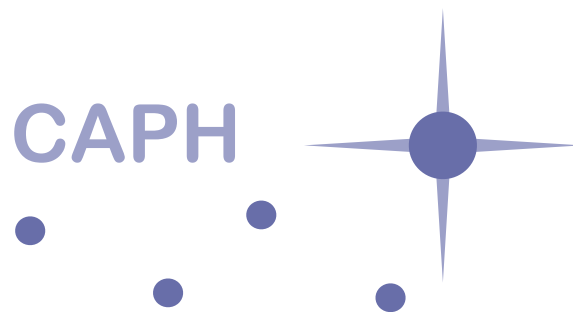
CAPH Reference Manual - 2.9

J. Sérot



Contents

1 Introduction **4**
- 1.1 Motivation, goals and principles . . . 5
- 1.2 Language principles . . . 6
- 1.3 Language Structure . . . 7
  - 1.3.1 The network language . . . 7
  - 1.3.2 The actor Language . . . 10
- 1.4 Tools and design flow . . . 11

2 CAPH Overview **13**
- 2.1 Language structure . . . 13
- 2.2 Types . . . 13
  - 2.2.1 Scalar types . . . 13
  - 2.2.2 Structured types . . . 14
  - 2.2.3 Translation to SystemC and VHDL types . . . 14
- 2.3 CAPH expression language . . . 16
  - 2.3.1 Constants . . . 17
  - 2.3.2 Variables . . . 17
  - 2.3.3 Function application . . . 17
  - 2.3.4 Conditionals . . . 17
  - 2.3.5 Local Declarations . . . 18
  - 2.3.6 Type coercion . . . 18
  - 2.3.7 Attributes . . . 19
- 2.4 CAPH declaration language . . . 20
  - 2.4.1 Type declarations . . . 20
  - 2.4.2 Value declarations . . . 23
  - 2.4.3 I/O declaration . . . 24
  - 2.4.4 Actor declarations . . . 24
  - 2.4.5 Polymorphism . . . 34
  - 2.4.6 Higher order actors . . . 39
  - 2.4.7 Network declarations . . . 42
- 2.5 A complete example . . . 59

3 Syntax **64**

4 Core Abstract Syntax **71**

5 Typing **73**
5.1 Notations . . . 74
5.2 Typing rules . . . 75
5.2.1 Programs . . . 75
5.2.2 Value declarations . . . 75
5.2.3 Actor declarations . . . 76
5.2.4 Expressions . . . 78
5.2.5 IOs . . . 79
5.2.6 Network declarations . . . 80
5.2.7 Type expressions . . . 81

6 Static semantics **83**
6.1 Programs . . . 86
6.2 Value declarations . . . 87
6.3 Expressions . . . 87
6.4 Actor declarations . . . 88
6.5 Stream declarations . . . 89
6.6 Network declaration . . . 90
6.6.1 Network expressions . . . 91
6.6.2 Recursive network definitions . . . 92

7 Dynamic semantics **95**
7.1 Semantic domain . . . 95
7.2 Programs . . . 96
7.2.1 Conversion from boxes to processes . . . 96
7.2.2 Conversion from wires to channels . . . 98
7.3 Processes . . . 98
7.3.1 Identifying and marking active processes . . . 99
7.3.2 Individual process execution . . . 101

8 Model of Computation **104**
8.1 Dataflow Process Networks . . . 104
8.2 The Caph Process Network model . . . 107
8.2.1 Token values . . . 107
8.2.2 Rule patterns . . . 108
8.2.3 Patterns . . . 108
8.2.4 Pattern matching . . . 108
8.2.5 Example . . . 108
8.2.6 Classification of CAPH actors . . . 108
8.2.7 Static computation of FIFO sizes . . . 115

9 Intermediate representation **119**
9.1 Network generation . . . 119
9.2 Behavioral description . . . 119

10 Using the caph compiler **123**
10.1 Generating a graphical representation of the program . . . 123
10.2 Running the simulator . . . 123
10.3 Generating SystemC code . . . 124
10.4 Generating VHDL code . . . 124

10.5 File I/O 125
10.5.1 Port I/O 126
10.5.2 File globbing 126
10.5.3 File IO when using the VHDL backend 127
10.5.4 Reading and writting image files 128
10.5.5 Blanking 128
10.6 File inclusion 129
10.7 Passing command-line options to programs 129
10.8 Conditional compilation 130
10.9 Adjusting FIFO size 130
10.10 Dumping box FSMs 132
10.11 The `caphmake` utility 132

11 The CAPH standard libraries **136**

12 Foreign function interfacing **154**

13 Compiler options **157**

14 Implementation of variant types **160**

A Writing your own FIFO model **163**

B The `txt2bin` command **164**

C The `bin2txt` command **166**

D The `pgm2txt` command **168**

E The `txt2pgm` command **169**

F The `pgm2bin` command **170**

G The `bin2pgm` command **172**

H The `mkdcimg` command **173**

I The `mkconv` command **175**

J The `caphmake` command **177**

Chapter 1

Introduction

This document describes the CAPH programming language. CAPH[1] is a domain-specific, high-level language for programming FPGAs. CAPH is based upon the *dataflow process network* model of computation [4] and produces implementations relying on a pure *data-driven* execution model. The CAPH compiler is able to generate synthetizable VHDL code from high-level descriptions of programs as networks of interconnected dataflow actors.

This report is structured as follows: the remainder of this chapter provides motivation and general background; Chapter 2 is an overview of the CAPH language design, including informal descriptions of the expression, network and actor sub-languages. The full concrete syntax is given in chapter 3 and the abstract syntax of the core language in chapter 4. Chapter 5 gives the formal typing rules for CAPH programs. Chapter 6 gives the formal static semantics, *i.e.* the interpretation of CAPH programs as *data-flow graphs*. Chapter 7 gives the formal dynamic semantics, which provides a way to *simulate* CAPH programs. Chapter 9 describes the intermediate representation of CAPH programs used as an input for the back-end code generators. Chapter 10 describes, pragmatically, how to use the compiler in order to obtain graphical representations of programs, simulate them or invoke the SystemC or VHDL backend in order to generate code. Chapter 11 gives the contents of some "standard libraries". Chapter 12 describes the mechanism by which existing C or VHDL functions can be used by CAPH programs. Chapter 13 is a summary of compiler options. Chapter 14 is a short, technical, overview of how *variant types* (described in chapter 2) are implemented in SystemC and VHDL.

[1] CAPH stands for *Caph just Ain't Plain HDL*. *Caph* is also the name of the second bright star in the constellation of Cassiopeia.

1.1 Motivation, goals and principles

The design of the CAPH language was guided by the following motivations and general principles:

Specificity. Although most of its underlying concepts are very general – and could indeed be applied to a large variety of programming languages targetting either software or hardware –, the CAPH language has been primarily designed to program FPGAs. This deliberate orientation allowed us to make clear and argumented choices regarding the set of supported features. Other languages relying on the same model of computation (MoC) – such as CAL [5] for example – are less specific, targeting both software and hardware implementations, and sometimes offer a richer set of features. But this richness comes at the price of ambiguity since it's frequent that not all features are supported in hardware implementations. Moreover, the set of supported features is not always clearly documented and must often be defined by tedious, repeated experiments. Our approach is more pragmatic, if less ambitious: if it can be described, it can be implemented.

Formal basis. We strongly adhere to the idea that any decent programming language should be based upon formal semantics, describing in a complete and unambiguous manner the meaning of programs. The essentially *functional* orientation of CAPH greatly eases the writing of such semantics. It also makes it possible to describe formally, and in a platform-independant way, the transformation process by which the high level language source code is turned into low-level VHDL code, a highly desirable property in our case, since it allows reasoning about this process (for certification and optimization purposes for example). Another advantage is the ability to derive (in a systematic way) a reference interpreter for the language. Such a reference interpreter can then be used latter to assess the correctness of the results produced by generated low-level code.

Minimum distance between the programming model and the execution model. This is the key idea for being able to produce efficient low-level harware solutions from high-level descriptions while keeping adherence to the second principle. Abstraction and efficiency being generaly perceived as contradictory requirements, this principle is often interpreted as : keep the the abstraction low at source level. Fortunately, this is not the case with the dataflow programming and execution models CAPH is relying on. A large part of this is due to the natural affinity between the dataflow MoC and the concepts used in purely functional programming languages (absence of side-effects, polymorphic type systems, higher-order constructs).

Beside these general principals, CAPH was also developped with a set of more technical / pragmatical goals in mind :

Strict separation of concerns between computation and communication. This is reflected by the structure of the language, which embeds an *expression language* (for describing computations) within a *declaration language* (for describing the global structure of the program) on the one hand, and provides two separate sub-languages for describing the behavior of actors and the communication network by which these actors interact, on the other hand.

Composability, modularity. This refers to the ability to build large, complex applications from smaller, simpler ones. This is naturally supported by the dataflow model of computation (no shared variables and hence no hidden dependencies) and the purely functional coordination language (referential transparency).

Reuse. Within the language, this is supported by *higher-order* constructs. At the actor level, higher-order actors can be used to encapsulate recurrent patterns of *computations*. At the network level, higher-order functions can be used to encapsulate recurrent patterns of *communication*. CAPH also offers the ability to import legacy existing code (in VHDL for synthesis, in C/SystemC for simulation).

Arbitrary data structuring. By this we mean the ability to encode values – and, jointly, to describe actors operating on these values – having an arbitrarily complex data structure. In particular,

the language should support the description of operations on non-regular and/or or variable-sized data structures (lists or trees, as opposed to fixed-sized arrays, for instance). In other words, the target applications should not be limited to regular signal or image processing. In CAPH, this is provided by separating the tokens exchanged by actors into two classes : *data tokens* (carrying actual values) and *control tokens* (acting as structuring delimiters).

1.2 Language principles

CAPH is based upon the *dataflow process network* model of computation [4]. Applications are described networks of computational units, called *actors*, exchanging *streams* of tokens through FIFO channels. Interaction between actors is strictly limited to token exchange through channels, so that the behavior of each actor can be completely described in terms of actions performed on its inputs to produce outputs (no side effect, strictly local control).

We will illustrate this model with a very simple example, involving four basic actors. These actors are depicted Fig. 1.1. Actor `INC` (resp. `DEC`) adds (resp. substracts) 1 to each element of its input stream. Actor `MUL` performs point-wise multiplication of two streams. Actor `DUP` duplicates its input stream[2]. Now, if we connect these four actors to build the network depicted in Fig 1.2, this network computes $f(x) = (x+1) \times (x-1)$ for each element x of its input stream. I.e. if the input stream i is `1,2,3,...`, then the output stream o will be `0,3,8,...`

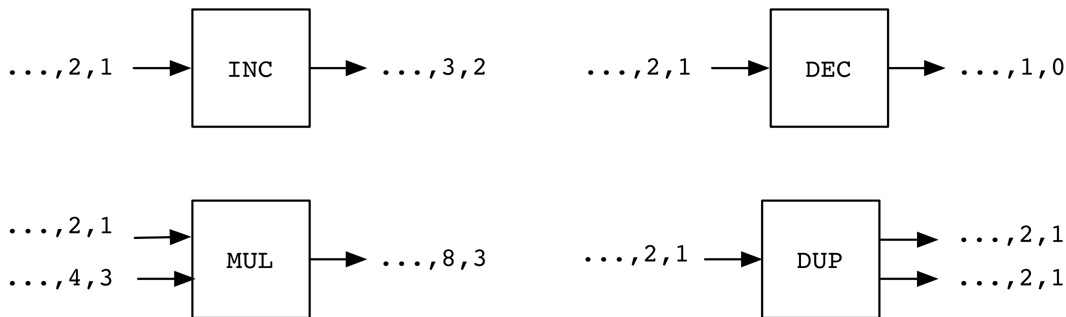


Figure 1.1: Four basic actors

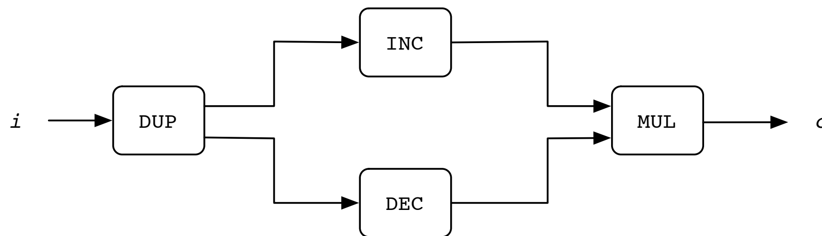


Figure 1.2: A dataflow process network

[2]In Fig. 1.1, streams are denoted (ordered) from right to left; for example, the actor `ADD` first processes the token `1`, then the token `2`, *etc.* Since streams are potentially infinite, their end is denoted "`...`". However, when describing actors *textually*, streams will be denoted from left to right; for example: `ADD:1,2,... = 2,3,...`

Such a model of computation is indeed very general and can be interpreted in different ways, depending in particular in

- the kind of behavior that can be assigned to actors (purely functional, stateful, firing semantics, . . .),
- the exact nature of channels (single place buffer / FIFO, bounded / unbounded, . . .),
- how networks are described.

In CAPH

- the behavior of actors is specified using a set of *transition rules* using *pattern matching* and actors are stateful,
- channels are bounded FIFOs,
- networks are described in a *implicit* manner, using a set of *functional equations.*

These aspects are detailed in the next section.

1.3 Language Structure

In common with other coordination language approaches such as Hume [6], CAPH takes a layered approach (see Fig. 1.3).

The outermost (*declaration*) layer declares types, global values (constant and functions), I/O streams, actors and network-level objects (wires and wiring functions).

The innermost (*expression*) layer is a small, purely functional language used for describing values and computations. This language is used both for defining the values assigned to global constants and functions and the values computed by actors.

At an intermediate level, two sub-languages can be distinguished : an *actor* sub-language, used for describing the interface and the behavior of actors, and a *network* sub-language, used for describing the structure of the dataflow process network. Both languages are functionnaly-based,

1.3.1 The network language

The CAPH *network language* is used for describing the structure of the dataflow process network describing the program. It is purely functional, polymorphic and supports higher-order functions. It is largely inspired from previous work on the FGN system [10].

The basic unit of coordination is the *box.* A *box* is an instance of an actor, viewed abstractly, *i.e.* only retaining its interface (parameters and input/output ports). The network language is responsible for describing the *wiring* of these boxes to form the dataflow process network corresponding to the program to be implemented (including the connections to the external devices).

The basic concept is that the network of actors is actually a dataflow graph (DFG) and that a DFG can be described by means of purely functional expressions [11].

Consider for example, the minimal network depicted in Fig. 1.4, in which `inc` is an actor with one input and one output, `i` an input stream and `o` an output stream[3]. It can be described with the following CAPH declaration :

```
net o = inc i;
```

This declaration

[3]This figure has been produced with the CAPH graph visualizer. Actors are drawn as rectangular boxes and I/O streams as triangles.

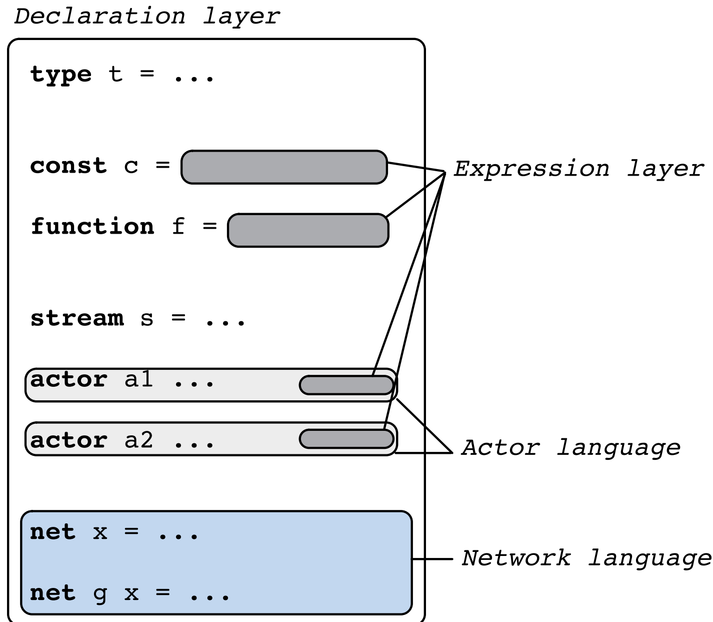


Figure 1.3: Caph language structure

1. instanciates the *inc* actor, i.e. creates a box in the network,
2. connects the network stream input `i` to the input of this box,
3. connects the network stream output `o` to the output of the `inc` box.

Usage of `net` declarations is of course not limited to binding network outputs. They are generally used to bind the output(s) of a box, in order to reuse it (them) in subsequent declarations, thus "wiring" the network. Such values are therefore called "wires" in the network language. A minimal example for this is given in Fig. 1.5, where two instances of the `inc` actor are wired together.

Note that the same network could have been described without naming the intermediate wire, by writing just : `net o = inc (inc i);`

The network depicted in Fig. 1.2 can be described with the following CAPH declarations[4], showing in particular how multiple outputs of an actor can be bound, using a tuple value :

```
net (i1,i2) = dup i;
net o = mul (inc i1, dec i2);
```

[4]Provided that the `dup`, `inc`, `dec` and `mul` actors have been previously declared with the right interface and that `i` and `o` designate the input and output of the network.

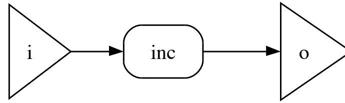


Figure 1.4: A minimal actor network

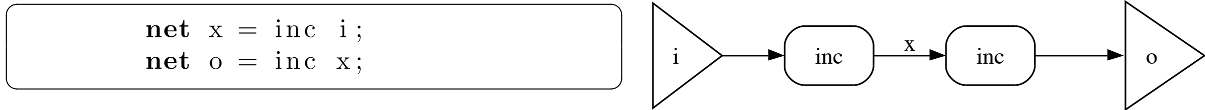


Figure 1.5: A small actor network

Compared to other textual or graphical network languages, this notation offers a significantly higher level of abstraction. In particular it saves the programmer from having to explicitly describe the wiring of channels between actors, a tedious and error-prone task. Moreover, ill-formed networks and inconsistent use of actors can be readily detected using a classical Hindley-Milner polymorphic type-checking phase.

Another advantage of "encoding" data-flow networks in a functional language is the ability to define reusable, polymorphic *graph patterns* in the form of higher-order functions, which offers an easy and safe compositional approach for building larger applications from smaller ones.

For example, the network of Fig 1.2 could also have been described with the following declarations, in which the `diamond` function *encapsulates* the diamond-shaped graph pattern examplified here :

```
net diamond (left,top,bottom,right) x =
  let (x1,x2) = left x in
  right (top x1, bottom x2);

net o = diamond (dup,inc,dec,mul) i;
```

The `diamond` function is called a *wiring function* in the CAPH network language. From a functional perspective, this is a *higher-order* function, i.e. a function taking other function(s) as argument(s). Once defined, such a function can be reused to freely instanciate graph patterns. For example, the network depicted in Fig 1.6, in which the "diamond" pattern is instanciated at two different hierarchical levels, can be simply described with the following declaration :

```
net o = diamond (dup, inc, diamond (dup,inc,dec,mul), mul) i;
```

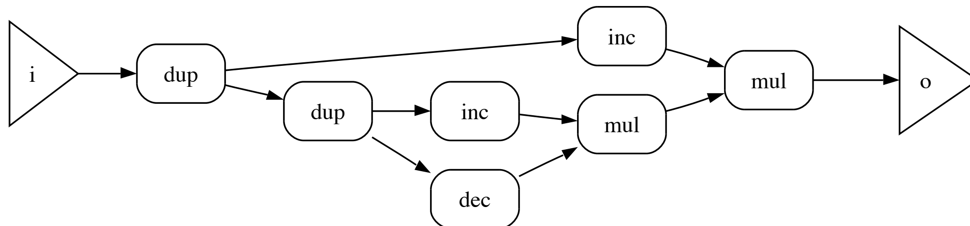


Figure 1.6: A hierachical network

1.3.2 The actor Language

The CAPH *actor language* is used to describe the behavior of individual dataflow actors. This is done using a set of *transition rules*, fired according to a generalized pattern-matching mechanism. The actions performed by each rule are described using a purely functional, primitive recursive language with a strict semantics.

A complete description of the actor language will be given in the next chapter. In this section, we will just introduce it using a simple example. Let us consider the actor `merge` described in Fig. 1.7. This actor takes two streams of tokens as inputs and produces one stream of tokens, taking input alternately from the first and second output. Its behavior description in CAPH is given on the left. It has two inputs and one output, all of type `int`. It also uses a local variable, `s`, of type `bool`. Its behavior is specified using a set of rules defined under the `rules` keyword. Each rule consists of a set of *patterns* refering to inputs or local variables and a corresponding set of *expressions*, refering to outputs or local variables. The first (resp. second) rule says : If `s` is `'false'` (resp. `'true'`) and a value `v` is available on input `i1` (resp. `i2`) then read[5] this value, write it to output `o` and set `s` to `'true'` (resp. `'false'`).

Fig. 1.8 gives the description in CAPH of the four actors introduced in Fig. 1.1.

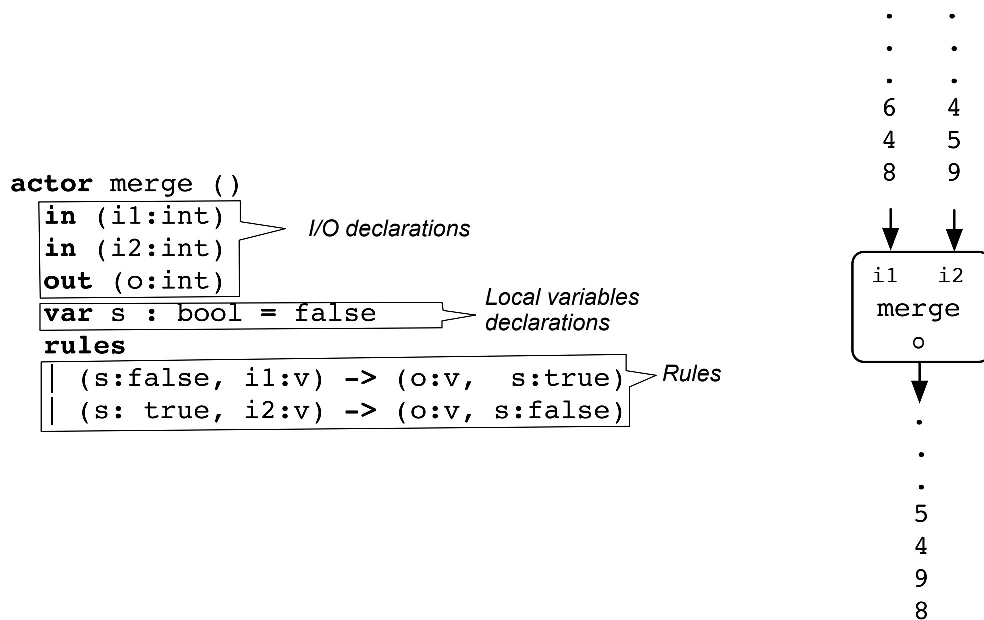
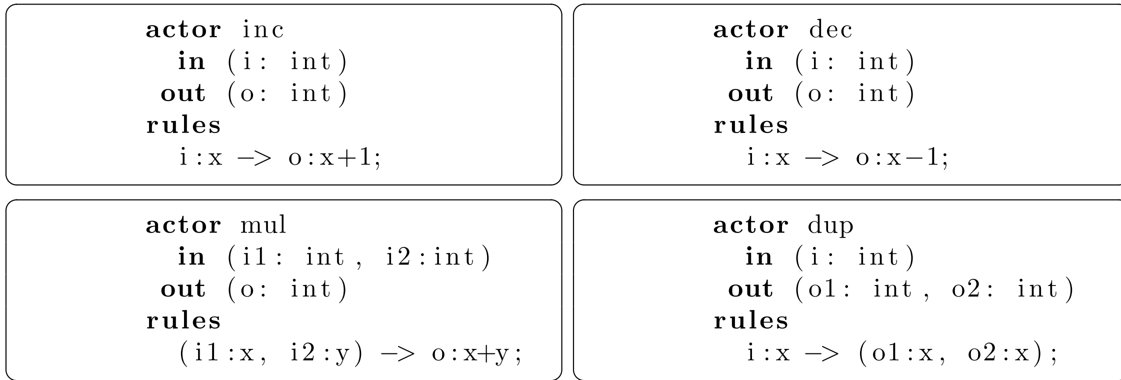


Figure 1.7: An example of actor description in CAPH

[5]Pop the connected channel.



```
actor inc
  in (i: int)
 out (o: int)
rules
  i:x -> o:x+1;
```

```
actor dec
  in (i: int)
 out (o: int)
rules
  i:x -> o:x-1;
```

```
actor mul
  in (i1: int, i2:int)
 out (o: int)
rules
  (i1:x, i2:y) -> o:x+y;
```

```
actor dup
  in (i: int)
 out (o1: int, o2: int)
rules
  i:x -> (o1:x, o2:x);
```

Figure 1.8: CAPH description of the four actors introduced in Fig.

1.4 Tools and design flow

The current tool chain supporting the CAPH language is sketched on Fig. 1.9. It comprises a graph visualizer, a reference interpreter and compiler producing both SystemC and synthetizable VHDL code.

The **graph visualizer** produces representations of the actor network in the `.dot` format [8] for visualisation with the GRAPHVIZ suite of tools [1].

The **reference interpreter** implements the dynamic semantics defined in Sec. 7. Its role is to provide reference results to check the correctness of the generated SystemC and VHDL code. It can also be used to test and debug programs, during the first steps of application development (in this case, input/output streams are read from/written to files). Several tracing and monitoring facilities are provided. For example, it is possible to compute statistics on channel occupation or on rule activation.

The **compiler** is the core of the system. It relies on an *elaboration phase*, turning the AST into a target-independant intermediate representation (described in chapter 9), and a set of dedicated back-ends.

Two back-ends are currently provided : the first produces cycle-accurate SystemC code for simulation and profiling, the second VHDL code for hardware synthesis. Execution of the SystemC code provides informations which are used to refine the VHDL implementation (for example : the actual size of the FIFOs used to implement channels).

The graph visualizer, the reference interpreter and the compiler all operate on an abstract syntax tree (AST) produced by the **front-end** after parsing and type-checking.

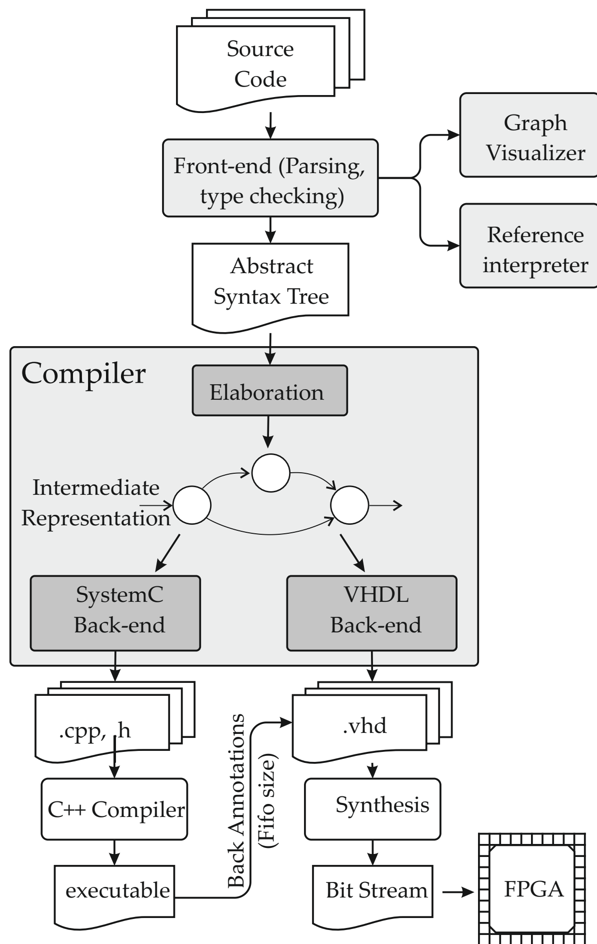


Figure 1.9:

Chapter 2

CAPH Overview

This chapter introduces the CAPH language. The basic structure of programs is presented. Its syntax and semantics are introduced informally, by means of examples. Formal accounts can be found in chapters 3, 4, 5, 6 and 7. In the examples, comments follow the CAPH syntax : they are single-line and starts with "--".

2.1 Language structure

As stated in introduction, CAPH is a layered language.

The outermost (*declaration*) layer declares types, global values (constant and functions), I/O streams, actors and network-level objects (wires and wiring functions). The innermost (*expression*) layer is used for describing computations. An *actor* sub-language is used for describing the interface and the behavior of actors and a *network* sub-language is used for describing the structure of the dataflow process network.

All layers essentially share a common type system, which is presented in the next section.

2.2 Types

Broadly speaking, two categories of types can be distinguished: *scalar* types and *structured* types[1].

2.2.1 Scalar types

Scalar types are described in Table 2.1. They include signed and unsigned fixed-precision integers, booleans and floating-point values[2].

Builtin operations provided on scalar types are summarized in Table 2.2. By *builtin*, we mean the operations which

- have a runtime implementation at the simulator level,
- can be a automatically translated to their SystemC and VHDL equivalent by the compiler backends[3].

[1]The concept of *type variable* is discussed separately in Sec. 2.4.5
[2]Support of floating-point values by the VHDL backend is platform-dependant (see Sec. 2.2.3).
[3]The support of floating-point values ultimately depends on the VHDL compiler.

`int`	Generic (signed) integer, range is implementation dependant
`signed<prec>`	Sized signed integer, range is $-2^{prec-1} \ldots 2^{prec-1} - 1$
`unsigned<prec>`	Sized unsigned integer, range is $0 \ldots 2^{prec} - 1$
`bool`	Boolean (`true`, `false`)
`float`	Floating-point value (minimal range : $-1.10^{38} \ldots + 1.10^{38}$)

Table 2.1: CAPH scalar types

<table>
<tr><td>bool</td><td>&& || not (logical and, or, not)</td></tr>
<tr><td>int
signed
unsigned</td><td>+ - * / mod
< <= = >= ></td></tr>
<tr><td>signed
unsigned</td><td>land lor lnand lnot (bitwise operations)
<<, >> (left/right logical shift)</td></tr>
<tr><td>float</td><td>+. -. *. /.
=. !=. <. >. <=. >=.</td></tr>
</table>

Table 2.2: Builtin operations on scalar types

2.2.2 Structured types

Two categories of structured types are currently supported : arrays and variants.

Arrays are indexed collections of values of the same type. The corresponding type, `array`, is predefined in CAPH. Arrays can have one, two or three dimensions :

- for 1D arrays, each item has a scalar type,
- 2D arrays are viewed as arrays of 1D arrays[4],
- 3D arrays are viewed as arrays of 2D arrays[5].

For each dimension, the size S is fixed and must defined at the declaration. Indexes range from 0 to $S-1$.

Arrays can be declared either as global constants or as local variables in actors (Table 2.3). In the first case, the (immutable) value of the array is part of the declaration and an optional type signature can be used to refine the type of the array items. In the second case, the type signature is not optional but the initial value can be omitted[6]. Two syntaxes – extension and comprehension – are provided for specifying initilisation values.

Individual array items can be accessed for reading and updating using the classical `[.]` notation (Table 2.4).

Variants allow values of different kinds to be mixed together in a common type by tagging them with a distinct label. They are described in Sec. 2.4.1.

2.2.3 Translation to SystemC and VHDL types

Table 2.5 describes how CAPH types are translated to SystemC or VHDL types by the compiler backends.

[4]And for this reason, sometimes called 1D×1D arrays.

[5]And for this reason, sometimes called 1D×1D×1D arrays.

[6]It is the progammer's responsability, of course, to ensure that no element of the array will be accessed before having been properly initialized.

<table>
<tr><td>const <id> "=" <array_init> [":" <array_type>]</td><td>global constant</td></tr>
<tr><td>var <id> ":" <array_type> ["=" <array_init>]</td><td>local variable</td></tr>
<tr><td>where</td><td></td></tr>
<tr><td><array_init> ::=</td><td></td></tr>
<tr><td> <array_ext1></td><td>1D extension</td></tr>
<tr><td> <array_ext2></td><td>2D extension</td></tr>
<tr><td> <array_ext3></td><td>3D extension</td></tr>
<tr><td> | "[" <expr> "|" <index_range> "," ..., "," <index_range></td><td>comprehension</td></tr>
<tr><td><array_ext1> ::=</td><td></td></tr>
<tr><td> | "[" <scalar_const>_1 "," ... "," <scalar_const>_n "]"</td><td></td></tr>
<tr><td><array_ext2> ::=</td><td></td></tr>
<tr><td> | "[" <array_ext1>_1 "," ... "," <array_ext1>_n "]"</td><td></td></tr>
<tr><td><array_ext3> ::=</td><td></td></tr>
<tr><td> | "[" <array_ext2>_1 "," ... "," <array_ext2>_n "]"</td><td></td></tr>
<tr><td><index_range> ::=</td><td></td></tr>
<tr><td> | <id> "=" <index_1> "to" <index_2></td><td></td></tr>
<tr><td>and</td><td></td></tr>
<tr><td><array_type> ::=</td><td></td></tr>
<tr><td> <scalar_type> "array" "[" <size> "]"</td><td>1D array</td></tr>
<tr><td> <scalar_type> "array" "[" <size> "]" "[" <size> "]"</td><td>2D array</td></tr>
<tr><td> <scalar_type> "array" "[" <size> "]" "[" <size> "]" "[" <size> "]"</td><td>3D array</td></tr>
<tr><td>Examples</td><td></td></tr>
<tr><td>var a : signed<4> array[4]</td><td></td></tr>
<tr><td>var b : unsigned<8> array[4] = [1,2,3,4]</td><td></td></tr>
<tr><td>var c : unsigned<8> array[4] = [i*2 | i=0 to 3][a]</td><td></td></tr>
<tr><td>const a1 = [1,2,3,4] : signed<4> array[4]</td><td></td></tr>
<tr><td>const a2 = [[1,2],[3,4],[5,6]] : signed<4> array[3][2]</td><td></td></tr>
<tr><td>const a3 = [[[1,2],[3,4]],[[5,6],[7,8]]] : signed<4> array[2][2][2]</td><td></td></tr>
</table>

Table 2.3: Array declarations

[a] This is equivalent to `var c : unsigned<8> array[4] = [0,2,4,6]`

<table>
<tr><td><array_id>"["<index_expr>"]"</td><td>for 1D arrays</td></tr>
<tr><td><array_id>"["<index_expr>"]" "["<index_expr>"]"</td><td>for 2D arrays</td></tr>
<tr><td><array_id>"["<index_expr>"]" "["<index_expr>"]" "["<index_expr>"]"</td><td>for 3D arrays</td></tr>
<tr><td>where</td><td></td></tr>
<tr><td><index_expr> is any expression of type int</td><td></td></tr>
<tr><td>Examples</td><td></td></tr>
<tr><td>t[1]</td><td></td></tr>
<tr><td>t[2*i+1]</td><td></td></tr>
<tr><td>t[i][j]</td><td></td></tr>
<tr><td>t[i][j][2]</td><td></td></tr>
</table>

Table 2.4: Array expressions

CAPH	SystemC	VHDL
bool	bool	boolean
int	int	signed<prec-1 downto 0>
signed<prec>	sc_int<prec> or int[a]	signed<prec-1 downto 0>
unsigned<prec>	sc_uint<prec> or unsigned int[b]	unsigned<prec-1 downto 0>
float	float[c]	float32
t array[sz][d]	std::array<t',sz>	array (0 to sz-1) of t' where t' is translation of type t
t array[sz][sz'][e]	std::array<std::array<t',sz>,sz'>	array (0 to sz-1) of array (0 to sz'-1) of t' where t' is translation of type t
t[f]	class ...[g]	std_logic_vector<m+n> where $m = \lceil log_2 N_c \rceil$ $n = max_{i=1 \ldots N_c} \lvert \tau_i \rvert$ N_c is the number of distinct tags τ_i is the type of argument for tag i $\lvert \tau \rvert$ is size (in bits) of the VHDL representation for type τ The m MSBs are used to encode the tag. The n LSBs are used to encode the argument[h]

Table 2.5: Translation of types in SystemC and VHDL

[a] Depending on whether prec resolves to a static constant or not.
[b] Depending on whether prec resolves to a static constant or not.
[c] C native float type (32 bits)
[d] Where t is a scalar type.
[e] Where t is a scalar type.
[f] Where t is a variant type.
[g] See chapter 14.
[h] See chapter 14.

In VHDL, Tokens circulating between actors are represented as std_logic_vectors and numeric values are represented using the signed or unsigned data types defined in the ieee.numeric_std library[7]. Booleans are encoded as VHDL booleans. Floating-point values are currently encoded using the float32 type provided by the *VHDL-2008 Support Library* available at http://www.vhdl.org/fphdl[8].

2.3 CAPH expression language

The expression language is a small, purely functional, first-order, polymorphic language with a strict semantics. This language is used both for defining the values assigned to global constants and functions at the declaration level and the values computed when a rule is fired at the actor level. Its syntax broadly follows that of Caml [2].

The expression language cannot manipulate functions as values, like in classical, higher-order functional languages. This makes sense since this concept cannot be translated, in general, into hardware.

[7] There's a compiler option to change the default library used for implementing arithmetic operations.

[8] This should be viewed as a temporary solution, before the full VHDL-2008 standard is supported by mainstream VHDL compilers and synthetizers. Support for floating-point values can be disabled if this library cannot be installed on the target platform.

2.3.1 Constants

Constants at the expression-level are simple constant values covering the basic CAPH types (integers and booleans). By default, integer constants are un-signed and un-sized. Signness and size can be provided using the type coercion operator ":" (see Sec 2.3.6). Signness can also specified using the `U` and `S` suffixes or implicitely for negative ints.

Example

```
23                  -- generic int
23S                 -- signed int
23U                 -- unsigned int
-12                 -- signed int
(12:unsigned<4>)    -- sized, unsigned int
(12:signed<8>)      -- sized, signed int
true                -- boolean
0xFA                -- hex constant
0b1001              -- binary constant
1.23                -- float constant
-2.3                -- float constant
```

Type coercion can be used to define constants with a more precise type (see Sec 2.3.6).

2.3.2 Variables

Variables appearing at the expression level are simple identifiers. These identifiers must have been bound at the declaration layer or with an upper `let` declaration within the same expression. Identifiers must start with a lowercase letter (symbols starting with an uppercase letter are used to designate data constructors).

2.3.3 Function application

Functions are always fully applied at the expression level[9]. The applied function must have been declared in the declaration layer or be a builtin primitive. Application of builtin primitives uses the classical infix notation.

Example

```
g (3)           -- g is function with arity 1
f (1,2)         -- f is function with arity 2
1+2             -- + is a builtin primitive
```

2.3.4 Conditionals

Conditional expressions always have two alternatives, depending on the value of the discriminating expression is (`true` or `false`).

Example

```
if x>1 then x-1 else x
```

[9] This is to be contrasted to functions defined at the network level, where higher-order functions and partial application are supported.

2.3.5 Local Declarations

Local declarations are used to bind variable(s) to expression(s) within the (limited) scope of a given expression. The names introduced by the bindings are only visible within the target expression. Bindings are evaluated before the target expression is evaluated.

Example

```
let x=1+2 in x*x                 -- expression value is 9
let x=1+2 and y=3*4 in x+y       -- expression value is 15
```

Local declarations can be nested. This offers a way for "breaking" complex computations and/or sharing intermediary results.

Example

```
let y=2*x+1 in let z=y*y-5 in z/4
```

The previous example is equivalent to writing `((2*x+1)*(2*x+1)-5)/4`.

2.3.6 Type coercion

The expression language has a builtin operator `(:)` for performing type coercion. Type coercion is often necessary to assign values a more "precise" type than that infered by the type checker in order to provide the compiler back-ends enough information.

Example

```
(1+2 : unsigned<8>)
```

Here the type inferred by the type-checker for the mere expression `1+2` is simply the *generic* type `int`[10]. This type is here explicitely refined to `unsigned<8>`[11].

The coercibility relation between base types is described in Table 2.6. A S at intersection of row `t` and column `t'` means that type `t` can be safely coerced to type `t'` (with no loss of information); a U (unsafe) indicates that some information may be lost and a F indicates a forbidden operation.

