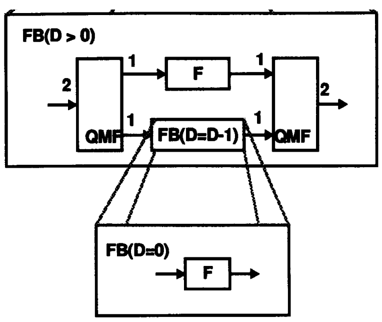
```
graph top
  in (i: int) out (o: int)
struct
  wire w1, w2: int
  box n1: f(i,w1)(o,w2)
  box n2: delay(‘0’,w2)(w1)
end;
```

- 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

Recursive graphs

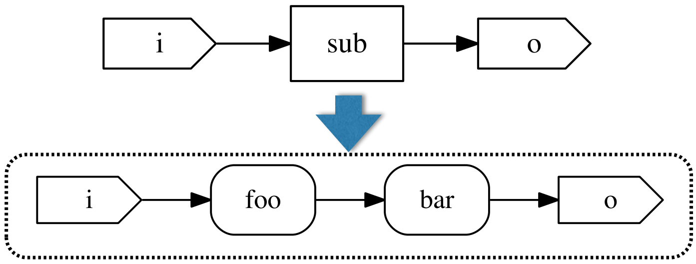
Example (from [Lee and Parks, 1995])






```
node f
   in (i: t)
  out (o: t);
node qmf
   in (i: t)
  out (o1: t, o2: t);

graph top
    in (i: t)
   out (o: t)
fun
  val rec fb d x =
    if d = 0 then f x
    else
     let x1,x2 = qmf x in
     qmf (f x1)
         (fb (d-1) x2)
   val o = fb 3 i
end;
```

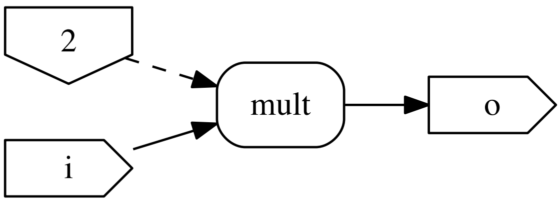
Hierarchical graphs




```
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

Parameters



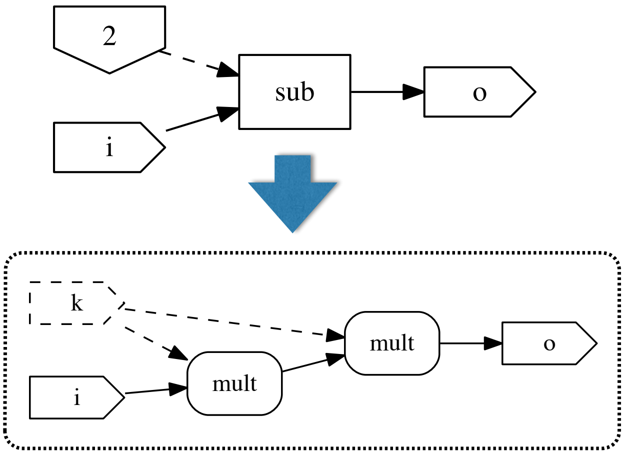
- Parameters are used to *configure* (*specialize*) nodes
- Parameters are distinguished from data by their type :
  - `t param` is the type of a parameter having itself type `t`
- In functional descriptions, this allows specifying their value using partial application of the corresponding function

```
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;
```

Parameter passing

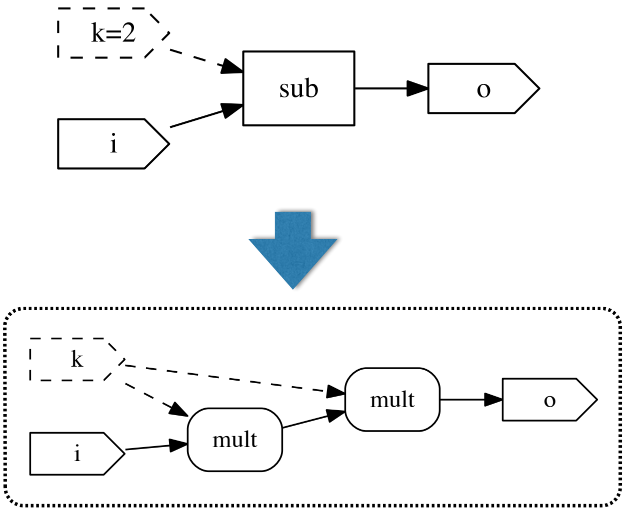
- Parameters can be passed from one hierarchy level to a nested one

