HoCL: High level specification of dataflow graphs

Jocelyn Sérot
Institut Pascal, UMR 6602 Université Clermont Auvergne, CNRS, SIGMA Clermont
Clermont-Ferrand, France

ABSTRACT

We introduce HoCL (Higher order Coordination Language), a domain specific language (DSL) for specifying hierarchical, parameterized dataflow graphs. HoCL leverages on a purely functional semantics to allow graph structures to be described in an abstract and concise manner. Generic graph patterns, in particular, can be encapsulated as user-definable higher-order functions. HoCL descriptions are independent of the underlying dataflow model of computation and the HoCL compiler is intended to be used as a front-end to existing dataflow visualization, analysis and implementation tools. HoCL and its documentation are freely available on Github [17].

CCS CONCEPTS

• Software and its engineering → Functional languages; Dataflow languages; Formal language definitions; Source code generation; Domain specific languages.

ACM Reference Format:
Jocelyn Sérot. 2020. HoCL: High level specification of dataflow graphs. In IFL ’20: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL ’20), September 2–4, 2020, Canterbury, United Kingdom. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462185

1 INTRODUCTION

Dataflow modeling is used extensively for designing digital signal processing (DSP) systems. With this approach, applications to be implemented are described as graphs of persistent processing entities, named actors, connected by first in, first out (FIFO) channels and performing processing (“firing”) when their incoming FIFOs contain enough data tokens. By varying the semantics of these firing rules, many dataflow models of computations (MoCs) can be defined, offering different trade-offs between expressivity and predictability, while keeping the key property of dataflow models: their ability to naturally express the intrinsic parallelism of DSP applications.

As a result, a wide variety of dataflow-based design tools have been developed, such as Ptolemy [3], LabView [10] or Preesm [13], for specification, simulation and synthesis, for hardware or software implementation, of dataflow-oriented applications.

With these tools, the specification of the application is typically carried out textually, using some form of graph notation, or graphically, using a dedicated Graphical User Interface (GUI). In both cases, the specification of large or/and complex graphs quickly becomes tedious and error-prone.

In this paper, we propose HoCL, a domain-specific language (DSL) aimed at simplifying and streamlining the description of large or/and complex dataflow graphs. The key feature of this language is the ability to describe graph structures as functions, so that several well-known and powerful concepts drawn from functional programming languages – such as polymorphic typing and higher order functions – can be applied both to ease and secure the task of describing these graphs.

The rest of this paper is organized in seven sections. Section 2 presents the main features of the HoCL language, by means of small examples. Section 3 gives some insights on its formal semantics. Section 4 describes the current implementation, in particular the available backends. Section 5 describes the design, with HoCL, of a complete DSP application. Section 6 is short review of related work and section 7 concludes the paper.

2 THE HOCL LANGUAGE

A small example of program is given in Listing 1, with the corresponding DFG in Fig. 1.

The abstract syntax of the HoCL language is given in Fig. 2. The meta-syntax is classical: Keywords are denoted in bold, other terminals in italics; non-terminals are enclosed in angle brackets. Vertical bars are used to indicate alternatives. Constructs enclosed in brackets are optional. An asterisk (*) indicates zero or more repetitions of the previous element, and a plus (+) indicates one or more repetitions.

Type expressions denote types attached to actor input and output ports. A type can be either a base type τ or the constructed type τ param. Base types are associated to "regular" data flows, param types to parameters (see Sec. 2.6). Base types are limited to the predefined types int and bool, abstract types (introduced by an explicit type declaration) and type variables. Type variables are used for declaring polymorphic actors. The type list is predefined but cannot be used in type expressions.

1Also with IETR, UMR 6164 Université Rennes 1, INSA Rennes.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

IFL ’20, September 2–4, 2020, Canterbury, United Kingdom
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8963-1/20/09... $15.00
https://doi.org/10.1145/3462172.3462185

*These types are propagated to graph edges during the type checking and inference step.
The interface gives the name of the node and the name and type of values, denoted by expressions. The syntax of expressions is classified to foster the adoption of the language among programmers not familiar with graph descriptions, introduced by the \texttt{struct} keyword. This possibility was introduced to foster the adoption of the language among programmers not familiar with functional programming. It is not discussed in this paper.

Listing 1: A simple HoCL program

\begin{center}
\begin{tikzpicture}
\node (f) at (0,0) {f};
\node (g) at (2,0) {g};
\node (k) at (1,1) {k};
\node (h) at (3,1) {h};
\node (i) at (-2,0) {i};
\node (o) at (4,0) {o};
\draw (i) -- (f);
\draw (f) -- (k);
\draw (k) -- (g);
\draw (g) -- (k);
\draw (k) -- (h);
\draw (h) -- (o);
\end{tikzpicture}
\end{center}

Figure 1: The DFG described by the program in Listing 1

Node declarations (introduced by the \texttt{node} keyword) define node models. A node model is made of an interface and a description. The interface gives the name of the node and the name and type of each input and output. In the example, for simplification, all inputs and outputs have type \( t \). The description can be either empty (as for node \( f \)) or be given as a list of definitions (as for node \( g \)). In the first case, the declaration describes an atomic node. In the second case, the given definitions describe a subgraph\(^3\).

Definitions (introduced by the \texttt{val} keyword) bind names to values, denoted by expressions. The syntax of expressions is classical. The usual boolean and arithmetic associated operations (+, ...) are pre-defined. The data constructors \( : \) and \([\ldots] \) (\texttt{cons} and \texttt{nil}, for building and manipulating lists) and \( () \) (\texttt{unit}) are also pre-defined.