This relation naturally extends to structured types with the same constructor and the same number of elements. For example, type `t array[m]` can be coerced to `t' array[n]` iff `m=n` and `t` can be coerced to `t'`.

Coercion between integer types is only accepted in two situations:

1. it corresponds to a "refinement" of the LHS type. Examples :
   - `(int<8>:unsigned<8>)` is ok,
   - `(int:unsigned<8>)` is ok,
   - `(signed<n>:signed<16>)` (where `n` is an unbound size variable) is ok,
   - `(unsigned<8>:int)` is forbidden,
   - `(signed<8>:signed<n>)` (where `n` is an unbound size variable) is forbidden[12].
2. it corresponds to a modification of an already known signness or size. Examples

[10] In fact, this is `int<s,n>` where `s` and `n` are respectively a *sign variable* and a *size* variable. The notion of type variable is discussed in Sec. 2.4.5.

[11] Where `unsigned<8>` is actually an abbreviation for `int<_unsigned,8>`.

[12] The known size 8 cannot be "erased".

	bool	int	signed<n>	unsigned<n>	float
bool	S	S[1]	S[1]	S[1]	S[1]
int	U[2]	S	S	S	S
signed<m>	U[2]	F	U[3]	U[4]	S
unsigned<m>	U[2]	F	U[5]	U[3]	S
float	U[2]	U[7]	U[7]	U[7]	S

Table 2.6: Type casting between base types

[a] true is translated to 1, false to 0.
[b] 0 is translated to false, all other values to true.
[c] Truncature may occur if $m > n$
[d] Sign is lost. Truncature may occur if $m > n$
[e] Truncature may occur if $m > n - 1$
[f] Some loss of precision may occur
[g] The fractional part is discarded. Truncature may occur

If value v has type...	v‘size gives...
bool	1
int<prec>	<prec>
signed<prec>	<prec>
unsigned<prec>	<prec>
float	32
t array[sz]	sz
$\tau_1 \times \ldots \times \tau_n$	n

Table 2.7: Interpretation of the size attribute

- (signed<8>:signed<16>) is ok
- (signed<8>:unsigned<8>) is ok (but the sign is lost),
- (unsigned<16>:unsigned<8>) is ok (but the result may be truncated).

The compiler accepts coercions of the first kind silently and emits a warning for the second kind when the signness/sizes of the LHS and RHS is/are different.

2.3.7 Attributes

An *attribute* is a property which can be attached to certain types of value. Examples are the size (number of elements) of an arrays or the width (in bits) of an integer value.

Attribute values can be retrieved using the following syntax : <value_name>‘<attribute_name>.

A typical use is of attributes is for initializing arrays :

Example

```
var z : array[8] = [ 0 | i=0 to z‘size-1 ]
```

The current version of the compiler only supports one kind of the attribute, named size. The meaning of this attribute directly depends on the *type* of the value to which it is attached. Table 2.7 gives its interpretation for all CAPH builtin types[13].

[13] The value of the size attribute is undefined for user-defined types.

2.4 CAPH declaration language

This is the outermost layer. A CAPH program is just a sequence of *declarations*, which are evaluated sequentially (in other words, the declaration of an object may refer to objects declared before but not the other way). Declarations concerns declares types, global values (constant and functions), I/O streams, actors and network-level objects (wires and wiring functions).

2.4.1 Type declarations

There are two kinds of type declarations : type synonym declarations and variant declarations.

Type synonym declarations are used to introduce abbreviations to existing types. For example, the following declaration defines a type `byte` which is a synonym to the builtin type `unsigned<8>` :

Example

```
type byte == unsigned<8>;  -- byte is now a synomym for unsigned<8>
```

Variant declarations are used to define *algebraic data types.* These types – also called *tagged unions* – allow values of different types to be mixed together by tagging them with a distinct label. Consider for example the situation in which a token can carry either a signed or unsigned 8-bit value. A type for this kind of tokens could be defined with the following type declaration :

```
type us8 =
    Signed of signed<8>
  | Unsigned of unsigned<8>
```

This declaration introduces the *type constructor* `us8`. A value of type `us8` is either a `signed<8>` value or a `unsigned<8>` value. The associated tag (`Signed` or `Unsigned`) is used to distinguish between these two cases.

More generally speaking, the declaration of a variant type lists all possible "shapes" for values of that type. Each case is identified by a specific tag, called a *value constructor*, which serves both for constructing values of the variant type and inspecting them by pattern-matching (see Sec. 2.4.4 for examples). Value constructors can take 0 to n arguments[14]. To distinguish them from variable names – which start with a lowercase letter – they must start with a capital letter.

Type constructors can be polymorphic, *i.e.* they can be parameterized over (an)other type(s), called the *argument types(s)* (see Sec. 2.4.5). For example, here's a definition of a type for representing *optional* values (*i.e.* values that can be either present or absent) :

Example 1

```
type $t option =
  Absent
| Present of $t
```

A value of type τ `option` is either `Absent` or `Present v`, where `v` is a value of type τ and τ can be any type.

The following definition introduces a type for representing optionally labeled values :

[14] Since version 2.6.2. In previous versions only nullary and unary value constructors were supported.

Example 2

```
type ($t1,$t2) labeled =
| Unlabeled of $t1
| Labeled of $t1 * $t2
```

A value of type (τ, λ) `labeled` is either `Unlabeled v`, where `v` is a value of type τ or `Labeled (v,l)` where `v` is a value of type τ and `l` a value of type λ. As above, τ and λ can be any type.

When an variant type involves sized integers, it can be parameterized over the corresponding sizes. The size parameters are then specified by listing them, between `<` and `>` after the name of the type constructor[15] For example, the type `us8` introduced above could be generalized as follows : matter

```
type us<n> =
    Signed of signed<n>
  | Unsigned of unsigned<n>
```

The type `us` can now be used to represent values having type `signed<n>` or `unsigned<n>`, where `n` can be any possible value.

The two kinds of polymorphism (type and size) can appear in the same declaration. Here's for example the declation of a type `tau` whose values are either a signed integers with size `n` or values of type `t` :

```
type $t tau<n> =
    Left of signed<n>
  | Right of $t
```

The CAPH standard library defines (in `lib/caph/dc.cph`) the following type[16] :

```
type $t dc =
  Data of $t
| SoS
| EoS
```

The **dc** type (abbreviation for *data/control*) can be used to encode *structured streams*, *i.e.* streams in which the sequence of tokens obeys to a certain structure (as opposed to "raw" streams in which the only structure is the order in which tokens appears). Consider for example the stream representing a (potentially infinite) sequence of $n \times n$ images. Representing this sequence as a simple stream of tokens, where each token carries a pixel value is generally not sufficient. Most often, actors operating on this stream will need to detect the start and end of a single image in this stream and the start and end of individual lines in this image. This need to *structure* the stream of tokens can be served by dividing the tokens, circulating on channels and manipulated by actors, into two broad categories : *data* tokens (carrying actual values) and *control* tokens (acting as structuring delimiters). For the aforementionned example, one could therefore introduce the following type to represent sequences of images :

Example

```
type $t image =
```

[15]This "hybrid" approach, in which *type* parameters are specified using a prefix notation and *size* parameters using a postfix notation may appear awkward to programmers familiar to purely prefix (OCaml, for example) of purely postfix-based (C++, for example) notations. It has been chosen because different trials have shown us that the other choices actually led to slightly more verbose formulations to avoid parsing ambiguities.

[16]This type was builtin in versions prior to 2.6.2.

```
  | SoI            -- Start of image
  | EoI            -- End of image
  | SoL            -- Start of line
  | EoL            -- End of line
  | Data of $t     -- Pixel
```

Tokens with values `SoI`, `EoI`, `SoL` and `EoL` will be control tokens indicating the start and end of images (resp. lines) and pixels will be carried by tokens having values `Data v`. With the scheme, the 4×4 image of Fig. 2.1 may be represented by the following stream of tokens:

```
SoI, SoL, Data(10), Data(30), Data(55), Data(90), EoL,
     SoL, Data(33), Data(53), Data(60), Data(12), EoL,
     SoL, Data(99), Data(56), Data(23), Data(11), EoL,
     SoL, Data(11), Data(82), Data(46), Data(11), EoL,  EoI
```

But it turns out that the type introduced above is sufficient for performing this encoding of images as structured streams. The idea is that an image can viewed as a *list* of lines, where each line can in turn be viewed as a *list* of pixels. For this, the `SoS` (resp. `EoS`) is interpreted as a *start of list* (resp. *end of list*) control token, and the 4×4 image of Fig. 2.1 is now represented by the following stream of tokens:

```
SoS, SoS, Data(10), Data(30), Data(55), Data(90), EoS,
     SoS, Data(33), Data(53), Data(60), Data(12), EoS,
     SoS, Data(99), Data(56), Data(23), Data(11), EoS,
     SoS, Data(11), Data(82), Data(46), Data(11), EoS,  EoS
```

The advantage[17] is that any kind of data structure can be represented this way : images, lists, but also trees, *etc.*, without requiring dedicated types.

Moreover, this structured representation of data nicely fits the stream-processing programming and execution models. Since the structure of the data is explicitly contained in the token stream no global control and/or synchronization is needed; this has strong and positive consequences both at the programming level (it justifies *a posteriori* the style of description we introduced in the previous subsection for actors) and the execution level (it will greatly eases the production of HDL code). Moreover, it naturally supports a pipelined execution scheme; processing of a line by an actor, for example, can begin as soon as the first pixel is read without having to wait for the entire structure to be received; this feature, which effectively allows concurrent circulation of successive "waves" of tokens through the network of actors is of course crucial for on-the-fly processing (like in real-time image processing).

10	30	55	90
33	53	60	12
99	56	23	11
11	82	45	11

Figure 2.1: A 4×4 image

[17]Which has probably not escaped to programmers familiar with the Lisp language...

2.4.2 Value declarations

There are two types of value declarations : constants and functions.

Identifiers bound in these declarations scope over both the actor and network sub-languages, but an important distinction must be made.

At the actor level, they can appear everywhere in the expressions defining the behavior of actors (in the right-hand side of actor rules – see Sec. 2.4.4). As a matter of fact, global functions are often used to improve the readability of actors, by allowing a separation between purely combinational and sequential aspects.

At the network level, they can only appear as parameters when an actor is instanciated (see Sec 2.4.7) and cannot be used to define network-level expressions.

Constant declarations

Constant declarations are like `#define` declarations in C. They give a name to a value which is computed statically. They are typically used to define application-specific parameters.

Example

```
const threshold = 1+2;                    -- scalar constant
const kernel = [1,2,1];                   -- 1D array constant
const kernels = [[1,2,1],[1,4,1]];        -- 1Dx1D array constant
```

Function declarations

Function declarations introduce functions, mapping an identifier (or a set of identifiers) to an expression. Functions with several arguments are represented as functions taking a tuple. An optional type signature can be specified to refine the type of the function.

Example

```
function incr x = x+1;
function scale (x,s) = x*s : signed<8> * signed<8> -> signed<16>;
```

External function declarations

External functions declarations introduce functions which are defined outside the CAPH language. Their main usage is to allow the SystemC or VHDL generated code to make use of pre-existing functions already written in these languages. The type of the corresponding function must be supplied. For the corresponding program to be simulated, a Caml implementation of the function must also be provided[18].

Example

```
function sqrt x =
  extern "sqrt_c","sqrt_vhd","sqrt_ml": unsigned<16> -> unsigned<16>;
```

In the above example, `sqrt_c`, `sqrt_vhd` and `sqrt_ml` are the names of the C, VHDL and Caml implementations of the `sqrt` CAPH function[19]. These functions are supposed to be defined in a file accessible when compiling the code generated by the back-ends. For the ML function, it must be defined in a specific file and registered using a dedicated function. The mechanism is detailed in Chap. 12

[18]It could be possible to interface the simulator directly to the C code but this is not currently implemented. Hence the necessity to provide the Caml version of the function.

[19]We have used three different names but in practice, the same name can be used for the three implementations.

Its the programmer's responsability to ensure that the actual types of the function arguments and result are compatible with the types specified in the declaration. There's currently no specific type-based translation mechanism for foreign values. As a result, only functions whose arguments and result can be safely coerced to integers are supported.

2.4.3 I/O declaration

These declarations specify the way by which the application will interact with the operating system (during simulation) or the physical devices (for the generated VHDL code for example).

Two types of I/Os are supported : `stream`s and `port`s.

Streams are used to model pure data flows, in which tokens are read (resp. written) from (resp. to) a sequential source (resp sink) using a fifo-like protocol.

Ports are used to model "asynchronous" I/Os, in which values are read (resp. written) using a RAM-like interface.

Both types of declarations specify a name, a type, a direction (input or output) and a "device". The device is a system-specific designator identifying the entity the input (resp. output) data will be read from (resp. written to). When using the simulator, designators will be simple file names. The SystemC and VHDL backends may use more system and platform specific designators. For port inputs, the declaration also specify an initial value[20].

Example

```
stream inp1 : unsigned<8> from "sample.txt";
stream outp : signed<8> to "display:0";

port inp2: unsigned<16> from "threshold.txt" init 64;
port inp3: signed<8> init -1;
port outp2: boolean to "acks.txt";
```

2.4.4 Actor declarations

Each actor involved in the dataflow process network must be declared. The declaration comprises an *interface* (which is the only visible part at the network level) and a *body*, describing its behavior.

Actor interface

The interface specifies the actor input(s), output(s) and optional parameter(s). All inputs, outputs and parameters are typed. When building the network, inputs and outputs will be connected to channels and parameters will be given values.

Example

```
-- This actor has one input, of type int, one output, of type bool and no
   parameter

actor a1
  in (x: int)
 out (y: bool)
 --   ...  actor body ...
```

[20]For port inputs, the device can be omitted; in this case the port behaves as a constant generator.

Example

```
-- This actor has two inputs, both of type signed<8>, one output, of type
     signed<8> and one parameter, of type unsigned<4>

actor a2 (k:unsigned<4>)
  in (e1: signed<8>, e2: signed<8>)
 out (s: signed<8>)
 --   ...  actor body ...
```

Actor body

The actor body comprises a set of local variable declarations and a set of transition rules.

The set of **local variable declarations** can be empty. Each variable is declared with a name, a type and an optional initial value. Variables are used to retain values between successive activations of the actor. Their scope is limited to the actor they are defined in.

In addition to the types defined in Sec. 2.2, local variables can also have an *enumerated* type. Two kinds of enumerated type are accepted : explicit enumerations and integer ranges.

Explicit enumerations are actually variant types for which all constructors have arity 0.

Example

```
actor ...
  ...
var state : { S0, S1, S2 }
...
```

Declaring a local variable with such a type *de facto* introduces a new type but the corresponding type constructor is anonymous and the scope of the introduced data constructors (`S0`, `S1`, ...) is limited to the englobing actor[21].

Integer ranges may be viewed as a subset of the `int` type.

Example

```
actor ...
  ...
var ctr : { 1,..,8 };
...
```

A variable with such a type can be used in any expression in which an `int` can be accepted. The main distinction is that it will be recognized as a potential *state* variable when performing abstract interpretation or FSM dumping (see Sec. 10.10 and chapter 8).

The behavior of an actor is specified using a set of **transition rules**.
Each rule consists of a set of *patterns*, involving inputs and local variables, and a set of *expressions*, describing modifications of outputs and/or local variables.
Each rule has the form

$$| \quad (\text{qual}_1 : \text{pat}_1, \ldots, \text{qual}_m : \text{pat}_m) \rightarrow (\text{qual'}_1 : \text{exp}_1, \ldots, \text{qual'}_n : \text{exp}_n)$$

[21] This feature was introduced precisely to avoid name conflicts that frequently arise if one has to declare global type and data constructors for locally defined values such as state variables.

where

- *qual* designates an input, a scalar variable or an element of an array variable,
- *pat* is a pattern,
- *qual′* designates an output, a scalar variable or an element of an array variable,
- *exp* is an expression.

Parens can be omitted if $m = 1$ (resp. $n = 1$).

A pattern can be

- a litteral constant[22] (ex: `0`, `true`, ...),
- a variable,
- a constant constructor (`SoS`, `EoS`, or any nullary constructor introduced by an enumerated type or a variant type declaration),
- `C` *p*, where
  - `C` is a constructor with arity 1 (`Data` or introduced by a variant type declaration)
  - *p* is a pattern,
- the "_" symbol

A pattern refers to an input or a local variable, the name of which is given by the attached qualifier.

An expression can be

- any expression of the expression language defined in Sec. 2.3,
- the "_" symbol

An expression refers to an output or a local variable, the name of which is given by the attached qualifier.

Identifiers appearing within right-hand side expressions of a rule can refer to

- variables introduced by patterns in the corresponding left-hand side,
- parameters of the defined actor,
- local variables of the defined actor,
- global variables.

Example

```
actor foo in (i:int) out (o:int)
var v : bool
rules
| (i:x, v:true) -> o:x+1
| (i:x, v:false) -> o:x-1;
```

[22]Constants defined in value declaration section (see Sec. 2.4.2) are not allowed, or, more precisely, they will be interpreted as a variable pattern, which is generally not what is expected.

In this example, we have two rules. Each rule depends on the value of the input `i` and the local variable `v` and affects the output `o`. The patterns for the first (resp. second) rule are `x` and `true` (resp. `x` and `false`). These patterns will match any configuration in which a token is present on input `i` (with a value `x`) and the local variable `v` has value `true` (resp. `false`). The expression for the first (resp. second) rule writes the value `x+1` (resp. `x-1`) to the output `o`.

Example

```
actor bar in (i:int) out (o:int)
var s : int = 0
rules
| i:x -> (o:x+s, s:s+1)
```

In this second example, the pattern of the rule will match any configuration in which a token is present on input `i`. The corresponding expressions will write the sum of the value of this token and the value of the local variable `s` to the output `o` and increment the local variable `s`.

Variant syntax for rules. When the rule section of actor contains several rules involving similar qualifiers for patterns and expressions, it is possible to simplify the formulation of these rules by prefixing them by a *rule format* and omitting the individual qualifiers on patterns and expressions. A rule format has the form

$$(\text{qual}_1, \ldots, \text{qual}_m) \rightarrow (\text{qual'}_1, \ldots, \text{qual'}_n)$$

where *qual* (resp. $qual'$) designates an input (resp. output), a scalar variable or an element of an array variable. This rule format tells to which input (resp. output, variable or array element) the corresponding[23] item of all the subsequent rules refers. As for rules themselves, the parens can be omitted when $m = 1$ (resp. $n = 1$).

For example, the actor `foo` introduced above can be reformulated as

Example

```
actor foo in (i:int) out (o:int)
var v : bool
rules (i,v) -> o
| (x, true) -> x+1
| (x, false) -> x-1;
```

Both formulation – individual qualifiers within rules or general rule format – are strictly equivalent[24]. In this manual, we will freely use one or the other.

Semantics of rules. At each execution cycle[25], a *fireable* rule is searched. A rule is fireable if the actual values of the inputs and local variables match the rule pattern and if the rule expression produces values that can be written to the involved outputs. The choice of the rule to be fired is done by sequential pattern-matching (in other words, the first rule is tried, then the second, etc. If no rule is fireable, the actor waits for the next execution cycle).

The special pattern "_" means "*ignore*" for inputs (*i.e.* don't even read the input) and "*don't care*" for local variables.

The special expression "_" means "*ignore*" (*i.e.* don't write the output) for outputs and "*don't modify*" for local variables.

[23] Where correspondance is established by position.
[24] In fact, the compiler front-end translates the latter into the former.
[25] The precise notion of *execution cycle* is defined by the dynamic semantics in chapter 7.

Examples

We now give several complete examples of actors to illustrate to concepts and notations introduced in the previous section.

Listing 2.1:

```
actor double in (i: int) out (o: int)
rules
| i:x -> o:x*2;
```

This actor defined in listing 2.1 ressembles the `inc` and `dec` actors introduced in Sec 1.3.1. It has one input and one output, of type `int`, no parameter and no local variable. There's only one rule, which says : whenever a token is available on input `i`, read it, bind the corresponding value to `x`, evaluate expression `x*2` and write the resulting value to output `o`. In effect, this actor will therefore doubles each value of the input stream : `inc:1,2,3,... = 2,4,6,...`

Listing 2.2:

```
actor scale (k:int) in (i: int) out (o: int)
rules
| i:x -> o:k*x;
```

The actor defined in listing 2.2 is a generalization of the previous one. It multiplies each value of the input stream by a constant factor. The factor is a parameter. Its value will be specified when the actor will be be instanciated (at the network level).

Listing 2.3:

```
actor mux in (i1: int, i2:int, sel:bool) out (o: int)
rules (sel,i1,i2) -> o
| (true,v1,v2) -> v1
| (false,v1,v2) -> v2;
```

The actor in listing 2.3 is a multiplexer : it routes its first (`i1`) or second (`i2`) input to its output (`o`) according to the value of its third input (`sel`). For example, if `i1=1,3,5,...`, `i2=2,4,6,...`, `sel=true,true,false,...`, then `o=1,3,6,...`. Note that a token must be present on each input for the actor to fire and that a token is consumed on each of these inputs at each firing. Using the "_" pattern, it is possible not to consume the unselected input, as described in listing 2.4. In this case, for `i1=1,3,5,...`, `i2=2,4,6,...` and `sel=true,true,false,...`, we have `o=1,3,2,...` (the tokens accumulate on the channel connected to `i2`).

Listing 2.4: A variant of the actor described in listing 2.3

```
actor mux_bis in (i1: int, i2:int, sel:bool) out (o: int)
rules (sel,i1,i2) -> o
| (true,v1,_) -> v1
| (false,_,v2) -> v2;
```

The actor `mux` can also be described with a single rule, using a `if` expression[26], as shown in listing 2.5.

Listing 2.5: Another formulation of the actor described in listing 2.3

```
actor mux_ter in (i1: int, i2:int, sel:bool) out (o: int)
```

[26] But the actor `mux_bis` cannot !

```
rules (sel,i1,i2) -> o
| (s,v1,v2) -> if s then v1 else v2;
```

Listing 2.6:

```
actor sum in (i: int) out (o: int)
  var sum : int = 0
rules
| i:v -> (o:sum, sum:sum+v);
```

The actor in listing 2.6 is a integrator : it produces the running sum of the values present on the input stream. For example, if `i=1,2,3,4,...`, then `o=0,1,3,7,...`. For this is uses a local variable `sum`. The single rule says : whenever a token is available on input `i` (with value `v`), writes the current value (`v`) of `sum` to output `o` and add `v` to `sum`.

Listing 2.7:

```
actor switch
  in (i:int)
  out (o1:int, o2:int)
  var s : bool = false
  rules (s,i) -> (o1,o2,s)
  | (false, v) -> (v, _, true)
  | (true, v) -> (_, v, false);
```

This actor in listing 2.7 is the dual of the `merge` actor introduced in Sec. 1.3.2. It reads tokens on its input channel and alternatively routes them to its first ("left") and second ("right") output. Given the stream `1,2,3,4,...` it will produce the stream `1,3,...` (resp. `2,4,...`) on its output output `o1` (resp. `o2`). Here, the `'_'` symbol used in the right-hand side of a rule means that no value is produced on the corresponding output channel.

Note : the previous actor can be rewritten in a slightly more self-documenting manner using an enumerated type for variable `s`, as shown in listing 8.3.

Listing 2.8:

```
actor switch_bis
  in (i:int)
  out (o1:int, o2:int)
  var s : {Left,Right} = Left
  rules (s,i) -> (o1,o2,s)
  | (Left, v) -> (v, _, Right)
  | (Right, v) -> (_, v, Left);
```

Listing 2.9:

```
actor incr
  in (a:int dc)
  out (c:int dc)
  rules a -> c
  | SoS -> SoS
  | EoS -> EoS
  | Data v -> Data (v+1);
```

The actor in listing 2.9 increments a *structured* stream of values, as evidenced by the type of its input and output, `int dc` (see Sec. 2.4.1). Given the structured stream

`SoS, Data 1, Data 2, Data 3, EoS`

on its input, it will produce the structured stream

`SoS, Data 2, Data 3, Data 4, EoS`

on its output. Here pattern-matching is used to discriminate between control and data tokens. The rules can be read as follows : if input is a *control* token (`SoS` or `EoS`) then write the same token on output; if input is a *data* token, increment the carried value and write the resulting data token on output.

Note. For convenience, the `SoS`, `EoS` and `Data` constructors may be abbreviated as `'<`, `'>` and `'` respectively. The actor of listing 2.9 can therefore be rewritten in a slightly more concise manner as shown in listing 2.10

Listing 2.10: A rewriting of listing 2.9

```
actor incr_bis
  in (a:int dc)
  out (c:int dc)
rules a -> c
  | '< -> '<
  | '> -> '>
  | 'v -> '(v+1);
```

Listing 2.11:

```
actor suml
  in (i:int dc)
  out (o:int)
var state : {S0,S1} = S0
var sum : int = 0
rules
  | (state:S0, i:    SoS) -> (sum:0, state:S1)
  | (state:S1, i:    EoS) -> (o:sum, state:S0)
  | (state:S1, i:Data v) -> (sum:sum+v);
```

The actor in listing 2.11 operates on a structured stream composed of a sequence of lists, each list starting with a `SoS` token and ending with a `EoS` token. For each list, it computes the sum of the elements. For example, if `i=<,1,2,3,>,<,4,5,>,<,6,7,8,>,...`, then `o=6,9,21,...`. For this, it uses pattern matching on the input to detect the start and end of each list and two local variables : a local state (`state`), indicating whether the actor is waiting for a new list or computing the sum, and a accumulator (`sum`) for computing the sum. The three transition rules can be read as follows :

- if we are in state S0 and input token is “`<`”, then initialize sum to 0 and go to state S1;
- if we are in state S1 and input token is “`>`”, then writes the accumulated sum to output and go back to state S0;
- if we are in state S1 and input token is a data, then add the corresponding value to the accumulator.

Note that the actor blocks if the input stream is ill-formed (for example, if `i=<,1,2,<,...`).

Listing 2.12:

```
type $t option =
   Absent
| Present of $t
;

actor count
  in (a:signed<8> option)
  out (c: signed<8>)
var s: signed<8> = 0
rules
| a:Absent -> c:s
| a:Present x -> (c:s+x, s:s+x)
;
```

Listing 2.12 illustrates the declaration and use of user-defined variant types. The `count` actors produces the running sum of optional values, represented with the `option` type. When the input token is `Present v` the value `v` is added to the current sum `s`. When the input token is `Absent` the current sum `s` is unchanged. In both cases, the current sum is output. For example, if the token stream on input `a` is

```
Present(1), Absent, Present(5), Absent, Absent, Present(9)
```

then the token stream on output `c` will be

```
1,    1,      6,    6,    6,     15
```

Listing 2.13:

```
type us8 =
   Signed of signed<8>
| Unsigned of unsigned<8>
;

actor add
  in (a:us8, b:us8)
 out (c: us8)
rules
| (a:Signed s1, b:Signed s2) -> c:Signed (s1+s2)
| (a:Signed s, b:Unsigned u) -> c:Signed (s+(u:signed<8>))
| (a:Unsigned u, b:Signed s) -> c:Signed ((u:signed<8>)+s)
| (a:Unsigned u1, b:Unsigned u2) -> c:Signed ((u1:signed<8>)+(u2:signed
    <8>))
;
```

Listing 2.13 shows how to define an actor performing "mixed" (signed/unsigned) arithmetic using a variant type. The `add` actor accepts both signed and unsigned 8-bit values and produces a sum as a signed 8-bit value, performing type coercion as needed. For example, if the input streams on inputs `a` and `b` resp. are

`Signed(1), Signed(2), Signed(3), Signed(-1), Signed(-2), Signed(-3), Unsigned(1), Unsigned(2), Unsigned(3)`

and

`Signed(1), Signed(-1), Unsigned(2), Signed(1), Signed(-1), Unsigned(2), Signed(1), Signed(-1), Unsigned(2)`

then the output stream on `c` will be

`Signed(2), Signed(1), Signed(5), Signed 0, Signed(-3), Signed(-1), Signed(2), Signed(1), Signed(5)`

Semantics of pattern matching

The precise semantics of rule evaluation is given in chapter 7. Basically, when a rule is selected, the expressions given in the RHS are all evaluated in an environment containing

- builtin values,
- globally defined values,
- actor parameters,
- actor local variables,
- variables bound by pattern-matching in the corresponding LHS of the rule.

Some remarks about pattern matching :

- a given variable can only appear once in the same LHS; for example, the following formulation is not allowed :

```
actor foo in (i1:t, i2:t') out (...)
rules
| (i1:x, i2:x) -> ...
```

  This makes sense because a reference to variable `x` in the RHS would be otherwise ambiguous.

- pattern-matching a local variable against a variable is possible, but not required, because local variable are implicitely part of the evaluation environment in the RHS; for example, the two following formulations are semantically equivalent[27] :

```
actor foo1 in (...) out (o:t)
var s:t
rules
| ..., s:v, ... -> ..., o:v, ...

actor foo2 in (...) out (o:t)
var s:t
rules
| ... -> ..., o:s, ...
```

[27] Note however that the SystemC and VHDL backends will allocate an extra variable for the former.

Of course, explicit pattern matching a local variable against a constant or a structured value is still useful.

- variables introduced by pattern matching in the LHS "shadow" actor parameters and local variables; for example, in the following example, the value written on output `o` when the first rule is selected is that of input `i` and not of local variable `s` :

```
actor foo in (i:t) out (o:t)
var s:t
rules
| i:s -> o:s
| ...
```

Guards

A guard is a boolean expression, the value of which is added to the conditions which are taken into account to decide whether a rule is fireable or not. More precisely, a rule containing a guard would be marked as fireable if :

- the actual values of the inputs and local variables match the rule patterns and
- the value of the guard expression is true and
- the involved outputs are writable.

In the current version, guards can only refer to inputs or variables appearing in the patterns of the corresponding rule or to actor parameters.

Guards do not modify the order in which rules are scanned, which is still sequential.

Example

```
actor thr (k:signed<8>)
  in (a:signed<8>)
  out (c:unsigned<1>)
rules a -> c
| p when p > k -> 1
| p -> 0
;
```

The `thr` actor binarizes a stream of values by comparing each of them to a given theshold (set as a parameter). For example, given the input stream `1,8,2,18`, the actor `thr(4)` will produce the output stream `0,1,0,1`. This actor could be written without guard with a simple conditionnal :

Example

```
actor thr_bis (k:signed<8>)
  in (a:signed<8>)
  out (c:unsigned<1>)
rules a -> c
| 'p -> if p > k then 1 else 0
;
```

IO-less actors

It is possible to define actors with no input port and/or no output. Actors with no input may be used, for instance, as data sources and actors with no output as data sinks. Two examples are given in Listings 2.14 and 2.15 respectively.

For the `src` actor, defined in Listing 2.14, the type of the input `i`, `unit`, indicates that the actor actually has no input[28]. The "`_`" pattern for `i` in the firing rule here means that no value is requested for the rule to fire. The `src` actor therefore produces the following stream on its output : `0, 1, 2, ...`.

For the `snk` actor, defined in Listing 2.15, the type of the output `i`, `unit` again, indicates that the actor actually has no output. This actor reads a stream of integers on its input and sum the values in its local variable `sum`[29].

Listing 2.14: Example of input-less actor

```
actor src
    in (i: unit)
   out (o: int)
var cnt: int = 0
rules
| i:_ -> (o:cnt, cnt:cnt+1)
;
```

Listing 2.15: Example of output-less actor

```
actor snk
    in (i: int)
   out (o: unit)
var sum: int = 0
rules
| i:x -> sum:sum+x
;
```

2.4.5 Polymorphism

The ability of define *polymorphic* actors and functions is an important feature of the CAPH language. Coupled with the concept of *higher-order wiring functions*, described in Sec. 2.4.7, it allows in particular highly generic solutions to be described and reused in a large variety of contexts.

Basically, a polymorphic actor (resp. function) is an actor (resp. function) for which the type of inputs or outputs (resp. arguments and result) is left unspecified at definition. This implies that the behavior of this actor (resp. function) does not depend on the actual value of this type. This is called *parametric polymorphism*[30].

To illustrate the need for such a feature consider for example, the basic `mux_s8` actor defined below :

```
actor mux_s8
  in (e1: signed<8>, e2: signed<8>, c:bool)
 out (s: signed<8>)
```

[28] The `unit` type can be viewed as the equivalent of the `void` type in C or C++.

[29] Since this variable is not visible outside the actor, the `snk` actor defined here is of little utility. In practice, output-less actors will perform side-effects.

[30] By opposition to *ad-hoc* polymorphism – a.k.a. *overloading* –, where a given function, for example, may accept arguments with different types, performing a computation which depends on the actual type of the arguments.

```
rules
| (c:true, e1:x, e2:_) -> s:x
| (c:false, e1:_, e2:x) -> s:x
```

This actor accepts two streams of integers and a stream of booleans. When the boolean token is `true` (resp. `false`), it forwards the token present on the first (resp. second) input to the output, discarding the token present on the second (resp. first) output. As defined above, this actor can only be used to multiplex streams of `signed<8>` quantities. If one wants to multiplex, let say, streams of `unsigned<4>` quantities, another actor must be written :

```
actor mux_u4
  in (e1: unsigned<4>, e2: unsigned<4>, c:bool)
 out (s: unsigned<4>)
rules
| (c:true, e1:x, e2:_) -> s:x
| (c:false, e1:_, e2:x) -> s:x
```

This is clearly redundant. Having to define a new actor for each possible type for these inputs and output is tiresome and error-prone.

In fact, the actual type of the `e1` and `e2` inputs and `s` output does not matter, since the corresponding values are just copied from input to output.

The solution, in this case is to define a polymorphic actor `mux` as follows :

```
actor mux
  in (e1: $t, e2: $t, c:bool)
 out (s: $t)
rules
| (c:true, e1:x, e2:_) -> s:x
| (c:false, e1:_, e2:x) -> s:x
```

Here the type `$t` is a *type variable*; it stands for "*any possible type t*". The actual value of this type variable will be decided when the actor is instanciated as a box : it will be the actual type of the input and output wires connected to the box. Note that the fact that *same* type variable is used for both inputs and the output implies that the corresponding wires will be required to have the same type (forbidding for example the instanciation of this actor as a box connected to one input of type `signed<8>` and `unsigned<4>` for example).

Type variables may also appear in the list of parameters and local variables of an actor. Below is a possible definition for an actor performing a one-sample delay on lists of values[31] :

```
actor dl (v:$t)
  in (a:$t dc)
 out (c:$t dc)
var s : {S0,S1} = S0
var z : $t
rules
| (s:S0, a:'<) -> (s:S1, c:'<, z:v)
| (s:S1, a:'p) -> (s:S1, c:'z, z:p)
| (s:S1, a:'>) -> (s:S0, c:'>)
;
```

[31] This actor is actually defined in the library `list_ops.cph`.

If the input stream has type, say, `unsigned<4>`, and is[32] :

```
'< '1 '2 '3 '4 '>
```

then, the processing of this stream by actor `d` (with its parameter `v` set to 0) will produce the following stream :

```
'< '0 '1 '2 '3  '>
```

Note that this justifies *a posteriori* why the definition of the `dc` type is polymorphic.

Size variables

Sometimes, a more restricted form of polymorphism is what is needed. Consider for example an actor performing the element-wise addition of two streams of signed integers. A possible definition of this actor could be, for example :

```
actor add
  in (a: signed<8>, b:signed<8>)
 out (c: signed<8>)
rules
| (a:x, b:y) -> c:x+y
```

But, again, this definition is too restrictive since a similar one should be written for `signed<4>` integers, `signed<10>` integers, *etc.* One could be tempted to resort to parametric polymorphism to “factorise out” this redundancy, writting the `add` actor as :

```
actor add
  in (a: $t, b:$t)
 out (c: $t)
rules
| (a:x, b:y) -> c:x+y
```

But this does not work since the builtin `+` operator, used in the right-hand-side of the rule is *not* defined for any possible type τ. What is needed here is a way of abstracting over the *size* of the integer arguments and result. For this, the `add` actor must be defined as follows :

```
actor add
  in (a: signed<s>, b:signed<s>)
 out (c: signed<s>)
rules
| (a:x, b:y) -> c:x+y
```

Here `s` denotes a *size variable*; it stands for “*any possible size s*”. Like for type variables, its actual value will be set when the `add` actor gets instanciated as a box.

Sign variables

From what precedes, one could deduce that the type of the `+` builtin function is

```
signed<s> * signed<s> -> signed<s>
```

[32] We use here the abbreviated syntax for values of type `dc`

Such a type would clearly be too restrictive because it forbids the application of this operator to `unsigned` quantities. In fact, the type[33] of `+` is

```
int<g,s> * int<g,s> -> int<g,s>
```

where `g` is a *sign variable*. A sign variable is an type variable which can only take two values : `_signed` or `_unsigned`. In fact the types `signed<n>` and `unsigned<n>` are just shorthands for `int<_signed,n>` and `int<_unsigned,n>` respectively. Explicit reference to sign variables is useful when the related actor (or function) should abstract over the signness of the manipulated integer quantities. For example, a fully generic version of the `add` actor introduced above can be written as

```
actor add
  in (a: int<g,s>, b:int<g,s>)
 out (c: int<g,s>)
rules
| (a:x, b:y) -> c:x+y
```

Note that the signature of the `add` actor enforces that its inputs and output have both the same signness and size.

This kind of signature is used largely in the CAPH standard library to define actors and wiring functions operating both on signed and unsigned data flows (filters and convolutions for example).

Note

The notions of size and sign variables introduced above are just an special form of classical Hindley-Millner style of parametric polymorphism[34]. Size variables, in particular, cannot be used to express dependencies richer than mere equality between sizes in type signatures. Concretely, this means that it is not possible to define actors like

```
signed<s> * signed<s> -> signed<s+1>
```

or

```
signed<s> * signed<s> -> signed<2*s>
```

where some kind of "computation" is allowed on type size parameters. Supporting this would require a significantly more complex type system than actually implemented[35].

Dependent types

CAPH offers a limited form a so-called *dependent typing*, in which the *type* of an actor can depend on the *value* of its parameters[36].

To understand why this feature is useful, consider the program given in listing 2.16 .

Listing 2.16: A small program exhibiting the need for dependent types

```
actor add
    in (i1:unsigned<s>, i2:unsigned<s>)
   out (o:unsigned<s>)
rules
| (x,y) -> x+y;
```

[33] Which can be displayed by invoking the compiler with the options `-dump_tenv` and `-phantom_types`

[34] Technically speaking, the type of `+` or `add` is simply the *type scheme* : $\forall \alpha, \beta.\ (\alpha, \beta)$ int $\times\ (\alpha, \beta)$ int $\rightarrow (\alpha, \beta)$ int.

[35] With full-fledged dependent types and constraint-solving based unification.

[36] This feature, introduced in version 2.6.0, is still largely experimental.

```
stream inp1: int<16> from "inp1.txt";
stream inp2: int<16> from "inp2.txt";
stream outp: int<16> to "res.txt";

net outp = add (inp1,inp2);
```

Now, suppose that the type network output `outp` – which is ultimately imposed by the hardware context – is finally changed to `unsigned<12>` (with the inputs still having type `unsigned<16>`). The actor `add` cannot be used "as is" any longer because its signatures enforces that its inputs and output have the same size. Of course, we could rewrite it as follows to meet the new requirements :

```
actor add
    in (i1:unsigned<16>, i2:unsigned<16>)
   out (o:unsigned<12>)
rules
| (i1:x, i2:y) -> o:(x+y:unsigned<12>);
```

But this obviously breaks the genericity of the actor and the more general principle of modularity. If we don't want – or can't, because it's part of a pre-existing, otherwise used library, for example – to modify the `add` actor, the only solution is to insert, between the output of the `add` actor and the network output `outp`, an actor – let's call it `resize` – whose function is precisely to adjust the size of its argument. In our particular case, the signature of this actor would be :

```
resize : unsigned<n> -> unsigned<12>
```

and our program could be rewritten as in listing 2.17.