```
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;
```



Parameter passing

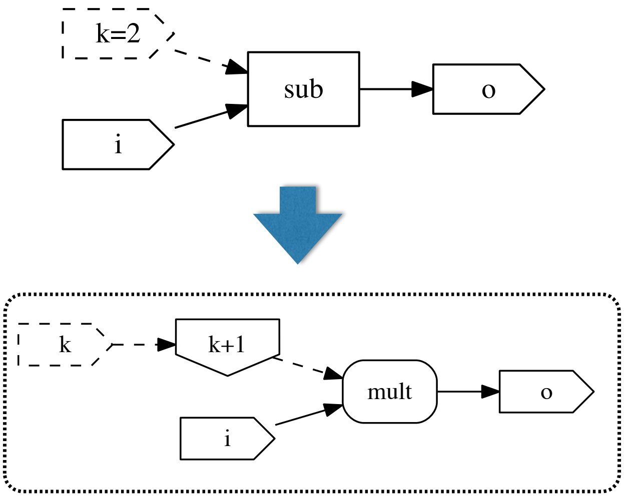
```
node sub
    in (k: int param, i: int)
  out (o: int)
fun
  val o =
    i |> mult k |> mult k
end;

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;
```

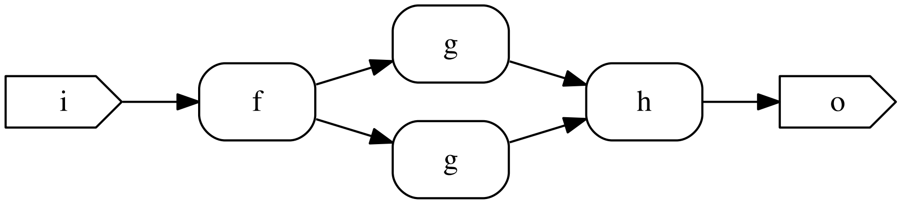


- The value of some parameters can depend on that of other parameters, defined at the same or at higher level(s) in the graph hierarchy
- Dependencies between parameter values create a *tree* in graph, which is “orthogonal” to the data flow

Higher order features

Ho..

Wiring functions

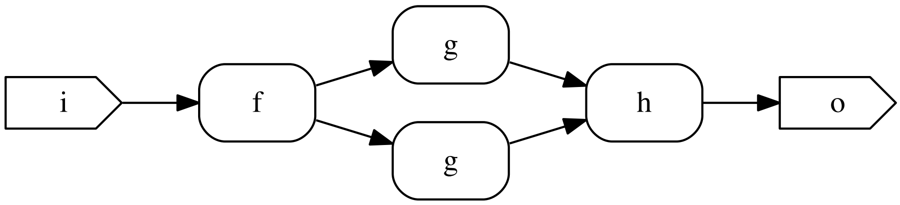


Another formulation :

```
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

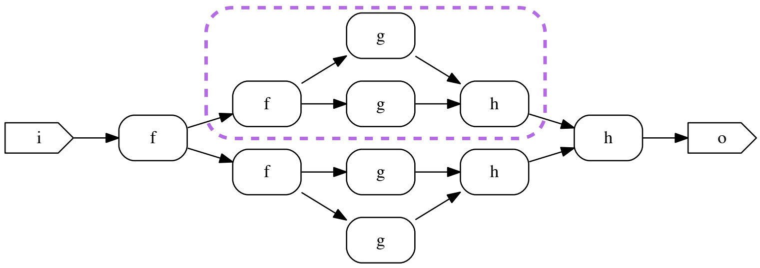


Pushing the abstraction a bit further :

```
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



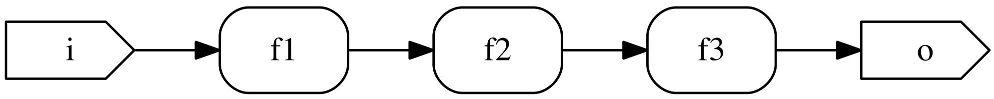

