HoCL v1.2
User manual

J. Sérot

Chapter 1

Using the HoCL compiler

The HoCL compiler is invoked with a command like :

```
hoclc [options] file1 ... filen
```

where `file1`,...,`filen` are the names of the file(s) containing the source code (by convention, these files should be suffixed `.hcl`).

The complete set of options is described in Chap. 3.

The set of generated files depends on the selected target. The output file `hocl.output` contains the list of the generated file.

1.1 Generating graphical representations

Example :

```
hoclc -dot main.hcl
```

The previous command generates a graphical representation of the graph(s) described in file `main.hcl` in `.dot` format[1]. Each toplevel graph (defined as `graph ... end`) and refined node (defined as `node ... struct ... end` or `node ... fun ... end`) gives a separate `.dot` file.

1.2 Generating XDF representations

Example :

```
hoclc -xdf main.hcl
```

The previous command generates a graphical representation of the graph(s) described in file `main.hcl` in `.xdf` format.

TODO: To be documented

1.3 Generating DIF representations

Example :

```
hoclc -dif main.hcl
```

The previous command generates a graphical representation of the graph(s) described in file `main.hcl` in `.dif` format.

TODO: To be documented

[1] `http://www.graphviz.org`.

1.4 Generating SystemC code

Example :

```
hoclc -systemc main.hcl
```

The previous command generates the SystemC code corresponding the dataflow graph described in file `main.hcl`. The following files are written :

- a pair of files `x_act.h`, `x_act.cpp` for each actor declared in the source file, containing respectively the interface and the implementation of the actor,
- a file `x_gph.h` for each defined graph (either as a toplevel graph or a refined node),
- a file `main.cpp` containing the toplevel description and driver for simulation.

The produced files can then compiled using the standard SystemC toolchain. When compiling (resp. linking) the HoCL-specific headers (resp. library) must be available[2]. Examples of `Makefiles` are provided in the `examples` sub-directories of the distribution.

[2] These headers and library are located in `$HOCL/lib/systemc` where `$HOCL` points to the installation directory of the HoCL toolset.

Chapter 2

Using the HoCL toplevel interpreter

The HoCL toplevel interpreter is launched by invoking the compiler with the `-interactive` option :

```
hoclc -interactive [other_options]
```

The interpreter reads and interprets *toplevel phrases* in loop, updating a global environment recording the type and value of each defined symbol and the state of a single dataflow graph, here called the *graph under construction* (GUC). The notion of graph hierarchy is not supported by the interpreter.

Each toplevel phrase is terminated by a semi-colon and can be

- a **type declaration**, introducing a new type name. *E.g.* :

  ```
  type tau;
  ```

- a **node declaration**, introducing a new node. Such nodes are here viewed as atomic, opaque actors. *E.g.* :

  ```
  node foo in (i:  tau) out (o:  int);
  ```

- an **input or output declaration**, introducing a graph input or output and adding it to the GUC. *E.g.* :

  ```
  input i1:  int;
  output o2:  tau;
  ```

- a **value declaration**, defining a new value, just like with the batch compiler, and possibly updating the GUC. *E.g.* :

  ```
  val o2 = foo i1;
  val twice f x = f (f x);
  ```

- a **directive**, for modifying the behavior of the interpreter. The `#help` directive gives the list of all available directives. *E.g.* :

  ```
  #display;
  #verbose 2;
  ```

A useful directive is `#dump_dot`. Invoking it puts the interpreter in *dump mode*. When in this mode, the interpreter writes a description of the GUC in a file after evaluating each phrase. This description is written in `.dot` format[1]. This file can be monitored by a DOT viewer such as the Graphviz application[2], thus providing some kind of "interactive" graph building mechanism. To do this

1. launch the DOT viewer application (either from an application menu or a terminal),
2. launch the HoCL compiler in interactive mode : `hoclc -interactive`[3],
3. enter the `#dump_dot;` directive,
4. from your DOT viewer, open the file `/tmp/hocl_top.dot` (this should display an empty graph),
5. enter phrases in the HoCL terminal; any modification in the GUC should be reflected immediately in the DOT viewer.