Listing 2.17: The program of listing 2.16 rewritten

```
actor add ... -- unchanged
actor resize
  in (i:unsigned<n)
 out (o:unsigned<12>)
rules
| i:x -> o:(x:unsigned<12);

stream inp1: int<16> from "inp.txt";
stream inp2: int<16> from "inp.txt";
stream outp: int<16> to "res.txt";

net outp = resize (add (inp1,inp2));
```

But, of course, the need then quickly arises for a *generic* version of the `resize` actor, so that we don't have to write a new one for each value of the output size. The idea is to make this size a *parameter* of the actor, so that, in our case, the last line of the last program could be written :

```
net outp = resize 12 (add (inp1, inp2));
```

For this, the `resize` actor has to be defined as follows :

Listing 2.18: none

```
actor resize (k:int)
  in (i:unsigned<n)
```

```
 out (o:unsigned<k>)
rules
| i:x -> o:(x:unsigned<k);
```

The type of the `resize` actor is now

```
resize : k:int -> unsigned<n> -> unsigned<k>
```

Such a type is called a *dependent type* because the *type* of some its components (the result here) depends on the *value* that will be assigned to some of the others (the first argument here). This dependency is here explicited by naming the first argument (parameter)[37].

In the previous example, the value of a parameter was used to define the size of some input/output types. This value can also be used to refine the type of some local variables of the actor. This is specially useful for arrays when the size of the array ultimately depends on the input data. In this case, specifying this size as a parameter value to the corresponding actor instance may be more efficient than fixing it in the actor definition itself. As an example, consider the `dkl` actor described in listing 2.19[38]. This actor inserts k predefined values at the beginning of a list, discarding the same number of values of values at the end[39]. For this it uses a local array (`z`) for memorizing the "delayed" values. The size of this array obviously depends on the value of k parameter. Dependent typing nicely supports this kind of dependency[40].

Listing 2.19: The `dkl` actor

```
actor dkl (k:int, v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2} = S0
var z : $t array[k] = [ v | i = 0 to k-1 ]
var i : int
rules
| (s:S0, a:'<)              -> (s:S1, c:'<, i:0)
| (s:S1, a:'p) when i<k-1 -> (s:S1, c:'v, i:i+1, z[i]:p)
| (s:S1, a:'p)              -> (s:S2, c:'v, i:0, z[i]:p)
| (s:S2, a:'p)              -> (s:S2, c:'z[i], i:if i<k-1 then i+1 else 0,
                                 z[i]:p)
| (s:S2, a:'>)              -> (s:S0, c:'>)
;
```

2.4.6 Higher order actors

Higher order actors are an important feature[41] of the the CAPH language. Just like higher order functions are functions taking other functions as argument in classical functional programming languages, higher order (HO) actors are actors for which at least one parameter is a function.

[37]Internally, the compiler uses on a nameless mechanism, similar to DeBruijn indices for representing dependent types. The type or `resize`, in particular, will be denoted $\forall n.$ `int` $\rightarrow$ `signed`$\langle n \rangle \rightarrow$ `signed`$\langle @1 \rangle$, where `@1` designates the first argument.

[38]This actor is included in the standard prelude.

[39]If lists represents fixed size of samples or lines of an images, this operation is a "delay", hence the name of the actor.

[40]Without it, the only solution is to set the size of the array to a "maximal" value which is both dangerous (what if the actor is instanciated actor violates this assumption) and inefficient (leading to a waste of resources if the size of the array is over-estimated) in a partcular case.

[41]Introduced in version 2.8.

The possibility to define and use HO actors significantly increases the abstraction level of programs. Consider for example, the program given in listing 2.20, which defines and then uses two actors, `inc` and `double`. The `inc` actor takes a stream of signed integers and produces a stream in which all data tokens have been incremented by one, whereas the `double` actor multiplies each data token by two[42]. Both actors exhibit a very similar structure, only differing in the function applied to data tokens (in the third rule). It is natural, then, to view these actors as two instances of a more general actor – let's call it `stream_apply` – taking a function f as parameter and applying this function to each data token of its input stream to produce the result stream. This actor `stream_apply` is defined in listing 2.21. The type of its parameter f is `signed<m> -> signed<m>`, *i.e.* the type of a function taking a value of type `signed<m>` and returning a value of the same type. This type is a hallmark of higher order actors. Note that it is closely related to those of the actor input and output : since the f function is applied to data tokens read on an input having type `signed<m> dc`, its domain type must be `signed<m>`. Respectively, since the result of this function is encapsulated in a token having type `signed<m> dc`, its co-domain type must be `signed<m>`. The program of listing 2.20 can be rewritten using the `stream_apply` actor as illustrated in listing 2.22.

Listing 2.20: A small program showing two similar actors

```
actor inc
  in (i:signed<m> dc)
  out (o:signed<m> dc)
rules i -> o
| '< -> '<
| '> -> '>
| 'x -> 'x+1
;

actor double
  in (i:signed<m> dc)
  out (o:signed<m> dc)
rules i -> o
| '< -> '<
| '> -> '>
| 'x -> 'x*2
;

stream i:signed<8> dc from ...
stream o1:signed<8> dc to ...
stream o2:unsigned<16> dc ...

net o1 = inc i;
net o2 = double i;
```

Listing 2.21: The `stream_apply` actor

```
actor stream_apply (f:signed<m> -> signed<m>)
  in (i:signed<m> dc)
  out (o:signed<m> dc)
rules i -> o
```

[42]So that, if the input stream `i` is, for example, `< 1 2 3 >`, then the output streams `o1` and `o2` will be `< 2 3 4>` and `< 2 4 6 >` respectively.

```
| '< -> '<
| '> -> '>
| 'x -> 'f(x)
;
```

Listing 2.22: The program of listing 2.20 rewritten using the actor defined in listing 2.21

```
function f_inc x = x+1 : signed<8> -> signed<8>;
function f_double x = x*2 : signed<8> -> signed<8>;

actor stream_apply (f:signed<m> -> signed<m>)
  in (i:signed<m> dc)
  out (o:signed<m> dc)
rules i -> o
| '< -> '<
| '> -> '>
| 'x -> 'f(x)
;

stream i:signed<8> dc from ...
stream o1:signed<8> dc to ...
stream o2:signed<8> dc to ...

net o1 = stream_apply f_inc i;
net o2 = stream_apply f_double i;
```

The `dc.cph` standard library (defined in `lib/caph`) defines several common higher order actors to operate on structured streams.

The `smap` actor, for instance, is a generalisation of the `stream_apply` actor introduced above, in which the applied function has the generic type $t_1 \rightarrow t_2$:

```
actor smap (f:$t1->$t2)
  in (i:$t1 dc)
 out (o:$t2 dc)
rules
| i:SoS     -> o:SoS
| i:Data x -> o:Data (f(x))
| i:EoS     -> o:EoS;
```

For example, if `f_inc` is a function incrementing its argument by one (just like in the first example)[43], we have :

$$\text{smap}_{f_inc}\ (\langle\ 1\ 2\ 3\ \rangle) = \langle\ f_inc(1)\ f_inc(2)\ f_inc(3)\ \ \rangle = \langle\ 2\ 3\ 4\ \rangle$$

The `smap2` actor is a variant of `smap` operating on two parallel streams :

```
actor smap2 (f:$t11*$t12->$t2)
  in (i1:$t11 dc, i2:$t12 dc)
 out (o:$t2 dc)
```

[43] and denoting "$\text{act}_p\ (\langle\ x_1\ \dots\ x_n\ \rangle)$" the application of actor *act*, with effective parameter *p*, to a streams $s = \langle\ x_1\ \dots\ x_n\ \rangle$.

```
rules
| (i1:SoS, i2:SoS)            -> o:SoS
| (i1:Data x, i2:Data y) -> o:Data (f(x,y))
| (i1:EoS, i2:EoS)            -> o:EoS ;
```

For example, if `add` is a function adding its two arguments, we have :

$$\text{smap2}_{add}\ (\langle\ x_1\ \ldots\ x_n\rangle, \langle\ y_1\ \ldots\ y_n\rangle) = \langle\ x_1 + y_1\ \ldots\ x_n + y_n\ \rangle$$

The `sfold` higher order actor "reduces" structured streams by applying a reducing function over each list of this stream. Formally :

$$\text{sfold}_{f,z}\ (\langle\ x_1\ x_2\ \ldots\ x_n\rangle) = f(f(f(f(z, x_1), x_2)..., x_n)$$

For example :

$$\text{sfold}_{+,0}\ (\langle\ 1\ 2\ 3\ \rangle) = 0 + 1 + 2 + 3 = 6$$

The `ssfold` actor is a generalisation of the `sfold` operating on lists of lists. Formally, if we note $l_i = \langle\ x_{i,1}\ x_{i,2}\ \ldots\ x_{i,m_i}\ \rangle$:

$$\text{ssfold}_{f,z}\ (\langle\ l_1\ \ldots\ l_n\rangle) = \langle\ \text{sfold}_{f,z}(l_1)\ \ldots\ \text{sfold}_{f,z}(l_n)\ \rangle$$

For example :

$$\text{ssfold}_{+,0}\ (\langle\ \langle\ 1\ 2\ 3\ \rangle\ \langle\ 4\ 5\ 6\ \rangle\ \rangle) = \langle 0 + 1 + 2 + 3 \quad 0 + 4 + 5 + 6\rangle = \langle\ 6\ 15\rangle$$

The `dc.cph` library defines several variant and extensions of the HO actors : `smapi`, when the applied function also depends on the position of the data in the stream, `sfold2`, for reducing two parallel stream, *etc.*

2.4.7 Network declarations

The network language is used to define the network of actors which describes the application. This involves defining how actors are instanciated and the interconnection between these instances. The basic concepts have been introduced in Sec. 1.3.1. Two kinds of values are declared for this : *wires* and *wiring functions*.

Wires

Wire declarations bind an identifier (resp. set of identifiers) to a wire (resp. set of wires) obtained when instanciating an actor or a network of actors.
Wire declarations have the form

`net` *<network_pattern>* = *<network_expression>*

where

- *<network_pattern>* is either a single identifier or a comma-separated list of identifiers enclosed in brackets (a so-called *tuple pattern*),
- *<network_expression>* is an expression representing a (sub)network of actors.

The previous declarations bind the output(s) of this network to the identifier(s) appearing in the pattern, offering a means to subsequently "wire" the corresponding output(s).

Network expressions

Network expressions can be classified into two main categories :

- expressions representing sub-network of actors,
- expressions denoting values to be used as actor parameters.

The **first category** is described by the syntax given in Fig. 2.2 (this is an excerpt from the full concrete syntax description given in Chap. 3) :

```
⟨net_expr⟩ ::= ⟨simple_net_expr⟩
           |  ⟨simple_net_expr⟩ ⟨simple_net_expr⟩+                      – application
           |  "(" ⟨net_expr⟩ "," ... "," ⟨net_expr⟩ ")"                 – t-uple
           |  "let" [ "rec" ] ⟨net_bindings⟩ "in" ⟨net_expr⟩            – local definition
           |  "function" ⟨net_pattern⟩ "->" ⟨net_expr⟩                  – function definition

⟨simple_net_expr⟩ ::= ⟨var⟩                                              – identifiers
           |  "()"
           |  "(" ⟨net_expr⟩ ")"
```

Figure 2.2: Syntax of the network language (excerpt)

Within this category

- *identifiers* refer either to previously defined wires, stream inputs or actors.
- *applications* "apply" a network expression to set of network expressions; two kind of values can be applied :
  - actors, like in the previous example,
  - wiring functions, which are described in the next section.

  In the former case, two kinds of arguments must be supplied, depending on whether the applied actor has been declared with parameters or not.

  - When the applied actor has no parameter, then the argument must be a single identifier, or a tuple of identifiers. For example, if an actor `add` has been defined as

    ```
    actor add
      in (a: signed<8>, b: signed<8>)
     out (c: signed<8>)
    ...
    ```

    then its application will be denoted as

    ```
    net x = ...
    net y = ...
    net r = add (x,y);
    ```

  - When the applied actor accepts parameters, then the value(s) of the parameter(s) must be passed as an extra argument *before* the wire arguments.

    Example 1. Consider an actor `scale`, accepting one parameter `k` and one input `a` defined as :

```
actor scale (k:signed<8>)
  in (a: signed<8>)
 out (c: signed<8>)
rules
| a:x -> c:k*x
```

then its application will be denoted, taking `k=2` here for example, as

```
net x = ...
net r = scale 2 x;
```

Example 2. Consider now an actor `scale2`, accepting two parameters, `k1` `k2`, and two inputs, `a` and `b`, defined as :

```
actor scale2 (k1:signed<8>, k2:signed<8>)
  in (a1: signed<8>, a2:signed<8>)
 out (c: signed<8>)
rules
| (a1:x, a2:y) -> c:k1*x+k2*y
```

then its application will be denoted, taking `k1=2` and `k2=3` here, as

```
net x = ...
net y = ...
net r = scale2 (2,3) (x,y);
```

- *Tuple patterns and expressions* are used to simultaneously bind several identifiers, like in

  ```
  net (x2,x3) = (scale 2 x, scale 3 x)
  ```

  where the `scale` actor defined above is instanciated twice.

- *local definitions*, introduced with the `let ...in ...`, construct, bind an identifier (or a set of identifiers) to an expression with a limited scope. The name(s) introduced by the binding is (are) only visible within the target expression. The special case of *recursive* local definitions is discussed in Sec 2.4.7.

The **second category** of network expressions is used to represent values to be passed as parameters to actors (such as `2` and `(2,3)`) in the examples given above, respectively). These values are limited to :

- identifiers refering to globally defined values,
- scalar or array constants,
- tuples of the above.

Wiring functions

As introduced in Sec. 1.3.1, wiring functions simplifies the description of complex networks by allowing the definition of reusable, polymorphic *network patterns*. Wiring function declarations have the form[44] :

`net` *fid* <*network_pattern*> = <*network_expression*>

where *fid* is the name of the function and *network_pattern* gives the name(s) of the formal argument(s)[45].

Application of such a function follows the classical strict, *call-by-value* evaluation strategy : each supplied argument is evaluated[46] and the resulting value is bound to the corresponding formal argument; the right-hand side expression is then evaluated in an environment augmented with the resulting bindings.

Listing 2.23:

```
net inc2f x = inc (inc x);
net o = inc2f i;
```

In the example given listing 2.23, the `inc2f` function represents a network in which the output wire (`y`) is obtained by having the input wire (`x`) traversing two `inc` boxes. It could be represented by the "sub-network"of Fig. 2.3[47]. The application of this function at line 2 *instanciates* this sub-network and creates the network of Fig. 2.4, in which the sub-network input (resp. output) has been bound to actual the input (resp. output) `i` (resp. `o`).

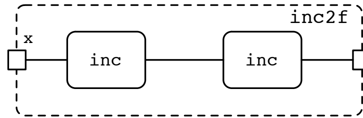


Figure 2.3: A representation of the `inc2f` wiring function defined in listing 2.23

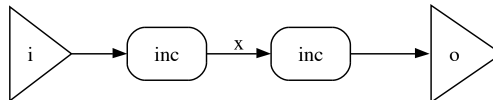


Figure 2.4: The network resulting from applying the `inc2f` function

Wiring functions can take several arguments. In this case, arguments can be passed either as a tuple or as a sequence.

[44] Actually, and as evidenced by the abstract syntax of the language given in Chap. 4, there's no distinction wire declarations and wiring function declarations. The latter is just a syntactic shorthand. In fact, there's an extra case to the rule defining the syntax of expressions :

⟨*net_expr*⟩ ::= ...
| "function" ⟨*simple_net_pattern*⟩ "->" ⟨*net_expr*⟩

and the declaration `net f <pat> = <exp>` is handled as `net f = function <pat> -> <exp>`.

[45] *Recursive* network definitions are discussed separately in Sec 2.4.7.

[46] Since all network expressions are pure – *i.e.* cannot involve side-effects –, the order of evaluation is irrelevant here.

[47] The term sub-network is used here because there's no real input nor output, only "slots" – drawn as small square boxes in Fig. 2.3 – intended to be connected to actual wires

Example

```
net foo (x,y) = add (x, inc y);
net o = foo (i1,i2);
```

The previous example could also have been written (strictly equivalent form) :

```
net foo x y = add (x, inc y);
net o = foo i1 i2;
```

When this second definition form for the `foo` function[48] is used as above, it is strictly equivalent to the first one (in other words, the network corresponding to the two previous programs are identical). But the second form has a bonus : it allows *partial application* of the `foo` function. Partial application means fixing the value of some arguments to obtain another function, which can be viewed as "specialized" version of the original one. In our case, partial application means fixing some part of the sub-network defined by the original function. For example, if we write

```
net foo1 = foo i;
```

then `foo1` is the function obtained by assigning the value `i` to `x` in the definition of `foo` (Fig. 2.5). In other words, the `foo1` function is equivalent to the function that could have been written as

```
net foo2 y = add (i, inc y);
```

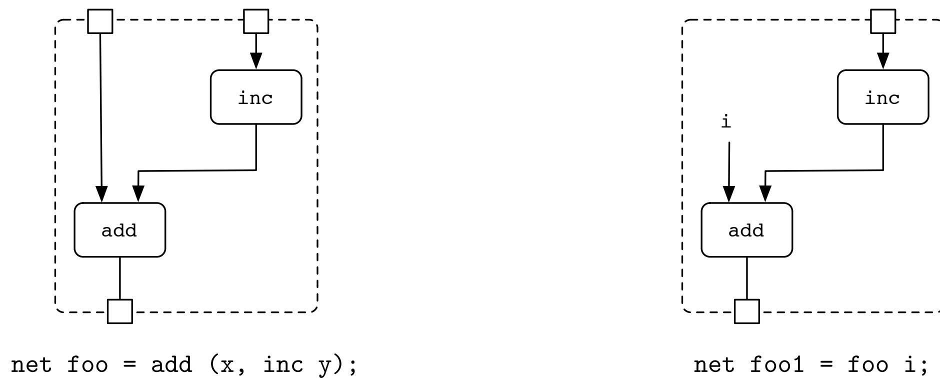
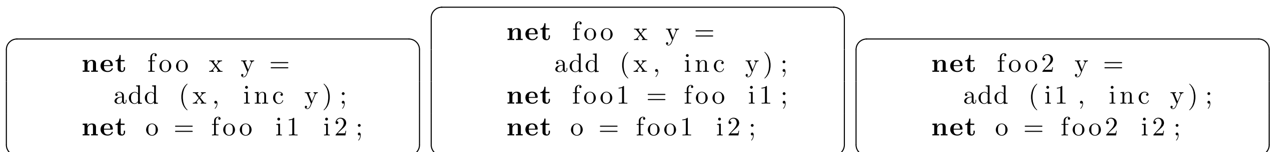


Figure 2.5: Partial application

Then, the three programs of Fig. 2.6 are stricly equivalent :



```
net foo x y =
  add (x, inc y);
net o = foo i1 i2;
```

```
net foo x y =
  add (x, inc y);
net foo1 = foo i1;
net o = foo1 i2;
```

```
net foo2 y =
  add (i1, inc y);
net o = foo2 i2;
```

Figure 2.6: Three equivalent programs

Higher-order wiring functions. For now we have been passing only *wires* as arguments to wiring functions. But functions can also be passed as arguments to wiring functions

[48]Technically known as the *curried* form.

Example

```
net twice (f,x) = f (f x);
```

The `twice` (higher-order) function takes two arguments, a function `f` and a wire `x`. It instanciates the sub-network corresponding to `f` a first time, binding its input to the `x` wire, and then a second time, binding the former output to the input of the latter. For example, writing :

```
net o = twice (inc2f,i);
```

produces the network depicted in Fig. 2.7.

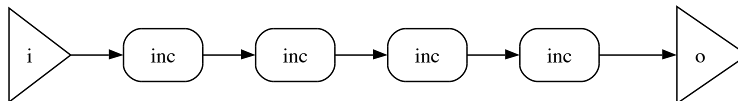


Figure 2.7: `net o = twice (inc2f, i)`

This is the combination of partial application and higher-order functions that allows the encapsulation of graph patterns as wiring functions, as illustrated in Sec. 1.3.1 with the function `diamond`. In the last example of this Section :

```
net o = diamond (dup, inc, diamond (dup,inc,dec,mul), mul) i;
```

the inner application of function `diamond` is partial, so that it can be passed, as a wiring function, as an argument to the outer application.

Recursive wiring

By recursive wiring, we mean the ability for a box (resp. set of boxes) to take as input some wires originating, directly or indirectly, from its (resp. their) output(s).

This is illustrated in Fig. 2.8, where the network on the left is described by the program on the right. Here, the second output of actor `A` is re-injected, after passing through actor `B` to its second input.

Mutually recursive wiring between actors is also possible, as shown on Fig. 2.9.

As shown above, recursive wiring is used to describe *cyclic* networks. In the classical dataflow model, cycles are typically used to implement iterations [3]. In CAPH, iterations are generally handled at the actor level, using local variables.

Functionnaly, the two formulations are equivalent : in the recursive version, the "state" memorized in the local variables is simply "externalized" and held in the looping wires.

To illustrate this, let's go back to the actor `suml` introduced at the end of section 2.4.4 :

Listing 2.24:

```
actor suml
  in (i:signed<16> dc)
  out (o:signed<16>)
var state : { S0, S1 } = S0
var sum : int
rules
  | (state:S0, i:'<) -> (sum:0, state:S1)
  | (state:S1, i:'>) -> (o:sum, state:S0)
  | (state:S1, i:'v) -> (sum:sum+v);
```

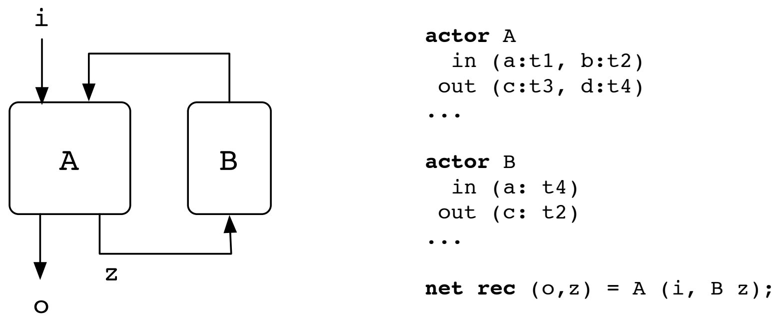


Figure 2.8: Example of recursive wiring

The `suml` actor computes the sum of each list given as input, using two local variables : one holding the state of the computation and the other the accumulator. This actor can be used, for example, in the following program :

Listing 2.25:

```
actor suml  ... -- as above

stream i:signed<16> dc from "sample.txt";
stream o:signed<16> to "result.txt";

net o = suml i;
```

If the input file `sample.txt` contains, let say

```
< 1 2 3 > < 4 5 6 >
```

then output file `result.txt` will contain

```
6 15
```

Listing 2.26 shows an equivalent program based on a variant of the `suml` actor in which local variables have been replaced by feedback wires :

Listing 2.26: A recursive reformulation of actor `suml`

```
actor suml_rec
  in (i:signed<16> dc, sum:signed<16>)
 out (o:signed<16>, nsum:signed<16>)
rules
| (i:'<          ) -> nsum:0
| (i:'>, sum:s) -> o:s
| (i:'v, sum:s) -> nsum:s+v
;
```

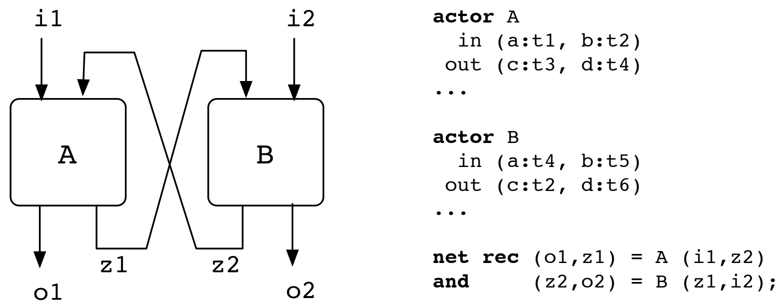


Figure 2.9: Example of mutually recursive wiring

```
stream i : signed<16> dc from "sample.txt";
stream o : signed<16> to "result.txt";

net rec (o,z) = suml_rec (i,z);
```

Note that `suml_rec` is a *state-less* actor : it has no local variable. The running sum is now implicitely kept on a external feedback wire, connecting its `nsum` output to its `sum` input. The three rules describing its behavior can be read as :

- when reading a "`<`" control token on input `i`, write the initial value of the accumulator on the feedback wire,
- when reading a "`>`" control token on input `i`, write the final value of the accumulator, available on input `sum`, on output `o`,
- when reading a value on both inputs, add these values and write the sum on the feedback wire.

The corresponding dataflow graph is given in Fig. 2.10.

Note that, if *functionally* equivalent, this second program actually generates less efficient code, since the recursive values must in this case pass through external links, which introduce latencies. For this reason, formulations using local variables should always be prefered, when possible, when writing CAPH programs[49].

Wiring of IO-less actors

The special network expression "`()`" (called `unit`) is used to instantiate input or output-less actors. An example is given in Listing 2.27 and Fig. 2.11, which use the actors `src`, `snk` and `double` previously defined in Listings 2.14, 2.15 and 2.1. Such networks, with no input nor output stream or port are said to be "*closed*". Their main usage is to model applications for which input and output streams are

[49] There's a noticeable exception to this rule; it concerns the `d1lr` actor defined in library `img_ops.cph` and implementing line delay on images. In this case, recursive wiring is deliberately used in order to have the delayed line stored in a external FIFO, because this scheme is more efficiently handled by the VHDL synthetisers.

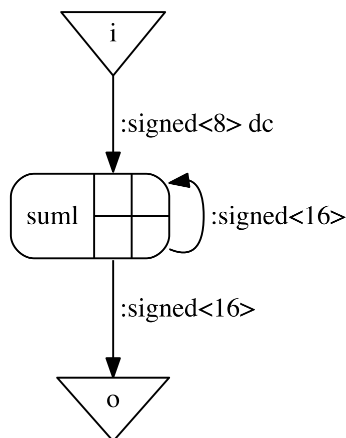


Figure 2.10: A cyclic dataflow network

not read from (resp. to) external devices[50] but are produced and consumed by actors performing their operations by side-effect. A typical example will be an actor reading output data from a memory or displaying results in a graphical window[51].

Listing 2.27: Example of network with IO-less actors

```
actor src in (i: unit) out (o: int) ...;
actor snk in (i: int) out (o: unit) ...;
actor double in (i: int) out (o: int) ...;

net () = snk (double (src ()));
```

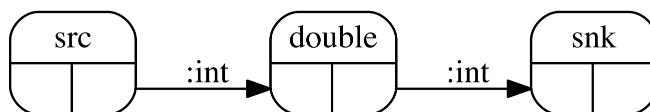


Figure 2.11: The network corresponding to the program of Listing 2.27

[50]Viewed as `streams` or `ports` at the network level.
[51]In practice, such actors will be provided using the `implemented` pragma.

Higher-order wiring primitives

Just like certain kinds of actor behavior can be encapsulated using higher order actors (see Sec. 2.4.6), certain patterns in data-flow graphs can be encoded concisely in CAPH using so-called *higher-order wiring primitives*. The four basic higher-order primitives are `map`, `napp`, `foldl` and `pipe`.

The **map** higher-order primitive[52] is illustrated in Fig. 2.12. It is used whenever the same processing has to be carried out on a set of separate data streams.

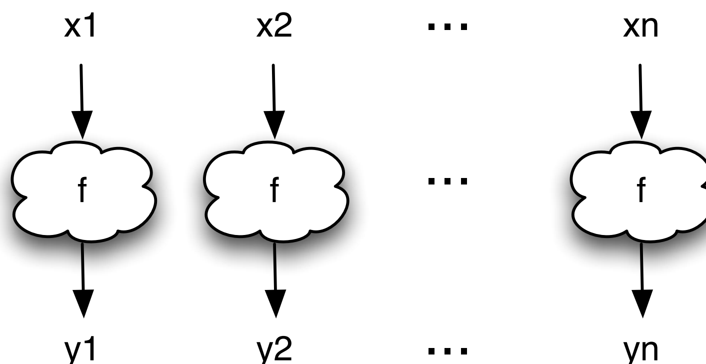


Figure 2.12: A parallel graph pattern

A simple description of the graph given in Fig. 2.12 can be given by simply replicating `net` definitions :

```
net y1 = f x1;
net y2 = f x2;
...
net yn = f xn;
```

The `map` higher-order primitive offers a more concise way of describing this graph :

```
net (y1,y2,... ,yn) = map f (x1,x2,... ,xn);
```

The `map` function takes two arguments :

- a wiring function f, with type $\tau \rightarrow \tau'$,
- a tuple $(x_1, \dots, x_n)$ of wires[53], each having type τ

and returns a tuple of wires, each of type τ', by applying the f function to each wire x_i. In other words :

$$\text{map } f \ (x_1, \dots, x_n) = (f \ x_1, \dots, f \ x_n)$$

[52] Not to be confused with the `smap` higher order *actor*.

[53] Actually, and for technical reasons, the type of the second argument `map` is not $(\tau, \dots, \tau)$ but (τ, α) *bundle*, where *bundle* is a built-in type constructor for representing bundles of homogeneous types. The type-checking phases unifies the types $\underbrace{(\tau, \dots, \tau)}_{n}$ and (τ, n) *bundle*. Since this unification is carried out automatically by the type checker, the `map` function can be viewed as operating and returning directly tuples of values (of the same type, of course).

The first argument of `map` can be an simple actor, with or without parameter(s) or a complex wiring function, as illustrated in Fig. 2.13, 2.14 and 2.15.

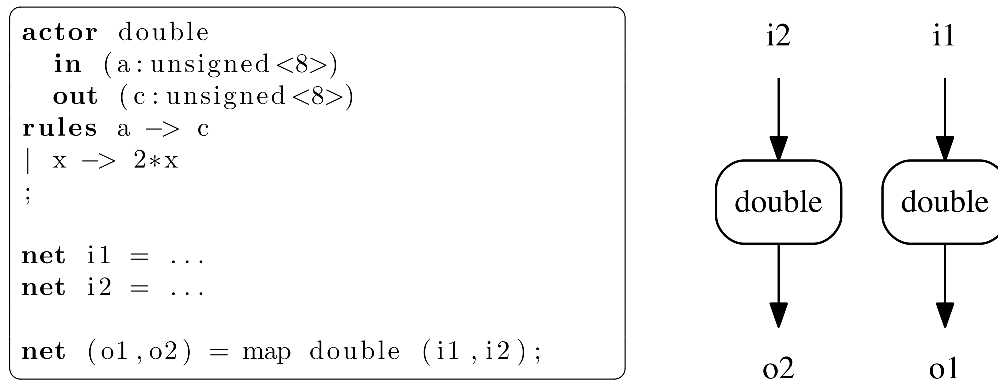


Figure 2.13: `map` - example 1

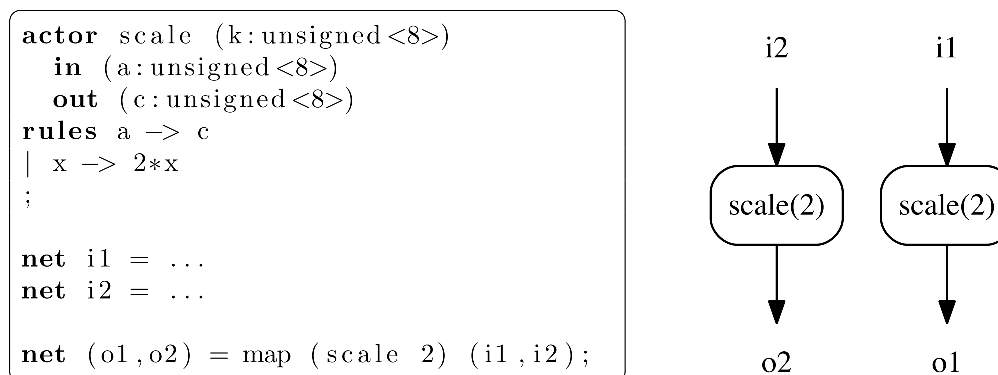


Figure 2.14: `map` - example 2

A very useful variant of the `map` function is `mapi`. As `map`, the `mapi` function takes two arguments, a wiring function f, with type unsigned $\rightarrow \tau \rightarrow \tau'$ and a tuple of wires, and returns a tuple of wires, obtained by applying the f function to each wire of the input tuple. But the f function here takes an

```
actor scale (k:unsigned<8>)
  in (a:unsigned<8>)
  out (c:unsigned<8>)
rules a -> c
| x -> 2*x
;

net i1 = ...
net i2 = ...

net f x = scale 2 (scale 4 x);

net (o1,o2) = map f (i1,i2);
```

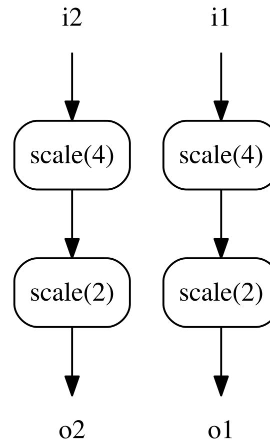


Figure 2.15: `map` - example 3

extra argument which is automatically set to the index[54] of the corresponding wire in the input tuple. In other words, and more precisely :

$$\text{mapi } f\ (x_1, \dots, x_n) = (f\ 0\ x_1, \dots, f\ (n-1)\ x_n)$$

A typical usage of the `mapi` function is given in Fig 2.16. Here the extra argument (ranging from 0 to 3) is used to adjust the scaling factor applied in each branch of the parallel pattern.

As for the `map` function, the first argument of the `mapi` function can be an actor or a any wiring function having a type of the form unsigned $\rightarrow \tau' \rightarrow \tau'$:

Another variant of the `map` function is `map2`. The `map2` function takes three arguments :

- a wiring function f, with type $\tau_1 \times \tau_2 \rightarrow \tau'$,
- a first tuple of wires, each with type τ_1,
- a second tuple of wires, each with type τ_2,

and returns a tuple of wires, obtained by applying the f function to each pair of input wires. In other words :

$$\text{map2 } f\ ((x_1, \dots, x_n), (y_1, \dots, y_n)) = (f\ (x_1, y_1), \dots, f\ (x_n, y_n))$$

Of course, the two input tuples must have the same size. Fig. 2.17 gives an example involving the `map2` function.

As for the `map` function, `map2` has a variant, named `map2i` for which the function argument takes an extra argument :

[54] Starting at 0, not 1.

```
const coeff = [2,4,8,16] : unsigned<8> array[4];

actor scale (k:unsigned<8>)
  in (a:unsigned<8>)
  out (c:unsigned<8>)
rules a -> c
| x -> k*x
;

net i1 = ...
net i2 = ...
net i3 = ...
net i4 = ...

net f i x = scale (coeff[i]) x;

net (o1,o2,o3,o4) = mapi f (i1,i2,i3,i4);
```

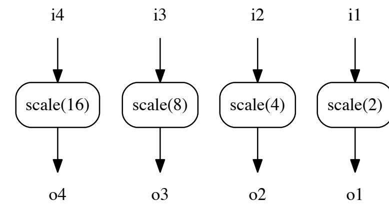


Figure 2.16: `mapi` - example

```
actor add (k:unsigned<8>)
  in (a:unsigned<8>, b:unsigned<8>)
  out (c:unsigned<8>)
rules (a,b) -> c
| (x,y) -> k*x+y
;

net i11 = ...
net i12 = ...
net i21 = ...
net i22 = ...

net (o1,o2) =
  map2 (add 2) ((i11,i12),(i21,i22));
```

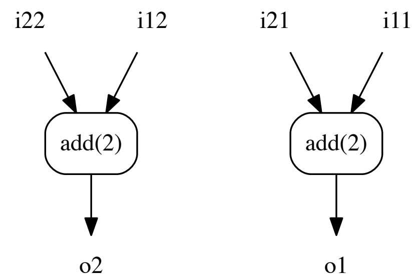


Figure 2.17: `map2` - example

$$\text{map2i } f \ ((x_1,\ldots,x_n),(y_1,\ldots,y_n)) = (f \ 0 \ (x_1,y_1),\ldots,f \ (n-1) \ (x_n,y_n))$$

With the `map` family of higher-order primitives, the same processing is replicated over a set of distinct data streams (the number of streams giving the “width” of the data-flow graph). The `napp` higher-order primitive offers a way of replicating a processing on the *same* data stream, the number of replications (the “width” of the graph) being here specified as an extra argument. Typical examples are given in Fig. 2.18 and 2.19.

The `napp` function takes three arguments :

- a integer n (which must be a statically bound constant),
- a wiring function f, with type $\tau \rightarrow \tau'$,

- a wire x, of type τ

and returns a tuple of n wires, each of type τ', by applying n times the f function to wire x. In other words :

$$\text{napp } n \ f \ x = (\underbrace{f \ x, \ldots, f \ x}_{n})$$

As for `map`, `nappi` is a variant of `napp` for which the applied fonction (f) takes a extra parameter automatically bound to the replication index :

$$\text{nappi } n \ f \ x = (f \ 0 \ x, \ldots, f \ (n-1) \ x)$$

The `nappi` higher-order primitive is illustrated in Fig. 2.20 (with $n = 3$).

```
actor double
  in (i:unsigned<8>)
  out (o:unsigned<8>)
rules
| i:x -> o:2*x
;

stream i:unsigned<8> ...
stream o1:unsigned<8> ...
stream o2:unsigned<8> ...
stream o3:unsigned<8> ...

net (o1,o2,o3) = napp 3 double i;
```

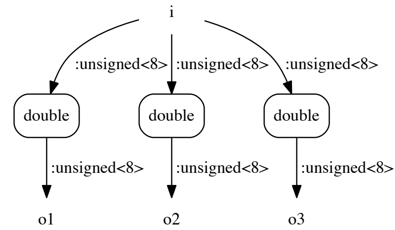


Figure 2.18: `napp` - example 1

```
actor addm
  in (i1:unsigned<8>, i2:unsigned<8>)
  out (o:unsigned<8>)
rules
| (i1:x1, i2:x2) -> o:x1+x2
;

stream i1:unsigned<8> ...
stream i2:unsigned<8> ...
stream o1:unsigned<8> ...
stream o2:unsigned<8> ...

net (o1,o2) = napp 2 addm (i1,i2);
```

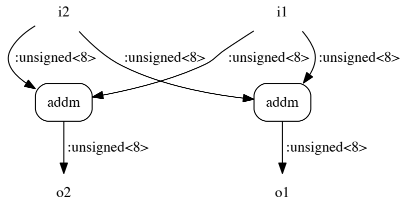


Figure 2.19: `napp` - example 2

```
actor scale (k: unsigned<8>)
  in (i:unsigned<8>)
  out (o:unsigned<8>)
rules
| i:x -> o:(k+1)*x
;

stream i:unsigned<8> ...
stream o1:unsigned<8> ...
stream o2:unsigned<8> ...
stream o3:unsigned<8> ...

net (o1,o2,o3) = nappi 3 scale i;
```

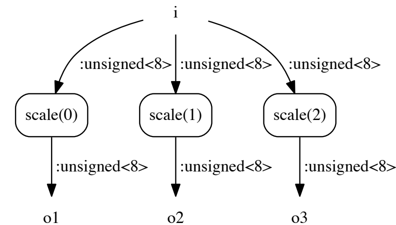


Figure 2.20: `nappi` - example

The `map` function (and its variants `mapi`, `map2` and `map2i`) encodes parallel *replication* graph patterns. Another higher-order wiring primitive, `foldl` (fold left)[55] is provided for encoding parallel *reduction* patterns. Formally, if

- x_1 is a value having type τ,
- $x = (x_2, \dots, x_n)$ a tuple of values having type τ and
- f a function having type $\tau \times \tau \to \tau$

then

$$\text{foldl } f \; x_1 \; (x_2, \dots, x_n) = f(\dots f(f(x_1, x_2), x_3) \dots, x_n)$$

In other words, the result of applying *foldl* to f and x is obtained by applying f as a binary reduction operator over the tuple $(x_1, \dots, x_n)$. An example showing the effect of the `foldl` function is given in Fig. 2.21.

As for `map`, the `foldl` function has a variant, named `foldli` for dealing with *indexed* reduction. Formally, if x is a value with type τ, $(x_2, \dots, x_n)$ is a tuple of values with type τ and f a function having type unsigned $\to \tau \times \tau \to \tau$, then

$$\text{foldli } f \; x_1 \; ((x_2, \dots, x_n) = f \; (n-1) \; (\dots f \; 1 \; (f \; 0 \; (x_1, x_2), x_3) \dots, x_n)$$

A last variant of the `foldl` function is `foldt`, which performs a *dyadic* reduction, as shown in Fig. 2.22.

The `pipe` higher-order wiring primitive can be used to encode linear “pipelined” graph patterns, as illustrated in Fig. 2.23.

The `chain` higher-order wiring primitive is similar to `pipe` but produces a tuple of values corresponding to each stage of the pipe, as illustrated in Fig. 2.24.

