A gentle introduction to functional graph description

J. Sérot

Abstract

This document is short, informal introduction to the basic concepts underpinning the HoCL language. It explains how the notions of function and types, as used in *functional programming languages* such OCaml and Haskell for example, can be applied to describe *dataflow graphs* in an abstract and concise manner.

The intended audience mainly includes programmers dealing with *dataflow models* and with little or no experience with functional programming languages.

1 Introduction

A *dataflow graph* (DFG) is a graph $\langle \mathrm{V}, \mathrm{E} \rangle$ where V a set of *nodes* and E a set of *edges*. A node is either an *actor*, representing a computation operating on input data streams and generating output data streams or another DFG (in other words, our definition includes hierarchical DFGs). Edges represent channels conveying data streams. Each channel connect an output port of a *source* node to an input port of a *destination* node.

A very simple DFG is depicted in Fig. 1. Nodes are here drawn as boxes and smaller, arrow-shaped boxes represent global input and output ports (*i.e.* ports through which data enter and leave the DFG). Nodes `g` and `h` both have one input and one output port. Node `f` has one input port and two output ports.

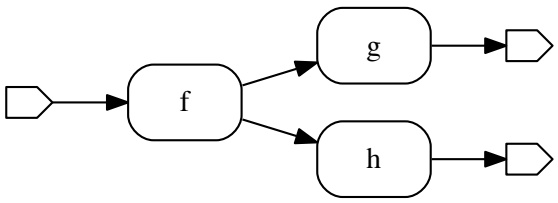


Figure 1: A simple dataflow graph

We are interested in a purely *textual* description of such a graph. Textual descriptions, though less intuitive, have numerous advantages : they can be easily communicated, versioned, stored in libraries, support syntax based analysis and transformations, *etc.*.

A possible solution is to adopt a *structural* approach, in which the graph is simply described by listing its nodes and edges. For example, listing 1 gives a structural description of the graph depicted in Fig. 1 using the corresponding language subset in HoCL. Edges are introduced by `wire` declarations[1] and nodes by `box` declarations. A `box` declaration specifies the edges connected to each input and output port of the corresponding node.

[1] Where `t` here denotes the type of the carried data, an attribute irrelevant here.

```
graph top in (i: t) out (o1: t, o2: t)
struct
  wire w1, w2: t
  box b1: f(i)(w1,w2)
  box b3: h(w2)(o2)
  box b2: g(w1)(o1)
end;
```

Listing 1: A structural description of the DFG in Fig. 1

Structural descriptions are easy to generate or parse. They are therefore widely used as intermediate representations in DFG-based modeling systems and tools[2]. But, when used as a *specification language*, they are very cumbersome. Describing this way a graph with more than a few nodes and edges is both tedious and error-prone, even with the help of dedicated GUIs, with which nodes and edges are "drawn" on a canvas instead of being declared in a text file.

The main HoCL offers – and will be described here – is an another way of describing dataflow graphs, using concepts and techniques drawn from *functional programming languages* (FPLs).

2 The very first steps

As an introductory example, let's consider the minimal DFG depicted in Fig. 2.

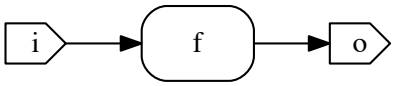


Figure 2: A minimal dataflow graph

In this graph, the presence of node `f` clearly establishes a *dependency* between input `i` and output `o`. This dependency is similar to that existing between the argument of a (mathematical) function and its result. Hence the idea of viewing node `f` as a function and describing the graph of Fig. 2 by the following equation :

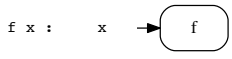
$$o = f(i)$$

Using a syntax closer to that of functional programming languages, we will describe this graph with the following declaration :

```
val o = f i
```

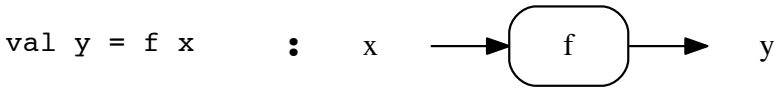
The `val` keyword here introduces a *definition*. This definition *binds* output `o` to the result of applying function `f` to input `i`. Note that, following a convention largely used in FPLs, function application is here denoted without brackets. This notation, which may look surprising at first sight, has several advantages which will be described further.