A sequence of snapshots illustrating this is given in Fig. 2.1. A video capture is available from the github repository[4].

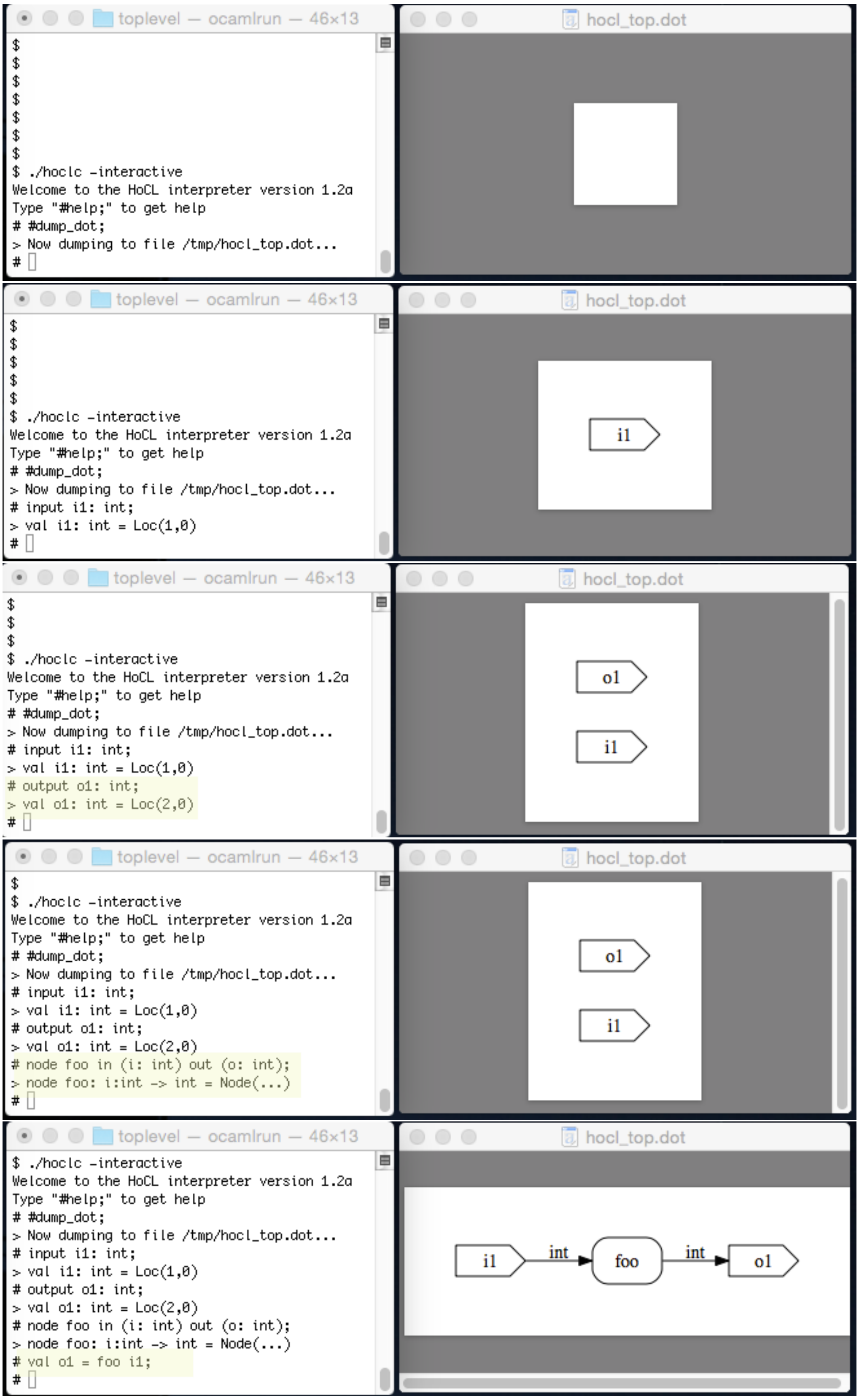
Invoking the `#dump_dot` directive when already in *dump mode* puts the interpreter back in normal mode.