[55]Not to be confused with the `sfold` and `ssfold` higher order *actors*.

```
actor madd (k: unsigned<8>)
  in (a:unsigned<8>, b:unsigned<8>)
  out (c:unsigned<8>)
rules (a,b) -> c
| (x,y) -> x*k+y
;

net i1 = ...
net i2 = ...
net i3 = ...
net i4 = ...

net f (x,y) = madd 2 (x,y);

net o = foldl f i1 (i2,i3,i4);
```

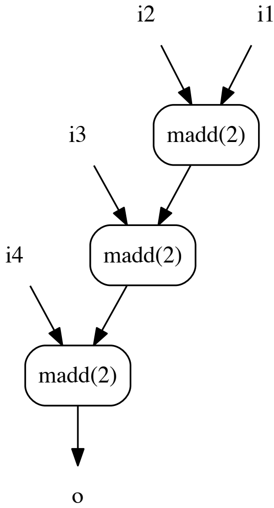


Figure 2.21: `foldl` - example

```
actor add
  in (a:unsigned<8>, b:unsigned<8>)
  out (c:unsigned<8>)
rules (a,b) -> c
| (x,y) -> x+y;

net i1 = ...
net i2 = ...
net i3 = ...
net i4 = ...

net o = foldt add (i1,i2,i3,i4);
```

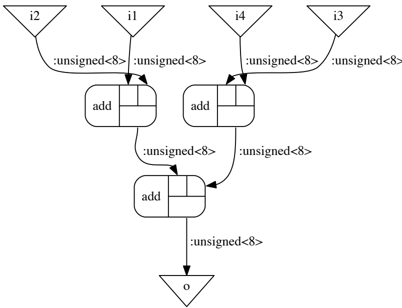


Figure 2.22: `foldt` - example

```
actor double
  in (a:unsigned<8>)
  out (c:unsigned<8>)
rules a -> c
| x -> 2*x ;

net i = ...

net o = pipe 4 double i;
```

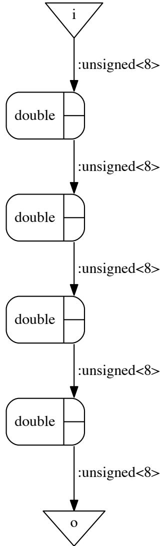


Figure 2.23: `pipe` - example

```
actor double ...;

actor incr
  in (a:unsigned<8>)
  out (c:unsigned<8>)
rules a -> c
| x -> x+1
;

stream i:unsigned<8> from ...;
stream o1:unsigned<8> to ...";
stream o2:unsigned<8> to ...";
stream o3:unsigned<8> to ...";
stream o4:unsigned<8> to ...";

net (o1,o2,o3,o4) = chain 4 double i;
```

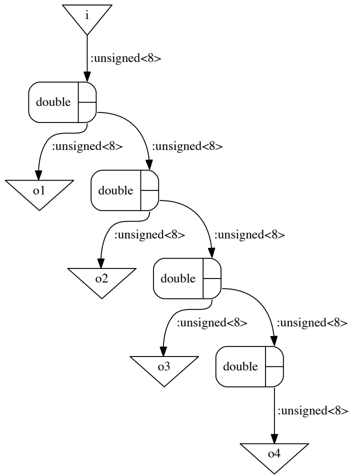


Figure 2.24: `pipe` - example

The higher-order primitives (HOP) introduced above can of course be combined to define powerful network building functions. As an illustration, the program of Listing 2.28 describes a generic $1 \times n$ FIR filter using the `chain` and `foldl` HOPs[56]. The former is used to build the values corresponding to the input stream delayed by $1, 2, \ldots n$ samples. The latter to describe the multiply-accumulate computation tree. The corresponding dataflow graph is given in Fig. 2.25.

Listing 2.28: A $1 \times n$ FIR filter described in CAPH using the `chain` and `foldl` higher-order primitives

```
#include "stream_ops.cph"  -- for [d], one-sample delay actor

const coeff = [1,2,3,2,1] : signed<12> array[5];

actor madd (j:unsigned<4>)
   in (acc:int<s,m>, tap:int<s,m>)
  out (o:int<s,m>)
rules
| (acc:s, tap:x) -> o:x*coeff[j]+s
;

net fir x =
  let xs = chain 5 (d 0) x in
  foldli madd x xs;

stream i:signed<12> from "sample.txt";
stream o:signed<12> to "result.txt";

net o = fir i;
```

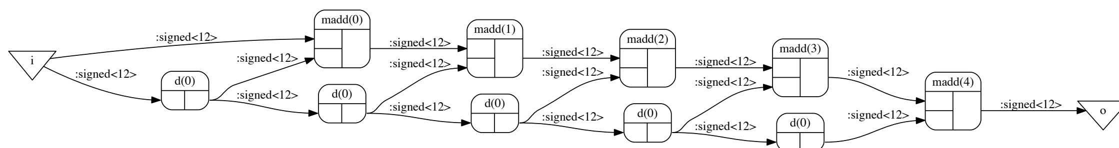


Figure 2.25: The dataflow graph obtained from the program in Listing 2.28

2.5 A complete example

Listing 2.29 gives the full CAPH source code of a basic application for extracting edges in images. Figure 2.27 gives the corresponding dataflow network.

For each pixel $P_{i,j}$, the local gradient magnitude is approximated by the sum of the absolute value of the horizontal and vertical derivatives ($|P_{i,j} - P_{i,j-1}| + |P_{i,j} - P_{i-1,j}|$) and the resulting value is compared to a fixed threshold for producing a binary image (with edge pixels encoded as 1 and background pixels as 0).

An example of input and output image (obtained with the simulator) is given Fig. 2.26.

Five actors are involved in this application :

[56] This program can be found in the directory `examples/fir/fir1n` of the distribution.

Input image Result image

Figure 2.26:

- the `d1p` and `d1l` actors are defined in the library `img_ops.cph`[57]. They are used by the `dx` and `dy` wiring functions (lines 31-32) to compute the horizontal and vertical derivatives. They delay an image by one pixel and by one line respectively, inserting a null value in the first colum (resp. line) and discarding the first column (resp. line). For example,
  - `d1p:< < 1 2 3 4 > < 5 6 7 8 > > = < < 0 1 2 3 > < 0 5 6 7 > >`
  - `d1l:< < 1 2 3 4 > < 5 6 7 8 > > = < < 0 0 0 0 > < 0 1 2 3 > >`
- the `add` actor (lines 4-10) performs the normalized addition of two structured streams, preserving the structure; the `n` parameter is the normalization factor;
- the `asub` actor (lines 13-19) computes the absolute value of the difference of two structured streams; for this it uses a global function `f_abs` defined lines 1-2 of the program;
- the `thr` actor (lines 22-28) binarizes a structured stream according to a given threshold; the threshold is specified as a parameter; the resulting stream contains only 0's and 1's.

Lines 31-32 defines the two wiring functions `dx` and `dy` computing the horizontal and vertical derivatives of their argument. The `[0]` argument passed to the `d1p` and `d1l` actors is the value of the pixel inserted in the first column (resp. line).

For simulation and SystemC execution, the input (resp. output) stream is here read (written) directly in (resp. to) a PGM file (this is an image) (lines 34-35). The input file will be specified at compile time using the macro mechanism described in Sec 10.

Lines 37-38 gives the network description. The value `gm` is the image of the gradient magnitude (computed as the sum of the absolute values of the horizontal and vertical derivatives). The result is obtained by passing this image to the `thr` actor. As for the input file, the threshold value will be specified here at compile time[58].

Listing 2.29: A basic edge extraction application in Caph

```
1  function f_abs x =
2    if x < 0 then -x else x : signed<m> -> signed<m>;
3
4  actor add (n:unsigned<4>)
5    in (a:signed<m> dc, b:signed<m> dc)
6    out (c:signed<m> dc)
7  rules (a,b) -> c
8  | ('<,'<) -> '<
```

[57]See chap. 11; `d1l` is actually a wiring function invoking the `d1lr` actor in a recursive manner, as shown in Fig. 2.27.

[58]A more adaptative approach would of course *compute* this threshold from some statistics extracted from the input image.

```
9  | ('>,'>) -> '>
10 | ('p1,'p2) -> '(p1+p2)>>n ;
11
12 actor asub
13   in (a:signed<m> dc, b:signed<m> dc)
14   out (c:signed<m> dc)
15 rules (a,b) -> c
16 | ('<,'<) -> '<
17 | ('>,'>) -> '>
18 | ('p1,'p2) -> 'f_abs(p1-p2) ;
19
20 actor thr (t:signed<m>)
21   in (a:signed<m> dc)
22   out (c:unsigned<1> dc)
23 rules a -> c
24 | '< -> '<
25 | '> -> '>
26 | 'p -> if p > t then '1 else '0 ;
27
28 net dx i = asub (i, d1p 0 i);
29 net dy i = asub (i, d1l 0 i);
30
31 stream inp:signed<10> dc from "%arg1";
32 stream res:unsigned<1> dc to "result.txt";
33
34 net gm = add 0 (dx inp, dy inp);
35 net res = thr %arg2 gm;
```

Note (version 2.8). The program given in listing 2.29 can be rewritten in a more concise manner using *higher order actors* (see Sec. 2.4.6). The definition of `thr` actor can be replaced by an instanciation of the `smap` actor, with the following function :

```
function f_thr(x) = if x>%th then (1:unsigned<1>) else (0:unsigned<1>) : signed<m>->unsigned<1>;
```

Similarly, the definition of `add` and `asub` actors can be replaced by an instanciation of the `smap2` actor, with the following functions, respectively :

```
function f_add(x,y) = x+y : signed<m> * signed<m> -> signed<m>;
function f_sub(x,y) = f_abs(x-y) : signed<m> * signed<m> -> signed<m>;
```

The corresponding program and resulting dataflow graph are given in listing 2.30 and figure 2.28 respectively.

Listing 2.30: The program of listing 2.29 rewritten with higher order actors

```
1 function f_abs x =
2   if x < 0 then -x else x : signed<s> -> signed<s>;
3 function f_thr(x) =
4   if x > %th then (1:unsigned<1>) else (0:unsigned<1>)
5   : signed<m> -> unsigned<1>;
6 function f_add(x,y) = x+y : signed<m> * signed<m> -> signed<m>;
7 function f_sub(x,y) = f_abs(x-y) : signed<m> * signed<m> -> signed<m>;
```

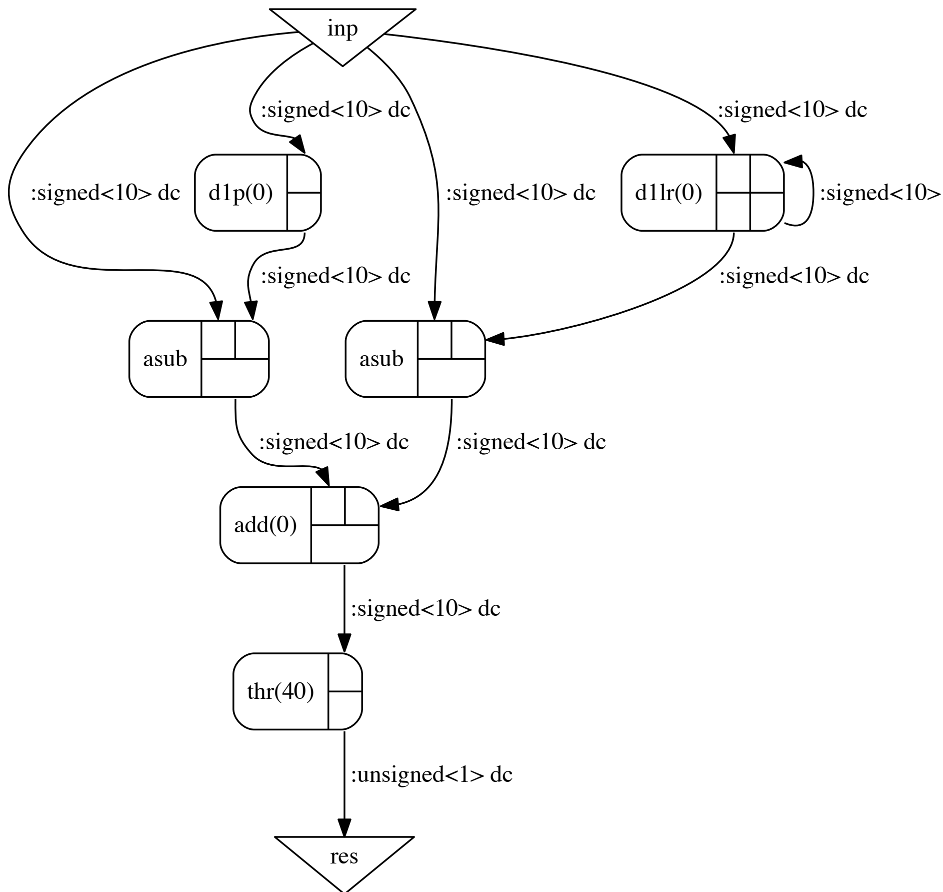


Figure 2.27: The data-flow graph for the edge extraction application

```
 8
 9  net dx i = smap2 f_sub (i, d1p 0 i);
10  net dy i = smap2 f_sub (i, d1l 0 i);
11
12  stream inp:signed<10> dc from %ifile;
13  stream res:unsigned<1> dc to "result.txt";
14
15  net res = smap f_thr (smap2 f_add (dx inp, dy inp));
```

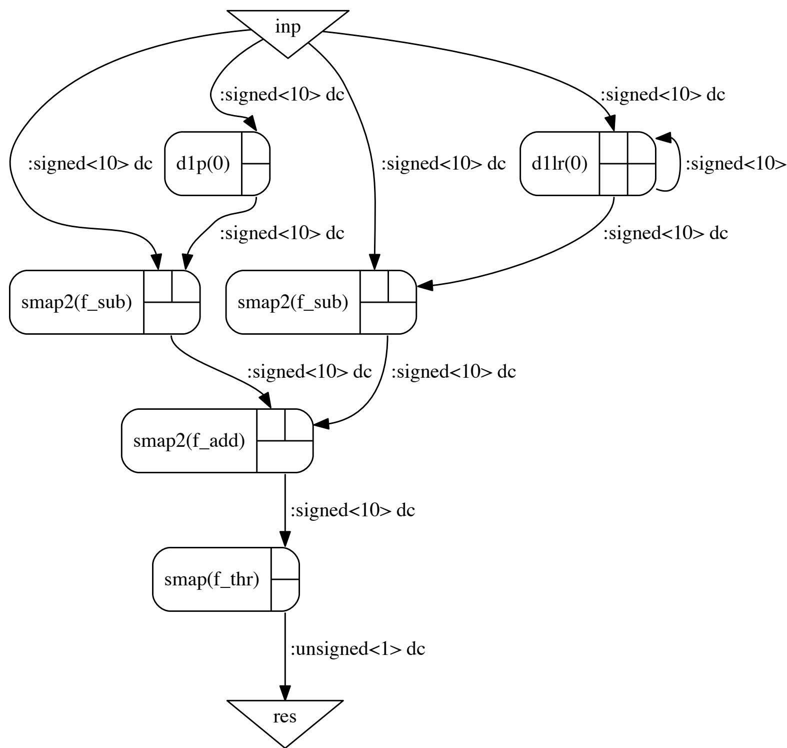


Figure 2.28: The data-flow graph derived from the program in listing 2.30 (to be compared with figure 2.27

Chapter 3

Syntax

This appendix gives a BNF definition of the concrete syntax for CAPH programs. The meta-syntax is conventional. Terminals are enclosed in double quotes `" ... "`. Non-terminals are enclosed in angle brackets `< ... >`. Vertical bars `|` are used to indicate alternatives. Constructs enclosed in brackets `[ ... ]` are optional. Parentheses `( ... )` are used to indicate grouping. Ellipses `(...)` indicate obvious repetitions. An asterisk (`*`) indicates zero or more repetitions of the previous element, and a plus (`+`) indicates one or more repetitions.

Programs

⟨*program*⟩ ::= ⟨*decl*⟩*

```
⟨decl⟩  ::= ⟨type_decl⟩ ";"
          | ⟨val_decl⟩ ";"
          | ⟨io_decl⟩ ";"
          | ⟨actor_decl⟩ ";"
          | ⟨net_decl⟩ ";"
          | ⟨pragma_decl⟩
```

Type declarations

```
⟨type_decl⟩ ::= ⟨abbrev_type_decl⟩
              | ⟨variant_type_decl⟩

⟨abbrev_type_decl⟩ ::= "type" ⟨id⟩ "==" ⟨type⟩

⟨variant_type_decl⟩ ::= "type" [⟨ty_params⟩] ⟨id⟩ [⟨sz_params⟩] "=" ⟨constr_decls⟩

⟨ty_params⟩ ::= ⟨ty_var⟩
              | "(" ⟨ty_var⟩ "," ... "," ⟨ty_var⟩ ")"

⟨sz_params⟩ ::= "<" ⟨sz_var⟩ ">"
              | "<" ⟨sz_var⟩ "," ... "," ⟨sz_var⟩ ">"

⟨constr_decls⟩ ::= ⟨constr_decl⟩ [ | ⟨constr_decls⟩ ]
```

⟨constr_decl⟩ ::= ⟨con_id⟩ [⟨impl_tag⟩] ["of" ⟨constr_args⟩]

⟨constr_args⟩ ::= ⟨type⟩
| ⟨type⟩ "*" . . . "*" ⟨type⟩

⟨impl_tag⟩ ::= "%" ⟨intconst⟩

Global value declarations

⟨value_decl⟩ ::= "const" ⟨id⟩ "=" ⟨expr⟩ [":" ⟨type⟩]
| "const" ⟨id⟩ "=" ⟨array_init⟩ [":" ⟨type⟩]
| "function" ⟨id⟩ ⟨fun_pattern⟩ "=" ⟨expr⟩ [":" ⟨type⟩]
| "function" ⟨id⟩ "=" "extern" ⟨vhdl_fn_name⟩ "," ⟨c_fn_name⟩ "," ⟨caml_fn_name⟩ ":" ⟨type⟩

⟨fun_pattern⟩ ::= ⟨id⟩
| "(" ⟨id⟩ "," . . . "," ⟨id⟩ ")"

⟨vhdl_fn_name⟩ ::= ⟨string⟩

⟨c_fn_name⟩ ::= ⟨string⟩

⟨caml_fn_name⟩ ::= ⟨string⟩

Actor declarations

⟨actor_decl⟩ ::= "actor" ⟨id⟩ ⟨actor_intf⟩ ⟨actor_body⟩

Actor interface

⟨actor_intf⟩ ::= ["(" ⟨actor_params⟩ ")"] "in" "(" ⟨actor_ins⟩ ")" "out" "(" ⟨actor_outs⟩ ")"

⟨actor_params⟩ ::= ⟨actor_param⟩ ["," ⟨actor_params⟩]

⟨actor_param⟩ ::= ⟨id⟩ ":" ⟨type⟩

⟨actor_ins⟩ ::= ⟨actor_ios⟩

⟨actor_outs⟩ ::= ⟨actor_ios⟩

⟨actor_ios⟩ ::= ⟨actor_io⟩ ["," ⟨actor_ios⟩]

⟨actor_io⟩ ::= ⟨id⟩ ":" ⟨type⟩

Actor body

⟨actor_body⟩ ::= ⟨actor_var⟩* "rules" ⟨actor_rules⟩

⟨actor_rules⟩ ::= ⟨rule_schema⟩ ⟨unqual_rule⟩+
| ⟨qual_rule⟩+

⟨*actor_var*⟩ ::= "var" ⟨*id*⟩ ":" ⟨*var_type*⟩ ["=" ⟨*var_init*⟩]

⟨*var_init*⟩ ::= ⟨*expr*⟩
| ⟨*array_init*⟩

⟨*rule_schema*⟩ ::= ⟨*qualifiers*⟩ "->" ⟨*qualifiers*⟩

⟨*qualifiers*⟩ ::= ⟨*qualifier*⟩
| "(" ⟨*qualifier*⟩ "," ... "," ⟨*qualifier*⟩ ")"

⟨*qualifier*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ "[" ⟨*simple_array_index*⟩ "]"
| ⟨*id*⟩ "[" ⟨*simple_array_index*⟩ "]" "[" ⟨*simple_array_index*⟩ "]"

⟨*unqual_rule*⟩ ::= "|" ⟨*unqual_rule_lhs*⟩ [⟨*rule_guard*⟩] "->" ⟨*unqual_rule_rhs*⟩

⟨*unqual_rule_lhs*⟩ ::= ⟨*rule_pattern*⟩
| "(" ⟨*rule_pattern*⟩ "," ... "," ⟨*rule_pattern*⟩ ")"

⟨*qual_rule*⟩ ::= "|" ⟨*qual_rule_lhs*⟩ [⟨*rule_guard*⟩] "->" ⟨*qual_rule_rhs*⟩

⟨*qual_rule_lhs*⟩ ::= ⟨*qual_rule_pattern*⟩
| "(" ⟨*qual_rule_pattern*⟩ "," ... "," ⟨*qual_rule_pattern*⟩ ")"

⟨*qual_rule_pattern*⟩ ::= ⟨*qualifier*⟩ ":" ⟨*rule_pattern*⟩

⟨*rule_pattern*⟩ ::= ⟨*simple_rule_pattern*⟩
| ⟨*con_id*⟩ [⟨*simple_rule_pattern*⟩]
| ⟨*con_id*⟩ "(" ⟨*simple_rule_pattern*⟩ "," ... "," ⟨*simple_rule_pattern*⟩ ")"
| "'<"[1]
| "'>"[2]
| "'" ⟨*simple_rule_pattern*⟩[3]
| "_"

⟨*simple_rule_pattern*⟩ ::= ⟨*var*⟩
| ⟨*scalar_constant*⟩
| "_"

⟨*rule_guard*⟩ ::= "when" ⟨*rule_guard_exprs*⟩

⟨*rule_guard_exprs*⟩ ::= ⟨ *rule_guard_expr*⟩ ["and" ⟨*rule_guard_exprs*⟩]

⟨*rule_guard_expr*⟩ ::= ⟨*simple_rule_expr*⟩

⟨*unqual_rule_rhs*⟩ ::= ⟨*rule_expr*⟩
| "(" ⟨*rule_expr*⟩ "," ... "," ⟨*rule_expr*⟩ ")"

[1] Synonym for the builtin constructor `SoS`.
[2] Synonym for the builtin constructor `EoS`.
[3] Synonym for the builtin constructor `Data`.

⟨*qual_rule_rhs*⟩ ::= ⟨*qual_rule_expr*⟩
| "(" ⟨*qual_rule_expr*⟩ "," ... "," ⟨*qual_rule_expr*⟩ ")"

⟨*qual_rule_expr*⟩ ::= ⟨*qualifier*⟩ ":" ⟨*rule_expr*⟩

⟨*rule_expr*⟩ ::= ⟨*simple_rule_expr*⟩
| ⟨*con_id*⟩ [⟨*simple_rule_expr*⟩]
| ⟨*con_id*⟩ "(" ⟨*simple_rule_expr*⟩ "," ... "," ⟨*simple_rule_expr*⟩ ")"
| "'<"
| "'>"
| "'" ⟨*simple_rule_expr*⟩
| "_"

⟨*simple_rule_expr*⟩ ::= ⟨*expr*⟩

IO declarations

⟨*io_decl*⟩ ::= "stream" ⟨*id*⟩ ":" ⟨*type*⟩ ("from" | "to") ⟨*device*⟩
| "port" ⟨*id*⟩ ":" ⟨*type*⟩ ["from" ⟨*device*⟩] "init" ⟨*simple_net_expr*⟩
| "port" ⟨*id*⟩ ":" ⟨*type*⟩ "to" ⟨*device*⟩]

⟨*device*⟩ ::= ⟨*string*⟩

Network declarations

⟨*net_decl*⟩ ::= "net" ["rec"] ⟨*net_bindings*⟩

⟨*net_bindings*⟩ ::= ⟨*net_binding*⟩ ["and" ⟨*net_bindings*⟩]

⟨*net_binding*⟩ ::= ⟨*net_pattern*⟩ "=" ⟨*net_expr*⟩
| ⟨*id*⟩ ⟨*net_pattern*⟩+ "=" ⟨*net_expr*⟩

⟨*net_pattern*⟩ ::= ⟨*var*⟩
| "(" ⟨*net_pattern*⟩ "," ... "," ⟨*net_pattern*⟩ ")"
| "()"

⟨*net_expr*⟩ ::= ⟨*simple_net_expr*⟩
| ⟨*simple_net_expr*⟩ ⟨*simple_net_expr*⟩+
| ⟨*net_exprs*⟩
| "let" ["rec"] ⟨*net_bindings*⟩ "in" ⟨*net_expr*⟩
| "function" ⟨*net_pattern*⟩ "->" ⟨*net_expr*⟩

⟨*simple_net_expr*⟩ ::= ⟨*var*⟩
| "()"
| "(" ⟨*net_expr*⟩ ")"
| "(" ⟨*simple_net_expr*⟩ ":" ⟨*type*⟩ ")"
| ⟨*param_value*⟩

⟨param_value⟩ ::= ⟨var⟩
| ⟨scalar_constant⟩
| ⟨array1_constant⟩
| ⟨array2_constant⟩
| "(" ⟨var⟩ "[" ⟨simple_array_index⟩ "]" ")"
| "(" ⟨var⟩ "[" ⟨simple_array_index⟩ "]" "[" ⟨simple_array_index⟩ "]" ")"
| "(" ⟨param_value⟩ ":" ⟨type⟩ ")"

⟨net_exprs⟩ ::= ⟨net_expr⟩ ["," ⟨net_exprs⟩]

Pragma declarations

⟨pragma_decl⟩ ::= "#pragma" ⟨id⟩ ["(" ⟨id⟩ "," ... "," ⟨id⟩ ")"]

Expressions

⟨expr⟩ ::= ⟨scalar_constant⟩
| ⟨var⟩
| ⟨var⟩ "`" ⟨attr⟩
| ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
| ⟨unop⟩ ⟨expr⟩
| ⟨var⟩ "(" ⟨expr⟩ "," ... "," ⟨expr⟩ ")"
| "let" ⟨var⟩ "=" ⟨expr⟩ "in" ⟨expr⟩
| "if" ⟨expr⟩ "then" ⟨expr⟩ "else" ⟨expr⟩
| ⟨var⟩ "[" ⟨array_index⟩ "]"
| ⟨var⟩ "[" ⟨array_index⟩ "]" "[" ⟨array_index⟩ "]"
| ⟨var⟩ "[" ⟨array_index⟩ "]" "[" ⟨array_index⟩ "]" "[" ⟨array_index⟩ "]"
| "(" ⟨expr⟩ ":" ⟨type⟩ ")"
| "(" ⟨expr⟩ ")"

⟨array_index⟩ ::= ⟨expr⟩

⟨simple_array_index⟩ ::= ⟨scalar_constant⟩
| ⟨var⟩

⟨array_init⟩ ::= ⟨array_ext1⟩
| ⟨array_ext2⟩
| ⟨array_ext3⟩
| "[" ⟨array_comprehension⟩ "]"

⟨array_ext1⟩ ::= "[" ⟨expr⟩ "," ... "," ⟨expr⟩ "]"

⟨array_ext2⟩ ::= "[" ⟨array_ext1⟩ "," ... "," ⟨array_ext1⟩ "]"

⟨array_ext3⟩ ::= "[" ⟨array_ext2⟩ "," ... "," ⟨array_ext2⟩ "]"

⟨array_comprehension⟩ ::= "[" ⟨expr⟩ "|" ⟨index_range⟩, "," ... "," ⟨index_range⟩ "]"

⟨index_range⟩ ::= ⟨id⟩ "=" ⟨array_index⟩ "to" ⟨array_index⟩

Constants

```
⟨scalar_constant⟩ ::= ⟨intconst⟩
              |  ⟨boolconst⟩
              |  ⟨floatconst⟩

⟨array1_constant⟩ ::= "[" ⟨scalar_constant⟩ "," ... "," ⟨scalar_constant⟩ "]"

⟨array2_constant⟩ ::= "[" ⟨array1_constant⟩ "," ... "," ⟨array1_constant⟩ "]"
```

Type expressions

```
⟨type⟩      ::= ⟨simple_type⟩
              |  ⟨type_product⟩
              |  ⟨type⟩ "->" ⟨type⟩

⟨type_product⟩ ::= ⟨simple_type⟩ [ "*" ⟨type_product⟩ ]

⟨simple_type⟩ ::= "signed" "<" ⟨size⟩ ">"
              |  "unsigned" "<" ⟨size⟩ ">"
              |  "int"
              |  "int" ["<" ⟨size⟩ ">"]
              |  "int" ["<" ⟨sign⟩ "," ⟨size⟩ ">"]
              |  [⟨simple_types⟩] ⟨id⟩ ["<" ⟨size⟩ ">"]
              |  ⟨ty_var⟩
              |  ⟨simple_type⟩ "array" "[" ⟨size⟩ "]"
              |  ⟨simple_type⟩ "array" "[" ⟨size⟩ "]" "[" ⟨size⟩ "]"[4]
              |  "(" ⟨type⟩ ")"

⟨simple_types⟩ ::= ⟨simple_type⟩
              |  "(" ⟨simple_type⟩ "," ... "," ⟨simple_type⟩ ")"

⟨size⟩      ::= ⟨int_const⟩
              |  ⟨sz_var⟩

⟨sign⟩      ::= "_signed" | "_unsigned"

⟨array_size⟩ ::= ⟨int_const⟩

⟨var_type⟩ ::= ⟨type⟩
              |  "{" ⟨ctors⟩ "}"
              |  "{" ⟨intrange⟩ "}"

⟨ctors⟩     ::= ⟨con_id⟩ [ "," ⟨ctors⟩ ]

⟨intrange⟩ ::= ⟨intconst⟩ ",..," ⟨intconst⟩
```

[4] `t array[m][n]` is actually syntactic sugar for `t array[n] array[m]`.

Lexical Syntax

⟨*var*⟩ ::= ⟨*id*⟩

⟨*attr*⟩ ::= ⟨*id*⟩

⟨*id*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩ | _ | ')*

⟨*letter*⟩ ::= "a" | ... | "z"

⟨*con_id*⟩ ::= ⟨*uid*⟩

⟨*ty_var*⟩ ::= "$" ⟨*id*⟩

⟨*sz_var*⟩ ::= ⟨*id*⟩

⟨*uid*⟩ ::= ⟨*uletter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩ | _ | ')*

⟨*uletter*⟩ ::= "A" | ... | "Z"

⟨*binop*⟩ ::= "+" | "-" | "*" | "/" | "mod" | "+." | "-." | "*." | "/." | "<" | ">" | "<=" | ">=" | "=" | "!=" | "&&" | "||" | "land" | "lor" | "lxor" "<<" | ">>"

⟨*unop*⟩ ::= "-" | "!"

⟨*intconst*⟩ ::= ["-"]⟨*digit*⟩+

⟨*fixintconst*⟩ ::= [⟨*radix*⟩] ⟨*digit*⟩+

⟨*radix*⟩ ::= "Ox" | "Ob"

⟨*boolconst*⟩ ::= "true" | "false"

⟨*floatconst*⟩ ::= ["-"]⟨*digit*⟩+["."⟨*digit*⟩*][["e"|"E"]["+"|"-"]⟨*digit*⟩+]

⟨*string*⟩ ::= "¨" ⟨*char*⟩* "¨"

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed.Comments are written in Java-style. They are single-line and start with "--".

The following identifiers are reserved as keywords, and cannot be employed otherwise:

```
type      of      net     let          in       rec
if        then    else    actor        out      var
rules     stream  to      from         when     and
function  const   extern  implemented  systemc  vhdl
true      false   or      not          lnot     signed
unsigned  array   port    init
```

The following character sequences are also keywords:

```
'<  '>   ->  <-  ..   <<  >>  &&
||  >=   <=  !=  +.   -.  *.  _
=.  !=.  >.  <.  >=.  <=.
```

Chapter 4

Core Abstract Syntax

This chapter gives the abstract syntax of a simplified version of the CAPH language[1]. Compared to the "full" CAPH language, this simplified version lacks :

- global type declarations,
- port declarations,
- external function declarations,
- rule formats (as defined in Sec. 2.4.4),
- guards in rules,

Moreover, types are limited to basic un-sized, un-signed, `int`s, `bool`s and builtin variants.

The omitted features are either classical, and hence not specific to CAPH[2], or just introduce too much technical details in the definition of the typing rules and static and dynamic semantics[3] without interfering with the global soundness of the formal system.

program ::=	`program` *valdecls actdecls iodecls netdefns*	
valdecls ::=	$valdecl_1 \; \ldots \; valdecl_n$	$n \geq 0$
valdecl ::=	`const` id = *expr* [: *tyexpr*]	
	| `fun` id $\langle \mathrm{var}_1, \ldots, \mathrm{var}_n \rangle \rightarrow$ *expr* [: *tyexpr*]	$n \geq 1$
actdecls ::=	$actdecl_1 \; \ldots \; actdecl_n$	$n \geq 1$
actdecl ::=	`actor` id *actparams actins actouts actvars actrules*	
actparams ::=	$param_1 \; \ldots \; param_n$	$n \geq 0$

[1] Called Core-Caph in the sequel.
[2] This is the case for global type definitions for example, which are handled exactly like in all other ML-like languages.
[3] In Chap. 5, 6 and 7 resp.

actins, *actouts* ::=	$actio_1 \; \ldots \; actio_n$	$n \geq 0$
actvars ::=	$actvar_1 \; \ldots \; actvar_n$	$n \geq 0$
param, *actio* ::=	id : *tyexpr*	
actvar ::=	id : *tyexpr*	
	| id : *tyexpr* = *expr*	
rules ::=	$rule_1 \ldots rule_n$	$n \geq 1$
rule ::=	$\langle qpat_1, \ldots, qpat_m \rangle \rightarrow \langle qexp_1, \ldots, qexp_n \rangle$	$m, n \geq 1$
qpat ::=	$\langle qual, rpat \rangle$	
rpat ::=	scalar_const	
	| var	
	| _	
	| con $\langle rpat_1, \ldots, rpat_n \rangle$	$n \geq 0$
qexp ::=	$\langle qual, expr \rangle$	
expr ::=	*const*	
	| var	
	| *con* $\langle expr_1, \ldots, expr_n \rangle$	$n \geq 0$
	| id ($expr_1, \ldots, expr_n$)	$n \geq 1$
	| `if` *expr* `then` *expr* `else` *expr*	
	| `let` var = *expr* `in` *expr*	
	| *expr* : *tyexpr*	
	| _	only in rule rhs
qual ::=	id	
iodecls ::=	$iodecl_1 \ldots iodecl_n$	$n \geq 0$
iodecl ::=	`stream` *id tyexpr dir* string	
dir ::=	`from` | `to`	
netdecls ::=	$ndecl_1 \ldots ndecl_n$	$n \geq 1$
ndecl ::=	`net` [`rec`] $\langle n_binding_1, \ldots, n_binding_n \rangle$	
n_binding ::=	*n_pattern* = *n_expr*	
n_expr ::=	| var	
	| *const*	
	| ($n_expr_1, \ldots, n_expr_n$)	$n \geq 1$
	| $n_expr_1 \;\; n_expr_2$	
	| `function` *n_pattern* → *n_expr*	
	| `let rec` $\langle n_binding_1, \ldots, n_binding_n \rangle$ `in` *n_expr*	
n_pattern ::=	| var	
	| ($n_pattern_1, \ldots, n_pattern_n$)	$n \geq 1$
tyexpr ::=	*tyctor* $\langle tyexpr_1, \ldots, tyexpr_n \rangle$	$n \geq 0$
	| $tyexpr_1 \times \ldots \times tyexpr_n$	$n \geq 1$
	| $tyexpr_1 \rightarrow tyexpr_2$	
const ::=	int | bool	

Chapter 5

Typing

This section gives the formal typing rules for the so-called *Core* CAPH language defined in Chap. 4.

The type language is fairly standard. A type τ is either[1]

- a type variable α
- a constructed type $\chi\ \tau_1 \ldots \tau_n$,
- a functional type $\tau_1 \rightarrow \tau_2$,
- a product type $\tau_1 \times \ldots \times \tau_n$,

A type schema σ is either

- a type τ,
- a type scheme $\forall \alpha . \sigma$.

For simplicity, conversions from type schemes to types (instanciation) and from types to type schemes (generalisation) have been left implicit since the rules are completely standard.

Typing occurs in the context of a *typing environment* consisting of :

- a type environment T, recording type and value constructors (*tycons* and *ctors* resp.),
- a variable environments V, mapping identifiers to (polymorphic) types.

The initial type environment TE_0 records the type of the *builtin* type and value constructors :

$$\begin{array}{lcl} \mathrm{TE}_0.\mathit{tycons} & = & [\mathbf{int} \mapsto \mathsf{Int}, \mathbf{bool} \mapsto \mathsf{Bool}, \\ & & \mathbf{dc} \mapsto \alpha \rightarrow \mathsf{Dc}\ \alpha] \\ \mathrm{TE}_0.\mathit{ctors} & = & [\mathbf{0} \mapsto \mathsf{Int}, \mathbf{1} \mapsto \mathsf{Int}, \ldots, \\ & & \mathbf{true} \mapsto \mathsf{Bool}, \mathbf{false} \mapsto \mathsf{Bool} \\ & & \mathbf{SoS} \mapsto \mathsf{Dc}\ \alpha, \mathbf{EoS} \mapsto \mathsf{Dc}\ \alpha, \mathbf{Data} \mapsto \alpha \rightarrow \mathsf{Dc}\ \alpha] \end{array}$$

The initial variable environment VE_0 contains the types of the expression-level builtin primitives.

$$\begin{array}{lcl} \mathrm{VE}_0 & = & [+ : \mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Int}, \\ & & = : \mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Bool}, \ldots] \end{array}$$

[1]Functional, product and enumerated types are here treated specially although they could be handled as "normal" constructed types.

5.1 Notations

Both type and variable *environments* are viewed as partial maps (from identifiers to types and from type constructors to types resp.). If E is an environment, the domain and co-domain of E are denoted by $\text{dom}(E)$ and $\text{codom}(E)$ respectively. The empty environment will be written $\varnothing$. $[x \mapsto y]$ denotes the singleton environment mapping x to y. We will note $E(x)$ the result of applying the underlying map to x (for ex. if E is $[x \mapsto y]$ then $E(x) = y$). We denote by $E[x \mapsto y]$ the environment that maps x to y and behaves like E otherwise. $E \oplus E'$ denotes the environment obtained by adding the mappings of E' to those of E. If E and E' are not disjoints, then the mappings of E are "shadowed" by those of E'. We will also use a specialized version of the $\oplus$ merging operator, denoted $\vec{\oplus}$, for which shadowing is replaced by type unification: if x appears both in E and E', and maps respectively to σ and σ', then it will map to $\text{unify}_E(\sigma, \sigma')$ in $E \vec{\oplus} E'$ (this simplifies the description of the semantics of definitions for outputs). $E \ominus E'$ denotes the environment obtained by removing any mapping $\{x \mapsto y\}$ for which $x \in dom(E')$ from E.

When an environment E is composed of several sub-environments E',E'', ..., we note $E = \{E', E'', \ldots\}$ and use the dot notation to access these sub-environments (ex: $E.E'$).

For convenience and readability, we try to adhere to the following naming conventions throughout this chapter :

Meta-variable	**Meaning**
TE	Type environment
VE	Variable environment
ty	Type expression
τ	Type
σ	Type scheme
α	Type variable
χ	Type constructor
c	Value constructor
id	Identifier
npat	Pattern (network level)
nexp	Network-level expression
qpat	Qualified rule pattern (actor level)
rpat	Rule pattern (actor level)
qexp	Qualified rule expression (actor level)
exp	Expression
qual	Rule qualifier (actor level)

Syntactical terminal symbols are written in **bold**. Non terminals in *italic.* Types values are written in serif.