The semantics of expressions is also classical, except for application. In HoCL, node models are viewed as functions, and a node declared as

\begin{verbatim}
node f in (i: t) out (o1: t, o2: t);
node k in (i: t) out (o: t);
node h in (i: t, i2: t) out (o: t);
node g in (i: t) out (o: t)
fun
  val o = k (k i)
end;

graph top in (i: t) out (o: t)
fun
  val (x1, x2) = f i
  val o = h (g x1) (g x2)
end;
\end{verbatim}

\begin{verbatim}
(program) ::= (type_decl)* (val_decl)* (node_decl)*
(type_decl) ::= type ident
(node_decl) ::= node ident \texttt{in} ( (io_decl)* ) out ( (io_decl)* ) [(node_impl)]
        | graph ident ( (gio_decl)* ) ( (io_decl)* ) ( (node_impl)
        | (io_decl)* ) (node_impl)
(io_decl) ::= ident : (type_expr)
(gio_decl) ::= ident : (type_expr) [ = (const_expr)]
(node_impl) ::= (val_decl)*
(valDecl) ::= val \texttt{[rec]} (binding)*
(binding) ::= (pattern) = (expr)
(pattern) ::= (simplepat)
        | ( (simplepat)* )
        | (simplepat) ::(simplepat)
        | [ (simplepat)* ]
\end{verbatim}

\begin{verbatim}
\begin{tabular}{l l l}
\texttt{simplepat} & ::= & \texttt{ident} | () \\
\texttt{expr} & ::= & \texttt{(const_expr) | ident | ()} \\
& | & ( (expr)* ) \\
& | & (expr) (expr) \\
& | & \texttt{fun} (funpat) \rightarrow \texttt{expr} \\
& | & \texttt{if} (expr) \texttt{then} (expr) \texttt{else} (expr) \\
& | & \texttt{match} (expr) \texttt{with} \texttt{(case)}* \\
& | & \texttt{let} \texttt{[rec]} (binding)* \texttt{and} \texttt{in} (expr) \\
& | & [ (expr)* ] \\
& | & \texttt{(expr)} (binop) \texttt{expr} \\
\texttt{case} & ::= & \texttt{(pattern)} \rightarrow \texttt{(expr)} \\
\texttt{binop} & ::= & + | - | :: | ... \\
\texttt{funpat} & ::= & \texttt{ident} \\
\texttt{const\_expr} & ::= & \texttt{int} | \texttt{true} | \texttt{false} \\
\texttt{type\_expr} & ::= & \texttt{(base\_type) | (base\_type) param} \\
\texttt{base\_type} & ::= & \texttt{int} | \texttt{bool} | \texttt{ident} | \texttt{\_var}
\end{tabular}
\end{verbatim}

Figure 2: Abstract syntax of the HoCL language

is interpreted as a function having type:

\begin{equation}
i_1 : \tau_1 \rightarrow \ldots \rightarrow i_m : \tau_m \rightarrow \tau'_1 \times \ldots \times \tau'_n
\end{equation}

where \( i_1, \ldots, i_m \) are argument labels (see Sec. 2.1).

Full application of the function associated to a node then instantiates this node in the graph. In Fig. 1, for example, the definition given for node \( g \) specifies that the corresponding subgraph is built by

- instantiating twice node \( k \),
- connecting the input of the first instance to the input of the subgraph,
- connecting the input of the second instance to the output of the first instance,
- connecting the output of the second instance to the output of the subgraph.
Graph declaration (introduced by the graph keyword) are also made up from an interface and a description but, at the difference of node declarations, the description is always a subgraph and the corresponding graph is automatically (implicitly) instantiated\(^4\). In the example of Listing 1, the toplevel graph top is built by

- first instantiating actor f, connecting its input \(i\),
- then instantiating twice subgraph g, connecting the first (resp. second) output of node \(f\)\(^5\) to the input of the first (resp. second) instance,
- finally instantiating actor h, connecting its first (resp. second) input to the output of the first (resp. second) instance of subgraph g.

It must be noted that the binding(s) performed by a \texttt{val} definition actually depends of the name(s) occurring in the right-hand side. If a name refers to the output of a (sub)graph, then this output is simply connected to the value denoted by the right-hand side (this value must correspond to a (sub)graph input or a node output port, which will be checked by the typing phase). This is the case, for example, for output \(o\) in the definition of subgraph g. Otherwise, the \texttt{val} definition is simply used to name intermediate values (like \texttt{let} declarations in functional programming languages). This is the case, for example, in the definition of graph top, where the first \texttt{val} definition is used to distinguish the two outputs of node \(f\).

\section{Labeled arguments}

As evidenced by the type signature (1), HoCL supports label-based passing of arguments. For example, the three following applications of node \(h\) (as declared in Listing 1) are all valid and equivalent:

\begin{verbatim}
val o1 = h x y
val o2 = h i1 : x i2 : y
val o3 = h i2 : y i1 : x
\end{verbatim}

This is useful for nodes having a large number of inputs, when passing the arguments to the corresponding function “in the right order” may become error-prone. This is especially true if a large proportion of these inputs have the same type, because the resulting error(s) will not be caught by the type checker in this case\(^6\).

As will be shown, in Sec. 2.6, labeled arguments are also useful for relaxing the constraints on parameterized node signatures.

\section{IO-less nodes}

Nodes with no input (resp. no output) can be described by specifying an empty input (resp. output) list. In applications, these nodes can represent data sources (resp. sinks)\(^7\). This is exemplified in Listing 2 and Fig. 3.

---

\(^4\)A valid specification in HoCL is therefore made up of at least one graph declaration.