The general idea is that if `f` is a function taking an argument of type τ and `x` a value of type τ, then `f x` denotes the DFG obtained by creating a node representing function `f` and connecting to its input port the wire representing the value `x`, a principle which can be depicted as follows :



[2]With ad-hoc extensions for attaching model or application-specific *attributes* to nodes and edges.

If we want to use the output of the node, we must name ("bind") the result of applying the corresponding function. This is exactly what is meant by a `val` declaration :



Depending on the context, the values bound by `val` declarations will represent either edges (like above) or graph outputs (like in Fig. 2).

3 More interesting graphs

Consider the graph depicted in Fig. 3.

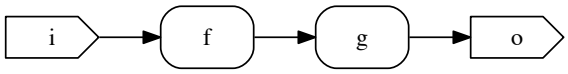
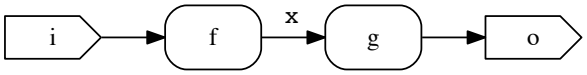


Figure 3: Another dataflow graph

A **first approach** to describe it is to name the intermediate wire, connecting the `f` and `g` nodes :



This is accomplished by the following HoCL "program" :

val x = f i
val o = g x

Note that the order of the `val` declarations is here important because, and by default, it is not possible to use a name before it has been defined[3]

A **second approach** is to interpret the "chaining" pattern depicted in Fig. 3 as function composition. In this view, the graph can be described by this single declaration :

val x = g (f i)

or, assuming that `@` denotes the composition operator on functions :

val x = (g @ f) i

Note the distinctive use of brackets in the two formulations above. In the expression `g (f i)`, the brackets around `f i` are required because function application is generally considered as a left-associative operation[4]. In the expression `(g@f) i`, the brackets are required because the function composition operator `@` has lower priority than function application.

The two above formulations have a little drawback : functions `f` and `g` appear in "reverse order" compared to the position of the corresponding node in Fig. 3. A workaround is to use an infix operator called "reverse function application", often denoted as `|>`, and defined as :

[3]This constraint can be removed by using mutually recursive definitions, as described in Sec. 8.
[4]*I.e.* `g f i` should be interpreted as `(g f) i`.

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

With this operator, and assuming it is left-associative, the graph of Fig. 3 will be described by the following declaration :

```
val o = i |> f |> g
```

in which the RHS expression nicely "mimics" the corresponding graph pattern.

The use of the `|>` operator is even more advantageous for long chain of applications, for which the classical notation for function application leads the numerous brackets. For example,

```
val o = i |> f1 |> f2 |> f3 |> f4 |> f5
```

is clearly much more readable than

```
val o = f5 (f4 (f3 (f2 (f1 i))))
```

A note on types

Clearly, to be composed, either by direct application or by using the `|>` operator, functions must have "compatible" types. More precisely, for the function composition `g @ f` to be *well-typed*

- function `f` must have type $\tau_1 \rightarrow \tau_2$ (*i.e.* taking arguments with type τ_1 and returning results with type τ_2),
- function `g` must have type $\tau_2 \rightarrow \tau_3$ (*i.e.* taking arguments with type τ_2 and returning results with type τ_3),

where τ_1, τ_2 and τ_3 are arbitrary types. The composed function `g @ f` then has type $\tau_1 \rightarrow \tau_3$.

In HoCL, like in most functional programming languages, checking this kind of constraints and, more generally, infering the type of all values occuring in a program, is carried out by a powerful algorithm known as *polymorphic type inference*. This approach effectively precludes the definition of "incoherent" graphs by detecting wrong connexion patterns as soon as possible.

4 Multi-IO nodes

Up to now, all our examples involved nodes having a single input and single output. For example, the `f` and `g` nodes appearing in Fig. 3 would be declared as follows in HoCL :

```
node f in (i: t1) out (o: t2)
node g in (i: t2) out (o: t3)
```

To illustrate how we can deal with nodes having several inputs and/or outputs, let's consider for example the graph depicted in Fig. 4, in which node `foo` (resp. `zib`) has two outputs (resp. inputs).