5.2 Typing rules

5.2.1 Programs

$$\boxed{\mathrm{TE}, \mathrm{VE} \vdash \text{Program} \Rightarrow \mathrm{VE}'}$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE}_0 \vdash \mathit{valdecls} \Rightarrow \mathrm{VE}_v \\ \mathrm{TE}, \mathrm{VE}_0 \oplus \mathrm{VE}_v \vdash \mathit{actdecls} \Rightarrow \mathrm{VE}_a \\ \mathrm{TE} \vdash \mathit{iodecls} \Rightarrow \mathrm{VE}_i, \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE}_0 \oplus \mathrm{VE}_v \oplus \mathrm{VE}_a \oplus \mathrm{VE}_i \vdash \mathit{netdecls} \Rightarrow \mathrm{VE}'\end{array}}{\mathrm{TE}_0, \mathrm{VE}_0 \vdash \textbf{program}\ \mathit{valdecls}\ \mathit{actdecls}\ \mathit{iodecls}\ \mathit{netdefns} \Rightarrow \mathrm{VE}' \overset{\rightarrow}{\oplus} \mathrm{VE}_o} \quad (\textsc{Program})$$

5.2.2 Value declarations

$$\boxed{\mathrm{TE}, \mathrm{VE}\ \vdash \text{ValDecls} \Rightarrow \mathrm{VE}'}$$

$$\frac{\forall i.\ 1 \le i \le n,\quad \mathrm{TE}, \mathrm{VE}_{i-1}\ \vdash \mathit{valdecl}_i \Rightarrow \mathrm{VE}_i,\quad \mathrm{VE}_0 = \mathrm{VE}}{\mathrm{TE}, \mathrm{VE}\ \vdash \mathit{valdecl}_1\ \ldots\ \mathit{valdecl}_n \Rightarrow \mathrm{VE}_n} \quad (\textsc{ValDecls})$$

$$\frac{\mathrm{TE}, \mathrm{VE}\ \vdash \mathit{exp} \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE}\ \vdash \texttt{const}\ \mathrm{id} = \mathit{exp} \Rightarrow [\mathrm{id} \mapsto \tau]} \quad (\textsc{ConstDecl1})$$

$$\frac{\mathrm{TE}, \mathrm{VE}\ \vdash \mathit{exp} \Rightarrow \tau \qquad \mathrm{TE}\ \vdash \mathit{ty} \Rightarrow \tau' \quad \mathit{coercible}(\tau, \tau')}{\mathrm{TE}, \mathrm{VE}\ \vdash \texttt{const}\ \mathrm{id} = \mathit{exp}\ :\ \mathit{ty} \Rightarrow [\mathrm{id} \mapsto \tau']} \quad (\textsc{ConstDecl2})$$

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n,\ \vdash \mathit{pat}_i, \tau_i \Rightarrow \mathrm{VE}_i \qquad \mathrm{VE}' = \bigoplus_{i=1}^{n} \mathrm{VE}_i \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}'\ \vdash \mathit{exp} \Rightarrow \tau'\end{array}}{\mathrm{TE}, \mathrm{VE}\ \vdash \texttt{fun}\ \mathrm{fid}\ \langle \mathit{pat}_1, \ldots, \mathit{pat}_n \rangle \rightarrow \mathit{exp} \Rightarrow [\mathrm{fid} \mapsto \langle \tau_1, \ldots, \tau_n \rangle \rightarrow \tau']} \quad (\textsc{FunDecl1})$$

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n,\ \vdash \mathit{pat}_i, \tau_i \Rightarrow \mathrm{VE}_i \qquad \mathrm{VE}' = \bigoplus_{i=1}^{n} \mathrm{VE}_i \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}'\ \vdash \mathit{exp} \Rightarrow \tau' \\ \mathrm{TE}\ \vdash \mathit{ty} \Rightarrow \tau'' \quad \mathit{coercible}(\langle \tau_1, \ldots, \tau_n \rangle \rightarrow \tau', \tau'')\end{array}}{\mathrm{TE}, \mathrm{VE}\ \vdash \texttt{fun}\ \mathrm{fid}\ \langle \mathit{pat}_1, \ldots, \mathit{pat}_n \rangle \rightarrow \mathit{exp}\ :\ \mathit{ty} \Rightarrow [\mathrm{fid} \mapsto \tau'']} \quad (\textsc{FunDecl2})$$

where

$$\vdash pat, \tau \Rightarrow \mathrm{VE}$$

means that declaring *pat* with type τ creates the variable environment VE.

The predicate *coercible* tells whether the infered type τ can be cast to the declared type τ'. The coercibility relation has been defined in Sec. 2.3.6.

5.2.3 Actor declarations

$$\boxed{\mathrm{TE}, \mathrm{VE} \vdash \mathrm{ActorDecls} \Rightarrow \mathrm{VE}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\quad \mathrm{TE}, \mathrm{VE} \vdash actdecl_i \Rightarrow \mathrm{VE}_i}{\mathrm{TE}, \mathrm{VE} \vdash actdecl_1 \ \ldots \ actdecl_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{VE}_i} \quad (\textsc{ActorDecls})$$

$$\frac{\begin{array}{c} params \neq \emptyset \\ \mathrm{TE} \vdash params \Rightarrow \tau, \mathrm{VE}_p \\ \mathrm{TE} \vdash ins \Rightarrow \tau', \mathrm{VE}_i \\ \mathrm{TE} \vdash outs \Rightarrow \tau'', \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_p \vdash localvars \Rightarrow \mathrm{VE}_v, \mathrm{TE}_v \\ \mathrm{TE} \oplus \mathrm{TE}_v, \mathrm{VE} \oplus \mathrm{VE}_p \oplus \mathrm{VE}_v \oplus \mathrm{VE}_i \oplus \mathrm{VE}_o \vdash rules \Rightarrow \mathrm{VE}_r \end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{actor}\ \mathrm{id}\ params\ ins\ outs\ localvars\ rules \Rightarrow [\mathrm{id} \mapsto \tau \to \tau' \to \tau'']} \quad (\textsc{ActorDecl1})$$

$$\frac{\begin{array}{c} params = \emptyset \\ \mathrm{TE} \vdash ins \Rightarrow \tau', \mathrm{VE}_i \\ \mathrm{TE} \vdash outs \Rightarrow \tau'', \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_p \vdash localvars \Rightarrow \mathrm{VE}_v, \mathrm{TE}_v \\ \mathrm{TE} \oplus \mathrm{TE}_v, \mathrm{VE} \oplus \mathrm{VE}_p \oplus \mathrm{VE}_v \oplus \mathrm{VE}_i \oplus \mathrm{VE}_o \vdash rules \Rightarrow \mathrm{VE}_r \end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{actor}\ \mathrm{id}\ params\ ins\ outs\ localvars\ rules \Rightarrow [\mathrm{id} \mapsto \tau' \to \tau'']} \quad (\textsc{ActorDecl2})$$

Note that the type assigned to an actor only depends on its interface (parameters, inputs and outputs). An actor `a` declared as

$\texttt{actor}\ a\ (p_1 : t_1, \ldots, p_k : t_k)\ \texttt{in}\ (i_1 : t'_1, \ldots, i_m : t'_m)\ \texttt{out}\ (o_1 : t''_1, \ldots, o_n : t''_n)$

will be assigned type

$t_1 \times \ldots \times t_k \to t'_1 \times \ldots \times t'_m \to t''_1 \times \ldots \times t''_n$

whereas an actor `a` declared as

$\texttt{actor}\ a\ \texttt{in}\ (i_1 : t'_1, \ldots, i_m : t'_m)\ \texttt{out}\ (o_1 : t''_1, \ldots, o_n : t''_n)$

will be assigned type

$t'_1 \times \ldots \times t'_m \to t''_1 \times \ldots \times t''_n$

$$\boxed{\mathrm{TE} \vdash \mathrm{ActParams} \Rightarrow \tau, \mathrm{VE}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \text{TE} \vdash \mathit{param}_i \Rightarrow \tau_i, \text{VE}_i}{\text{TE} \vdash \mathit{param}_1 \ \ldots \ \mathit{param}_n \Rightarrow \tau_1 \times \ldots \times \tau_n, \bigoplus_{i=1}^{n} \text{VE}_i} \quad (\textsc{ActParams})$$

$$\frac{\text{TE} \vdash \mathit{ty} \Rightarrow \tau}{\text{TE} \vdash \text{id} \ : \ \mathit{ty} \Rightarrow \tau, \ [\text{id} \mapsto \tau]} \quad (\textsc{ActParam})$$

$$\boxed{\text{TE} \vdash \text{ActIOs} \Rightarrow \tau, \text{VE}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \text{TE} \vdash \mathit{io}_i \Rightarrow \tau_i, \text{VE}_i}{\text{TE} \vdash \mathit{io}_1 \ \ldots \ \mathit{io}_n \Rightarrow \tau_1 \times \ldots \times \tau_n, \bigoplus_{i=1}^{n} \text{VE}_i} \quad (\textsc{ActIOs})$$

$$\frac{\text{TE} \vdash \mathit{ty} \Rightarrow \tau}{\text{TE} \vdash \text{id} \ : \ \mathit{ty} \Rightarrow \tau, \ [\text{id} \mapsto \tau]} \quad (\textsc{ActIO})$$

$$\boxed{\text{VE}, \text{TE} \vdash \text{ActVars} \Rightarrow \text{VE}', \text{TE}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \text{VE}, \text{TE} \vdash \mathit{var}_i \Rightarrow \text{VE}_i, \text{TE}_i}{\text{VE}, \text{TE} \vdash \mathit{var}_1 \ \ldots \ \mathit{var}_n \Rightarrow \bigoplus_{i=1}^{n} \text{VE}_i, \bigoplus_{i=1}^{n} \text{TE}_i} \quad (\textsc{ActVars})$$

$$\frac{\text{TE} \vdash \mathit{ty} \Rightarrow \tau, \text{TE}'}{\text{VE}, \text{TE} \vdash \text{id} \ : \ \mathit{ty} \Rightarrow \tau, [\text{id} \mapsto \tau], \text{TE}'} \quad (\textsc{ActVar})$$

$$\frac{\text{TE} \vdash \mathit{ty} \Rightarrow \tau, \text{TE}' \quad \text{TE}, \text{VE} \vdash \mathit{exp} \Rightarrow \tau' \quad \mathit{coercible}(\tau', \tau)}{\text{VE}, \text{TE} \vdash \text{id} \ : \ \mathit{ty} \ = \ \mathit{exp} \Rightarrow [\text{id} \mapsto \tau], \text{TE}'} \quad (\textsc{ActVar'})$$

Typing of actor local variables is the only situation where the type environment TE can be augmented. This happens when a variable is declared with an enumerated type. In this case, the (nullary) enumerated constants are added, as nullary value constructors, to the type environment. Such declarations can be viewed as purely local type declarations, since the scope of the declared constructors is limited to the rule set of the enclosing actor).

Actor rules

$$\boxed{\text{TE}, \text{VE} \vdash \text{ActRules} \Rightarrow \text{VE}'}$$

$$\frac{\forall i.\ 1 \le i \le n, \quad \mathrm{TE}, \mathrm{VE} \ \vdash \mathit{rule}_i \Rightarrow \tau_i}{\mathrm{TE}, \mathrm{VE} \ \vdash \mathit{rule}_1 \ \ldots \ \mathit{rule}_n \Rightarrow [1 \mapsto \tau_1, \ldots, n \mapsto \tau_n]} \quad (\textsc{ActRules})$$

Typing the rule set produces a environment mapping rule numbers to types. Each rule can have a distinct type.

$$\boxed{\mathrm{TE}, \mathrm{VE} \ \vdash \mathrm{ActRule} \Rightarrow \tau \to \tau'}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le m, \quad \mathrm{TE}, \mathrm{VE} \ \vdash \mathit{qpat}_i \Rightarrow \tau_i, \mathrm{VE}'_i \\ \forall j.\ 1 \le j \le n, \quad \mathrm{TE}, \mathrm{VE} \oplus \bigoplus_{i=1}^{n} \mathrm{VE}'_i \vdash \mathit{qexp}_j \Rightarrow \tau'_j\end{array}}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathit{qpat}_1, \ldots, \mathit{qpat}_m \rangle \ \to \ \langle \mathit{qexp}_1, \ldots, \mathit{qexp}_n \rangle \Rightarrow \tau_1 \times \ldots \times \tau_m \to \tau'_1 \times \ldots \times \tau'_n} \quad (\textsc{ActRule})$$

Rule patterns

Typing a (qualified) rule pattern gives a type and an environment, mapping each identifier introduced in the pattern to a type. The type is retrieved using the pattern qualifier.

$$\boxed{\mathrm{TE}, \mathrm{VE} \ \vdash \mathrm{QPat} \Rightarrow \tau, \mathrm{VE}}$$

$$\frac{\mathrm{VE}(\mathrm{id}) = \mathsf{Int}}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, \mathrm{int} \rangle \Rightarrow \mathsf{Int}, \ \varnothing} \quad (\textsc{RPatConstInt})$$

$$\frac{\mathrm{VE}(\mathrm{id}) = \mathsf{Bool}}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, \mathrm{int} \rangle \Rightarrow \mathsf{Bool}, \ \varnothing} \quad (\textsc{RPatConstBool})$$

$$\frac{\mathrm{VE}(\mathrm{id}) = \tau}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, \mathrm{var} \rangle \Rightarrow \tau, \ [\mathrm{var} \mapsto \tau]} \quad (\textsc{RPatVar})$$

$$\frac{\mathrm{VE}(\mathrm{id}) = \tau}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, _ \rangle \Rightarrow \tau, \ \varnothing} \quad (\textsc{RPatWild})$$

$$\frac{\mathrm{VE}(\mathrm{id}) = \tau \qquad \mathrm{TE}.\mathit{ctors}(c) = \tau}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, \mathrm{c} \ \langle \rangle \rangle \Rightarrow \tau, \ \varnothing} \quad (\textsc{RPatCon0})$$

$$\frac{\begin{array}{c}\mathrm{VE}(\mathrm{id}) = \tau \qquad \mathrm{TE}.\mathit{ctors}(c) = \tau_1 \ \times \ldots \times \ \tau_n \to \tau \\ \forall i.\ 1 \le i \le n, \mathrm{TE}, \mathrm{VE} \ \vdash \mathit{qpat}_i \Rightarrow \tau_i, \ \mathrm{VE}_i\end{array}}{\mathrm{TE}, \mathrm{VE} \ \vdash \langle \mathrm{id}, \mathrm{c} \ \langle \mathit{qpat}_1, \ldots, \mathit{qpat}_n \rangle \rangle \Rightarrow \tau, \ \bigoplus_{i=1}^{n} \mathrm{VE}_i} \quad (\textsc{RPatCon})$$

5.2.4 Expressions

$$\boxed{\mathrm{TE}, \mathrm{VE} \ \vdash \mathrm{QExp} \Rightarrow \tau}$$

$$\frac{\text{TE}, \text{VE} \vdash exp \Rightarrow \tau \qquad \text{VE}(\text{id}) = \tau}{\text{TE}, \text{VE} \vdash \langle \text{id}, exp \rangle \Rightarrow \tau}$$

The infered type for a qualified expression must match the type assigned to the corresponding output or local variable.

$$\boxed{\text{TE}, \text{VE} \ \vdash \text{Exp} \Rightarrow \tau}$$

$$\frac{}{\text{TE}, \text{VE} \vdash \text{int/bool} \Rightarrow \mathsf{Int/Bool}} \quad (\text{EConst})$$

$$\frac{\text{VE}(id) = \tau}{\text{TE}, \text{VE} \vdash \text{id} \Rightarrow \tau} \quad (\text{EVar})$$

$$\frac{}{\text{TE}, \text{VE} \vdash _ \Rightarrow \alpha} \quad (\text{EWild})$$

$$\frac{\text{TE}.ctors(c) = \tau}{\text{TE}, \text{VE} \ \vdash \text{c} \ \langle\rangle \Rightarrow \alpha} \quad (\text{ECon0})$$

$$\frac{\begin{array}{c}\text{TE}.ctors(c) = \tau_1 \ \times \ldots \times \ \tau_n \to \tau \\ \forall i.\ 1 \le i \le n, \quad \text{TE}, \text{VE} \vdash exp_i \Rightarrow \tau_i\end{array}}{\text{TE}, \text{VE} \ \vdash \text{c} \ \langle exp_1, \ldots, exp_n \rangle \Rightarrow \tau} \quad (\text{ECon})$$

$$\frac{\begin{array}{c}\text{VE}(id) = \tau_1 \times \ldots \times \tau_n \to \tau' \\ \forall i.\ 1 \le i \le n, \quad \text{TE}, \text{VE} \vdash \ exp_i \Rightarrow \tau_i\end{array}}{\text{TE}, \text{VE} \vdash \text{id} \ (\ exp_1, \ \ldots, \ exp_n \ ) \ \Rightarrow \tau'} \quad (\text{EApp})$$

$$\frac{\text{TE}, \text{VE} \vdash exp \Rightarrow \mathsf{Bool} \qquad \text{TE}, \text{VE} \vdash exp_1 \Rightarrow \tau \qquad \text{VE}, \text{TE} \vdash exp_2 \Rightarrow \tau}{\text{TE}, \text{VE} \vdash \textbf{if} \ \ exp \ \textbf{then} \ exp_1 \ \texttt{else} \ exp_2 \Rightarrow \tau} \quad (\text{ECond})$$

$$\frac{\vdash_p \text{id}, \tau' \Rightarrow \text{VE}' \qquad \text{TE}, \text{VE} \vdash exp_2 \Rightarrow \tau' \qquad \text{TE}, \text{VE} \oplus \text{VE}'' \vdash exp_1 \Rightarrow \tau}{\text{TE}, \text{VE} \vdash \textbf{let} \ \text{id} = exp_2 \ \textbf{in} \ exp_1 \Rightarrow \tau} \quad (\text{ELet})$$

$$\frac{\text{TE}, \text{VE} \vdash \ exp \Rightarrow \tau \qquad \text{TE} \vdash ty \Rightarrow \tau' \qquad coercible(\tau, \tau')}{\text{TE}, \text{VE} \ \vdash exp : ty \Rightarrow \tau'} \quad (\text{ECoerce})$$

5.2.5 IOs

$$\boxed{\text{TE} \ \vdash \text{IoDecls} \Rightarrow \text{VE}_i, \text{VE}_o}$$

$$\frac{\forall i.\ 1 \le i \le n, \quad \text{TE} \ \vdash iodecl_i \Rightarrow \text{VE}_i, \text{VE}_o}{\text{TE} \ \vdash iodecl_1 \ \ldots \ iodecl_n \Rightarrow \bigoplus_{i=1}^{n} \text{VE}_i, \bigoplus_{i=1}^{n} \text{VE}_o} \quad (\text{IoDecls})$$

$$\frac{\text{TE} \ \vdash ty \Rightarrow \tau}{\text{TE} \ \vdash \texttt{stream} \ \ \text{id} \ ty \ \text{from id'} \Rightarrow [\text{id} \mapsto \tau], \varnothing} \quad (\text{StreamDecl})$$

$$\frac{\text{TE} \vdash \textit{ty} \Rightarrow \tau}{\text{TE} \vdash \texttt{stream}\ \text{id}\ \textit{ty}\ \text{to id'} \Rightarrow \varnothing, [\text{id} \mapsto \tau]} \quad (\text{StreamDecl})$$

5.2.6 Network declarations

$$\boxed{\text{TE}, \text{VE} \vdash \text{NetDecls} \Rightarrow \text{VE}'}$$

$$\frac{\text{VE}_0 = \text{VE} \qquad \forall i.\ 1 \leq i \leq n,\ \ \text{TE}, \text{VE}_{i-1} \vdash \textit{netdecl}_i \Rightarrow \text{VE}_i}{\text{VE} \vdash \textit{netdecl}_1\ \ldots\ \textit{netdecl}_n \Rightarrow \text{VE}_n} \quad (\text{NetDecls})$$

$$\boxed{\text{TE}, \text{VE} \vdash \text{NetDecl} \Rightarrow \text{VE}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ \ \text{TE}, \text{VE} \vdash \textit{nbind} \Rightarrow \text{VE}_i \qquad \text{VE}' = \bigoplus_{i=1}^{n} \text{VE}_i}{\text{TE}, \text{VE} \vdash \textbf{net}\ \langle \textit{nbind}_1, \ldots, \textit{nbind}_n \rangle \Rightarrow \text{VE} \oplus \text{VE}'} \quad (\text{NetDecl})$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ \ \text{TE}, \text{VE} \oplus \text{VE}' \vdash \textit{nbind} \Rightarrow \text{VE}_i \qquad \text{VE}' = \bigoplus_{i=1}^{n} \text{VE}_i}{\text{TE}, \text{VE} \vdash \textbf{net rec}\ \langle \textit{nbind}_1, \ldots, \textit{nbind}_n \rangle \Rightarrow \text{VE} \oplus \text{VE}'} \quad (\text{NetRecDecl})$$

$$\boxed{\text{TE}, \text{VE} \vdash \text{NetBind} \Rightarrow \text{VE}'}$$

$$\frac{\text{TE}, \text{VE} \vdash \textit{nexp} \Rightarrow \tau \qquad \vdash_p \textit{npat}, \tau \Rightarrow \text{VE}'}{\text{TE}, \text{VE} \vdash \textit{npat} = \textit{nexp} \Rightarrow \text{VE}'} \quad (\text{NetBind})$$

Patterns

The following rules are used to handle pattern binding at the network level :

$$\boxed{\vdash_p \text{NPat}, \tau \Rightarrow \text{VE}}$$

$$\frac{}{\vdash_p \text{id}, \tau \Rightarrow [\text{id} \mapsto \tau]} \quad (\text{NPatVar})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash_p \textit{npat}_{\ i}, \tau_i \Rightarrow \text{VE}_i}{\vdash_p \texttt{(}\textit{npat}_{\ 1}, \ldots \textit{npat}_{\ n}\texttt{)}, \tau_1 \times \ldots \times \tau_n \Rightarrow \bigoplus_{i=1}^{n} \text{VE}_i} \quad (\text{NPat Tuple})$$

where

$$\vdash_p \textit{npat}, \tau \Rightarrow \text{VE}$$

means that declaring *npat* with type τ creates the variable environment VE.

Network Expressions

$$\boxed{\text{TE}, \text{VE} \ \vdash \text{NExp} \Rightarrow \tau}$$

$$\frac{\text{VE}(id) = \tau}{\text{TE}, \text{VE} \vdash \text{id} \Rightarrow \tau} \quad (\text{NVar})$$

$$\frac{}{\text{TE}, \text{VE} \vdash \text{int/bool} \Rightarrow \mathsf{Int/Bool}} \quad (\text{NConst})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \text{VE} \vdash \ nexp_i \Rightarrow \tau_i}{\text{TE}, \text{VE} \vdash (nexp_1, \ \ldots \ nexp_n) \Rightarrow \tau_1 \times \ldots \times \tau_n} \quad (\text{NTuple})$$

$$\frac{\text{VE} \vdash nexp_1 \Rightarrow \tau \to \tau' \qquad \text{VE} \vdash nexp_2 \Rightarrow \tau}{\text{TE}, \text{VE} \vdash nexp_1 \ nexp_2 \Rightarrow \tau'} \quad (\text{NApp})$$

$$\frac{\vdash_p npat\ , \tau \Rightarrow \text{VE}' \qquad \text{VE} \oplus \text{VE}' \vdash nexp \Rightarrow \tau'}{\text{TE}, \text{VE} \vdash \textbf{function}\ npat\ \to nexp \Rightarrow \tau \to \tau'} \quad (\text{NFun})$$

$$\frac{\vdash_p npat\ , \tau' \Rightarrow \text{VE}' \qquad \text{VE} \vdash nexp_2 \Rightarrow \tau' \qquad \text{VE} \oplus \text{VE}' \vdash nexp_1 \Rightarrow \tau}{\text{TE}, \text{VE} \vdash \textbf{let nonrec}\ npat = nexp_2\ \textbf{in}\ nexp_1 \Rightarrow \tau} \quad (\text{NLet})$$

$$\frac{\vdash_p npat\ , \tau' \Rightarrow \text{VE}' \qquad \text{VE} \oplus \text{VE}' \vdash nexp_2 \Rightarrow \tau' \qquad \text{VE} \oplus \text{VE}' \vdash nexp_1 \Rightarrow \tau}{\text{TE}, \text{VE} \vdash \textbf{let rec}\ npat = nexp_2\ \textbf{in}\ nexp_1 \Rightarrow \tau} \quad (\text{NLet Rec})$$

5.2.7 Type expressions

$$\boxed{\text{TE} \ \vdash ty \Rightarrow \tau, \text{TE}'}$$

$$\frac{\text{TE}.tycons(\chi) = \tau}{\text{TE} \vdash \chi \Rightarrow \tau, \varnothing} \quad (\text{TyCon0})$$

$$\frac{\begin{array}{c}\text{TE}.tycons(\chi) = \tau_1 \ \times \ldots \times \ \tau_n \to \tau \\ \forall i.\ 1 \leq i \leq n, \quad \text{TE} \vdash ty_i \Rightarrow \tau_i, \varnothing\end{array}}{\text{TE} \vdash \chi\ \langle ty_1, \ldots, ty_n \rangle \Rightarrow \tau, \varnothing} \quad (\text{TyCon})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \text{TE} \vdash \ ty_i \Rightarrow \tau_i, \varnothing}{\text{TE} \vdash ty_1 \times \ldots \times ty_n \Rightarrow \tau_1 \times \ldots \times \tau_n, \varnothing} \quad (\text{TyTuple})$$

$$\frac{\text{TE} \vdash ty \Rightarrow \tau, \varnothing \qquad \vdash ty' \Rightarrow \tau', \varnothing}{\text{TE} \vdash ty \ \to \ ty' \Rightarrow \tau \to \tau', \varnothing} \quad (\text{TyFun})$$

$$\frac{\text{tyname}, \tau = \text{new_tyname}()}{\text{TE} \vdash \textbf{enum}\langle \text{c}_1, \ldots, \text{c}_n \rangle \Rightarrow \tau, \{tycons = [\text{tyname} \mapsto \tau]; ctors = [c_1 \mapsto \tau, \ldots, c_n \mapsto \tau]\}} \quad (\text{TyEnum})$$

The function *new_tyname* generates a fresh type name and type constructor.

$$\boxed{\mathrm{TE} \vdash ty \Rightarrow \tau}$$

$$\frac{\mathrm{TE} \vdash ty \Rightarrow \tau, \mathrm{TE}'}{\mathrm{TE} \vdash ty \Rightarrow \tau}$$

The main originality here is the introduction of local, anonymous declarations for enumerated types. These declarations are given when declaring local variables within actors. The scope of the implicitely declared type is limited to the enclosing actor. As a result, typing a type expression may result in an updated type environment, which is reflected in the signature of the above rules.

Chapter 6

Static semantics

The static semantics of CAPH programs is in the form of a set of *boxes* interconnected by *wires*. Boxes result from the instanciation of actors and wires from the data dependencies expressed in the definition section.

The static semantics is built upon the semantic domain *SemVal* given below.

In the semantic rules, tuples will be denoted $\langle x_1, \ldots, x_n \rangle$. The size of a tuple t will be denoted $|t|$. The notation D^* is used to define the domain of all tuples of D: $\langle\rangle$, $\langle D \rangle$, $\langle D, D \rangle$, ... The notation D^+ is used to define the domain of all non-empty tuples of D: $\langle D \rangle$, $\langle D, D \rangle$, ... A single value and a tuple of size one will be considered as semantically equivalent (this simplifies the description of the rule dealing with the instanciation of actors). We will use the notation . to denote the "*dont care*" (wildcard) value in rules (in order to avoid confusion with the "*dont care*" value at the syntactical level (_).

If E is an environment, and f a function, then we denote by $\Pi_f(E)$ the environment obtained by applying f to each element of codom(E) : $\Pi_f(E) = \{x \mapsto f(E(x)), x \in dom(E)\}$. If F is a set $\{f_1, \ldots, f_n\}$ of functions, then we abbreviate $\Pi_F(E)$ the successive applications $\Pi_{f_1}(\Pi_{f_2}(\ldots \Pi_{f_n}(E) \ldots))$.

Variable	Set ranged over	Definition	Meaning
v	EVal	$\mathsf{Int} + \mathsf{Bool} + \mathsf{ETuple}$	
		$+\mathsf{ECon} + \mathsf{EPrim}$	
		$+\mathsf{EFun} + \mathsf{Unknown}$	Expression-level semantic value
ι	Int	$\{\ldots, -2, -1, 0, 1, \ldots\}$	
β	Bool	$\{\mathrm{true}, \mathrm{false}\}$	Builtin constant
c	ECon	id EVal^*	Constructed value
vs	ETuple	EVal^+	Expression-level Tuple
EE	EEnv	$\{\mathrm{id} \mapsto \mathsf{EVal}\}$	Expression-level environment
ρ	NVal	$\mathsf{Loc} + \mathsf{Act} + \mathsf{Clos} + \mathsf{Tuple}$	
		$+\mathsf{Box} + \mathsf{Wire} + \mathsf{NEVal}$	Network-level semantic value
NE	NEnv	$\{\mathrm{id} \mapsto \mathsf{NVal}\}$	Network-level environment
ρs	NTuple	NVal^+	Network-level tuples
cl	Clos	$\langle n_pattern, n_expr, \mathsf{NEnv}\rangle$	Network-level closure
a	Act	$\langle \mathsf{id},\ \mathsf{id}^*,\ \mathsf{id}^*,\ \mathsf{id}^*,\ \{\mathrm{id} \mapsto expr\},\ rules\rangle$	Static actor description
ℓ	Loc	$\langle \mathsf{bid}, \mathsf{sel}\rangle$	Location
b	Box	$\langle \mathsf{id},\ \mathsf{btag},\ \{\mathsf{id} \mapsto \mathsf{EVal}\},\ \{\mathsf{id} \mapsto \mathsf{EVal}\}$	Box
		$\{\mathsf{id} \mapsto \mathsf{wid}\},\ \{\mathsf{id} \mapsto \mathsf{wid}\},\ rules)$	
t	btag	$\mathsf{BBox} + \mathsf{BIn} + \mathsf{BOut} + \mathsf{BDummy}$	Box tag
w	Wire	$\langle\langle \mathsf{bid}, \mathsf{sel}\rangle, \langle \mathsf{bid}, \mathsf{sel}\rangle\rangle$	Wire
ρ'	NEVal	$\mathsf{Int} + \mathsf{Bool}$	
B		$\{\mathsf{bid} \mapsto \mathsf{Box}\}$	Box environment
W		$\{\mathsf{wid} \mapsto \mathsf{Wire}\}$	Wire environment
l, l'	bid	$\{1, 2, \ldots\}$	Box id
k, k'	wid	$\{1, 2, \ldots\}$	Wire id
s, s'	sel	$\{1, 2, \ldots\}$	Slot indexes

The semantic categories EVal (for expression-level values), Tuple and Clos are classical. The semantic category NEVal is used to wrap a subset of expression-level semantic values (integer and boolean constants) as network values. This is required to be able to manipulate actor parameters at the network level. Unknown values are used for uninitialized actor variables.

Act values describe actors. An actor is a 6-tuple consisting of

- a name,
- a list of parameter names,
- a list of input and output names,
- a list of local variables, with an optional expression describing the initial value,
- a list of rules.

Note that initial value of actor variables is described as an *unevaluated* expression since the corresponding semantic value can only be computed when the actor is instanciated as a box (it may depend on the actual value of the actor parameters, set at the network level).

The actor rules are just copied from the abstract syntaxt representation, since the static semantics does not deal with the *behavior* of actors (actors are essentially viewed as “black boxes” at this level). They will be used by the dynamic semantics.

Box values describe boxes (instanciated actors). A box is a 7-tuple consisting of

- the name of corresponding actor,
- a tag, for distinguishing "ordinary" boxes (resulting from the instanciation of actors) from boxes corresponding to i/o streams[1],
- a list of parameters, with their actual values,
- a list of local variables, with their actual values,
- a list of inputs and outputs slots, each associating a name with a wire index,
- a list of rules (copied verbatim from the actor description).

Wire values are used to describe interconnexions between boxes. Each box has a unique index (bid). A *location* (Loc) identifies a given i/o slot of a box (inputs and outputs are numbered from 1 to n). So wires are actually described by a pair of locations.

Semantic rules are given in a context consisting of

- a *type environment* TE recording type constructors[2],
- a *expression-level environment* EE, mapping identifiers to expression-level semantic values,
- a *network-level environment* NE, mapping identifiers to network-level semantic values,
- a *box environment* B, mapping indexes to *boxes*,
- a *wire environment* W, mapping indexes to *wires*.

The notations for accessing and manipulating environments are similar to those defined in chapter 5. Empty environments are noted $\varnothing$.

[1] The tag BDummy is used internally for handling recursive definitions.
[2] This environment is required for evaluating type coercion operations.

6.1 Programs

$$\boxed{\mathrm{TE}_0, \mathrm{EE}_0 \vdash \mathrm{Program} \Rightarrow \mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W}}$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{EE} \vdash \mathit{valdecls} \Rightarrow \mathrm{EE}' \\ \mathrm{TE}, \mathrm{EE}' \vdash \mathit{actdecls} \Rightarrow \mathrm{NE}_a \\ \mathrm{TE} \vdash \mathit{strdecls} \Rightarrow \mathrm{NE}_s, \mathrm{B} \\ \mathrm{TE}, \mathrm{EE} \oplus \mathrm{EE}', \mathrm{NE}_a \oplus \mathrm{NE}_s, \mathrm{B} \vdash \mathit{netdecls} \Rightarrow \mathrm{B}', \mathrm{W}' \\ \mathrm{W}' \vdash \mathrm{B} \oplus \mathrm{B}' \Rightarrow \mathrm{B}''\end{array}}{\mathrm{TE}, \mathrm{EE} \vdash \textbf{program}\ \mathit{valdecls}\ \mathit{actdecls}\ \mathit{strdecls}\ \mathit{netdecls} \ \Rightarrow\ \mathrm{TE}, \mathrm{EE}', \mathrm{NE}_a, \mathrm{B}'', \mathrm{W}'} \ (\textsc{Program})$$

The initial environment EE contains the internal value[3] of the expression-level builtin primitives (+, -, . . .).

The result consists of

- a type environment, containing the declared type constructors[4],
- an expression-level environment, containing the declared constants and functions,
- a network-level environment, containing the declared actors.
- a indexed set of boxes and wires, describing the network,

Evaluation of network-level definitions produces a set of boxes and a set of wires, in which the latter refer to the former. The last subgoal of the previous rule – detailed in the following rules – makes boxes also refer to the wires (a kind of "reverse-wiring" step).

$$\boxed{\mathrm{W} \vdash \mathrm{B} \Rightarrow \mathrm{B}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\quad \mathrm{W} \vdash b_i \Rightarrow b'_i}{\mathrm{W} \vdash \{b_1,\ \ldots,\ b_n\} \ \Rightarrow \{b'_1,\ \ldots,\ b'_n\}}$$

$$\boxed{\mathrm{W} \vdash \mathrm{b} \Rightarrow \mathrm{b}'}$$

$$\frac{\mathrm{W} \vdash \mathrm{bid}, \mathit{bins} \Rightarrow \mathit{bins}' \qquad \mathrm{W} \vdash \mathrm{bid}, \mathit{bouts} \Rightarrow \mathit{bouts}'}{\mathrm{W} \vdash \mathrm{bid} \mapsto \langle \mathit{id}, \mathit{tag}, \mathit{bparams}, \mathit{bins}, \mathit{bouts}\rangle \ \Rightarrow \mathrm{bid} \mapsto \langle \mathit{id}, \mathit{tag}, \mathit{bparams}, \mathit{bins}', \mathit{bouts}'\rangle}$$

$$\boxed{\mathrm{W} \vdash \mathrm{bid}, \mathit{bios} \Rightarrow \mathit{bios}'}$$

[3] Function from semantic values to semantic values.
[4] These constructors are required to evaluate *cast* expressions.

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{W} \vdash \mathrm{bid}, i \Rightarrow \mathrm{wid}_i}{\mathrm{W} \vdash \mathrm{bid},\ \{\mathrm{id}_1 \mapsto 0,\ \ldots,\ \mathrm{id}_n \mapsto 0\} \ \Rightarrow \{\mathrm{id}_1 \mapsto \mathrm{wid}_1,\ \ldots,\ \mathrm{id}_n \mapsto \mathrm{wid}_n\}}$$

$$\boxed{\mathrm{W} \vdash \mathrm{bid}, \mathrm{sel} \Rightarrow \mathrm{wid}}$$

$$\frac{\mathrm{W}(k) = \langle ., \langle \mathrm{bid}, \mathrm{sel} \rangle \rangle}{\mathrm{W} \vdash \mathrm{bid}, \mathrm{sel} \Rightarrow k}$$

6.2 Value declarations

$$\boxed{\mathrm{TE}, \mathrm{EE} \vdash \mathrm{ValDecls} \Rightarrow \mathrm{EE}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE}_{i-1} \vdash \mathit{valdecl}_i \Rightarrow \mathrm{EE}_i, \quad \mathrm{EE}_0 = \varnothing}{\mathrm{TE} \vdash \mathit{valdecl}_1\ \ldots\ \mathit{valdecl}_n \Rightarrow \mathrm{EE}_n} \qquad (\textsc{ValDecls})$$

A value definition can only refer to a value defined before (hence the order of declaration is relevant).

$$\frac{\mathrm{TE}, \mathrm{EE} \vdash \mathit{exp} \Rightarrow v \quad \mathit{is_static_const}(v)}{\mathrm{TE}, \mathrm{EE} \vdash \texttt{const}\ \mathrm{id} = \mathit{exp} \Rightarrow [\mathrm{id} \mapsto v]} \qquad (\textsc{ConstDecl})$$

The predicate *is_static_const* is true only for integer and boolean constants.

$$\frac{}{\mathrm{TE}, \mathrm{EE} \vdash \texttt{fun}\ \mathrm{id} = \mathit{pat}\ \rightarrow\ \mathit{exp} \Rightarrow [\mathrm{id} \mapsto \mathsf{EFun}(\mathit{pat}, \mathit{exp})]} \qquad (\textsc{FunDecl})$$

6.3 Expressions

This set of rules is classical. Note that the type environment TE is needed to process *cast* expressions.

$$\boxed{\mathrm{TE}, \mathrm{EE} \vdash \mathsf{Exp} \Rightarrow \mathrm{v}}$$

$$\frac{}{\mathrm{TE}, \mathrm{EE} \vdash \mathrm{int/bool} \Rightarrow \mathsf{Int/Bool}} \qquad (\textsc{EConst})$$

$$\frac{\mathrm{EE}(\mathrm{id}) = \mathrm{v}}{\mathrm{TE}, \mathrm{EE} \vdash \mathrm{id} \Rightarrow \mathrm{v}} \qquad (\textsc{EVar})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE} \vdash \mathit{exp}_i \Rightarrow \mathrm{v}_i}{\mathrm{TE}, \mathrm{EE} \vdash \mathrm{con}\ \langle \mathit{exp}_1, \ldots, \mathit{exp}_n \rangle \Rightarrow \mathsf{ECon}\ \langle \mathrm{v}_1, \ldots, \mathrm{v}_n \rangle} \qquad (\textsc{ECon})$$

$$\frac{\begin{array}{c}\mathrm{EE}(\mathrm{id}) = \mathsf{EPrim}\ f \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE} \vdash \mathit{exp}_i \Rightarrow \mathrm{v}_i \\ \overline{f}(\mathrm{v}_1, \ldots, \mathrm{v}_n) = \mathrm{v}\end{array}}{\mathrm{TE}, \mathrm{EE} \vdash \mathrm{id}\ (\mathit{exp}_1, \ldots, \mathit{exp}_n) \Rightarrow \mathrm{v}} \qquad (\textsc{EFunApp1})$$

Rule EFunApp1 deals with the application of builtin functions. For each function f, we assume that $\overline{f}(v_1, \ldots, v_k)$ denotes the proper resulting value, provided that k is equal to the function arity and that the arguments $v_1, \ldots, v_k$ are of the proper type[5].

$$\frac{\begin{array}{c}\text{EE(id)} = \mathsf{EFun}\ \langle\langle \text{id}_1, \ldots, \text{id}_n\rangle, \mathit{exp}\rangle \\ \forall i.\ 1 \leq i \leq n,\quad \text{TE, EE} \vdash \mathit{exp}_i \Rightarrow \text{v}_i \\ \text{TE, EE} \oplus [\text{id}_1 \mapsto \text{v}_1, \ldots, \text{id}_n \mapsto \text{v}_n] \vdash \mathit{exp} \Rightarrow \text{v}\end{array}}{\text{TE, EE} \vdash \text{id}\ (\mathit{exp}_1, \ldots, \mathit{exp}_n) \Rightarrow \text{v}} \quad (\text{EFunApp2})$$

Rule EFunApp2 deals with the application of globally defined functions. It follows the classical call-by-value strategy (the function expression is evaluated in an environment augmented with the bindings resulting from binding its pattern to the argument).

$$\frac{\text{TE, EE} \vdash \mathit{exp}_2 \Rightarrow \text{v} \qquad \text{TE, EE} \oplus [\text{id} \mapsto \text{v}] \vdash \mathit{exp}_1 \Rightarrow \text{v}'}{\text{TE, EE} \vdash \texttt{let}\ \text{id=}\ \mathit{exp}_2\ \texttt{in}\ \mathit{exp}_1 \Rightarrow \text{v}'} \quad (\text{ELet})$$

$$\frac{\text{TE, EE} \vdash \mathit{exp} \Rightarrow \mathsf{true} \qquad \text{TE, EE} \vdash \mathit{exp}_1 \Rightarrow \text{v}}{\text{TE, EE} \vdash \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}_1\ \texttt{else}\ \mathit{exp}_2 \Rightarrow \text{v}} \quad (\text{ECond0})$$

$$\frac{\text{TE, EE} \vdash \mathit{exp} \Rightarrow \mathsf{false} \qquad \text{TE, EE} \vdash \mathit{exp}_2 \Rightarrow \text{v}}{\text{TE, EE} \vdash \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}_1\ \texttt{else}\ \mathit{exp}_2 \Rightarrow \text{v}} \quad (\text{ECond1})$$

$$\frac{\text{TE, EE} \vdash \mathit{exp} \Rightarrow \text{v} \qquad \text{TE} \vdash \mathit{ty} \Rightarrow \tau \text{TE} \vdash \mathit{coerce}(\text{v}, \tau) = \text{v}'}{\text{TE, EE} \vdash \mathit{exp} : \mathit{ty} \Rightarrow \text{v}'} \quad (\text{ECast})$$

The function *coerce* coerces a value to a given type. Coercibility has been checked by the at the typing stage. The coercibility relation and the behavior of the *coerce* function have been defined in Sec. 2.3.6.