\(^5\)Strictly speaking, to the second output of the instance of node \(f\). But, for the sake of concision, and unless explicitly noted, we will now denote a node instance by the name of the corresponding node model.

\(^6\)This is frequently the case in DSP applications because the sequential functions implementing the node behaviors often (over)use the \texttt{int} type to represent data.

\(^7\)Whether these sources (resp. sinks) are represented as top-level graph inputs (resp. outputs) or as input-less (resp. output-less) nodes generally depends on the actual implementation.

\(^8\)These operators are written \(\triangleright\triangleright\) and \(\triangleright\triangleright\triangleright\) in the HoCL concrete syntax respectively.
val rec pipe fs x = match fs with
    | [] -> x
    | f::fs -> pipe fs (f x);

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 4: The pipe higher-order wiring function

val rec map f xs = match xs with
    | [] -> []
    | x::xs -> f x :: map f xs;

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

graph top in (i1 : int, i2 : int,i3 : int)
out (o1 : int, o2 : int, o3 : int)
fun
    val [o1;o2;o3] = map f [i1;i2;i3]; end;

Figure 5: The map higher-order wiring function

An important feature is that all these functions are defined using regular HoCL declarations, i.e. within the language itself\footnote{In file lib/hocl/stdlib.hcl of the distribution, technically.}. The set of available higher order graph patterns is therefore not fixed but can be freely modified and extended by the application programmer to suit her specific needs. This is in strong contrast with most dataflow-based design tools (those listed in the introduction in particular) in which similar abstraction mechanisms, when available, rely on a predefined and fixed set of patterns.

2.4 Recursive graphs

In a dataflow context, a recursive graph is a graph in which the refinement of some specific nodes is the graph itself. A typical example is provided by Lee and Parks in their classical paper on dataflow process networks [11].

This example is an analysis/synthesis filter bank. The corresponding dataflow graph has a regular structure which can be characterized by its "depth". Fig. 6, for example, shows a graph of depth three\footnote{The meaning of the QMF, QMF2 and F actors is irrelevant here.}.

Figure 6: A filter bank of depth 3 (redrawn from [11], Sec III-C, p. 792)

For the sake of generality, Lee and Parks propose to view this graph as an instance of a "recursive template", depicted in Fig. 7.

Figure 7: A “recursive template” for filter bank of depth D (redrawn from [11], Sec III-C, p. 793)

The recursive nature of this description is evidenced by the occurrence, in the definition of the graph labeled $FB(D>0)$, of a node labeled $FB(D=0)$. The graph labeled $FB(D=0)$ provides the base case for the recursion.

This graph structure can be readily encoded in HoCL as follows:

\[
\begin{align*}
\text{val rec } & \text{ fb d x = if d=0 then f x else let (x1, x2) = qmf x in qmf2 (f x1) (fb (d-1) x2) end;}
\end{align*}
\]

\[\text{In file 11b/hocl/std11b.hcl of the distribution, technically.}\]
so that the graph of Fig. 6 can be simply defined as
\[
\text{graph leeparks3 in } (i : \text{int}) \text{ out } (o : \text{int}) \\
\text{fun } o = fb \ 3 \ i \\
\text{end;}
\]
where the type of the data tokens has here been arbitrarily set to \text{int}.

2.5 Cyclic graphs

In most of dataflow models, cycles in a graph – i.e. sequences of edges connecting the output of an actor to one of its inputs – denote dependencies between two successive activations of this graph. For example, the graph in Fig. 8 describes a recursive filter, taking as input a sequence \(x_1, x_2, \ldots\) of tokens and producing as output the sequence \(y_1, y_2, \ldots\), where

- \((y_i, z_i) = f(x_i, z_{i-1})\) for all \(i > 0\),
- \(z_0\) is an arbitrary constant.

The special actor named \text{delay} is here used to store the \(z\) value between two successive activations and to provide the initial value \(z_0\).

\[\text{delay}\]

\[f\]
\[i_1\]
\[o_1\]
\[i_2\]
\[o_2\]
\[\text{delay}\]

\[g\]
\[i_2\]
\[o_2\]
\[\text{delay}\]

\[f\]
\[i_1\]
\[o_1\]
\[i_2\]
\[o_2\]

Figure 8: A cyclic dataflow graph

This kind of cyclic structure can be described as shown in Listing 3 in HoCL. The recursive nature of the definition – the \(z\) symbol occurring both in the left and right-hand sides – is evidenced by the \text{rec} keyword.

\[
\text{graph top in } (i : \text{int}) \text{ out } (o : \text{int}) \\
\text{fun } \text{rec } (o, z) = f \ i \ z \\
\text{end;}
\]

Listing 3: The DFG of Fig. 8 described in HoCL

Mutual recursion is also possible, as exemplified by the following description of the graph depicted in Fig. 9:

\[
\text{node } f \text{ in } (i_1 : t, i_2 : t) \text{ out } (o_1 : t, o_2 : t) \\
\text{node } g \text{ in } (i_1 : t, i_2 : t) \text{ out } (o_1 : t, o_2 : t) \\
\text{graph top in } (i_1 : t, i_2 : t) \text{ out } (o_1 : t, o_2 : t) \\
\text{fun } \text{rec } ((o_1, z_1), (z_2, o_2)) = f \ i_1 \ z_2, \ g \ z_1 \ i_2 \\
\text{end;}
\]

Figure 9: A DFG showing mutual recursion between nodes

It is important to note that HoCL describes dataflow graph without making any assumption on their underlying dynamic semantics. A definition like

\[
\text{val rec } (o, z) = f \ i \ z
\]

for example, is accepted and gives a graph in which the second output of node \(f\) is directly connected to its second input. Such a graph clearly does not satisfy the liveness property of the synchronous dataflow (SDF) model for example (in the absence of an initial token on its second input, the \(f\) actor will never fire). The reason is that checking this kind of properties ultimately depends on the underlying model of computation, something that HoCL, as a pure coordination language, does not take into account, leaving the corresponding analysis to dedicated tools taking its output as input.