Because the two outputs of node `foo` must be used distinctively, we must *name* them. This is accomplished by writing :

```
val (x1,x2) = foo i
```

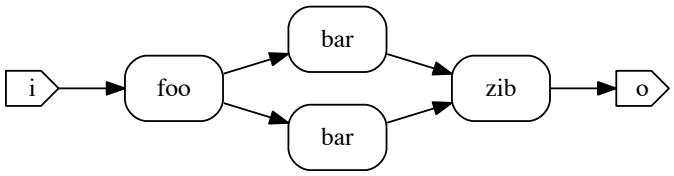
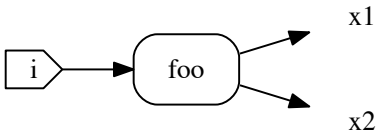


Figure 4: A dataflow graph with multi-IO nodes

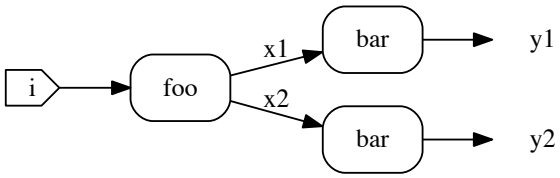
The notation `(x1,x2)` denotes a *tuple*[5]. The `val` declaration binds the first output of node `f` to the first component of this tuple and the second output to the second component[6]. The graph resulting from this first declaration can then be depicted as follows :



We can then apply `bar` to `x1` and `x2` respectively, obtaining two values (edges) which can be named, for example, `y1` and `y2` :

```
val y1 = bar x1
val y2 = bar x2
```

This gives the following graph :



The last step consists in connecting `y1` and `y2` to the inputs of node `h`. This is accomplished by passing these values as arguments to the corresponding function :

```
val o = zib y1 y2
```

Note that, contrary to what might have been expected, the two arguments are here not passed as a *tuple*, *i.e.* writing `zib (y1,y2)`, but simply one after the other. This style of function application is called *curried* form[7] and is very common in functional programming languages. Its main advantage is to allow *partial application* of functions, a feature which will be discussed in Sec. 7.

Introducing explicit bindings for `y1` and `y2` was in fact not necessary and we could have described the graph of Fig. 4 using only two definitions, as follows :

```
val (x1,x2) = foo i
val o = zib (bar x1) (bar x2)
```

[5]A *pair* in this particular case but tuple with any number $n > 1$ of components are allowed.
[6]Here again, the type checker ensures that sizes are compatibles.
[7]After the name of H. Curry, the logician who invented this notation.

The two formulations (with or without definitions for `y1` and `y2`) are strictly equivalent[8] and using one or the other is purely a matter of style.

5 IO-less nodes

Some DFGs may contain input-less or output-less nodes. These nodes are typically used to model the interaction of the graph with the "external orld" (reading or writing data to a file for instance). Fig. 5 gives an example of such a graph, with an input-less node `inp` and an output-less node `outp`.

Figure 5: A dataflow graph with IO-less nodes

In a functional setting, these nodes will be represented by functions taking no argument and producing no result respectively. In functional programming languages, such functions are classicaly represented as functions taking an argument and returning a result of type `unit` respectively. The `unit` type has only one value, denoted `()`.

In HoCL, the graph depicted in Fig. 5 can therefore be described as follows :

$$\textbf{val}\ \text{x} = \text{inp}\ ()$$
$$\textbf{val}\ \text{y} = \text{proc}\ \text{x}$$
$$\textbf{val}\ ()\ = \text{outp}\ \text{y}$$

or, equivalently, as :

$$\textbf{val}\ ()\ = \text{outp}\ (\text{proc}\ (\text{inp}\ ()))$$

or, using the `|>` operator, as :

$$\textbf{val}\ ()\ = ()\ |{>}\ \text{inp}\ |{>}\ \text{proc}\ |{>}\ \text{outp}$$

Because, having input data provided by input-less actors is quite frequent in dataflow-based applications, HoCL has a special operator for expressing the corresponding graph pattern (and hence avoiding the "`() |>`" syntax, which may appear a bit weird. This operator is denoted `|->` and defined as

$$\text{f}\ |{-}{>}\ \text{g} = ()|{>}\ \text{f}\ |{>}\ \text{g}$$

The graph of Fig. 5 can therefore be described with this very simple definition in HoCL :

$$\textbf{val}\ ()\ = \text{inp}\ |{-}{>}\ \text{proc}\ |{>}\ \text{outp}$$

[8]This is in fact a property of side-effect free functional programming languages.

6 Wiring functions

Consider the graph depicted in Fig. 6, which is a variation of that of Fig. 3 in which the *same* function `foo` is applied in sequence.