6.4 Actor declarations

$$\boxed{\text{TE, EE} \vdash \text{ActorDecls} \Rightarrow \text{NE}}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\quad \text{TE, EE} \vdash \mathit{actdecl}_i \Rightarrow \text{NE}_i}{\text{TE, EE} \vdash \mathit{actdecl}_1\ \ldots\ \mathit{actdecl}_n \Rightarrow \bigoplus_{i=1}^{n} \text{NE}_i} \quad (\text{ActorDecls})$$

$$\frac{\begin{array}{c}\vdash \text{params} \Rightarrow \text{params}' \\ \vdash \text{ins} \Rightarrow \text{ins}' \\ \vdash \text{outs} \Rightarrow \text{outs}' \\ \vdash \text{vars} \Rightarrow \text{vars}'\end{array}}{\begin{array}{c}\text{TE, EE} \vdash \texttt{actor}\ \ \text{id params ins outs vars rules} \\ \Rightarrow [\text{id} \mapsto \mathsf{Act}\ \langle \text{id}, \text{params}', \text{ins}', \text{outs}', \text{vars}', \text{rules}\rangle]\end{array}} \quad (\text{ActorDecl})$$

The rule ActorDecl builds Act semantic values from the corresponding description at the abstract syntax level.

[5]This is ensured by the the type cheking stage.

$$\boxed{\vdash \text{params/ins/outs} \Rightarrow \text{params}'/\text{ins}'/\text{outs}'}$$

$$\frac{}{\vdash \text{id}_1 : ty_1 \ \ldots \ \text{id}_n : ty_n \Rightarrow \{\text{id}_1, \ldots, \text{id}_n\}} \quad (\textsc{ActorParamsInsOuts})$$

$$\boxed{\vdash \text{vars} \Rightarrow \text{vars}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash var_i \Rightarrow \text{EE}_i}{\vdash var_1 \ \ldots \ var_n \Rightarrow \bigoplus_{i=1}^{n} \text{EE}_i} \quad (\textsc{ActorVars})$$

$$\boxed{\vdash \text{var} \Rightarrow \text{var}'}$$

$$\frac{}{\vdash \text{id} \ : \ ty \ = \ exp \Rightarrow [\text{id} \mapsto exp]} \quad (\textsc{ActVar})$$

6.5 Stream declarations

$$\boxed{\text{TE} \vdash \text{StreamDecls} \Rightarrow \text{NE}, \text{B}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash strdecl_i \Rightarrow \text{NE}_i, \text{B}_i}{\text{TE} \vdash strdecl_1 \ \ldots \ strdecl_n \Rightarrow \bigoplus_{i=1}^{n} \text{NE}_i, \bigoplus_{i=1}^{n} \text{B}_i} \quad (\textsc{StreamDecls})$$

$$\frac{\text{b} = \langle \text{id}, \mathsf{BIn}, \langle\rangle, \langle\rangle, \langle \text{"}o\text{"} \mapsto 0 \rangle\rangle \qquad \text{l} = \textsc{NewBid}()}{\text{TE} \vdash \texttt{stream}\ \text{id}\ ty\ \text{from id'} \Rightarrow [\text{id} \mapsto \mathsf{Loc}(l, 1)], [l \mapsto \text{b}]} \quad (\textsc{InStreamDecl})$$

$$\frac{\text{b} = \langle \text{id}, \mathsf{BOut}, \langle\rangle, \langle \text{"}i\text{"} \mapsto 0 \rangle, \langle\rangle\rangle \qquad \text{l} = \textsc{NewBid}()}{\text{TE} \vdash \texttt{stream}\ \text{id}\ ty\ \text{to id'} \Rightarrow [\text{id} \mapsto \mathsf{Loc}(l, 1)], [l \mapsto \text{b}]} \quad (\textsc{OutStreamDecl})$$

Each stream declaration creates a new box and enters the corresponding location in the environment. The BIn (resp. BOut) boxes have no parameter, no input (resp. output). Wire identifiers in the input (resp. output) list are set to 0 at this stage. They will be updated when all definitions have been processed (see last sub-goal of rule Program).

The function NewBid returns a new, fresh box index (i.e. an index l such as $l \notin Dom(\text{B})$).

6.6 Network declaration

This section only deals with *non recursive* declarations. Recursive declarations are handled in Sec. 6.6.2.

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathrm{NetDecls} \Rightarrow \mathrm{NE}', \mathrm{B}, \mathrm{W}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE}, \mathrm{NE}_{i-1}, \mathrm{B}_{i-1}, \mathrm{W}_{i-1} \vdash \mathit{netdecl}_i \Rightarrow \mathrm{NE}_i, \mathrm{B}_i, \mathrm{W}_i \qquad \mathrm{NE}_0 = \mathrm{NE}}{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathit{netdecl}_1, \ldots, \mathit{netdecl}_n \Rightarrow \mathrm{NE}_n,\ \mathrm{B}_n,\ \mathrm{W}_n} \ (\textsc{NetDecls})$$

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathrm{NetDecl} \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathit{npat}_i{=}\mathit{nexp}_i \Rightarrow \mathrm{NE}_i, \mathrm{B}_i, \mathrm{W}_i \\ \mathrm{NE}' = \bigoplus_{i=1}^{n} \mathrm{NE}_i,\ \mathrm{B}' = \bigoplus_{i=1}^{n} \mathrm{B}_i,\ \mathrm{W}' = \bigoplus_{i=1}^{n} \mathrm{W}_i\end{array}}{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathbf{net}\ \langle \mathit{npat}_1{=}\mathit{nexpr}_1, \ldots, \mathit{npat}_n{=}\mathit{nexpr}_n \rangle \Rightarrow \mathrm{NE} \oplus \mathrm{NE}', \mathrm{B} \oplus \mathrm{B}', \mathrm{W} \oplus \mathrm{W}'} \ (\textsc{NetDecl})$$

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \langle \mathit{npat}_1{=}\mathit{nexpr}_1, \ldots, \mathit{npat}_n{=}\mathit{nexpr}_n \rangle \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathit{npat}_i{=}\mathit{nexp}_i \Rightarrow \mathrm{NE}_i, \mathrm{B}_i, \mathrm{W}_i \\ \mathrm{NE}' = \bigoplus_{i=1}^{n} \mathrm{NE}_i,\ \mathrm{B}' = \bigoplus_{i=1}^{n} \mathrm{B}_i,\ \mathrm{W}' = \bigoplus_{i=1}^{n} \mathrm{W}_i\end{array}}{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \langle \mathit{npat}_1{=}\mathit{nexpr}_1, \ldots, \mathit{npat}_n{=}\mathit{nexpr}_n \rangle \Rightarrow \mathrm{NE} \oplus \mathrm{NE}', \mathrm{B} \oplus \mathrm{B}', \mathrm{W} \oplus \mathrm{W}'} \ (\textsc{NetBindings})$$

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathit{npat}{=}\mathit{nexpr} \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathit{nexp} \Rightarrow \mathrm{v}, \mathrm{B}', \mathrm{W}' \qquad \mathrm{NE}, \mathrm{B}' \vdash_n \mathit{npat}, \mathrm{v} \Rightarrow \mathrm{NE}', \mathrm{W}''}{\mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \vdash \mathit{npat} =\ \mathit{nexp} \Rightarrow \mathrm{NE} \oplus \mathrm{NE}',\ \mathrm{B} \oplus \mathrm{B}',\ \mathrm{W} \oplus \mathrm{W}' \oplus \mathrm{W}''} \ (\textsc{NetBinding})$$

Each definition potentially augments the variable environment and the box and wire sets. Evaluating an expression potentially creates boxes and wires. Binding a pattern may only create wires.

Network-level pattern binding is handled using the following rules, where

$$\mathrm{NE}, \mathrm{B} \vdash_n \mathit{npat}, \rho \Rightarrow \mathrm{NE}', \mathrm{W}'$$

means that in the context of NE and B binding *npat* to value ρ results in a network-level environment NE and a set of wires W'.

$$\boxed{\mathrm{NE}, \mathrm{B} \vdash_n \mathrm{NPat}, \rho \Rightarrow \mathrm{NE}', \mathrm{W}}$$

$$\overline{\text{NE}, \text{B} \vdash_n id, \rho \Rightarrow [id \mapsto \rho],\ \varnothing} \quad (\text{TNPatVar})$$

$$\frac{\forall i.\ 1 \le i \le n, \quad \text{NE}, \text{B} \vdash_n npat_i, \rho_i \Rightarrow \text{NE}_i,\ \text{W}_i}{\text{NE}, \text{B} \vdash_n (npat_1, \ldots, npat_n), \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \bigoplus_{i=1}^{n} \text{NE}_i,\ \bigoplus_{i=1}^{n} \text{W}_i} \quad (\text{TNPatTuple})$$

$$\frac{\text{NE}(id) = \mathsf{Loc}(l', 1) \qquad \rho = \mathsf{Loc}(l, s) \qquad \text{B}(l) = \langle \mathsf{BOut}, \ldots \rangle \qquad k = \text{NewWid}()}{\text{NE}, \text{B} \vdash_n id, \rho \Rightarrow \varnothing,\ [k \mapsto \langle \langle l, s \rangle, \langle l', 1 \rangle \rangle]} \quad (\text{TNPatOutput})$$

The last rule handes the case where a an identifier previously declared as a stream output is bound to an expression. In this case, a wire is inserted, connecting the box resulting from the evaluation of this expression to the box which has been created when instanciating the stream output.

The function NewWid returns a new, fresh wire index (i.e. an index k such as $k \notin Dom(\text{W})$).

6.6.1 Network expressions

Here the rules are fairly standard, except for those dealing with instanciation of actors and recursive `let` definitions.

$$\boxed{\text{TE}, \text{EE}, \text{NE} \vdash \text{NExp} \Rightarrow \rho, \text{B}, \text{W}}$$

$$\frac{\text{NE}(id) = \rho}{\text{TE}, \text{EE}, \text{NE} \vdash id \Rightarrow \rho,\ \varnothing,\ \varnothing} \quad (\text{NNVar})$$

$$\frac{\text{EE}(id) = \text{v}, \quad wrapable(\text{v})}{\text{TE}, \text{EE}, \text{NE} \vdash id \Rightarrow \mathsf{NEVal}(\text{v}),\ \varnothing,\ \varnothing} \quad (\text{NEVar})$$

The value of an identifier appearing in a network expression is searched first in the network-level environment and, if this fails, in the expression-level environment. The predicate *wrapable* indicates whether an expression-level value v can be wrapped as network-level value (in other words whether this value can be used as a parameter value for an actor). It is only true for Int and Bool constants.

$$\overline{\text{TE}, \text{EE} \vdash \text{int/bool} \Rightarrow \mathsf{NEVal(Int/Bool)}} \quad (\text{NEConst})$$

$$\frac{\forall i.\ 1 \le i \le n, \quad \text{NE} \vdash nexp_i \Rightarrow \rho_i, \text{B}_i, \text{W}_i}{\text{TE}, \text{EE}, \text{NE} \vdash (nexp_1, \ldots, nexp_n) \Rightarrow \langle v_1, \ldots, v_n \rangle,\ \bigoplus_{i=1}^{n} \text{B}_i,\ \bigcup_{i=1}^{n} \text{W}_i} \quad (\text{NTuple})$$

$$\overline{\text{TE}, \text{EE}, \text{NE} \vdash \texttt{function}\ npat \rightarrow nexp\ \Rightarrow \mathsf{Clos}(npat, nexp, \text{NE}),\ \varnothing,\ \varnothing} \quad (\text{NFun})$$

$$\frac{\begin{array}{c}\text{TE}, \text{EE}, \text{NE}, \emptyset, \emptyset \vdash \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n \rangle \Rightarrow \text{NE}', \text{B}, \text{W} \\ \text{TE}, \text{EE}, \text{NE} \oplus \text{NE}' \vdash nexp_2 \Rightarrow \rho, \text{B}', \text{W}''\end{array}}{\text{TE}, \text{EE}, \text{NE} \vdash \texttt{let}\ \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n \rangle\ \texttt{in}\ nexp_2 \Rightarrow \rho,\ \text{B},\ \text{W}} \quad (\text{NLetDef})$$

The above rule is for *non recursive* definitions. Recursive definitions are handled in Sec. 6.6.2.

The following rules deals with applications.

Rule NAppClo 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 its pattern to the argument).

$$\frac{\begin{array}{c}\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1 \Rightarrow \mathsf{Clos}(\mathit{npat},\mathit{nexp},\mathrm{NE}'),\mathrm{B},\mathrm{W} \\ \mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_2 \Rightarrow \rho,\mathrm{B}',\mathrm{W}' \\ \mathrm{NE},\mathrm{B} \vdash_n \mathit{npat},\rho \Rightarrow \mathrm{NE}'',. \\ \mathrm{NE}' \oplus \mathrm{NE}'' \vdash \mathit{nexp} \Rightarrow \rho',\mathrm{B}'',\mathrm{W}''\end{array}}{\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1\ \mathit{nexp}_2 \Rightarrow \rho',\mathrm{B}\oplus\mathrm{B}'\oplus\mathrm{B}',\mathrm{W}\cup\mathrm{W}'\cup\mathrm{W}''} \quad (\textsc{NAppClo})$$

Rules NAppAct1 and NAppAct2 deals with the instanciation of actors. Both insert a new box and a set of new wires connecting the argument to the inputs of the inserted box. The former deals with actors with parameters, the latter with actors without parameters.

$$\frac{\begin{array}{c}\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1 \Rightarrow \\ \mathsf{Act}\langle \mathit{id}, \langle p_1,\ldots,p_q\rangle, \langle i_1,\ldots,i_m\rangle, \langle o_1,\ldots,o_n\rangle, [v_1 \mapsto \mathit{exp}_1,\ldots,v_k \mapsto \mathit{exp}_k], \mathit{rules}\rangle, \quad \mathrm{B}',\mathrm{W}' \\ \forall i.\ 1\le i\le k,\ \mathrm{TE},\ \mathrm{EE}\oplus \bigoplus_{i=1}^{q} [p_i \mapsto \mathrm{v}'_i] \vdash \mathit{exp}_i \Rightarrow \mathrm{v}_i \\ \mathit{TE},\mathrm{EE},\mathit{NE} \vdash \mathit{nexp}_2 \Rightarrow \langle \mathsf{NEVal}(\mathrm{v}'_1),\ldots,\mathsf{NEVal}(\mathrm{v}'_q)\rangle, \mathrm{B}',\mathrm{W}' \\ \mathit{TE},\mathrm{EE},\mathit{NE} \vdash \mathit{nexp}_3 \Rightarrow \langle \mathsf{Loc}(l_1,s_1),\ldots,\mathsf{Loc}(l_m,s_m)\rangle, \mathrm{B}'',\mathrm{W}'' \\ \mathrm{b} = \langle \mathit{id}, \mathsf{BBox}, [p_1\mapsto \mathrm{v}'_1,\ldots,p_q\mapsto \mathrm{v}'_q], [v_1\mapsto \mathrm{v}_1,\ldots,v_k\mapsto \mathrm{v}_k], \langle \mathit{bi}_1,\ldots,\mathit{bi}_m\rangle, \langle \mathit{bo}_1,\ldots,\mathit{bo}_n\rangle, \mathit{rules}\rangle \\ l = \textsc{NewBid}() \\ \forall j.\ 1\le j\le m,\quad k_j = \textsc{NewWid}(),\quad \mathrm{w}_j = \mathsf{Wire}\langle\langle l_j,s_j\rangle,\langle l,j\rangle\rangle \\ \forall j.\ 1\le j\le m,\quad \mathit{bi}_j = \langle i_j, 0\rangle \\ \forall j.\ 1\le j\le n,\quad \mathit{bo}_j = \langle o_j, 0\rangle \\ \mathrm{W}''' = \{k_1\mapsto \mathrm{w}_1,\ldots,k_m\mapsto \mathrm{w}_m\}\end{array}}{\begin{array}{c}\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1\ \mathit{nexp}_2\ \mathit{nexp}_3 \\ \Rightarrow \langle \mathsf{Loc}(l,1),\ldots,\mathsf{Loc}(l,n)\rangle, \quad [l\mapsto \mathrm{b}]\oplus\mathrm{B}''\oplus\mathrm{B}'\oplus\mathrm{B}, \quad \mathrm{W}'''\cup\mathrm{W}''\cup\mathrm{W}'\cup\mathrm{W}\end{array}} \quad (\textsc{NAppAct1})$$

$$\frac{\begin{array}{c}\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1 \Rightarrow \\ \mathsf{Act}\langle \mathit{id}, \langle\rangle, \langle i_1,\ldots,i_m\rangle, \langle o_1,\ldots,o_n\rangle, [v_1 \mapsto \mathit{exp}_1,\ldots,v_k \mapsto \mathit{exp}_k], \mathit{rules}\rangle, \quad \mathrm{B}',\mathrm{W}' \\ \forall i.\ 1\le i\le k,\ \mathrm{TE},\mathrm{EE} \vdash \mathit{exp}_i \Rightarrow \mathrm{v}_i \\ \mathit{TE},\mathrm{EE},\mathit{NE} \vdash \mathit{nexp}_2 \Rightarrow \langle \mathsf{Loc}(l_1,s_1),\ldots,\mathsf{Loc}(l_m,s_m)\rangle, \mathrm{B}'',\mathrm{W}'' \\ \mathrm{b} = \langle \mathit{id}, \mathsf{BBox}, [], [v_1\mapsto \mathrm{v}_1,\ldots,v_k\mapsto \mathrm{v}_k], \langle \mathit{bi}_1,\ldots,\mathit{bi}_m\rangle, \langle \mathit{bo}_1,\ldots,\mathit{bo}_n\rangle, \mathit{rules}\rangle \qquad l = \textsc{NewBid}() \\ \forall j.\ 1\le j\le m,\quad k_j = \textsc{NewWid}(),\quad \mathrm{w}_j = \mathsf{Wire}\langle\langle l_j,s_j\rangle,\langle l,j\rangle\rangle \\ \forall j.\ 1\le j\le m,\quad \mathit{bi}_j = \langle i_j, 0\rangle \\ \forall j.\ 1\le j\le n,\quad \mathit{bo}_j = \langle o_j, 0\rangle \\ \mathrm{W}''' = \{k_1\mapsto \mathrm{w}_1,\ldots,k_m\mapsto \mathrm{w}_m\}\end{array}}{\begin{array}{c}\mathrm{TE},\mathrm{EE},\mathrm{NE} \vdash \mathit{nexp}_1\ \mathit{nexp}_2 \\ \Rightarrow \langle \mathsf{Loc}(l,1),\ldots,\mathsf{Loc}(l,n)\rangle, \quad [l\mapsto \mathrm{b}]\oplus\mathrm{B}''\oplus\mathrm{B}'\oplus\mathrm{B}, \quad \mathrm{W}'''\cup\mathrm{W}''\cup\mathrm{W}'\cup\mathrm{W}\end{array}} \quad (\textsc{NAppAct2})$$

6.6.2 Recursive network definitions

Recursive definitions may appear both within network expressions or at the network declaration level.

$$\boxed{\mathrm{TE, EE, NE} \vdash \mathrm{NExp} \Rightarrow \rho, \mathrm{B}, \mathrm{W}}$$

$$\frac{\begin{array}{c}\mathrm{TE, EE, NE}, \emptyset, \emptyset \vdash \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n\rangle_{rec} \Rightarrow \mathrm{NE}', \mathrm{B}, \mathrm{W} \\ \mathrm{TE, EE, NE} \oplus \mathrm{NE}' \vdash nexp_2 \Rightarrow \rho, \mathrm{B}', \mathrm{W}''\end{array}}{\mathrm{TE, EE, NE} \vdash \texttt{let rec}\ \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n\rangle\ \texttt{in}\ nexp_2 \Rightarrow \rho,\ \mathrm{B},\ \mathrm{W}} \ (\textsc{NLetRecDef})$$

$$\boxed{\mathrm{TE, EE, NE, B, W} \vdash \mathrm{NetRecDecl} \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\mathrm{TE, EE, NE}, \emptyset, \emptyset \vdash \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n\rangle_{rec} \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}{\mathrm{TE, EE, NE, B, W} \vdash \texttt{net rec}\ \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n\rangle \Rightarrow \mathrm{NE} \oplus \mathrm{NE}', \mathrm{B} \oplus \mathrm{B}', \mathrm{W} \oplus \mathrm{W}'}$$

(NetRecDecl)

Both cases are handled with a common rule NetRecBindings. This rule supports only two kinds of recursive bindings :

- all the bound identifiers are *functions*,
- all the bound identifiers *wires*.

In the former case, the result is a circular closure (or a set of mutually recursive closures in case of multiple bindings).

In the latter case, the recursively defined values correspond to *cycles* in the network.

Recursive functions

$$\boxed{\mathrm{TE, EE, NE, B, W} \vdash \langle npat_1{=}nexpr_1, \ldots, npat_n{=}nexpr_n\rangle_{rec} \Rightarrow \mathrm{NE}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n,\quad \mathrm{NE}_i = [\mathrm{id} \mapsto \mathrm{cl}_i] \\ \forall i.\ 1 \leq i \leq n,\quad \mathrm{cl}_i = \mathsf{Clos}(npat_i, nexp_i, \mathrm{NE} \oplus \mathrm{NE}')] \\ \mathrm{NE}' = \bigoplus_{i=1}^{n} \mathrm{NE}_i\end{array}}{\begin{array}{c}\mathrm{TE, EE, NE} \vdash \langle\ \mathrm{id}_1{=}\ \texttt{function}\ npat_1\ \rightarrow\ nexp_1, \ldots, \mathrm{id}_n{=}\ \texttt{function}\ npat_n\ \rightarrow\ nexp_n\rangle_{rec} \\ \Rightarrow \mathrm{NE}', \varnothing, \varnothing\end{array}}$$

(NRecBindingsF)

Recursive wires

If the defined value is *not* a function, then the recursively defined values correspond to *cycles* in the network. Evaluation is then carried out as follows:

1. First, we create a *recursive environment* NE′, by binding each identifier occuring in the LHS patterns to the location of a temporary, freshly created, box with tag BDummy. This can be formalized with the following set of rules:

$$\boxed{\vdash_r \mathrm{NPat} \Rightarrow \mathrm{NE}, \mathrm{B}}$$

$$\frac{\mathrm{b} = \langle \mathrm{id}, \mathsf{BDummy}, \langle\rangle, \langle "i" \mapsto 0\rangle, \langle "o" \mapsto 0\rangle\rangle \qquad \mathrm{l} = \textsc{NewBid}()}{\vdash_r id \Rightarrow [id \mapsto \mathsf{Loc}(l, 0)],\ [l \mapsto \mathrm{b}]}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash_r npat_i \Rightarrow \mathrm{NE}_i,\ \mathrm{B}_i}{\vdash_r (npat_1, \ldots, npat_n) \Rightarrow \bigoplus_{i=1}^{n} \mathrm{NE}_i,\ \bigoplus_{i=1}^{n} \mathrm{B}_i}$$

2. Second, all the RHS expressions are evaluated in an environment augmented with NE'. The resulting values are bound to the LHS patterns using the usual rules defined in section 6.6, leading an environment NE''.
3. Third, for each identifier r occuring in the recursive environment NE', we build a *substitution* $\phi = \{u \mapsto v\}$, where $u = \mathrm{NE}'(r) = Loc(l, s)$ and $v = \mathrm{NE}''(r) = Loc(l', s')$. This results in a set Φ of substitutions.
4. Fourth, we apply each $\phi \in \Phi$ to the set W' of wires produced by the evaluation of the expressions evaluated at step 2. More precisely, we replace each pair $\langle \gamma, \gamma' \rangle \in \mathrm{W}'$ by $\langle \phi(\gamma), \phi(\gamma') \rangle$, where $\phi(\gamma)$ is defined as follows:

   if $\phi = \{\mathsf{Loc}(k, s) \mapsto \mathsf{Loc}(k', s')\}$ and $\gamma = \mathsf{Loc}(l, s'')$ then

$$\phi(\gamma) = \begin{cases} \mathsf{Loc}(k', s') & \text{if } l = k, \\ \gamma & \text{otherwise.} \end{cases} \tag{6.1}$$

The process can be summarized in the following rule:

$$\frac{\begin{array}{c} \forall i.\ 1 \leq i \leq n,\ \vdash_r npat_i \Rightarrow \mathrm{NE}'_i, \mathrm{B}_i \\ \mathrm{NE}' = \bigoplus_{i=1}^{n} \mathrm{NE}'_i \qquad \mathrm{B} = \bigoplus_{i=1}^{n} \mathrm{B}_i \\ \forall i.\ 1 \leq i \leq n,\ \mathrm{TE}, \mathrm{EE}, \mathrm{NE}' \oplus \mathrm{NE} \vdash nexp_i \Rightarrow \rho_i, \mathrm{B}'_i, \mathrm{W}'_i \\ \forall i.\ 1 \leq i \leq n,\ \mathrm{NE}, \mathrm{B} \vdash_n npat_i, \rho_i \Rightarrow \mathrm{NE}''_i, \mathrm{W}'''_i \\ \mathrm{NE}'' = \bigoplus_{i=1}^{n} \mathrm{NE}''_i \\ \Phi = \{\mathrm{NE}'(r) \mapsto \mathrm{NE}''(r),\ \forall r \in dom(\mathrm{NE}')\} \\ \mathrm{W}'' = \Pi_\Phi(\mathrm{W}') \end{array}}{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \langle npat_1{=}nexp_1, \ldots, npat_n{=}nexp_n \rangle_{rec} \Rightarrow \mathrm{NE}'', \mathrm{B}', \mathrm{W}''} \ (\textsc{NRecBindingsV})$$

Chapter 7

Dynamic semantics

This semantics is the one used to define the reference interpreter.

It is defined in an axiomatic style. It assumes that type-checking has been properly carried out and that all translations defined by the static semantics are valid and have been properly carried out.

7.1 Semantic domain

The dynamic semantics is given in terms of the semantic domain $DVal$ defined below. We use the same notations than for the static semantics for tuples, environments, *etc.* In addition, we denote by $E[x \mapsto y]$ the environment that maps x to y and behaves like E otherwise. We write $[\,]$ to denote the empty list and $[a, b, c]$ for the list containing the elements a, b and c. The concatenation of two lists is written $l_1 ++ l_2$. If h is a suitable element and t is a (possibly empty) list, then $h :: t$ is the list obtained by prepending h to t; while $t ++ [h]$ denotes the list obtained by attaching element h at the end of list t. The cardinality of a list l is denoted by $|l|$.

Variable	Set ranged over	Definition	Meaning
v	DVal	$\mathsf{EVal} + \mathsf{Unknown}$	Expression-level semantic value
EE	EEnv	$\{\mathrm{id} \mapsto \mathsf{DVal}\}$	Dynamic environments
π	Process	$\langle\{\mathsf{id} \mapsto \mathsf{DVal}\},$ $\{\mathsf{id} \mapsto \mathsf{cid}\},\ \{\mathsf{id} \mapsto \mathsf{cid}\},$ $\{\mathsf{id} \mapsto \mathsf{DVal}\},\ \mathsf{Rule}^*,\ \mathsf{RIndex}\rangle$	Processes
r	Rule	$\langle \mathsf{QPat}^+, \mathsf{QExp}^+ \rangle$	Actor rule
qpat'	QPat	$\langle \mathsf{Qual}, \mathrm{id}, Pat \rangle$	Qualified rule pattern
qexp'	QExp	$\langle \mathsf{Qual}, \mathrm{id}, Exp \rangle$	Qualified rule expression
q	Qual	$\mathsf{In} + \mathsf{Out} + \mathsf{Var}$	Qualifier
theta	Channel	$\langle \mathrm{bool}, \mathrm{bool}, [\mathsf{DVal}] \rangle$	Channels
P, I, A		$\{\mathsf{pid} \mapsto \mathsf{Process}\}$	Process environment
C		$\{\mathsf{cid} \mapsto \mathsf{Channel}\}$	Channel environment
B		$\{\mathsf{bid} \mapsto \mathsf{Box}\}$	Box environment
W		$\{\mathsf{wid} \mapsto \mathsf{Wire}\}$	Wire environment
l, l'	pid	$\{1, 2, \ldots\}$	Process ids
k, k'	cid	$\{1, 2, \ldots\}$	Channel ids
j	RIndex	$\{0, 1, \ldots\}$	Rule index

Expression-level semantic values are the same as those defined for the static semantics, with the

addition of an Unknown value for dealing with uninitialized local variable.

Processes are the dynamic view of boxes. They consist of

- a list of parameters (mapping names to values),
- a list of inputs and outputs (mapping names to channel ids),
- a list of local variables (mapping names to values),
- a list of rules.
- a rule index, indicating the current fireable rule for an active process (rules are numbered from 0 to N_r; 0 means that no rule is fireable)

Channels are the dynamic view of wires. They consist of

- two boolean flags indicating whether the channel is ready for reading (not empty) and ready for writing (not full).
- a list of memorized values (modeling the buffering capabilities of the channel),

7.2 Programs

$$\boxed{\mathrm{TE}_0, \mathrm{EE}_0 \vdash \mathrm{Program} \Rightarrow \mathrm{P}, \mathrm{C}}$$

$$\frac{\begin{array}{c}\mathrm{TE}_0, \mathrm{EE}_0 \vdash \textbf{program}\ \textit{valdecls actdecls strdecls netdecls} \ \Rightarrow\ \mathrm{TE}, \mathrm{EE}, \mathrm{NE}, \mathrm{B}, \mathrm{W} \\ \mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathrm{B} \Rightarrow \mathrm{P} \qquad \vdash \mathrm{W} \Rightarrow \mathrm{C} \\ \mathrm{EE}_0 \oplus \mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{C}', \mathrm{P}'\end{array}}{\mathrm{SE}_0, \mathrm{TE}_0, \mathrm{EE}_0 \vdash \textbf{program}\ \ \textit{valdecls actdecls strdecls netdecls} \ \Rightarrow\ \mathrm{C}', \mathrm{P}'} \quad (\textsc{Program})$$

The program is first translated to set of type, expression-level, network-level, box and wire environments according to the static semantics. Boxes and wires are turned into initial sets of processes and channels and the processes are executed. The result is an final set of processes (in which no process can be executed any longer) and channels.

7.2.1 Conversion from boxes to processes

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathsf{B} \Rightarrow \mathsf{P}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash b_i \Rightarrow \pi_i}{\mathrm{W} \vdash \{l_1 \mapsto b_1, \ldots, l_n \mapsto b_n\} \ \Rightarrow \{l_1 \mapsto \pi_1, \ldots, l_n \mapsto \pi_n\}}$$

$$\boxed{\mathrm{TE}, \mathrm{EE}, \mathrm{NE} \vdash \mathsf{b} \Rightarrow \pi}$$

$$\frac{\begin{array}{c} \mathrm{b} = \langle id, \mathsf{BBox}, \mathrm{params}, \mathrm{vars}, \mathrm{bins}\rangle \mathrm{boutsrules} \\ \vdash \mathrm{bins} \Rightarrow \mathrm{ins} \\ \vdash \mathrm{bouts} \Rightarrow \mathrm{outs} \\ \mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{rules} \Rightarrow \mathrm{rules'} \end{array}}{\vdash \mathrm{l} \mapsto \mathrm{b} \;\; \Rightarrow \mathrm{l} \mapsto \langle \mathrm{params}, \mathrm{ins}, \mathrm{outs}, \mathrm{vars}, \mathrm{rules'}\rangle}$$

$$\boxed{\vdash \mathrm{bins/bouts} \Rightarrow \mathrm{ins/outs}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash \mathrm{wid}_i \Rightarrow \mathrm{cid}_i}{\vdash \{\mathrm{id}_1 \mapsto \mathrm{wid}_1, \ldots, \mathrm{id}_n \mapsto \mathrm{wid}_n\} \;\; \Rightarrow \{\mathrm{id}_1 \mapsto \mathrm{cid}_1, \ldots, \mathrm{id}_n \mapsto \mathrm{cid}_n\}}$$

$$\boxed{\vdash \mathrm{wid} \Rightarrow \mathrm{cid}}$$

$$\frac{\vdash \mathrm{wid} = i}{\vdash \mathrm{wid} \Rightarrow i}$$

Identifying wire and channel ids simplifies the translation of box ios.

The following rules refine the qualification of each pattern (resp. expression) appearing in the actor rules.

$$\boxed{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{rules} \Rightarrow \mathrm{Rules}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{rule}_i \Rightarrow \mathrm{Rule}}{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \{\mathrm{rule}_1, \ldots, \mathrm{rule}_n\} \Rightarrow \{\mathrm{rule'}_1, \ldots, \mathrm{rule'}_n\}}$$

$$\boxed{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{rule} \Rightarrow \mathrm{Rule}}$$

$$\frac{\begin{array}{c} \forall i.\ 1 \leq i \leq m, \quad \mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{qpat}_i \Rightarrow \mathrm{qpat'}_i \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{qexp}_i \Rightarrow \mathrm{qexp'}_i \end{array}}{\begin{array}{c} \mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \langle \mathrm{qpat}_1, \ldots, \mathrm{qpat}_m\rangle \rightarrow \langle \mathrm{qexp}_1, \ldots, \mathrm{qexp}_n\rangle \\ \Rightarrow \langle \mathrm{qpat'}_1, \ldots, \mathrm{qpat'}_m\rangle \rightarrow \langle \mathrm{qexp'}_1, \ldots, \mathrm{qexp'}_n\rangle \end{array}}$$

$$\boxed{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{qpat} \Rightarrow \mathrm{QPat}}$$

$$\frac{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \mathrm{id} \Rightarrow \mathrm{q}}{\mathrm{ins}, \mathrm{outs}, \mathrm{vars} \vdash \langle \mathrm{id}, pat\rangle \Rightarrow \langle \mathrm{q}, \mathrm{id}, pat\rangle}$$

$$\boxed{\text{ins}, \text{outs}, \text{vars} \vdash \text{qexp} \Rightarrow \text{QExp}}$$

$$\frac{\text{ins}, \text{outs}, \text{vars} \vdash \text{id} \Rightarrow \text{q}}{\text{ins}, \text{outs}, \text{vars} \vdash \langle \text{id}, \textit{exp} \rangle \Rightarrow \langle \text{q}, \text{id}, \textit{exp} \rangle}$$

$$\boxed{\text{ins}, \text{outs}, \text{vars} \vdash \text{id} \Rightarrow \text{q}}$$

$$\frac{q = \begin{cases} \mathsf{In} \text{ if id} \in \text{ins}, \\ \mathsf{Out} \text{ if id} \in \text{outs}, \\ \mathsf{Var} \text{ if id} \in \text{Dom(vars)} \end{cases}}{\text{ins}, \text{outs}, \text{vars} \vdash \text{id} \Rightarrow q}$$

7.2.2 Conversion from wires to channels

$$\boxed{\vdash \text{W} \Rightarrow \text{C}}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash w_i \Rightarrow c_i}{\text{W} \vdash \{k_1 \mapsto w_1, \ldots, k_n \mapsto w_n\} \ \Rightarrow \{k_1 \mapsto c_1, \ldots, k_n \mapsto c_n\}}$$

$$\boxed{\vdash \mathsf{Wire} \Rightarrow \mathsf{Channel}}$$

$$\frac{}{\vdash \langle \langle l, s \rangle, \langle l', s' \rangle \rangle \Rightarrow \langle \mathsf{false}, \mathsf{true}, [] \rangle}$$

Channels are initially empty, ready for writing and herit their id from the corresponding wire.

7.3 Processes

Execution of processes proceeds by successive synchronous cycles. Each execution cycle can be decomposed into 2 steps :

1. processes are split into two subsets : the *active* (A) subset and *inactive* (I) subset; a process is active *iff* at least one of its rules is fireable
2. all active processes are executed; during execution, a process can read values from its input channels, updates local variables and write values to its output channels.

Execution cycle are repeated until the active set becomes empty. The separation of each cycle into two steps ensures that the order in which the processes are executed at step 2 does not matter (as long as the channel have sufficient capacity). This synchronous style of scheduling, in which all active processes are executed at each cycle is coherent with a "hardware-oriented" interpretation of processes[1].

[1] A more more "software-oriented" view could prefer to execute only one process at a time.