2.6 Parameterized graphs

The term parameterized dataflow was introduced in [1] to describe a meta-model which, when applied to a given dataflow model of computation (MoC), extends this model by adding dynamically reconfigurable actors. Reconfigurations occur when values are dynamically assigned to parameters of such actors, causing changes in the computation they perform and/or their consumption and production rates. The precise nature of changes triggered by reconfigurations and the instants at which these reconfigurations can occur both depend on the target MoC. HoCL offers a MoC-agnostic

\[11\]

It could be argued that the HoCL compiler could, at least, check that any cycle contains at least one \text{delay} actor. But, unless some MoC or target-specific annotations are used, such actors cannot be reliably identified.
interface to this feature using a dedicated type to distinguish parameters from "regular" data flows.

Consider, for example, the delay actor occurring in Fig. 8. This actor can be parameterized by the value initially produced on its output. For this, it will be declared as follows:

```haskell
node delay
  in (init: α param, i: α) out (o: α)
```

where α designates a type variable

This makes delay a polymorphic actor with type:

```haskell
init: α param → i: α → α
```

The example given in Fig. 8 can be reformulated using the configurable version of the delay actor as shown in Listing 4 and Fig. 10.

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

Listing 4: A reformulation of the example given in Listing 3 using a configurable delay

Figure 10: The DFG described by the program in Listing 4

In Fig. 10, parameter values are drawn as house-shaped nodes and parameter dependencies using dashed lines. In Listing 4, the value of the init parameter has been set to 0. The enclosing quotes are here used to turn a value of type int into a value of type int param.

It can be noted that, with this approach, actor (re)configuration is interpreted as the partial application of the corresponding function. The following definition

```haskell
val delay0 = delay '0'
```

for example, defines an actor delay0, with type int → int, taking and producing flows of integers and producing 0 as its initial value.

It can be further noted that, using labeled arguments, this interpretation can hold even when parameter(s) are not specified at first position in the list of node inputs. For example, would the delay actor have been defined as:

```haskell
node delay
  in (i: α, init: α param) out (o: α)
```

the definition of the delay0 actor would still be possible by writing:

```haskell
val delay0 = delay init: '0'
```

### 2.7 Parameters and hierarchy

When a parameterized node is refined as a subgraph, the value of the parameter(s) can be used to parameterize the nodes of the subgraph, either directly or by means of some dependent computations. This allows parameters to be propagated across graph hierarchies. This is illustrated by the following program, which expands into the graphs depicted in Fig. 11.

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

```haskell
graph top in (i: int) out (o: int)
fun
  val o = i * sub '2'
end;
```

In graph sub, k is viewed as an input parameter (drawn as a dashed input port in Fig. 11) and used to parameterize both instances of the mult actor, first directly and second by through the parameter expression k+1. It is important to note that, although this could make sense in this particular example, parameter expression are not statically evaluated by the HoCL compiler since their interpretation ultimately depends on the target MoC (which controls, in particular, when parameters are evaluated to trigger the reconfiguration of the dependent actors).

Parameter dependencies create dependency trees. The root of these trees can be either constants, as in the previous example, or specified as top level input parameters, as illustrated in the following program, which is an equivalent reformulation of the previous example. Note that, at the difference of node parameters, toplevel parameters must be given a value.

```haskell
graph top
  in (n:int param=2, i:int) out (o:int)
fun
  val o = i * sub n
end;
```

Figure 11: A hierarchical graph with parameter passing

Written ’a in ASCII-encoded source code.
3 SEMANTICS

The semantics of the HoCL language is specified in natural (big step) style. It gives the interpretation of HoCL programs, described with the abstract syntax given in Fig. 2, as a set of *dataflow graphs*, where each graph is defined as a set of boxes connected by wires. The formulation given here assumes that the program has been successfully type checked\(^\text{13}\). This semantics is built upon the semantic domain described in Fig. 12, using the following meta-syntax: The + symbol denotes union; tuples are denoted between angle brackets ((...)); sets between curly brackets ({...}); an asterisk (\(^*\)) (resp. \(^+\)) in superscript position denotes zero (resp. 1) or more repetitions of the scripted element.

**Values** in the category Int and Bool denotes scalar values. The categories Clos, Prim and Tuple respectively correspond to

- closures, denoting functional values,
- builtin functions, operating on integer or boolean values (+, =, ...),
- tuples.

Values in the category Loc denote *graph locations*. Such locations are made of a box index and a selector. Boxes correspond to actor instances. Selectors are used to distinguish inputs (resp. outputs) when the box has several of them. Valid selectors start at 1. The selector value 0 is used for incomplete box definitions.

**Nodes** are described by

- a node category, indicating whether the node is a toplevel graph or an ordinary node\(^\text{14}\),
- a list of inputs, each with an attached value\(^\text{15}\),
- a list of outputs,
- an implementation, which is either empty (in case of opaque actors) or given as a graph.

**Boxes** are described by

- a box category,
- a input environment, mapping selector values (1,2,...) to wire identifiers,
- a output environment, mapping selector values to sets of wire identifiers\(^\text{16}\),
- an optional value.

Box categories separate boxes

- resulting from the instantiation of a node,
- materializing graph inputs and outputs,
- materializing graph input parameters,
- materializing graph local parameters.