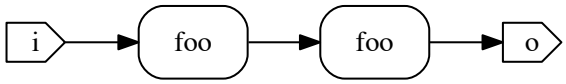


Figure 6: A dataflow graph with a repetition pattern

As introduced in Sec. 3, this graph can be described as follows :

```
val o = i |> foo > foo
```

We are interested here in capturing the pattern exemplified in this graph, *i.e.* the fact that a given function is applied "twice in a row" to a given input. In HoCL, this can be done by writing the following definition :

```
val twice f x = x |> f |> f
```

The value `twice` defined by the declaration is a *function*, taking two arguments, named `f` and `x`[9]. This function, as described in the RHS expression, simply applies twice its first argument to its second argument.

The graph in Fig. 6 can then simply be described by *applying* function `twice` to the actual input `i` and function `foo` :

```
val o = twice foo i
```

Note that the function `twice` is a bit "special" in that its first argument is itself a function. In functional programming languages, these functions are classicaly called *higher order functions*[10]. They are very useful in the context of graph description since they allow specific (and possibly recurrent) graph patterns to be "encapsulated" – by storing their definition in a library for example – and reused latter. For example, the "diamond-shaped" pattern depicted in Fig. 4 can be encapsulated in the following definition :

```
val diamond f g h x =
  let (x1,x2) = f x in
  h (g x1) (g x2)
```

where the `let ... in ...` construct is used to introduce a *local definition*.

The graph of Fig. 4 can now be described by applying the function `diamond` to `foo`, `bar`, `zib` and `i` respectively :

```
val o = diamond foo bar zib i
```

Note. An interesting question is whether the `twice` function can be generalized to describe a graph where n (instead of two) instances of a node are chained linearly, *i.e.*, we could define a function – let's call it `iter` – so that the function `twice` could be defined as `twice f x = iter 2 f x`. The answer is yes and will be given in Sec. 9.

[9] The syntax `val f x = e` is actually a shorthand for `val f = fun x -> e`.

[10] This is where the "Ho" in HoCL comes from.

7 Partial application

As stated in Sec. 4, nodes with multiple inputs are represented by functions taking their arguments in *curried* form. More precisely, a node declared as

node f **in** (i1:t1, ..., im:tm) **out** (o:t)

will be represented by a function f having type

$$t_1 \to t_2 \to \ldots \to t_m \to t.$$

This type should be interpreted as

$$t_1 \to (t_2 \to \ldots \to t_m \to t).$$

In other words, f is a function taking an argument of type t_1 and *returning a function* of type $t_2 \to \ldots \to t_m \to t$. The returned function can itself be applied to an argument of type t_2 and so on until the last argument of type t_m is provided to give the result of type t.

In functional programming languages like OCaml or Haskell for example, it is common to define a function adding its two arguments as[11] :

val add x y = x+y

so that a function adding one to its argument can be defined as :

val inc = add 1

i.e. by *partially applying* the function `add`. Because of this partial application, `inc` is a function taking an single argument and can itself be applied :

val r = inc 2

gives r=3.

Allowing partial application is the main reason why n-ary functions are generally defined in curried form in functional programming languages rather than as functions taking a n-tuple as argument[12].

In HoCL, partial application provides an elegant solution to a problem occuring when dealing with graph containing *parameterized* nodes.

A parameterized node is a node for which at least one input is not bound to a data stream but is used to *configure* it. The instants at which these configurations occur and their effect on the node behavior depends on the underlying dataflow *model of computation* and do need to be detailed here. At the specification level, it is ony required to distinguish parameters from "regular" data (flows). In HoCL, this distinction is made using types. For example, an actor `mult` accepting an input stream of integers and producing an output stream containing each input multiplied by a constant factor `k` will be declared as :

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

The type `int param` assigned to input `k` identifies this input as a *parameter* input. Fig. 7 gives an example of a graph containing an instance of node `mult`. Parameters and parameter dependencies have here been drawn using dashes to distinguish them from the data flows and a default value of 2 has been attached to the parameter.

[11] The actual concrete syntax differ.

[12] Though the latter is also possible.

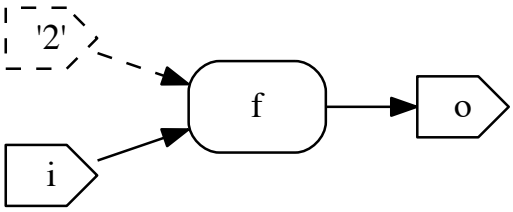


Figure 7: A dataflow graph containing a parameterized node

A complete description of this graph in HoCL is given in Listing 2.