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{C}', \mathrm{P}'}$$

$$\frac{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{I}, \mathrm{A} \qquad \mathrm{EE}, \mathrm{C} \vdash \mathrm{I}, \mathrm{A} \Rightarrow \mathrm{C}', \mathrm{P}'}{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{C}', \mathrm{P}'} \quad (\textsc{EvalPs})$$

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \mathrm{I}, \mathrm{A} \Rightarrow \mathrm{C}', \mathrm{P}'}$$

$$\frac{}{\mathrm{EE}, \mathrm{C} \vdash \mathrm{I}, \{\} \Rightarrow \mathrm{C}, \mathrm{I}} \quad (\textsc{RunPs0})$$

$$\frac{\begin{array}{c} \mathrm{A} \neq \{\} \\ \mathrm{EE} \vdash \mathrm{C}, \mathrm{A} \Rightarrow \mathrm{C}', \mathrm{A}' \\ \mathrm{EE}, \mathrm{C}' \vdash \mathrm{A}' \cup \mathrm{I} \Rightarrow \mathrm{I}', \mathrm{A}'' \\ \mathrm{EE}, \mathrm{C}' \vdash \mathrm{I}', \mathrm{A}'' \Rightarrow \mathrm{C}'', \mathrm{P}'' \end{array}}{\mathrm{EE}, \mathrm{C} \vdash \mathrm{I}, \mathrm{A} \Rightarrow \mathrm{C}'', \mathrm{P}''} \quad (\textsc{RunPs1})$$

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{C}', \mathrm{P}'}$$

$$\frac{\begin{array}{c} \mathrm{P} = \{\pi_1, \ldots, \pi_n\} \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{EE}, \mathrm{C}_{i-1} \vdash \pi_i \Rightarrow \mathrm{C}_i, \pi'_i \qquad \mathrm{C}_0 = \mathrm{C} \qquad \mathrm{C}' = \mathrm{C}_n \\ \mathrm{P}' = \{\pi'_1, \ldots, \pi'_n\} \end{array}}{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{C}', \mathrm{P}'} \quad (\textsc{ExecPs})$$

7.3.1 Identifying and marking active processes

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \mathrm{P} \Rightarrow \mathrm{P}, \mathrm{P}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{EE}, \mathrm{C} \vdash \pi_i \Rightarrow \mathrm{I}_i, \mathrm{A}_i}{\mathrm{EE}, \mathrm{C} \vdash \{\pi_1, \ldots, \pi_n\} \Rightarrow \bigcup_{i=1}^{n} \mathrm{I}_i,\ \bigcup_{i=1}^{n} \mathrm{A}_i} \quad (\textsc{SplitPs})$$

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \pi \Rightarrow \mathrm{P}, \mathrm{P}'}$$

$$\frac{\mathrm{EE}, \mathrm{C} \vdash \pi \Rightarrow \beta, \pi' \qquad \text{if } \beta \text{ then } \{\}, \{\pi'\} \text{ else } \{\pi'\}, \{\}}{\mathrm{EE}, \mathrm{C} \vdash \pi = \mathrm{I},\ \mathrm{A}} \quad (\textsc{SplitP})$$

$$\boxed{\mathrm{EE}, \mathrm{C} \vdash \pi \Rightarrow \mathsf{Bool}, \pi'}$$

$$\frac{\begin{array}{c}\pi = \langle ., ., ., ., \langle \mathrm{r}_1, \ldots, \mathrm{r}_n \rangle, . \rangle \\ \exists j.\ 1 \leq j \leq n, \quad \mathrm{EE}, \mathrm{C}, \pi \vdash \mathrm{r}_j \Rightarrow \mathsf{true} \\ \pi' = \langle ., ., ., ., \langle \mathrm{r}_1, \ldots, \mathrm{r}_n \rangle, j \rangle\end{array}}{\mathrm{EE}, \mathrm{C} \vdash \pi \Rightarrow \mathsf{true}, \pi'}$$

$$\frac{\begin{array}{c}\pi = \langle ., ., ., ., \langle \mathrm{r}_1, \ldots, \mathrm{r}_n \rangle, . \rangle \\ \forall j.\ 1 \leq j \leq n, \quad \mathrm{EE}, \mathrm{C}, \pi \vdash \mathrm{r}_j \Rightarrow \mathsf{false} \\ \pi' = \langle ., ., ., ., \langle \mathrm{r}_1, \ldots, \mathrm{r}_n \rangle, 0 \rangle\end{array}}{\mathrm{EE}, \mathrm{C} \vdash \pi \Rightarrow \mathsf{false}, \pi'}$$

A process is active if at least one of its rules is fireable. The semantics described here does not tell *which* rule is selected when several are fireable. It is therefore non-deterministic. Making it deterministic is straightforward, by assigning an fixed index to each rule and by requiring that the rule with the lowest (or highest) index is always selected, for example. Determinism is of course a highly desirable property for hardware implementations. For software implementations, other strategies can be chosen. For example, one could implement *fair* matching – like in HUME [6] – by reordering rules after selection to ensure that each fireable rule is eventually selected.

$$\boxed{\mathrm{EE}, \mathrm{C}, \pi \vdash \mathrm{Rule} \Rightarrow \mathsf{Bool}}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq m, \quad \mathrm{C}, \pi \vdash \mathrm{qpat}_i \Rightarrow \beta_i \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{C}, \pi \vdash \mathrm{qexp}_i \Rightarrow \beta'_i\end{array}}{\mathrm{EE}, \mathrm{C}, \pi \vdash \langle \mathrm{qpat}_1, \ldots, \mathrm{qpat}_n \rangle \rightarrow \langle \mathrm{qexp}_1, \ldots, \mathrm{qexp}_n \rangle \Rightarrow \beta_1 \ \wedge \ldots \wedge \ \beta_m \wedge \beta'_1 \ \wedge \ldots \wedge \ \beta'_n}$$

(FIREABLERULE)

A process rule is fireable if all its LHS patterns and RHS expressions are ready.

The following rules specify when an rule pattern is ready, depending on whether this pattern is attached to a local variable or an input channel.

$$\boxed{\mathrm{C}, \pi \vdash \mathsf{QPat} \Rightarrow \mathsf{Bool}}$$

$$\frac{\pi.vars(\mathrm{id}) = \mathrm{v} \qquad \vdash_p \mathrm{pat}, \mathrm{v} \Rightarrow \beta, \mathrm{EE}'}{\mathrm{C}, \pi \vdash \langle \mathsf{Var}, \mathrm{id}, \mathrm{pat} \rangle \Rightarrow \beta}$$

Patterns attached to variables are ready as soon as the value of the variable matches the pattern.

$$\frac{}{\mathrm{C}, \pi \vdash \langle \mathsf{In}, \mathrm{id}, _ \rangle \Rightarrow \mathrm{true}}$$

Ignored inputs are always ready.

$$\frac{\pi.ins(\mathrm{id}) = \mathrm{k} \qquad \mathrm{C}(\mathrm{k}) = \langle \mathrm{false}, ., . \rangle}{\mathrm{C}, \pi \vdash \langle \mathsf{In}, \mathrm{id}, \mathrm{pat} \rangle \Rightarrow \mathrm{false}}$$

Inputs connected to empty input channels are not ready.

$$\frac{\text{pat} \neq _ \qquad \pi.ins(\text{id}) = \text{k} \qquad \text{C(k)} = \langle \text{true}, ., v :: vs \rangle \qquad \vdash_p \text{pat}, \text{v} \Rightarrow \beta, \text{EE}'}{\text{C}, \pi \vdash \langle \mathsf{In}, \text{id}, \text{pat} \rangle \Rightarrow \beta}$$

Inputs connected to an non-empty channel are ready *iff* the first available value in the channel matches the corresponding rule pattern.

The three following rules specify when an rule expression is ready, depending on whether this expression is attached to a local variable or an output channel.

$$\boxed{\text{C}, \pi \vdash \mathsf{QExp} \Rightarrow \mathsf{Bool}}$$

$$\frac{}{\text{C}, \pi \vdash \langle \mathsf{Var},.,. \rangle \Rightarrow \mathsf{true}} \quad (\text{OutVarRdy})$$

Variables are always ready for updating.

$$\frac{}{\text{C}, \pi \vdash \langle \mathsf{Out}, \text{id}, _ \rangle \Rightarrow \text{true}}$$

Ignored outputs are always ready.

$$\frac{\text{pat} \neq _ \qquad \pi.outs(\text{id}) = \text{k} \qquad \text{C(k)} = \langle ., \beta, . \rangle}{\text{C}, \pi \vdash \langle \mathsf{Out}, \text{id}, \text{pat} \rangle \Rightarrow \beta} \quad (\text{OutChanRdy})$$

An non-ignored output is ready for *iff* the connected channel can be written (is not full).

7.3.2 Individual process execution

Execution of an active process is done as follows :

1. retrieve the index of the selected fireable rule (there must be one, since the process has been tagged as active)
2. bind the rule LHS patterns to the corresponding values (consuming the values on the involved input channels)
3. add these bindings to an environment containing global values, actor parameters and actor local variables,
4. evaluate the RHS of the selected rule in this environment, environment producing values to update outputs and local variables

$$\boxed{\text{EE}, \text{C} \vdash \pi \Rightarrow \text{C}', \pi'}$$

$$\frac{\pi = \langle \text{params}, ., ., \text{vars}, \langle \text{r}_1, \ldots, \text{r}_n \rangle \rangle j \qquad 1 \leq j \leq n \qquad \text{r}_j = \langle \text{qpat}, \text{qexps} \rangle \qquad \text{C}, \pi \vdash \text{qpats} \Rightarrow \text{EE}', \text{C}' \qquad \text{EE} \oplus \text{EE}' \oplus \text{params} \oplus \text{varsC}', \pi \vdash \text{qexps} \Rightarrow \text{C}'', \pi'}{\text{EE}, \text{C} \vdash \pi \Rightarrow \text{C}'', \pi'} \quad (\text{ExecP})$$

Binding of rule patterns

$$\boxed{\mathrm{C}, \pi \vdash \mathsf{QPats} \Rightarrow \mathrm{EE}', \mathrm{C}'}$$

$$\frac{\mathrm{C}_{i-1}, \pi \vdash \mathrm{qpat}_i \Rightarrow \mathrm{EE}_i, \mathrm{C}_i \qquad \mathrm{C}_0 = \mathrm{C}}{\mathrm{C}, \pi \vdash \langle \mathrm{qpat}_1, \ldots, \mathrm{qpat}_n \rangle \Rightarrow \bigoplus_{i=1}^{n} \mathrm{EE}_i,\ \mathrm{C}_n} \quad (\textsc{BindRPats})$$

The previous rule creates a local environment by binding the patterns of the rule RHS. The four next rules details the process of pattern matching depending on whether the pattern is attached to a variable or an input channel.

$$\boxed{\mathrm{C}, \pi \vdash \mathsf{QPat} \Rightarrow \mathrm{EE}', \mathrm{C}'}$$

$$\frac{}{\mathrm{C}, \pi \vdash \langle \mathsf{Var}, \mathrm{id}, _ \rangle \Rightarrow \emptyset, \mathrm{C}} \quad (\textsc{MatchVar1})$$

$$\frac{\mathrm{rpat} \neq _ \qquad \pi.vars(\mathrm{id}) = \mathrm{v} \qquad \vdash_p \mathrm{rpat}, \mathrm{v} \Rightarrow \mathrm{EE}}{\mathrm{C}, \pi \vdash \langle \mathsf{Var}, \mathrm{id}, \mathrm{rpat} \rangle \Rightarrow \mathrm{EE}, \mathrm{C}} \quad (\textsc{MatchVar2})$$

$$\frac{}{\mathrm{C}, \pi \vdash \langle \mathsf{In}, \mathrm{id}, _ \rangle \Rightarrow \emptyset, \mathrm{C}} \quad (\textsc{MatchInp1})$$

$$\frac{\begin{array}{c}\mathrm{rpat} \neq _ \qquad \pi.ins(\mathrm{id}) = \mathrm{k} \qquad \mathrm{C}(\mathrm{k}) = \langle \mathsf{true}, \beta, \mathrm{v} :: \mathrm{vs} \rangle \\ \mathrm{c}' = \langle |\mathrm{vs} > 0|, \beta, \mathrm{vs} \rangle \\ \vdash_p \mathrm{rpat}, \mathrm{v} \Rightarrow \mathrm{true}, \mathrm{EE}' \end{array}}{\mathrm{C}, \pi \vdash \langle \mathsf{In}, \mathrm{id}, \mathrm{rpat} \rangle \Rightarrow \mathrm{EE}', \mathrm{C}[\mathrm{k} \leftarrow \mathrm{c}']} \quad (\textsc{MatchInp2})$$

When a pattern is attached to an input channel, the value is consummed on this channel (except when the pattern is _).

Pattern matching

Pattern matching at the rule level is handled using the following rules, where $\vdash_p \mathrm{rpat}, \mathrm{v} \Rightarrow \mathsf{Bool}, \mathrm{EE}$, means that binding *rpat* to value v succeeds or not and results in a dynamic environment EE :

$$\boxed{\vdash_p \mathsf{RPat}, \mathrm{v} \Rightarrow \mathsf{Bool}, \mathrm{EE}}$$

$$\frac{}{\vdash \text{int/bool v'},\ \mathsf{Int/Bool}\ \mathrm{v} \Rightarrow \mathrm{v'} = \mathrm{v}, \varnothing}$$

$$\frac{}{\vdash \text{var v'},\ \mathrm{v} \Rightarrow \mathrm{true}, [\mathrm{v'} \mapsto \mathrm{v}]}$$

$$\frac{}{\vdash _,\ \mathrm{v} \Rightarrow \mathrm{true}, \varnothing}$$

$$\frac{\text{con} \neq \text{con}'}{\vdash \text{con}\langle \text{rpat}_1, \ldots, \text{rpat}_n \rangle, \text{con}'\langle \text{v}_1, \ldots, \text{v}_n \rangle \Rightarrow \text{false}, \varnothing}$$

$$\frac{\text{con} = \text{con}' \qquad \forall i.\ 1 \leq i \leq n, \quad \vdash_p \mathit{rpat}_i,\ \text{v}_i \Rightarrow \beta_i, \text{EE}_i}{\vdash \text{con}\langle \text{rpat}_1, \ldots, \text{rpat}_n \rangle, \text{con}'\langle \text{v}_1, \ldots, \text{v}_n \rangle \Rightarrow \beta_1 \ \wedge \ldots \wedge \ \beta_n,\ \bigoplus_{i=1}^{n} \text{EE}}$$

Evaluation of rule expressions

$$\boxed{\text{EE}, \text{C}, \pi \vdash \mathsf{QExps} \Rightarrow \text{C}', \pi'}$$

$$\frac{\text{C}_0 = \text{C} \qquad \pi_0 = \pi \qquad \forall i.\ 1 \leq i \leq n,\ \text{EE}, \text{C}_{i-1}, \pi_{i_1} \vdash \text{qexp}_i \Rightarrow \text{C}_i, \pi_i}{\text{EE}, \text{C}, \pi \vdash \langle \text{qexp}_1, \ldots, \text{qexp}_n \rangle \Rightarrow \text{C}_n, \pi_n} \quad (\text{EvalQExps})$$

$$\boxed{\text{EE}, \text{C}, \pi \vdash \mathsf{QExp} \Rightarrow \text{C}', \pi'}$$

$$\frac{\pi = \langle \ldots, \text{vars}, \ldots \rangle \qquad \text{EE} \vdash \exp \Rightarrow \text{v} \qquad \pi' = \langle \ldots, \text{vars}[\text{id} \leftarrow \text{v}], \ldots \rangle}{\text{EE}, \text{C}, \pi \vdash \langle \mathsf{Var}, \text{id}, \exp \rangle \Rightarrow \text{C}, \pi'} \quad (\text{UpdVar})$$

A variable update is directly reflected in the process local state.

$$\frac{}{\text{EE}, \text{C}, \pi \vdash \langle \mathsf{Out}, \text{id}, _ \rangle \Rightarrow \text{C}, \pi} \quad (\text{UpdOut1})$$

Nothing happens when an output is assigned the value _ (ignore).

$$\frac{\exp \neq _ \qquad \pi.\mathit{outs}(\text{id}) = \text{k} \qquad \text{C}(\text{k}) = \langle ., \mathsf{true}, \text{vs} \rangle \qquad \text{EE} \vdash \exp \Rightarrow \text{v} \qquad \text{c}' = \langle \mathsf{true}, \mathsf{true}, \text{vs} +\!\!+ [\text{v}] \rangle}{\text{EE}, \text{C}, \pi \vdash \langle \mathsf{Out}, \text{id}, \exp \rangle \Rightarrow \text{C}[\text{k} \leftarrow \text{c}'], \pi} \quad (\text{UpdOut 2})$$

Updating an output channel adds the value at the end of the channel buffer (FIFO behavior). The above semantics assumes undbounded FIFOs (the written channel is always ready for writting). Assuming FIFOs with a finite capacity κ would just require modifying the last premisse of the above rule into :

$$\text{c}' = \langle \mathsf{true}, |\text{vs} +\!\!+ [\text{v}]| < \kappa, \text{vs} +\!\!+ [\text{v}] \rangle$$

Chapter 8

Model of Computation

In this chapter, we describe a *model of computation* (MoC) which can be used to describe the behavior of CAPH programs. The goal is threefold.

First is to place CAPH on the "map" of dataflow-based formalisms and languages, which is generally organized in terms of associated MoCs (KPN, DPN, SDF, CSDF, ...), for the sake of comparison and communication.

Second is to give a formal description of the behavior of CAPH programs at a higher level than that provided by the structural operational semantics rules given in chapter 7.

Third is a to settle the basis for a mechanism allowing (when applicable) the static computation of FIFO sizes.

The CAPH MoC is a variant of the *Dataflow Process Network* model. We therefore start by recalling the main features of the DPN model, in Sec. 8.1. The CAPH MoC is presented in this context in Sec. 8.2. In Sec. 8.2.7 we discuss the application of this model to the third of the objectives listed above, namely the prediction of FIFO sizes.

8.1 Dataflow Process Networks

The *Dataflow Process Network* (DPN) model has been introduced and formalized by Lee and Parks in [4]. It can be viewed as a generalization of the *Kahn Process Network* model introduced by Kahn in [7]. The presentation of the DPN model in this section is directly inspired by that given in [4], with slight variations in the notations in order to ease the derivation of the CAPH MoC from it.

In the DPN model, a program is a collection of processes communicating through unidirectional, unbounded FIFO channels. Both read and write from/to these channels are unblocking[1]. Each FIFO channel carries a sequence of *tokens* and each process consists of repeated *firings* of a dataflow *actor*. Each firing can read (and possibly consumes) tokens on the channels connected to the input ports and can write (produce) tokens on the output ports of the corresponding actor.

In the DPN model, each token carries a **value** x taken from an uninterpreted **domain** D.

Channels contains (ordered, finite or infinite) **sequences** of tokens. A sequence will be denoted

$$X = [x_1, x_2, \ldots] \qquad \text{where} \quad x_i \in D \quad \forall i$$

[1] This is the main distinction with the KPN model, where write is unblocking but read is always blocking.

We will write [] for the empty sequence and denote $\sqsubseteq$ the classical **prefix** ordering relation between sequences. For example

$$[x_1, x_2] \quad \sqsubseteq \quad [x_1, x_2, x_3]$$

and

$$[] \quad \sqsubseteq \quad [x_1]$$

The behavior of an actor with m input ports and m' output ports is the defined by a set $\mathcal{R}$ of N **firing rules**

$$\mathcal{R} \quad = \quad \{R_1, R_2, \ldots, R_N\}$$

where each firing rule R_i is defined as a set of m **rule input patterns** and m' **rule output patterns**

$$R_i \quad = \quad \{R_{i,1}, \ldots, R_{i,m},\ R'_{i,1}, \ldots, R'_{i,m'}\}$$

A **rule input pattern** is a (finite, possibly empty) sequence of **input patterns**

$$R_{i,j} \quad = \quad [p_1, p_2, \ldots, p_n] \quad (j \in \{1, \ldots, m\},\ n \geq 0)$$

where a **input pattern** p is either

- a *constant value* $x \in D$,
- the *wildcard* pattern $*$.

A **rule output pattern** is a (finite, possibly empty) sequence of wildcard patterns[2]

$$R'_{i,j} \quad = \quad [*_1, *_2, \ldots, *_{n'}] \quad (j \in \{1, \ldots, m\},\ n' \geq 0)$$

The **context** of a process is a, ordered set composed of the input sequences $\{X_1, \ldots, X_m\}$ present on the m channels connected to the input of this actor.

A rule R_i is said to be **fireable** in a context $C = \{X_1, \ldots, X_m\}$ if and only if

$$\forall j = 1, \ldots, m \quad R_{i,j} \models_r X_j$$

[2]A rule output pattern then simply indicates *how many* of tokens produced on a given output port when the corresponding rule is fired, regardless of the actual values carried by these tokens.

where $\models_r$ is the relation indicating whether a given input sequence X_j "matches" the corresponding rule pattern $R_{i,j}$. This relation can be defined recursively as follows, using ":" as the sequence concatenation operator[3] :

$$\begin{array}{rcll} [] & \models_r & X & \forall X \\ p : ps & \models_r & x : xs & \text{iff} \quad p \models_p x \quad \text{and} \quad ps \models_r xs \end{array}$$

and where the **pattern matching relation** $\models_p$ is defined as follows :

$$\begin{array}{rcll} * & \models_p & x & \forall x \in D \\ x & \models_p & x' & \text{iff} \quad x = x' \end{array}$$

As an example, consider an actor with $m = 3$ inputs and $m' = 2$ outputs and the following firing rule

$$R_1 \quad = \quad \{[*, *], [], [1], \ [*], [*, *]\}$$

This rule will be fireable in any context $C = \{X_1, X_2, X_3\}$ in which

- X_1 contains at least two unconsummed tokens (regardless of their value),
- X_3 contains at least one token with value equals to 1,
- X_2 contains any sequence (possibly, but not necessarily, empty).

and, when fired, it produces one token on the first output port and two tokens on the second.

An actor is said to be **fireable** in a given context if at least one of its rules is fireable. If, in any case, *at most* one rule is fireable, the actor is said to be **deterministic**.

A classical example of a DPN actor is that described in Fig. 8.1. This actor reads a token carrying a boolean value on its third input and, depending on this value, copy the token present on its first or second input to its output. The behavior of the `select` actor can be described by a set of two firing rules $\{R_1, R_2\}$ where

$$\begin{array}{rcl} R_1 & = & \{[*], [], [\mathsf{true}], \ [*]\} \\ R_2 & = & \{[], [*], [\mathsf{false}], \ [*]\} \end{array}$$

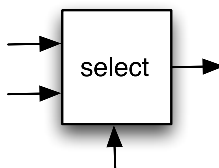


Figure 8.1: The `select` actor

The **signature** $|R_i|$ of a firing rule R_i gives the number of tokens consummed on each input channel and produced on each output channel when the rule is fired :

[3] $x_1 : [x_2, x_3, \ldots] = [x_1, x_2, x_3, \ldots]$.

$$|R_i| = \{|R_{i_1}|, \ldots, |R_{i,m}|,\ |R'_{i_1}|, \ldots, |R'_{i,m'}|\}$$

where the "length" of the rule pattern $R_{i,j}$ is defined as

$$\begin{aligned} |R_{i,j}| &= |[p_{i,j,1}, \ldots, p_{i,j,n}]| \\ &= n \end{aligned}$$

For example, for the `select` actor introduced above, we have

$$\begin{aligned} |R_1| &= \{1, 0, 1,\ 1\} \\ |R_2| &= \{0, 1, 1,\ 1\} \end{aligned}$$

The set of rule signatures can be used to refine the model of computation of an actor. For example, an actor which consumes (resp. produces) a fixed number of tokens on each input (resp. output) at each activation, *i.e.* an actor for which

$$|R_i| = \{k_1, \ldots, k_m,\ k'_1, \ldots, k'_{m'}\} \qquad \forall i = 1, \ldots, N$$

is categorized as belonging to **SDF** (Synchronous Dataflow) model of computation.

8.2 The Caph Process Network model

The model of computation for CAPH – let's call it CPN, for *Caph Process Network* – can be viewed as a variant of the DPN model summarized in the previous section. The variations concern the type of values carried by tokens on the one hand and the form of the patterns used to define the firing rules on the other hand.

8.2.1 Token values

The set of values carried by tokens is obtained by enriching the basic domain D used in the DPN model with two types of *structured* values :

- tuples of atomic values, denoted $(x_1, \ldots, x_n)$ (where $n \geq 1$ and x_i are atomic values),
- constructed values, of the form $\chi\ (x_1, \ldots, x_n)$ where $n \geq 1$, χ is a n-ary value constructor (taken from a predefined set of constructors Con) and the x_is are atomic values.

Formally, we therefore define the domain D of token values in the CPN model as

$$\begin{array}{lll} D & := & \mathsf{a} \quad \text{atomic value (int, bool, \ldots)} \\ & | & (\mathsf{a}_1, \ldots, \mathsf{a}_n) \quad \text{(tuples)} \\ & | & \chi\ (\mathsf{a}_1, \ldots, \mathsf{a}_n) \quad \text{where } \chi \in \mathsf{Con} \quad \text{(constructed values)} \end{array}$$

8.2.2 Rule patterns

In the CPN model, rule patterns cannot have a length ≥ 1 :

$$\begin{array}{lll} \forall i = 1, \ldots, N, & \forall j = 1, \ldots, m & |R_{i,j}| \in \{0, 1\} \\ & \forall j = 1, \ldots, m' & |R'_{i,j}| \in \{0, 1\} \end{array}$$

In other words, deciding whether a given rule is fireable can only involve the *first* token present of each input sequence and the activation of a rule can produce at most one token of each output. The main reason for this is that it allows hardware implementations to rely on a simple, fixed-interface FIFO model[4]. Moreover, our experience, based upon a large set of realistic applications, showed that this restriction does not in practice limit the expressiveness of the language.

8.2.3 Patterns

The set of individual patterns is augmented to accomodate the new definition of the domain D for values :

$$\begin{array}{lll} p & := & * \quad \text{(wildcard pattern)} \\ & | & c \in D \quad \text{(constant pattern)} \\ & | & v \in \mathsf{Var} \quad \text{(variable pattern)} \\ & | & (p_1, \ldots, p_n) \quad \text{(tuple pattern)} \\ & | & \chi\ (p_1, \ldots, p_n) \quad \text{(constructed pattern)} \end{array}$$

Variable patterns are used to *bind* values associated to tokens to names in order to use them in for constructing the outputs of a given rule.

8.2.4 Pattern matching

The pattern matching relation $\models_p$ defined in Sec. 8.1 is modified as follows to take account the previous modifications :

$$\begin{array}{rcll} * & \models_p & \mathsf{a} & \forall \mathsf{a} \\ c & \models_p & \mathsf{a} & \text{iff} \quad x = \mathsf{a} \\ v & \models_p & \mathsf{a} & \forall \mathsf{a} \\ (p_1, \ldots, p_n) & \models_p & (\mathsf{a}_1, \ldots, \mathsf{a}_n) & \text{iff} \quad p_i \models_p \mathsf{a}_i \quad \forall j = 1, \ldots, n \\ \chi'\ (p_1, \ldots, p_n) & \models_p & \chi\ (\mathsf{a}_1, \ldots, \mathsf{a}_n) & \text{iff} \quad \chi = \chi' \quad \text{and} \quad p_i \models_p \mathsf{a}_i \quad \forall j = 1, \ldots, n \end{array}$$

8.2.5 Example

Figure 8.2 shows an actor defined in CAPH and the three firing rules defining the behavior of this actor in the CPN model of computation.

8.2.6 Classification of CAPH actors

The rule signatures of an actor can be used to classify the behavior of this actor into three categories : SDF, CSDF and DDF.

[4]Allowing an actor to read – and *a fortiori* "pop" – $n > 1$ tokens from a FIFO requires a significantly more complex interface, especially if n cannot be computed statically.

```
type opt = None | Some of int;

actor foo
  in (i1: opt, i2: int)
  out (o:int)
rules
  | (i1: Some x) -> x
  | (i1: None, i2:0) -> 1
  | (i1: None, i2:y) -> y
;
```

$$
\begin{aligned}
R_1 &= \{[\mathsf{Some}\ x], [],\ [*]\} \\
R_2 &= \{[\mathsf{None}, [0],\ [*]\} \\
R_3 &= \{[\mathsf{None}, [y],\ [*]\}
\end{aligned}
$$

Figure 8.2: A CAPH actor and the associated firing rules in the corresponding model of computation

SDF Actors

SDF (Synchronous Data Flow) actor are those for which all firing rules have the same signature

$$|R_i| \quad = \quad \{\rho_1, \ldots, \rho_m,\ \rho'_1, \ldots, \rho'_{m'}\} \qquad \forall i = 1, \ldots, N$$

The CAPH model of computation imposes that $\rho_j \in \{0,1\}\ \forall j = 1, \ldots, m$ and $\rho'_j \in \{0,1\}\ \forall j = 1, \ldots, m'$. But for an SDF actor, having $\rho_j = 0$ (resp. $\rho'_j = 0$) would mean that this actor *never* consumes (resp. produces) a token on the jth input (resp. output) channel. There's no loss in expressivity in excluding this type of behaviors. As a result, SDF actors in CAPH are those for which all firing rules have a signature

$$|R_i| \quad = \quad \{1, \ldots, 1,\ 1, \ldots, 1\}$$

Listings 8.1 and 8.2 give two examples of actors which can classified as SDF. The `add` actor operates (here) on unstructured streams os signed integers whereas the `scale` actor operates on streams structured as lists using the `dc` type[5]. For both actors, each rule consumes and produces exactly one token when fired[6].

Listing 8.1: A simple SDF actor in CAPH

```
actor add
  in (i1:signed<s>, i:signed<s>)
  out (o:signed<s>)
rules
| (i1:x1,i2:x2) -> o:x1+x2          -- Rule R1 ({1,1, 1})
;
```

Listing 8.2: Another SDF actor

```
actor scale (k:unsigned<s>)
  in (i:unsigned<s> dc)
```

[5]The type `t dc` is a variant type. Values belonging to this type can be either the constant constructors `SoS` or `EoS` (*Start of Structure* or *End of Structure* or the constructed value `Data v`, where `v` has type `t`.

[6]The signature of the rules has been here indicated as a comment. In practice, it will be computed by the compiler.

```
  out (o:unsigned<s> dc)
rules
| i:SoS -> o:SoS                  -- Rule R1 ({1, 1})
| i:Data(v) -> o:Data(v*k)        -- Rule R2 ({1, 1})
| i:EoS -> o:EoS                  -- Rule R3 ({1, 1})
;
```

CSDF Actors

For CSDF (Cyclo Static Data Flow) actors, several firing rules, with distinct signatures, coexist but the order at which these rules are fired is statically predictable. The behavior of such an actor can generally be described as a finite state machine whose transitions depend solely on the value of local variables (and not on the value of tokens read on inputs).

An example of actor which can be classified as CSDF is given is Listing 8.3. The `switch` actor[7] reads tokens on its input channel and alternatively routes them to its first ("left") and second ("right") output. This is accomplished using the local variable `s`, which is alternately set to the (enumerated) values `Left` and `Right`. Given the input stream

$$x_1, x_2, x_3, x_4, x_5, x_6 \dots$$

it produces the following output streams on its first and second output

$$x_1, x_3, x_5, \dots$$
$$x_2, x_4, x_6, \dots$$

Listing 8.3: A simple CSDF actor in CAPH

```
actor switch
  in (i:$1)
  out (o1:$1, o2:$1)
  var s : {Left,Right} = Left
  rules
  | (s:Left, i:v) -> (o1:v, s:Right)  -- Rule R1
  | (s:Right, i:v) -> (o2:v, s:Left)  -- Rule R2
;
```

The signature of the first and second rule are respectively $|R_1| = \{1,\ 1, 0\}$ and $|R_2| = \{1,\ 0, 1\}$ but is should be clear from the definition of these rules that they fire alternatively and cyclically as follows

$$R_1, R_2, R_1, R_2, R_1, \dots$$

In fact, this property can be established formally using a technique known as *abstract interpretation.* Application of this technique to the classification of dataflow actors has been described for example in [12]. It can here prove that the only sequence of values that the local variable `s` can take is

`Left, Right, Left, Right, ...`

[7] This actor has already been described in Sec. 2.4.4.

which is sufficient to prove that the rules fires in the aforementionned order.

Another example of CSDF actor is given in Listing 8.4. The `sample` actor[8] performs $n \to 1$ subsampling. Given the input stream

$$x_1, x_2, x_3, x_4, x_5, x_6 \ldots$$

it produces the output stream

$$x_n, x_{2n}, x_{3n}, \ldots$$

Using abstract interpretation, it is still possible to classify the `sample` actor as CSDF, as soon as the value of its parameter n is known[9]. For example, Fig. 8.3 gives the FSM obtained by instanciating this actor with $n = 4$, from which it is possible to predict this cyclic sequence of consumption/production rates on `(i,o)` :

$$\{1,0\},\{1,0\},\{1,0\},\{1,1\},\ \{1,0\},\{1,0\},\{1,0\},\{1,1\},\ \ldots$$

Listing 8.4: A $n \to 1$ subsampling actor

```
actor sample (n: int)
  in (i: $t)
  out (o: $t)
var k : {1,..,n} = 1
rules
| i:x when k<n -> k:k+1
| i:x when k=n -> (k:1, o:x)
;
```

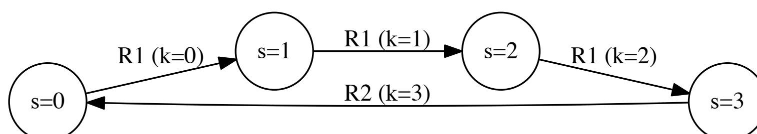


Figure 8.3: FSM for the `sample` actor with $n = 4$

DDF Actors

This category (Dynamic Data Flow) gathers all actors which cannot be classified as SDF or CSDF. For such actor it is not possible to statically predict the consumption/production rate on the I/O channels because this rate ultimately depends on the *value* of the tokens read by the actor.

A first example of DDF actor is given in listing 8.5. The `bswitch` actor is a variant of the `switch` actor described in the previous section. It routes the token read on its first input to either its first or second

[8] Which is taken from the CAPH standard library.

[9] This underlines a important distinction between the definition of an *actor*, in which the behavior may depend on formal *parameters* (such as `n` in the current example), and the actual *instances* of this actor in the program graph (*boxes* in the CAPH terminology), in which all parameters have been bound to values.

output depending on the (boolean) *value* of its second input. For example, given the following input streams on inputs `i1` and `i2` o respectively

$$0, 1, 2, 3, 4, 5, \ldots$$
$$true, false, false, true, false, true, \ldots$$

it will produce the output streams on the outputs `o1` and `o2`

$$0, 3, 5, \ldots$$
$$1, 2, 4, \ldots$$

Because the selection of fired rule depends on the *value* read on input `i2`, it is not possible to statically decide which one will be selected and hence predict the production rate on the outputs.

Listing 8.5: A simple DDF actor in CAPH

```
actor bswitch
  in (i1:$t, i2:bool)
  out (o1:$t, o2:$t)
  rules
  | (i1:x, i2:true) -> o1:x        -- Rule R1 ({1,1, 1,0})
  | (i1:x, i2:false) -> o2:x       -- Rule R2 ({1,1, 0,1})
;
```

A second, example of a DDF actor is described in listing 8.6. This actor[10] operates on a structured stream composed of a sequence of lists, each list starting with a `SoL` (*StartOfList*) token and ending with a `EoL` (*EndOfList*) token. For each list, it computes the sum of the elements. For example, if

$$\texttt{i} \;=\; \texttt{SoL},\ 1,\ 2,\ 3,\ \texttt{EoL},\ \texttt{SoL},\ 4,\ 5,\ \texttt{EoL},\ \ldots$$

then

$$\texttt{o} \;=\; 6,\ 21,\ \ldots$$

For this, it uses pattern matching on the input to detect the start and end of each list and two local variables : a local state (`state`), indicating whether the actor is waiting for a new list or computing the sum, and a accumulator (`sum`) for computing the sum. The three transition rules can be read as follows :

- if we are in state S0 and input token is "`SoL`", then initialize sum to 0 and go to state S1;
- if we are in state S1 and input token is a data, then add the corresponding value to the accumulator.
- if we are in state S1 and input token is "`EoL`", then writes the accumulated sum to output and go back to state S0;

This style of description, in which the *size* of the processed data structures is neither hard-coded nor provided using "external" parameters is actually very common in CAPH. It allows the formulation of actors being able to operate on data structures (lists, images, ...) of *any* size.

[10] Also previously described in Sec. 2.4.4.

Listing 8.6: Another DDF actor

```
type $t list =
  SoL
| EoL
| Data of $t;

actor suml
  in (i:signed<16> list)
  out (o:signed<16>)
var state : { S0, S1 } = S0
var sum : int
rules
| (state:S0, i:SoL)    -> (sum:0, state:S1)       -- Rule R1 ({1, 0})
| (state:S1, i:Data v) -> (sum:sum+v);            -- Rule R2 ({1, 0})
| (state:S1, i:EoL)    -> (o:sum, state:S0)       -- Rule R3 ({1, 1})
;
```

As for the `bswitch` actor, the exact sequence of rule firings cannot be predicted at compile time. In fact, and provided that the input stream is well structured[11], the sequence of firings can be described as

$$\underbrace{\{1,0\},\ \{1,0\},\ \ldots,\ \{1,0\}}_{\text{n activations}},\ \{1,1\}$$

where n depends on the actual length of the input lists and hence cannot be predicted.

Implementation

A prototype implementation of MoC-based actor classification has been implemented in the CAPH compiler since version 2.8.4.

Classification is obtained by invoking the compiler with the `-infer_mocs` option. Results are written in a file named `<prefix>_mocs.dat`, where `<prefix>` is the name of the application[12]. They can also be obtained with `-dump_boxes` option.

It is important to note that classification actually operates on *boxes* and not on the actors themselves. This is because this classification can depend on the actual value of actor parameters, which are only known when actors are instanciated as boxes in the final process network.

As an example, consider the program given in Fig. 8.4. In this program, the actors `switch`, `bswitch` and `sample` are those introduced in the previous sections and their code has not been reproduced. The `merge` actor reads one token on each of its inputs and alternatively writes that read on the first (resp. second) input to its output.

Classification is performed by invoking the compiler as follows (assuming that the program is contained in file `main.cph`, in directory `test`) :

```
caphc -infer_mocs main.cph
```

This gives the following output

[11] I.e. that it is effectively constituted of successive lists, otherwise, the actor simply blocks.
[12] As specified with the `-prefix` option or, by default, the current directory name.

```
actor switch ...

actor bswitch ...

actor sample (n: int) ...

actor merge
  in (i1: $t, i2:$t)
  out (o:$t)
var st : {Left,Right} = Left
rules
| (st:Left, i1:x, i2:y) -> (o:x, st:
    Right)
| (st:Right, i1:x, i2:y) -> (o:y, st:
    Left)
;

stream i1:signed<8> from "sample1.txt";
stream i2:bool from "sample2.txt";
stream o1:signed<8> to "result1.txt";
stream o2:signed<8> to "result2.txt";

net (s1,s2) = switch i1;
net (s3,s4) = bswitch (s2,i2);
net o1 = sample 2 s1;
net s5 = sample 4 s3;
net o2 = merge (s5,s4);
```

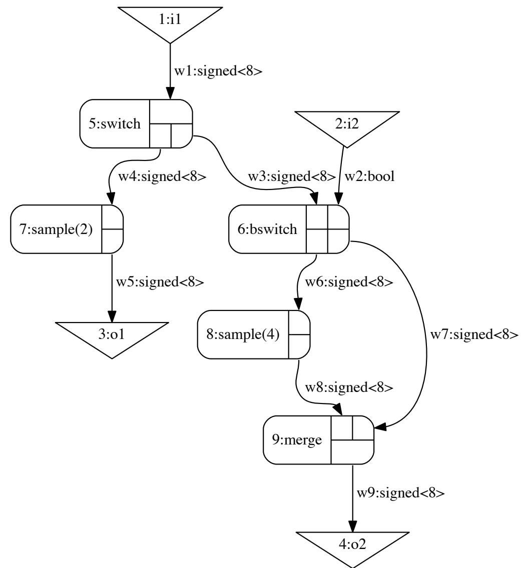


Figure 8.4: Example program for MoC-based actor classification

```
This is the Caph compiler, version 2.8.4
...
> Running abstract interpreter on box B5(switch) to infer Moc... Done.
> Running abstract interpreter on box B6(bswitch) to infer Moc... Done.
> Running abstract interpreter on box B7(sample) to infer Moc... Done.
> Running abstract interpreter on box B8(sample) to infer Moc... Done.
Wrote file ./test_mocs.dat
```

The contents of the generated `test_mocs.dat` file is given in Fig. 8.5. In this listing

- the first column gives the box identifier (as defined as a prefix number in the dataflow graph shown in Fig. 8.4),
- the second column gives the name of the instanciated actor,
- the third column gives the name of the infered MoC (`sdf`, `csdf` or `ddf`),
- the fourth column, in case of a `sdf` (resp. `csdf`) model, gives the box signature (resp. cyclic sequence of signatures).

This result shows that the compiler has correctly infered the model for each box. Note, in particular, the difference in the infered sequence of signatures for the two instances of the `sample` actor (boxes `B7` and `B8`), due to the dependance of the activation sequence on the value of the parameter `n` of the actor.

```
B5; switch; csdf; [1,1,0],[1,0,1]
B6; bswitch; ddf
B7; sample; csdf; [1,0],[1,1]
B8; sample; csdf; [1,0],[1,0],[1,0],[1,1]
B9; merge; sdf; 1,1,1
```

Figure 8.5: Result of MoC-based box classification for the program of Fig. 8.4

As evidenced by the compiler output reproduced above, classification was here obtained using *abstract interpretation*.

The abstract interpreter used for MoC-based classification is based upon a modified version of the dynamic semantics described in chapter 7, in which a special value (typically noted $\perp$) is used to represent values which cannot be statically evaluated (such the values carried by input tokens for instance). The interpreter iteratively fires the analysed box until one of the following conditions is met :

1. an execution cycle is detected (in other words, the value of all known local variables is the same as it was at the first activation),
2. the number of firings exceeds a given predefined limit.

The first condition is the one occuring for SDF for CSDF actors. The second condition is used to detect DDF actors (with the assumption that CSDF actors generally have "short" periods and, hence, that a sufficiently large value for the limit will be discriminant).