```
graph top
  in (i: int) out (o: int)
fun
  val diamond l m r x = …
  val sub = diamond f g h
  val o = diamond f sub h i
end;
```

```
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
- Example :



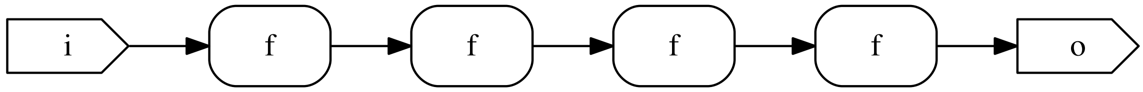

```
graph top
  in (i: int)
 out (o: int)
fun
  val o = i |> pipe [f1;f2;f3]
end;
```

where :

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

« Classic » higher order wiring functions

- Many recurrent graph patterns can be encapsulated using higher-order wiring functions
- Example :



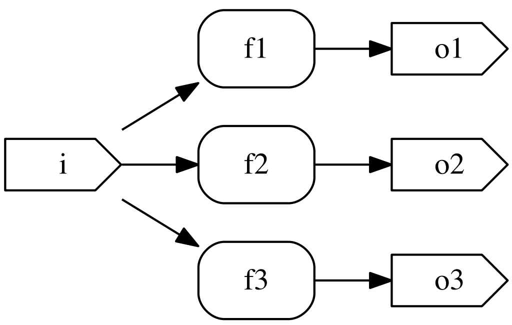

```
graph top
  in (i: int)
 out (o: int)
fun
  val o = i |> iter 4 f
end;
```

where :

```
val rec iter n f x =
  if n = 0 then x
  else iter (n-1) f (f x)
```

« Classic » higher order wiring functions

- Many recurrent graph patterns can be encapsulated using higher-order wiring functions
- Example :




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

where :

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

Higher order wiring functions

- Higher order wiring functions
  - promote abstraction
  - allow common graph patterns to be encapsulated for reuse
- In HoCL, they are defined within the language itself
  - the set of available reusable patterns can therefore be freely extended to suit the application domain
  - this is in contrast with existing dataflow-based design tools in which similar abstraction mechanisms rely on a predefined and fixes set of patterns

In practice

Implementation

- Prototype compiler written in OCaml
- Based upon a fully formalized static semantics (natural style)
- Source code available on github (`jserot/hocl`)
- Two versions
  - a command line compiler
  - a toplevel interpreter
- The CL compiler currently has four backends
  - a .dot backend (for visualizing the DFGs)
  - a DIF backend (for interfacing to DF-based analysis tools)
  - a Preesm backend (for generating code on heterogeneous many-core embedded platforms)
  - a SystemC backend (for simulation under the DDF and SDF MoCs)

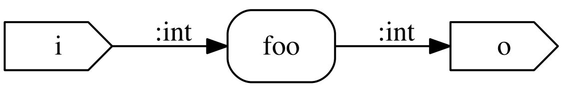
Example : using the SystemC backend

- Used to simulate the described DFGs
- 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

Conclusion

- Another attempt to bring the benefits of functional programming outside its « classical » circle
  - programmers in the DSP field are *not* familiar with concepts such as polymorphic typing and higher order functions
- Drawing of previous experience in a similar context with the CAPH project (http://dream.ispr-ip.fr/CAPH)
  - provide interfaces to existing, already used, tools
  - demonstrate practical benefits wrt. this tools
  - introduce disruptive concepts only if it serves a well identified goal

Conclusion

- Work in progress
  - injection of MoC-specific features into specifications
  - design of large scale DSP applications with HoCL for assessing gains if programmer's productivity

Thanks for your (remote) attention