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

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

Listing 2: HoCL description of the graph depicted in Fig. 7

It is interesting to remember that, in the formulation given in Listing 2, the expression

f k i

should be interpreted as as

(f k) i

In other words, the value `o` is obtained by first applying `f` to `k`, obtaining a *specialized* version of `f` and then applying this specialized version to input `i`. This interpretation nicely fits with the idea of parameters acting as *configuration values* not to be mixed with "regular" data flows. It also allows the `val` definition in Listing 2 to be rewritten as :

```
val o = i |> f k
```

something that was not obvious at first sight.

The use of partial application as an "enabler" for the `|>` operator is even more convincing for the graph in Fig. 8.

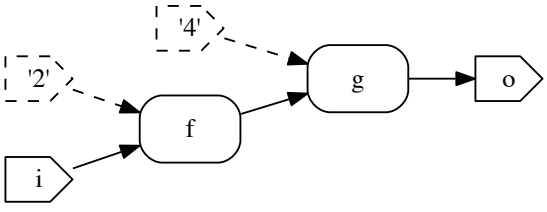


Figure 8: A dataflow graph containing two parameterized nodes

Without partial application, the graph in Fig. 8 would have to be described as in Listing 3

```
node f in (k: int param, i:int) out (o: int);
node g in (p: int param, i:int) out (o: int);

graph top in (k1: int param=2, k2: int param=4, i: int) out (o: int)
fun
  val o = g k2 (f k1 i)
end;
```

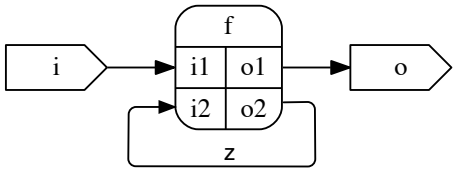
Listing 3: HoCL description of the graph depicted in Fig. 8

With partial application, the `val` definition can be rewritten in a more "natural" manner as :

```
val o = i |> f k1 |> g k2
```

8 Recursive wiring

Cyclic graph structures – in which the output of a node is fed back to one of its inputs – can also be described functionally, as exemplified in Fig.9[13]. Note that the value `z`, corresponding to the feedback edge, is used both as an input and an output of `f` function call (in the RHS and LHS of the `val` declaration, respectively). This is the justification of the `rec` (shorthand for *recursive*) keyword in the `val` declaration. Without it, the symbol `z`, which is *defined* by declaration would not be visible in the RHS expression (`f i z`).



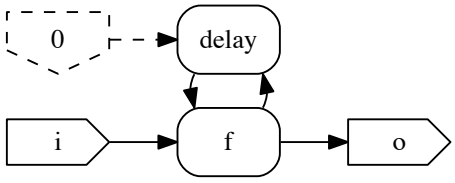

```
node f
  in (i1: int, i2: int)
 out (o1: int, o2: int);

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

Figure 9: A cyclic dataflow graph and its description in HoCL

This kind of "recursive wiring" is not limited to self loops, like in Fig. 9, but can be used to describe any cyclic path. Fig. 10, for example, shows a very common situation in dataflow modeling, where the cycling path contains a "delay" actor. Note that the initial value of the delay actor (*i.e.* the value of the token initially present on its output) is here specified as a parameter. In the graph of Fig. 10, this value is statically set to 0.

[13]In this figure, the `f` node has been drawn as a box with distinct "slots" for each of its input and output, for clarity.



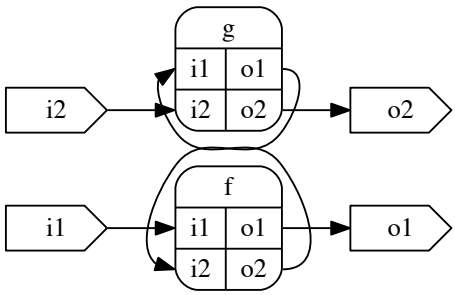

```
node f
  in (i1: int, i2: int)
 out (o1: int, o2: int);

node delay
  in (iv: int param, i: int)
 out (o: int);