The box category rec is used internally for building cyclic graphs. The box value is only meaningful for local parameters bound to constants or for toplevel input parameters (giving in this case the bound value).

**Wires** are pairs of graph locations: one for the source box and the other for the destination box.

The environments E, B and W respectively bind

- identifiers to semantic values,
- box indices to box description,
- wire indices to wire description.

In this paper, the description of the semantics is deliberately limited to a subset of the associated inference rules. A complete version is available on the Github repository [18]. In these rules, all environments are viewed as partial maps from keys to values. If \(E\) is an environment, the domain of \(E\) is denoted by \(\text{Dom}(E)\). The empty environment is written \(\emptyset\). \([x \mapsto y]\) denotes the singleton environment mapping \(x\) to \(y\). \(E(x)\) denotes the result of applying the underlying map to \(x\) (for ex. if \(E\) is \([x \mapsto y]\) then \(E(x) = y\) and \(E @ E'\) the environment obtained by adding the mappings of \(E'\) to those of \(E\), assuming that \(E\) and \(E'\) are disjoints.

Fig. 14 gives the main rules for node declarations. Each declaration is evaluated in a pair of value and box environments and produces a pair of updated environments. Rule NDeclA concerns nodes with no attached definition. These are mapped to opaque actors. The Unit value initially attached to inputs here means “yet unconnected”. Rule NDeclG concerns nodes with attached definitions. These definition are evaluated in an environment augmented with node input and output declarations, and the resulting graph (a pair of boxes and wires) is attached to the node declaration. The auxiliary rules B \(\triangleright I\) \(\ldots\) \(\rightarrow E',B'\) and B \(\triangleright o\) \(\ldots\) \(\rightarrow E',B'\) (not detailed here) augment the value and box environments with the bindings and box descriptions attached to the node inputs and outputs respectively\(^\text{17}\).

The semantics of value declarations (either at the program top level or within node declarations) is described in Fig. 15. Rule VDeclg gives the semantics of a sequence of such declarations and rule VDecl of a single declaration. Declarations are interpreted in order. Each declaration updates the value, box and wire environments. Rule Binding gives the semantics of bindings occurring in definitions. The \(\oplus\) operator used in this rule merges box descriptors. If a box appears in both argument environments, the resulting environment contains a single occurrence of this box in which the respective input and output environments have been merged. For example

\[
\begin{align*}
[l \mapsto \text{Box(actor, [1 \mapsto 0], [1 \mapsto (2)]})] \\
\oplus [l \mapsto \text{Box(actor, [1 \mapsto 4], [1 \mapsto (3)]})] \\
= [l \mapsto \text{Box(actor, [1 \mapsto 4], [1 \mapsto (2, 3)]})]
\end{align*}
\]

The rule \(E, B \triangleright p \ pat, v \Rightarrow E', B', W'\), not detailed here, gives the semantics of pattern matching. It returns the augmented value, box and wire environments obtained by binding pattern \(pat\) to value \(v\).

---

\(^{13}\)The type checking rules are classical and not discussed here.

\(^{14}\)This avoids having two distinct but almost identical semantic values for nodes and toplevel graphs.

\(^{15}\)These values are used to handle partial application.

\(^{16}\)A box output can be broadcasted to several other boxes.

\(^{17}\)Outputs create boxes with category snk, inputs create boxes with category src or inParam depending on their type.
| Category | Variable | Definition | Meaning |
|----------|----------|------------|---------|
| Val      | ν        | Loc + Node + Tuple + Clos | Value   |
| Loc      | ℓ        | ⟨bid, sel⟩ | Graph location |
| Node     | n        | ⟨NCat, {id → Val}, {id}, Nimpl⟩ | Node description |
| NCat     | κ        | node + graph | Node category |
| Tuple    | vs       | Val⁺ | Tuple |
| Clos     | cl       | ⟨pattern, expr, Env⟩ | Closure |
| Env      | E        | {id → Val} | Value environment |
| Nimpl    | η        | actor + Graph | Node implementation |
| Graph    | g        | ⟨Boxes, Wires⟩ | Graph description |
| Boxes    | B        | {bid → Box} | Box environment |
| Wires    | W        | {wid → Wire} | Wire environment |
| Locs     | L        | Loc⁺ | Location set |
| Box      | b        | ⟨BCat, {sel → wid}, {sel → wid⁺}, Val⟩ | Box |
| BCat     | c        | actor + graph + src + snk + rec | Box category |
| Wire     | w        | ⟨⟨bid, sel⟩⟩ | Wire |
| bid      | l, l'    | {0, 1, 2 . . .} | Box id |
| wid      | k, k'    | {0, 1, 2 . . .} | Wire id |
| sel      | s, s'    | {0, 1, 2 . . .} | Slot selector |
| Int      |          | {..., −2, −1, 0, 1, . . .} | Integer value |
| Bool     |          | true + false | Boolean value |
| Prim     | π        | {Val → Val} | Primitive function |

Figure 12: Semantic domain

Figure 13: The DFG of the `multifilt` application, as specified using the Preesm CAD tool
Fig. 16 describes the semantics of application, the most salient feature of the HoCL language. Rule EAppC deals with the application of closures and follows the classical call-by-value strategy (the closure body is evaluated in an environment augmented with the bindings resulting from binding the pattern to the value of argument). Rules EAppNP and EAppNF deal with the application of nodes. The former is used for partial application. The value resulting from the evaluation of the argument (which must be a graph location) is simply “pushed” on the list of supplied inputs. The latter describes the full application of a node. Here, a new box b is created and its inputs are connected to wires w, representing the arguments. The function cat used in rule EAppNF is trivially defined as $\text{cat(actor)} = \text{actor}$ and $\text{cat(Graph)} = \text{graph}$. For simplicity, the formulation of the rule EAppNF assumes that single values and tuples of size one are semantically equivalent\(^{18}\). Note that the outputs of the inserted box are left unconnected at this level. They will be connected when the result of the application is bound by the rule Binding described in Fig. 15.