[1] This file is named `/tmp/hocl_top.dot` by default. This name can be changed with the `#dot_file` directive.

[2] www.graphviz.org

[3] Assuming the `hoclc` command is in your `PATH` environment.

[4] `github.com/jserot/hocl`.



```
$
$
$
$
$
$
$
$ ./hoclc -interactive
Welcome to the HoCL interpreter version 1.2a
Type "#help;" to get help
# #dump_dot;
> Now dumping to file /tmp/hocl_top.dot...
# 
```

```
$
$
$
$
$
$ ./hoclc -interactive
Welcome to the HoCL interpreter version 1.2a
Type "#help;" to get help
# #dump_dot;
> Now dumping to file /tmp/hocl_top.dot...
# input i1: int;
> val i1: int = Loc(1,0)
# 
```

```
$
$
$
$ ./hoclc -interactive
Welcome to the HoCL interpreter version 1.2a
Type "#help;" to get help
# #dump_dot;
> Now dumping to file /tmp/hocl_top.dot...
# input i1: int;
> val i1: int = Loc(1,0)
# output o1: int;
> val o1: int = Loc(2,0)
# 
```

```
$
$ ./hoclc -interactive
Welcome to the HoCL interpreter version 1.2a
Type "#help;" to get help
# #dump_dot;
> Now dumping to file /tmp/hocl_top.dot...
# input i1: int;
> val i1: int = Loc(1,0)
# output o1: int;
> val o1: int = Loc(2,0)
# node foo in (i: int) out (o: int);
> node foo: i:int -> int = Node(...)
# 
```

```
$ ./hoclc -interactive
Welcome to the HoCL interpreter version 1.2a
Type "#help;" to get help
# #dump_dot;
> Now dumping to file /tmp/hocl_top.dot...
# input i1: int;
> val i1: int = Loc(1,0)
# output o1: int;
> val o1: int = Loc(2,0)
# node foo in (i: int) out (o: int);
> node foo: i:int -> int = Node(...)
# val o1 = foo i1;
# 
```

Figure 2.1: A sequence of snapshots illustrating the behavior of the toplevel interpreter

Chapter 3

Compiler options

General options

-stdlib	set location of the standard library file
-no_stdlib	do not use the standard library
-interactive	launch the toplevel interpreter instead of compiling files
-prefix	set prefix output file names (default is main source file basename)
-target_dir	set target directory for generated files (default is current directory)
-dump_tenv	dump builtin typing environment (for debug only)
-dump_typed	dump typed program (for debug only)
-dump_senv	dump builtin static environment (for debug only)
-dump_ir	dump intermediate representation (for debug only)
-dump_boxes	dump static representation of boxes
-insert_bcasts	insert broadcast boxes
-version	print version of the compiler
–v	print version of the compiler

DOT-specific options

-dot	generate .dot representation of the program
-dot_rank_dir	set rank direction for DOT output graph (default: LR)
-dot_unlabeled_edges	do not annotate graph edges
-dot_no_io_rates	do not annotate ports with resp. rates
-dot_show_indexes	print box and wire indexes
-dot_slotted_boxes	print boxes with i/o slots

SystemC-specific options

-systemc	activate the SystemC backend
-sc_stop_time	stop after n ns
-sc_clock_period	set clock period (ns) (default: 10)
-sc_default_fifo_capacity	set default fifo capacity (systemc only) (default: 256)
-sc_trace	set trace mode
-sc_dump_fifos	dump fifo contents
-sc_trace_fifos	trace fifo usage in .vcd file
-sc_dump_fifo_stats	dump fifo usage statistics after run
-sc_fifo_stats_file	set file for dumping fifo statistics (default: fifo_stats.dat)

XDF-specific options

-xdf	generate .xdf representation of the network
-xdf_package	set package name for the generated XDF code

DIF-specific options

-dif	generate .dif representation of the program