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

Figure 10: Another cyclic dataflow graph and its description in HoCL

Recursive declarations can even be used to describe mutually recursive nodes, as shown in Fig. 11. The construct `val rec ... and ...` introduces *mutually recursive definitions*. Each of the `val` declaration can refer to any symbol defined in another declaration (here the first declaration refers to the symbol `z2` defined in the second and *vice versa*).




```
node f
  in (i1: int, i2: bool)
 out (o1: int, o2: bool);

node g
  in (i1: bool, i2: int)
 out (o1: bool, o2: int);

graph top
  in (i1: int, i2: int)
 out (o1: int, o2: int)
fun
  val rec (o1,z1) = f i1 z2
  and     (z2,o2) = g z1 i2
end;
```

Figure 11: A dataflow graph showing mutually recursive nodes and its description in HoCL

9 Generalized wiring functions

In the examples given up to now, the manipulated values were either representing *edges* (*wires*) or *functions*. HoCL can also manipulate scalar values such as integers or booleans or structured values such as lists. Performing *computations* on such values can be used to describe rich graph structures.

As an illustration,let's go back to the question raised at the end of Sec. 6, *i.e.* the definition of a function `iter` taking three arguments, an integer n, a function f and a value x, and returning the result of applying n times the function f to x[14] :

$$iter\ n\ f\ x = \underbrace{f(\ldots(f(fx))\ldots)}_{\text{n applications of f}}$$

[14]Assuming $n > 0$ here.

For any input `i` and node `f`, the graph depicted in Fig. 12 will be described by the following definition :

```
val o = iter n f i
```

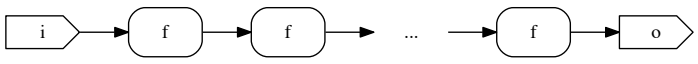


Figure 12: The graph described by `val o = iter n f x`

Defining the `iter` function in HoCL is done exactly as in any functional programming language, using *recursion* :

- if n is zero, then `iter n f x` is simply `x`,
- if $n > 0$, then `iter n f x` is obtained by first computing `f x` and then applying `iter (n-1) f` to the result.

This gives the following definition for `iter`, which is a direct translation of the above principle :

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

Listing 4: HoCL definition of the `iter` function depicted in Fig. 12

As all functional programming languages, HoCL also has a `list` type and this type can be used to define powerful higher-order graph building functions.

Listing 5, for example, defines a `pipe` higher-order function which can be used to describe graphs built by chaining (pipelining) a sequence of functions. The `pipe` function is, here again, defined by recursion and by inspecting the structure of its list argument. This inspection is carried out using the `match | ... | ...` construct. There are two cases :

- either the list (`fns`) is empty (`[]`),
- either the list is not empty and hence can be viewed as a *head* (`f`) followed by a *tail* (`fs`) (which is denoted as `f::fs`, using the `::` operator).

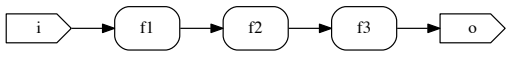
In the first case, the result of the application `pipe fns x` application is simply `x`. In the second case, this result is obtained by first applying `f` to `x` and then applying `pipe` to `fs` and this first result. Here's the sequence of reduction corresponding to the evaluation of `pipe [f1;f2;f3] x` for example :

$$\begin{aligned}
\text{pipe } [f_1; f_2; f_3]\ x &\Rightarrow \text{pipe } (f_1 :: f_2 :: f_3 :: [])\ x \\
&\Rightarrow \text{pipe } (f_2 :: f_3 :: [])\ (f_1\ x) \\
&\Rightarrow \text{pipe } (f_3 :: [])\ (f_2(f_1\ x)) \\
&\Rightarrow \text{pipe } []\ (f_3(f_2(f_1\ x))) \\
&\Rightarrow f_3(f_2(f_1\ x))) = x \triangleright f_1 \triangleright f_2 \triangleright f_3
\end{aligned}$$

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

Listing 5: HoCL definition of the `pipe` function

An example of using the `pipe` higher-order function is given in Fig. 13.




```
node f1 in (i: int) out (o: int);
node f2 in (i: int) out (o: int);
node f3 in (i: int) out (o: int);

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

Figure 13: An example of using the `pipe` wiring function defined in Listing 5

The HoCL *standard library* available in the distribution defines several useful higher-order wiring functions using the `list` type for describing various graph patterns (`map`, `fold`, *etc.*).