4 IMPLEMENTATION

A prototype compiler, implementing the semantics described in the previous section has been written in OCaml and is available on Github [17]. The distribution includes a command-line compiler, turning HoCL source files into various dataflow graph representations, and a toplevel interpreter, supporting interactive building of dataflow graphs.

The command-line compiler currently supports four distinct backends: DOT, DIF, Preesm and SystemC.

The DOT backend produces graphical representations of the generated graphs in .dot format, to be visualized with the graphviz [5] set of tools. All the graph representations used in this paper have been produced by this backend from the corresponding programs.

The DIF backend produces representations in the Dataflow Interchange Format. DIF [9] provides a standard, textual notation for dataflow graphs aimed at fostering tool cooperation. By using DIF as an intermediate format, graphs specified in HoCL can be passed to a variety of tools for analysis, optimization and implementation.

The Preesm backend generates code for PREESM [13], an open source prototyping tool for implementing dataflow-based signal processing applications on heterogeneous multi/many-core embedded systems. Using this backend is illustrated in Sec. 5.

The SystemC backend generates executable SystemC code for the simulation of simple DDF (Dynamic DataFlow) and SDF (Synchronous DataFlow) graphs (for which the behavior of the actors is described in C or C++).

A short video illustrating the use of the toplevel interpreter is available online [16].

5 A COMPLETE EXAMPLE

In order to demonstrate the gain in abstraction and programmer’s productivity offered by the HoCL language, we consider a small DSP application consisting in applying in parallel a sequence of three filters on a single data stream and selecting the “best” output according to a given criterion. Apart from the fact that it’s typical of the kind of processing performed in the DSP domain, this application was chosen because we already had a working implementation, obtained with the Preesm [13] tool.

The dataflow graph, initially specified “by hand” using the Preesm GUI is depicted in Fig. 13. In this figure:

- gray boxes denote actors,
- orange boxes denote dedicated broadcasting nodes,
- blue triangle-shaped boxes denote parameter sources,
- black arrows denote data wires and
- dashed, blue arrows denote parameter wires.

Input data, generated by the src node, is passed, through the bcast node to three parallel chains of nodes. In the first chain (bottom), data goes first through filter $f_1$, then $f_2$ and finally $f_3$. In the second (middle), the order is $f_3$, then $f_1$ and finally $f_2$. In the third (top), it is $f_2$, $f_3$, $f_1$. The respective output data are finally given as input to the select node. Each filter node $f$ takes a parameter input named $p$. For simplicity, the value of this parameter has here been considered as constant for all filters. The select node also takes a parameter, named thr.
\[
E_0 = E, B_0 = B, W_0 = \emptyset \\
\forall i. 1 \leq i \leq n, E_{i-1}, B_{i-1}, W_{i-1} \vdash valdecl_i \Rightarrow E_i, B_i, W_i \\
\]

\[
E, B \vdash valdecl_1 \ldots valdecl_n \Rightarrow E_n, B_n, W_n \\
\]

(VDECLS)

\[
E, B \vdash pat = expr \Rightarrow E', B', W' \\
E, W \vdash val pat = expr \Rightarrow E \uplus E', B \uplus B', W \uplus W' \\
\]

(VDECL)

\[
E, B \vdash pat = expr \Rightarrow E', B', W' \\
\]

(BINDING)

**Figure 15: Semantics, part 2 (value declarations)**

\[
E, B \vdash exp_1 exp_2 \Rightarrow v, B', W' \\
\]

(EAPP)

\[
E, B \vdash exp_1 \Rightarrow \text{Clos}(\text{pat}, exp, E'), B_f, W_f \\
E, B \vdash exp_2 \Rightarrow v, B_a, W_a \\
\emptyset, \emptyset \vdash p \text{ pat}, v \Rightarrow E_p, B_p, W_p \\
E' \uplus E_p, B \vdash exp \Rightarrow v', B', W' \\
\]

(EAPPNP)

\[
E, B \vdash \text{exp}_1 \Rightarrow \text{Node}(\kappa, [id_1 \mapsto \ell_1, \ldots, id_{k-1} \mapsto \ell_{k-1}, id_k \mapsto \text{Unit}, \ldots, id_m \mapsto \text{Unit}], [id'_1, \ldots, id'_n], \eta), B_f, W_f \\
k < m - 1 \\
E, B \vdash \text{exp}_2 \Rightarrow \ell, B_a, W_a \\
n = \text{Node}(\kappa, [id_1 \mapsto \ell_1, \ldots, id_{k-1} \mapsto \ell_{k-1}, id_k \mapsto \ell, \ldots, id_m \mapsto \text{Unit}], [id'_1, \ldots, id'_n], \eta) \\
\]

(EAPPNF)

\[
E, B \vdash \text{exp}_1 \Rightarrow \text{Node}(\kappa, [id_1 \mapsto \ell_1, \ldots, id_{m-1} \mapsto \ell_{m-1}, id_m \mapsto \text{Unit}], [id'_1, \ldots, id'_n], \eta), B_f, W_f \\
E, B \vdash \text{exp}_2 \Rightarrow \ell_m, B_a, W_a \\
1 \notin \text{Dom}(B) \\
\forall j. 1 \leq j \leq m, k_j \notin \text{Dom}(W), w_j = (\ell_j, \text{Loc}(l, j)) \\
W' = [k_1 \mapsto w_1, \ldots, k_m \mapsto w_m] \\
b = \text{Box}([\text{cat}(\kappa), [1 \mapsto k_1, \ldots, m \mapsto k_m], [1 \mapsto \emptyset, \ldots, n \mapsto \emptyset])] \\
B' = [1 \mapsto b] \\
v' = (\text{Loc}(l, 1), \ldots, \text{Loc}(l, n)) \\
\]

(EAPPNF)

\[
E, B \vdash \text{exp}_1 \Rightarrow v', B_f \uplus B_a \uplus B', W_f \uplus W_a \uplus W' \\
\]

**Figure 16: Semantics, part 3 (application)**
Listing 5 gives a possible description of the graph depicted in Fig. 13 in HoCL.

```plaintext
| Line | HoCL Code |
|------|-----------|
| 1    | type f16; |
| 2    | node src in () out (o:f16); |
| 3    | node snk in (o: f16) out (); |
| 4    | node f1 |
| 5    | in (p: int param, i : f16) out (o:f16); |
| 6    | node f2 |
| 7    | in (p: int param, i : f16) out (o:f16); |
| 8    | node f3 |
| 9    | in (p: int param, i : f16) out (o:f16); |
| 10   | node select |
| 11   | in (thr: int param, |
| 12   | i1 : f16, i2 : f16, i3 : f16) |
| 13   | out (o : f16); |
| 14   | graph top |
| 15   | in (p: int param=2, thr : int param=128) |
| 16   | out () |
| 17   | fun |
| 18   | val fs = [ f1 p; f2 p; f3 p ] |
| 19   | val chain s x = x x pipe (shuffle s fs) |
| 20   | val sel c1 c2 c3 x = |
| 21   | select thr (c1 x) (c2 x) (c3 x) |
| 22   | val o = src ▼ sel (chain [0;1;2]) |
| 23   | (chain [1;2;0]) |
| 24   | (chain [2;0;1]) |
| 25   | end; |
```

Listing 5: A description of the graph depicted in Fig. 13 in HoCL.

Lines 3–14 declare the involved atomic actors. It has been assumed here that all processed data has type f16 (a shorthand for the fix16 type used in the original implementation). Both the p parameter of the f1, f2 and f3 actors and the thr parameter of the select actors are here declared as int.

The graph itself is described in the top declaration, lines 16–26. The global parameters p and thr, with a default value (here arbitrarily set to 2 and 128), are declared as input parameters of this graph.

The value fs, defined at line 20, is a list made of the three filters, with their supplied parameter.

The wiring function chain, defined at line 21, is used to build the horizontal chains of filters depicted in Fig. 13. It takes a list of integers s and a input wire x and connects x to the sequence of nodes obtained by permuting the elements of the fs list. Permutation is done by the shuffle function and chaining by the pipe function. The pipe function has been introduced in Sec. 2.3. The shuffle function has type

\[
\text{shuffle} : \text{int list} \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}
\]

and is defined as follows in HoCL:

\[
\text{val rec} \quad \text{shuffle} \; \text{ks} \; \text{xs} = \text{match} \; \text{ks} \; \text{with} \\
[\] \rightarrow [] \\
| \text{k :: ks} \rightarrow \text{nth k xs :: shuffle} \; \text{ks} \; \text{xs}
\]

where nth is the function returning the \(k^{\text{th}}\) element of a list. Informally:

\[
\text{shuffle} \; [k_1, \ldots, k_n] \; [x_1, \ldots, x_n] = [x_{k_1}, \ldots, x_{k_n}]
\]

These two functions are included in the HoCL standard library.

The wiring function `sel`, defined at lines 22–23, encodes the main graph pattern: it applies its arguments c1, c2 and c3 in parallel to its argument x and routes the three results to the select actor.

The top level graph is built, at lines 24–26 by applying the `select` function to the three chains of filters, themselves obtained by applying the `chain` function to the corresponding lists of permutation indices.

Writing the program in Listing 5 took less than 15 minutes and the resulting dataflow graph was obtained immediately. By contrast, describing the initial version of the graph using the Preesm GUI took more than one hour. This includes the definition of the node interfaces, the placement of the nodes on the canvas and, above all, the manual, cumbersome, drawing of the connections between the nodes. This represent a four time increase in productivity. Moreover, and most importantly, whereas it’s straightforward, with the HoCL formulation, to modify the graph (adding or modifying the number of chains, changing the permutation choices, etc.) to test new application configurations, this task is much more tedious and error-prone with the purely GUI-based representation.

6 RELATED WORK

The idea of describing dataflow graphs using a functional programming language goes back to the VAL [12] and SISAL [6] dataflow languages. Since then, it has been exploited in languages such as Lava [2], CLaSH [7] (for designing digital circuits), Fran [4] (in the context of functional reactive programming) or in synchronous programming languages such as Lustre [8] or Lucid Synchronè [14].

Like HoCL, these functional programming languages offer the possibility to encode graph patterns using higher order functions. But, because their goal is to assign both a static and a dynamic semantics to programs – in other words to describe not only the topology of dataflow graphs but also their behavior – they do not really meet the needs of programmers when the goal is simply, and pragmatically, to avoid the “manual”, GUI-based, specification of large dataflow graphs, to be passed further to existing analysis and implementation tools. Hence the need for a simple coordination language, acting as a front-end for such existing tools, which is precisely the goal of HoCL.

In their seminal paper on dataflow process networks [11], Lee and Parks noted that the thé replication of a given actor on parallel streams can be denoted using the map higher order function. But no attempt was made to generalize the correspondence between functional expressions and graph structures beyond the particular pattern captured by the map HOF. The work of Sane et al. [15] is more closely related to ours. They proposed an extension to the DIF [9] notation supporting the use of so-called topological patterns for explicit and scalable representation of regular structures in DFGs. The definition of these patterns explicitly relies on a indexing mechanism for nodes and edges. HoCL is more general in the sense that any dependency pattern can be described, and not only those
based on explicit indexing. Moreover, in the work described in [15], patterns are built-in and the set of available patterns is therefore fixed. By contrast, patterns are first class values in HoCL and can therefore be defined directly by the programmer, \textit{within the language}.

The HoCL language was inspired, in part, by the network description language used in the CAPH language for dataflow-based high-level synthesis [19]. Some design decisions were also motivated by conclusions of a retrospective assessment of the CAPH project reported in [20]. The idea of a language playing the role of a front-end to existing analysis and implementation tools, in particular, can be viewed as an answer to the “invasiveness” problem mentioned in [20].

7 CONCLUSION

The design and development of the HoCL language started recently and this paper should be viewed more as a draft specification than as a definite language reference.

Work is undergoing for reformulating in HoCL complex DSP applications, initially developed with tools using lower-order specification formalisms, such as Ptolemy, DIF or Preem, in order to further assess the gain in expressivity and in the effort required by the specification of the input dataflow graph.

An important issue which remains to be investigated, in particular, is whether MoC-specific features can be “injected” into the language without compromising its use as a general and MoC-agnostic coordination language. The current version, for example, allows actor ports to be annotated with production and consumption rates, to be used by backends supporting an SDF semantics (DIF or Preem for example). It is still uncertain, however, whether such an annotation-based approach is always feasible or whether some specific MoCs may require deeper changes to the syntax or semantics of the language itself.

REFERENCES

[1] B. Bhattacharya and S. S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing 49, 10 (2001), 2408–2421. https://doi.org/10.1109/78.950795
[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 2001. Lava: Hardware Design in Haskell. ACM SIGPLAN Notices 36 (05 2001). https://doi.org/10.1145/289423.289440
[3] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. Proc. IEEE 91, 1 (2003), 127–144. https://doi.org/10.1109/JPROC.2002.865829
[4] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In International Conference on Functional Programming. http://conal.net/papers/icfp97/
[5] J. Ellison, E. Ganster, Y. Hu, and S. North. 2021. Graphviz Visualization Software. http://www.graphviz.org
[6] John Feo. 2011. Staal. Springer US, Boston, MA, 1827–1831. https://doi.org/10.1007/978-0-387-09766-4_118
[7] M. Gerards, C. Baaij, J. Kuper, and M. Kooijman. 2011. Higher-Order Abstraction in Hardware Descriptions with C!aSH. In 2011 14th Euromicro Conference on Digital System Design, 495–502. https://doi.org/10.1109/SDSD.2011.69
[8] N. Hallbwachs, P. Caspi, P. Raymond, and D. Pileau. 1991. The synchronous dataflow programming language LUSTRE. Proc. IEEE 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300
[9] Chia-Jui Hsu, Fuat Kececi, Ming-Yung Ko, Shahroz Shaheen, and Shuvra S. Bhattacharyya. 2004. DIF: An Interchange Format for Dataflow-Based Design Tools. In Computer Systems: Architectures, Modeling, and Simulation. Andy D. Finenlent and Stamatis Vassiliadis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 423–432.
[10] National Instruments. 2021. LabVIEW. Retrieved April 2021 from https://www.ni.com/labview
[11] E. A. Lee and T. M. Parks. 1995. Dataflow process networks. Proc. IEEE 83, 5 (1995), 773–801. https://doi.org/10.1109/5.381846
[12] James R. McGraw. 1982. The VAL Language: Description and Analysis. ACM Trans. Program. Lang. Syst. 4, 1 (Jan. 1982), 44–82. https://doi.org/10.1145/357153.357157
[13] Maxime Pellet, Karol Denis, Julien Heulot, Clément Guy, Jean François Nesz, and Sahedde责任心 Aribdi. 2014. PREESM: A Dataflow-Based Prototype Tools Framework for Simplifying Multicore DSP Programming. In EDERC Italy. 36. https://hal.archives-ouvertes.fr/hal-01059313
[14] Marc Pouzet. 2006. Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI. Distribution available at: www.lri.fr/~pouzet/lucid-synchrone.
[15] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya. 2010. Scalable representation of dataflow graph structures using topological patterns. In 2010 IEEE Workshop On Signal Processing Systems. 13–18. https://doi.org/10.1109/SIPS.2010.5624821
[16] J. Sérot. 2020. Demonstration of the HoCL toplevel interpreter. Retrieved April 2021 from https://www.youtube.com/watch?v=WQSacRxYgGM
[17] J. Sérot. 2020. HoCL, A Higher Order dataflow Coordination Language. Retrieved April 2021 from https://github.com/jserot/hocl
[18] J. Sérot. 2021. The HoCL documentation. Distribution available at: github.com/jserot/hocl
[19] J. Sérot and F. Berry. 2014. High-Level Dataflow Programming for Reconfigurable Computing. In 2014 International Symposium on Computer Architecture and High Performance Computing. 72–77. https://doi.org/10.1109/SAHC-PAD.2014.18
[20] J. Sérot and F. Berry. 2019. The CAPH Language, Ten Years After. In Embedded Computer Systems: Architectures, Modeling and Simulation (19th International Conference, SAMOS 2019). 336–347. https://doi.org/10.1007/978-3-030-27562-4_24