It is possible to trace the execution of the abstract interpreter by inkoking the compiler with the `-absint` option with the identifier of the target box as argument[13]. For our example, invoking the compiler as follows

```
caphc -infer_mocs -absint 8 main.cph
```

gives the following output

```
This is the Caph compiler, version 2.8.4
...
> abstract interpretation of box B8(sample) ...
> cycle detected at t=4:
>   [k=1] i:x, k<n -> k:k+1 [1,0]
>   [k=2] i:x, k<n -> k:k+1 [1,0]
>   [k=3] i:x, k<n -> k:k+1 [1,0]
>   [k=4] i:x, k=n -> k:1, o:x [1,1]
> done
```

This trace shows how the classification of box `B8` as CSDF here results from the detection of a cycle (of length 4) in the sequence of rule activations.

8.2.7 Static computation of FIFO sizes

A potential application of actor classification can be the static computation of FIFO sizes.

Predicting the size (depth) of the FIFOs implementing the channels connecting actors is important because an underestimation of these sizes may lead to runtime failure by overflow. When generating VHDL code, overestimations also lead to a waste of hardware resources.

[13] A complementary option, `ai_max_cycles`, serves to adjust the limit used to detect DDF actors.

As described in Sec. 10.9, this prediction is typically carried out using runtime monitoring of the code generated by the SystemC backend. This approach may be the only applicable for some programs involving actors with DDF behaviors. It is intuitively clear that the size of the required FIFOs depends on the consumption and production rates of connected actors. If these rates depend on the *values* carried by tokens, so do the size of the FIFOs.

But not all programs involve DDF actors. For programs involving only SDF actors – and, to a certain extent[14], CSDF actors –, it is possible to accurately predict these sizes on the basis of a purely static analysis.

This is described, for SDF graphs, in the next section.

SDF graphs

It is well known (see for example [9]) that for "pure" acyclic SDF graphs – *i.e.* graphs for which all actors can be classified as SDF and exhibiting to feedback loops – the size of the buffers implementing channels can be statically computed. The basic technique relies on solving the so-called *balance equations*, relating production and consumption rates on extremities of all edges.

In the case of CAPH programs, for which all production and consumption rates are equal to 1, these sizes can be computed in a more straigthforward way using the algorithm described below. The algorithm basically works by propagating *phases* along all edges of the dataflow graph, where the phase (ϕ) measures the propagation time (counted in execution cycles) of the input token(s). For a box to fire, each input edge must have the same phase. Otherwise, a FIFO must be inserted on the "late" edge(s). The required FIFO capacity (*cap*) can be computed from the difference of phases. Moreover, the phase of the output edge(s) can be simply computed by adding a constant (representing the actor delay, in execution cycles) to the latest input phase.

Implementation

A prototype implementation of the mechanism described in the previous section is available in version 2.8.4 of the compiler. This implementation generates an annotation file similar to that produced by the SystemC backend used with the `sc_dump_fifo_stats` (as described in Sec. 10.9 of the manual). This file can passed directly as argument to the `vhdl_annot_file` option when invoking the VHDL backend in order to specify the size of each hardware FIFO.

The mechanism is invoked by passing the `-dump_sdf_fifo_sizes` option to the compiler. Of course, a necessary condition is that all boxes can be classified as SDF[15].

As an example consider the program given in Listing 8.2.7. The actual computations performed by actors is irrelevant here, only their SDF signature is.

```
actor foo
  in (i:signed<8>)
  out (o:signed<8>)
rules
|  i:x -> o:x
;

actor bar
  in (i1:signed<8>, i2:signed<8>)
  out (o:signed<8>)
rules
```

[14] Which remains to be investigated...

[15] Classification is performed as a preliminary step without the need to explicitely pass the `-infer_mocs` option.

Algorithm 1 Compute FIFO sizes of DFG G

Require: Each node n of G has been classified as SDF
Ensure: Each edge e of G is assigned a phase ϕ and a required FIFO capacity *cap*

1: **for each** $n \in SourceNodes(G)$ **do**
2:     **for each** $e \in v.outps$ **do**
3:         $e.\phi \leftarrow 0$
4:     **end for**
5: **end for**
6: $N \leftarrow TopologicalSort(G)$
7: **while** Nnot empty **do**
8:     $n \leftarrow ExtractFirst(N)$
9:     $\phi_m \leftarrow max_{e \in n.ins}(e.\phi)$
10:     **for each** $e \in n.ins$ **do**
11:         **if** $e.\phi < \phi_m$ **then**
12:             $e.cap \leftarrow 1 + \phi_m - e.\phi$
13:         **else**
14:             $e.cap \leftarrow 1$
15:         **end if**
16:     **end for**
17:     **for each** $e \in n.outs$ **do**
18:         $e.\phi \leftarrow \phi_m + n.delay$
19:     **end for**
20: **end while**

```
|   (i1:x1, i2:x2)-> o:x1+x2
;

actor zib
  in (i:signed<8>)
  out (o1:signed<8>, o2:signed<8>)
rules
|   i:x-> (o1:x, o2:x)
;

stream inp:signed<8> from "sample.txt";
stream outp:signed<8> to "result.txt";

net (x2,x3) = zib inp;
net outp = bar (foo inp, bar (foo x2, foo (foo x3)));
```

Computing the FIFO sizes is performed by invoking the compiler as follows (assuming that the program is contained in file `main.cph`, in directory `test`) :

```
caphc -dot -dump_sdf_fifo_sizes main.cph
```

This gives the following output

```
This is the Caph compiler, version 2.8.4
...
Wrote file ./test_mocs.dat
Wrote file ./test_sdf_fifo_sizes.dat
Wrote file ./test.dot
```

The contents of the generated `test_sdf_fifo_sizes.dat` file is given in Fig. 8.6. These results can also visualized graphically by invoking the compiler with the `-dot` as `-dot_wire_annot` options, producing the graph given in Fig. 8.7. In this graph, each edge has been annotated with two integers p/s, where p is the phase and s the required FIFO capacity.

```
w1 fifo_size = 1
w9 fifo_size = 4
w8 fifo_size = 1
w7 fifo_size = 1
w6 fifo_size = 2
w5 fifo_size = 1
w4 fifo_size = 1
w3 fifo_size = 1
w2 fifo_size = 1
w10 fifo_size = 1
```

Figure 8.6: Result of static computation of FIFO sizes for the program of Fig. 8.2.7

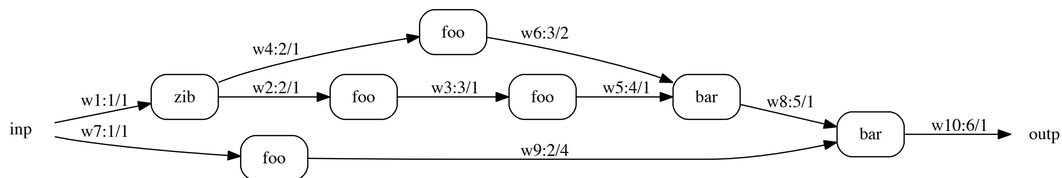


Figure 8.7: The dataflow graph generated by the CAPH compiler from the program in Listing. 8.2.7, with edges annotated with the infered phase and FIFO size information

Chapter 9

Intermediate representation

This chapter briefly describes the target-independant intermediate representation (IR) used as an input for the back-ends.

The intermediate representation (IR) is basically a process network in which each process is represented as a finite-state machine (FSM) and channels as unbounded FIFOs.

Generation of this intermediate representation involves two steps : first generating a *structural* representation of the actor network and then generating a *behavioral* description of each actor involved in the network.

9.1 Network generation

Generating the structural representation of the actor network consists in instantiating each actor – viewed as a black box at this level – and "wiring" the resulting instances according to the dependencies expressed by the functional definitions. This process is fully formalized by the static semantics described in Chap. 6. The resulting network is set of *boxes* interconnected by *wires*. Boxes result from the instanciation of actors and wires from the data dependencies expressed in the definition section.

9.2 Behavioral description

Generating the behavioral description of an instantiated actor (box) consists in turning the set of pattern-matching rules of the corresponding actor into a finite state machine with operations (FSMD level). This process is depicted in Fig. 9.1.

At each rule r_i, consisting of a list of patterns $pats_i$ and a list of expressions $exps_i$, we associate a set of *conditions* $C[\![r_i]\!]$ and a set of actions $A[\![r_i]\!]$. The set $C[\![r_i]\!]$ denotes the firing conditions for rule r_i, i.e. the conditions on the involved inputs, outputs and local variables that must be verified for the corresponding rule to be selected. The set $A[\![r_i]\!]$ denotes the firing actions for rule r_i, i.e. the read operations and write operations that must be performed on the involved inputs, outputs and variables when the corresponding rule is selected.

There are four possible firing conditions:

- $Avail_r(i)$, meaning that input i is ready for reading (the connected FIFO is not empty),
- $Match_i(i, pat)$ (resp.$Match_v(v, pat)$), meaning that input i (resp. variable v) matches pattern pat,
- $Avail_w(o)$, meaning that output o is ready for writing (the connected FIFO is not full),

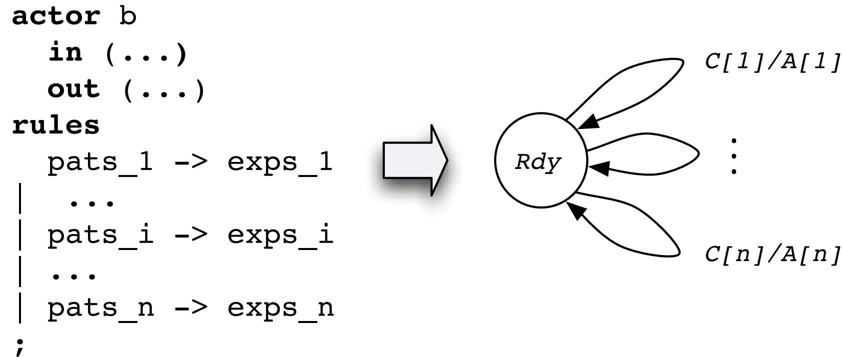


Figure 9.1: Translation of a box into a FSM

- $Cond(exp)$, meaning that guard expression exp (involving inputs and/or variables) is true

and four possible firing actions:

- $Read(i)$, meaning "read input i (pop the corresponding from the connected FIFO)", ignoring the read value,
- $Bind_i(i, pat)$, meaning "read input i (pop the corresponding from the connected FIFO) and match the corresponding value against pattern pat", binding the variable(s) occurring in the pattern to the corresponding value(s),
- $Bind_v(v, pat)$, meaning "match variable v against pattern pat",
- $Write_o(o, exp, \rho)$ (resp. $Write_v(v, exp, \rho)$), meaning "evaluate expression exp, using environment ρ and write the resulting value on output o (pushing the value on the connected FIFO) or in variable v.

Fig 9.3 summarizes the rules for computing the sets $C[\![r]\!]$ and $A[\![r]\!]$ from the patterns and expressions composing a rule. In these rules _ denotes the "empty" pattern (resp. expression), *const* a constant pattern and *var* a variable pattern, vars(pat) is a function returning the name of all variables bound by a pattern and ρ_0 is the "default" environment for evaluating a RHS expression, containing of the values of the global and local variables.

To illustrate the generation of the intermediate representation, let's take again the `suml` actor introduced in Sec. 2.4.4. The code of this actor is given again below :

```
actor suml
  in (i:signed<16> dc)
  out (o:signed<16>)
```

```
var st : {S0,S1} = S0
var sum : int
rules
  | (st:S0, i:'<) -> (sum:0, st:S1)
  | (st:S1, i:'>) -> (o:sum, st:S0)
  | (st:S1, i:'v) -> (sum:sum+v);
```

The corresponding intermediate representation is given in Fig. 9.2. The small number appearing beside each transition is the index of the corresponding rule.

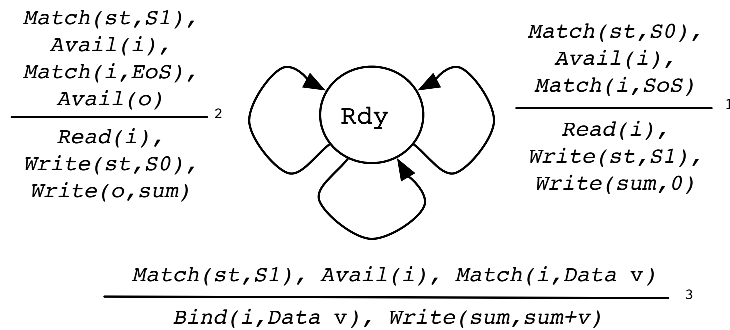


Figure 9.2: Intermediaire representation for the suml actor

$$\frac{C_r[\![qpat_1,\ldots,qpat_m]\!] = C \qquad C_g[\![gexp_1,\ldots,gexp_p]\!] = C'' \qquad C_w[\![qexp_1,\ldots,qexp_n]\!] = C'}{C[\![qpat_1,\ldots,qpat_m \quad \texttt{when} \quad gexp_1,\ldots,gexp_p \quad \rightarrow \quad qexp_1,\ldots,qexp_n]\!] = C \cup C'' \cup C'}$$

$$\frac{A_r[\![qpat_1,\ldots,qpat_m]\!] = C, \rho \qquad \rho_0 \oplus \rho \vdash A_w[\![qexp_1,\ldots,qexp_n]\!] = A'}{A[\![qpat_1,\ldots,qpat_m \rightarrow qexp_1,\ldots,qexp_n]\!] = A \cup A'}$$

$$\frac{\forall i.\ 1 \leq i \leq m, \quad A_r[\![qpat_i]\!] = A_i, \rho_i}{A_r[\![qpat_1,\ldots,qpat_m]\!] = \bigcup_{i=1}^{m} A_i, \bigoplus_{i=1}^{n} \rho_i}$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \rho \vdash A_w[\![qexp_i]\!] = A_i}{\rho \vdash A_w[\![qexp_1,\ldots,qexp_n]\!] = \bigcup_{i=1}^{n} A_i}$$

$$\begin{aligned}
C_r[\![qpat_1,\ldots,qpat_m]\!] &= \textstyle\bigcup_{j=1}^{m} C'_r[\![qpat_j]\!] \\
C_g[\![gexp_1,\ldots,gexp_p]\!] &= \textstyle\bigcup_{j=1}^{p} C''_r[\![gexp_j]\!] \\
C_w[\![qexp_1,\ldots,qexp_n]\!] &= \textstyle\bigcup_{j=1}^{n} C'_w[\![qexp_j]\!]
\end{aligned}$$

$$\begin{aligned}
C'_r[\![\mathsf{In}, i, _]\!] &= \emptyset \\
C'_r[\![\mathsf{In}, i, pat]\!] &= \{Avail_r(i), Match_i(i, pat)\} \\
C'_r[\![\mathsf{Var}, v, _]\!] &= \emptyset \\
C'_r[\![\mathsf{Var}, v, pat]\!] &= \{Match_v(v, pat)\}
\end{aligned}$$

$$\begin{aligned}
C'_w[\![\mathsf{Out}, o, _]\!] &= \emptyset \\
C'_w[\![\mathsf{Out}, o, exp]\!] &= \{Avail_w(o)\} \\
C'_w[\![\mathsf{Var}, v, exp]\!] &= \emptyset
\end{aligned}$$

$$C''_r[\![exp]\!] = \{Cond(exp)\}$$

$$\begin{aligned}
A'_r[\![\mathsf{In}, i, _]\!] &= \emptyset, \emptyset \\
A'_r[\![\mathsf{In}, i, const]\!] &= \{Read(i)\}, \emptyset \\
A'_r[\![\mathsf{In}, i, pat]\!] &= \{Bind_i(i, pat)\}, \mathrm{vars}(pat) \\
A'_r[\![\mathsf{Var}, v, _]\!] &= \emptyset, \emptyset \\
A'_r[\![\mathsf{Var}, v, const]\!] &= \emptyset, \emptyset \\
A'_r[\![\mathsf{Var}, v, var\ v]\!] &= \emptyset, \emptyset \\
A'_r[\![\mathsf{Var}, v, pat]\!] &= \{Bind_v(v, pat)\}, \mathrm{vars}(pat)
\end{aligned}$$

$$\begin{aligned}
\rho \vdash A'_w[\![\mathsf{Out}, o, _]\!] &= \emptyset \\
\rho \vdash A'_w[\![\mathsf{Out}, o, exp]\!] &= \{Write_o(o, exp, \rho)\} \\
\rho \vdash A'_w[\![\mathsf{Var}, v, _]\!] &= \emptyset \\
\rho \vdash A'_w[\![\mathsf{Var}, v, var\ v]\!] &= \emptyset \\
\rho \vdash A'_w[\![\mathsf{Var}, v, exp]\!] &= \{Write_v(v, exp, \rho)\}
\end{aligned}$$

Figure 9.3: Rules for computing the C and A sets for actor rules

Chapter 10

Using the caph compiler

The CAPH compiler can be used to

- obtain graphical (`.dot`) representations of program,
- simulate programs or
- generate SystemC or VHDL code.

This chapter describes how to invoke compiler on the command-line (on Unix systems). A separate document describes the graphical IDE (running under MacOS and Windows platforms).

The compiler is invoked with a command like :

```
caphc [options] file
```

where `file` is the name of the file containing the source code (by convention, this file should be suffixed `.cph`).

The complete set of options is described in App. 13.

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

10.1 Generating a graphical representation of the program

Example :

```
caphc -dot foo.cph
```

The previous command generates a graphical representation of the program contained in file `foo.cph` and writes it in file `foo.dot`. This representation can be viewed with the `Graphviz` suite of tools[1].

10.2 Running the simulator

Example :

```
caphc -sim -stop_after 200 foo.cph
```

The previous command runs the program contained in file `foo.cph` for 200 execution cycles.

[1] Available freely from `http://www.graphviz.org`.

10.3 Generating SystemC code

Example :

```
caphc -systemc foo.cph
```

The previous command generates the SystemC code corresponding the program contained in file `foo.cph`. The following files are written :

- a file `foo_expanded.dot`, containing a modified version of the program graphical description, in which explicit flow-splitting boxes have been inserted,
- a file `foo_net.cpp`, containing the network description,
- a pair of files `x_act.h`, `x_act.cpp` for each instance[2] appearing in the program, containing respectively the interface and the implementation of the actor,
- if some global constants and/or functions are defined, a pair of files `foo_globals.h`, `foo_globals.cpp`, containing the C++ prototypes (resp. definitions) of the corresponding values. These files also contains the interface (resp. implementation) of the user defined variant types when such types have been declared in the program (see Sec. cha:variants-impl),
- if needed, a file `foo_splitters.h` containing the interface and implementation of actors for performing $1 \rightarrow n$ flow replication.

The produced files can then compiled using the standard SystemC toolchain. When compiling (resp. linking) the CAPH-specific headers (resp. library) must be available[3]. The easiest way to compile the generated code is to use the `-make` option of the compiler and to rely on the predefined skeleton makefile `$(CAPH)/lib/etc/Makefile.core` (see Sec. 10.11.

10.4 Generating VHDL code

Example :

```
caphc -vhdl foo.cph
```

The previous command generates the VHDL code corresponding the program contained in file `foo.cph`. The following files are written :

- a file `foo_expanded.dot`, containing a modified version of the program graphical description, in which FIFO buffers and explicit flow-splitting boxes have been inserted,
- a file `foo_net.vhd`, containing the (structural) network description,
- a file `xxx_act.vhd` for each instance of actor `xxx` appearing in the program, containing the interface and implementation of the actor,
- if some global constants and/or functions are defined, a pair of files `foo_globals.vhd`, containing the VHDL description of the corresponding values,

[2]If the actor is monomorphic, there will a single instance; otherwise there will be as many instances as distinct type and size specialisations of this actor. Each instance goes in a distinct pair of files. A name mangling mechanism is used to distinguish between files.

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

- if the program declares some variant types, a file `foo_types.vhd` containing the description of the VHDL package implementing the corresponding types[4],
- if needed, a file `foo_splitters.vhd` containing the interface and implementation of actors for performing $1 \rightarrow n$ flow replication,
- a file `foo_tb.vhd`, containing a *test bench* for simulating the resulting design.

The produced files can then compiled, simulated and synthetized using a standard VHDL toolchain[5]. When compiling the CAPH-specific `dc` library, containing the `dcflow` package, must be available[6]. As with the SystemC backend, the easiest way to compile the generated code is to use the `-make` option of the compiler and to rely on the predefined skeleton makefile `$(CAPH)/lib/etc/Makefile.core` (see Sec. 10.11.

10.5 File I/O

When performing simulations of either using the reference interpreter or the generated SystemC code input data streams and output streams are read from (resp. written to) text files.

For example, if the program contains the following lines :

```
stream inp : signed<10> dc from "sample.txt";
stream res : unsigned<1> dc to "result.txt";
```

then the input stream will be read from file `sample.txt` and the output stream written to file `result.txt`[7].

For streams containing values with a scalar type (see Sec. 2.2.1), these files will simply contain the sequence of input (resp.output) tokens, separated by white space(s). For example, here's the contents of a file containing eight tokens of type `unsigned<8>` :

```
12 34 67 6 99 0 0 55
```

For streams containing values with an array type (see Sec. 2.2.2), each array will be denoted as a comma-separated list of values and enclosed between braces. For example, a file describing a stream of four arrays of size 4 is :

```
{ 1,2,3,4 } { 10, 20, 30, 40 } { 100, 200, 300, 400 } { 1000, 2000, 3000, 4000 }
```

For streams containing values with a variant type (see Sec. 2.2.2), each token will denote, textually, either a nullary constructor or a n-ary constructor, followed by the associated value(s).

For example, a file describing a stream of values with type `unsigned<8> option` (as introduced in Sec. 2.4.1), is :

```
Absent Present 2 Present 4 Absent Absent Present 8
```

and a file describing a stream of values with type `(signed<8>,bool) pair` is :

```
Pair 1 true Pair 8 false Pair 0 100
```

[4]In case of polymorphic variants, there's one package per distinct monomorphic instanciation of the declared type.

[5]We use the Quartus II toolchain from Altera.

[6]This library, and the corresponding package(s) and library are located in `$CAPH/lib/vhdl` where `$CAPH` points to the installation directory of the CAPH toolset.

[7]In the directory from which the CAPH compiler is invoked. Absolute and relative pathnames are also possible.

In particular, input or input files representing streams structured with the `t dc` type can be denoted using the `Data`, `EoS` and `SoS` constructors. Listing 10.1, for example, gives the contents of a file describing a 3×3 image.

Listing 10.1: A text input file describing a 3x3 image

```
SoS SoS Data 10 Data 30 Data 55 EoS SoS Data 53 Data 60 Data 12 EoS SoS Data 56 Data 23
     Data 11 EoS EoS
```

Because denoting images this way quickly becomes very verbose, the interpreter and the SystemC generated code also accept (resp. produce) input (resp. output) files written with an abbreviated syntax, in which the `SoS`, `EoS` and `Data v` token are respectively written as `<`, `>` and `v`. For this, the simulator and the SystemC backend must be invoked with the `-abbrev_dc_ctors` option. With this option, the file listed in listing. 10.1 can be replaced by that of listing 10.2.

Listing 10.2: A text input file describing a 3x3 image (abbreviated syntax)

```
< < 10 30 55 > < 53 60 12 > < 56 23 11 > >
```

10.5.1 Port I/O

The previous sections dealt with *stream*-based I/O. A similar mechanism is provided to simulate programs using *port*-based I/O.

For **input ports**, if a file is specified this file is expected to contain a list of *events*, where a *event* consists in a date and value. The specified port will then hold its initial value until the current simulation time reaches the date(s) specified in the file; at this time the value of the port is updated with the specified value and this continues for each *event* of the input file (or the simulation stops).

Because the notion of "simulation" time depends on the nature of the simulation (using the interpreter, the code generated by the SystemC backend or the code generated by the VHDL backend), the date of each event in the event file is actually specified using three distinct values, respectively representing a simulation cycle number (for the interpreter), a SC_TIME value, in ns (for the SystemC code) and a simulated clock time, also in ns (for the VHDL code). Listing 10.3 is an example of event file, containing two events. The first event is scheduled at cycle 60 (when using the reference interpreter) and at $t = 200ns$ when using the SystemC and VHDL backends, and the second at cycle 60 and $t = 600ns$[8].

Listing 10.3: A sample event file, to be attached to a input port declaration

```
# sim_cycle sc_time vhdl_time value
        40      200       200    10
        60      600       600    20
```

For **output ports**, the file specified in the declaration will simply contain, at the end of the simulation, a list of events which occured on the corresponding port, where an event is here defined as a pair of a value and the date (simulation cycle/time) when this value was written on the port.

10.5.2 File globbing

A minimal support for file globbing is offered in the current version. For example, specifying, in the CAPH source file

```
stream inp : signed<10> dc from "im[1-3].pgm";
```

[8]Unfortunately, there's no simple relation between the first number and the two others; these two others should be the same, provided the period of the clock used in the respective testbenches are equal.

will cause input streams to be read successively from files `im1.pgm`, `im2.tx` and `im2.pgm`. There can be only one globing pattern in a file specification and the only accepted pattern[9] is a *range pattern*, *i.e.* a pattern having form `[n1-n2]`, where `n1` and `n2` are integers[10].

Patterns can also be used in output file specifications but only in conjonction with the `-split_output_frames` compiler option. In this case, when the output stream is composed of tokens having type `t dc` and contains a sequence of images, the interpreter (resp. the SystemC and VHDL testbench) will write each successive image in a separate file, the name of these files being given by expanding the file pattern. For example, writing

```
stream res : unsigned<1> dc to "results/result[1-3].pgm";
```

in the CAPH source file will cause the images produced on output `res` to be written in files `results/result1.pgm`, `results/result2.pgm` and `results/result3.pgm`. The behavior is undefined if the output stream is not of type `t dc`, does not encode an image or if the number of successive images does not match the number of files after pattern expansion.

10.5.3 File IO when using the VHDL backend

Because of the limitations of file IO library in VHDL, direct reading and writting of structured text files is not supported. When running simulations using the VHDL testbench generated by the CAPH compiler, text files must be converted to and from a special custom format. Files having this format have the `.bin` extension[11]. Two utility programs, `txt2bin` and `bin2txt` are provided in the CAPH distribution for converting between `.txt` and `.bin` files[12].

For example, the command to convert the file `sample1.txt`, containing the representation of an stream of type `signed<16>`, is[13] :

```
txtbin -out sample1.bin sint 16 sample1.txt
```

and the command to convert the file `sample2.txt`, containing the representation of structured stream of type `unsigned<8> dc`, is :

```
txtbin -dc -abbrev -out sample2.bin uint 8 sample2.2txt
```

The `txt2bin` and `bin2txt` also supports the generation (resp. interpretation) of *event files* attached to input (resp. output) ports, using the `-eventf` option. Note that only ports carrying *scalar* values can be programmed this way.

A complete description of the `txt2bin` and `bin2txt` commands can be found in the appendices.

Note. Since version 2.8.1, and when using the `caphmake` utility described in Sec. 10.11, calls to `txt2bin` and `bin2txt` utilities are automatically inserted in the generated `Makefile.vhdl` file.

[9] In the current version.

[10] In particular, *wildcard* patterns `*` and `?`, accepted in previous versions of the compiler, are no longer supported because they cannot be used for specifying *output* filesets.

[11] These files are actually text files containing a sequence of `words`, one word per line, where each word is the ASCII representation of a binary word.

[12] These programs are written in C and have to be compiled on the target platform.

[13] Provided the corresponding command is in the current execution path.

10.5.4 Reading and writting image files

The CAPH distribution also includes four utility programs for converting image files, in the PGM format, to and from `.txt` and `.bin` files. This makes it possible to process images and view the results using external image viewers[14].

For example, if a CAPH source program, `foo.cph`, contains the following lines

```
stream inp : unsigned<8> dc from "im1.txt";
stream res : unsigned<8> dc to "im2.txt";
```

then the complete list of commands for using it to process, using the reference interpreter, an image stored in file `im1.pgm`, producing an image in file `im2.pgm` will be

```
pgm2txt -abbrev im1.pgm im1.txt
caphc -sim -abbrev_dc_ctors foo.cph
txt2pgm -abbrev 256 im2.txt im2.pgm
```

In this example, using the `-abbrev` option for both programs is not mandatory; this simply reduces the size of the generated text files. The first non optional argument to the `txt2pgm` command (256 in this case) gives the maximum pixel value of the corresponding image.

A pair of similar programs is provided to convert `.pgm` files to and from `.bin` for running VHDL simulations.

A complete description of the `txt2pgm`, `pgm2txt`, `bin2pgm` and `pgm2bin` commands can be found in the appendices.

Note. Since version 2.8.1, and when using the `caphmake` utility described in Sec. 10.11, calls to `txt2pgm`, `pgm2txt`, `bin2pgm` and `pgm2bin` utilities are automatically inserted in the generated target specific Makefiles.

10.5.5 Blanking

By default, when reading values from a file, the read tokens are put in the input stream at a fixed period. When using the reference interpreter, one token is injected at each execution cycle[15]. When running the code generated by the SystemC or VHDL backend, one token is injected every N clock cycle(s), where N is set to 1 by default and can be adjusted using the `-sc_istream_period` (resp. `-vhdl_istream_period`).

But in certain situations, such a fixed input rate does not model accurately enough the behavior of the application on the hardware target platform. When processing streams coming from a digital camera, in particular, a certain number of clock cycles without pixels are frequently inserted between the end of a line and the start of the next line and between the end of a frame (image) and the next one. This is usually called *blanking* ("horizontal" and "vertical" respectively). Ideally, the presence or the absence of blank clock cycles should be transparent. But some dataflow actors may actually rely on the presence of these cycles to operate correctly[16]. For these actors the behavior observed at simulation leval (either SystemC or VHDL) matches the behavior observed after synthesis on the target platform only if blanking cycles are generated by the testbench.

When using the SystemC backend, insertion of such blanking cycles in the testbench code can be requested by invoking the compiler with the `-sc_istream_hblank` and `-sc_istream_vblank` options.

[14] Such as `xv` or `xloadimage` under linux.

[15] The notion of execution cycle is defined in Sec. 7.3.

[16] This is the case, in particular of certain actors using external FIFOs to perform pixel or line delays, for which the blank cycles are used to flush the FIFOs, such as the `cconvXXX` actors defined in the `convol.cph` standard library.

For example, the command for generating the code processing a sequence of images, inserting 8 blank cycles between lines of 32 blank cycles between images will be :

```
caphc -systemc -sc_stop_time 2000 -sc_istream_hblank 8 -sc_istream_vblank
   32 -split_output_frames appli.cph
```

When using the VHDL backend, blanking is carried out by

- inserting special tokens in the `.bin` file generated by the `txt2bin` utility (using the `-hblank` and `-vblank` options of this command),
- passing the `-vhdl_istream_blanking` options when invoking the CAPH compiler.

For example, to process a sequence of images stored in files `im1.txt` to `im16.txt`, inserting 8 blank cycles between lines of 32 blank cycles between images, the following commands can be issued :

```
txt2bin -dc -abbrev -hblank 8 -vblank 32 -out im.bin uint 8 im[1-16].txt
caphc -vhdl -sc_stop_time 8000 -vhdl_istream_blanking appli.cph
bin2txt -dc -abbrev -split_output_frames -out result.txt uint 8 result.bin
```

Both in the SystemC and VHDL case, blanking only makes sense when dealing with input streams representing images encoded using the `t dc` type. The results is otherwise undefined.

10.6 File inclusion

The compiler implements a simple mechanism for file inclusion. When compiling a CAPH source file, the directive

```
#include "file.cph"
```

will cause the contents of the file `file.cph` to be included textually and compiled. Filenames can be relative or absolute. In the former case, they are searched relatively to the working directory (the directory from which the CAPH compiler is invoked). The search path can be extended with the `-I` option. A typical usage is pass the `-I $(CAPHLIB)` option to the compiler, where `CAPHLIB` is the location of the standard libraries.

Several `#include` directives can be issued in the same file and they can also be nested (a file included this way can itself contains an `#include` directive).

The mechanism works exactly like the `#include` directive for C compilers[17]. It therefore offers no protection against symbol redefinition.

10.7 Passing command-line options to programs

There's a rudimentary macro mechanism for passing command-line arguments to programs. As for file inclusion, this mechanism imitates the one used for C programs. The corresponding option is `-D`. This option has two forms :

- `-D name=value`,
- `-D name`

[17]Technically, this is achieved by just switching the lexing buffer.

The first form defines a symbol `name` and binds it to the value `value`, where `value` can be either

- a string (without double quotes) (ex: `sample.txt`),
- a integer (ex: `12`),
- an explicitely unsigned integer (ex: `23U`),
- an explicitely signed integer (ex: `23S`).

Any occurrence of "`%name`" in the program source will then be textually substituted by the attached value.

The second form simply defines a symbol `name` without assigning it a value. This form is particularly used in conjunction with the conditional compilation mechanism described in Sec. 10.8.

For example, suppose we want to adjust the input file connected to the input stream, without having to edit the program itself. We would write the corresponding line of the program

```
stream inp:unsigned from %ifile;
```

and invoke the simulator, for ex., as :

```
caphc -D ifile=input_file_name.txt ... program.cph
```

Like C macros, this mechanism is handled using textual substitution at the lexical level and is therefore fragile.

10.8 Conditional compilation

The compiler supports a minimal form of conditional compilation using `#ifdef`, `#else` and `#endif` directives. As for the `#include` directive, these directives works exactly like with a C pre-processor. For example, in the following program

```
1 #ifdef SYM1
2 ...
3 #else
4 ...
5 #endif
```

the code section between the lines 1-3 will be included only if the symbol `SYM1` is defined[18]. Otherwise, the code section between lines 3 and 5 will be included. The `#else` section can be omitted. In this case, nothing in included if the corresponding symbol is not defined.

Symbols can be defined when invoking the compiler with the `-D` option (see Sec. 10.7).

In the current version, nesting of conditionnal compilation directives is not allowed.

10.9 Adjusting FIFO size

When running the simulator, the size of the FIFO channels connecting the actors may be adjusted using the `-chan_cap` option. The default value is 64. The option `-warn_channels` can be used to detect situations in which some channels get full (and therefore block program execution).

[18] Note that when using symbols with `#ifdef` directive, the symbol name is *not* prefixed with "`%`".

The default size for the FIFOs implementing the channels in the SystemC code is also 64. This value can be adjusted with `-sc_fifo_capacity` option. The `-sc_dump_fifo_stats` option can be used to obtain statistics about FIFO occupation during program execution. This option will generate a file named `fifo_stats.dat` which summarizes the maximum occupation of each FIFO during the program execution[19]. More accurate information can be obtained with option `-sc_dump_fifos`, which dumps (on stdout) the contents of each FIFO whenever it changes at runtime. An intermediate option is `-sc_trace_fifos`, which generates a .vcd file tracing the occupation of each FIFO and which can be viewed after run using a VCD viewer such as `gtkwave`.

When generating the VHDL code, the option `-vhdl_default_fifo_capacity` can be used to set the default size for the FIFOs. If not used, the value is 4. In some cases, it may be necessary to adjust this size individually for each FIFO. This can be done with `-vhdl_annot_file` option. This option accepts a file as argument. This file is supposed to contain a set of *back annotations* for customizing the final VHDL code. Currently, only one kind of annotation is available[20] :

- `<fid> fifo_size=<n>` : this annotation will set the size of the FIFO named `<fid>` in the VHDL RTL description to $n + m$, where m is an *offset* value. The default *offset* value is 2. It can be changed with the `-vhdl_fifo_offset` option.

This annotation mechanism allows the file `fifo_stats.dat` generated by running the SystemC generated code with the `-sc_dump_fifo_stats` to be passed directly as an argument to the `-vhdl_annot_file` option.

When targeting a FPGA using the VHDL backend, there are two possibilities for implementing FIFOs :

- using the registers included in the standard logic elements (LEs),
- using embedded RAM blocks.

Up to version 2.8.5, the CAPH standard VHDL library contained two FIFO models : one for generating LE-based implementations and the other for RAM-based implementations. This makes sense since using LE-based implementations for "big" FIFOs can can consume a larg number of LEs[21]. A pragmatical problem is that writing a platform-independant model for a RAM-based FIFO is actually very difficult[22] because it ultimately relies on the actual target hardware and on synthetizer-specific settings. We view it as essential that CAPH essentially remains a *platform-agnostic* tool, *i.e.* that it does not rely on hardware and/or vendor specific facilities. Since version 2.9.0, the CAPH VHDL library therefore only contains one FIFO model, generating, by default, LE-based FIFO implementations. For a given platform, it is possible to supply an alternate model – generating RAM-based implementations in particular – and to decide when using this model by using two compiler options[23] :

- the `-vhdl_big_fifo_model` option is used to specify the VHDL model for "big" FIFO (which have to be implemented using RAM-blocks typically),
- the `-vhdl_fifo_model_threshold` option is used to decide when to switch from the default ("small") model to the alternate ("big") one; the default threshold value is 32.

[19] The name of this file can be changed with the `-sc_fifo_stats_file` option.

[20] The set of annotations may be augmented in future versions of the CAPH compiler.

[21] A 256×10 bits FIFO, for example, required for storing a single line of a 256×256 image, will consume *at least* 2560 LEs !

[22] The `fifo_big.vhd` model provided in the CAPH library up to v2.8.5 did generate RAM-based implementations, but only on Altera FPGAs using the Quartus synthetizer.

[23] These options were already present in pre-2.9.0 versions; they are just used slightly differently now.

For example, suppose we have written an optimized RAM-based implementation of a FIFO in file my_ram_fifo.vhd. If we compile the program foo.cph with

```
caphc -vhdl -vdhl_big_fifo_model my_ram_fifo -vhdl_fifo_model_threshold 128 foo.cph
```

then, in the generated VHDL code, all FIFOs with a size ≤ 128 will be instanciated from the default fifo component provided in the CAPH standard library and all FIFOs with a size > 128 from the supplied my_ram_fifo component. For this to work, of course :

- the file my_ram_fifo.vhd must be available when compiling / synthetizing the generated VHDL code (either by supplying the appropriate options to the VHDL tools or by copying this file to the working directory),
- the interface and the behavior of the supplied FIFO must be conformant to the protocol used by CAPH actors to communicate to FIFOs; this protocol is described in Appendix cha:fifos.

Note. A third option, -vhdl_small_fifo_model, also makes it possible to change the VHDL model used for implementing "small" FIFOs (those with a depth smaller under the default or specified threshold). If not used, and as specified above, the model defined in lib/vhdl/fifo.vhd is used.

10.10 Dumping box FSMs

The behavior of an actor can often be described as a finite state machine (FSM), with a dedicated local variable playing the role of the state. When inkoked with the -dump_fsms option, the caphc compiler generates a graphical representation of all boxes with such an FSM behavior. The name of the generated file is <aaa>_act_b<nnn>.dot, where <aaa> is the name of the instanciated actor and <nnn> the box identifier[24]. Boxes associated with a FSM behavior are those having a least one local variable with a enumerated type or an ranged integer type. When several such variables are present the first declared one is considered to be the state variable.

Examples of generated FSMs are given in figures 10.1, 10.2 and 10.3. The two former examples have already been described in Sec. 2.4.4. The latter has been introduced in chapter 8. This last example shows how the actual value of a box parameter can be used to instanciate the FSM.

10.11 The caphmake utility

This utility, introduced in version 2.8.1, greatly simplifies the usage of the compiler by reducing the configuration of a CAPH project to the strict minimum[25].

Generating code and running simulations using the CAPH compiler now proceeds as follows :

1. generate a top-level Makefile by invoking caphmake,
2. generate backend specific Makefiles by typing make makefiles,
3. generate code and/or run simulations by invoking the ad-hoc rule; the most useful rules are[26] :
    - make dot to generate the .dot representation of the program (make dot.show to display it),
    - make sim.run to run the simulation using the interpreter (make sim.show to display results),

[24]The box identifier can be retrieved with the -dump_senv option or by inspecting the .dot file representation of the program.

[25]It can be viewed as the analogue of the qmake utility in Qt distributions.

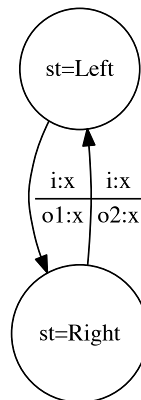
[26]See the generated Makefiles for a complete list of rules.

```
actor switch
  in (i:$t)
  out (o1:$t, o2:$t)
var st : {Left,Right} = Left
rules
| (st:Left, i:x) -> (o1:x, st:Right)
| (st:Right, i:x) -> (o2:x, st:Left)
;

stream i:signed<8> from "sample.txt";
stream o1:signed<8> to "result1.txt";
stream o2:signed<8> to "result2.txt";

net (o1,o2) = switch i;
```

Program



FSM of the `switch` actor

Figure 10.1: Example of FSM generation

- `make systemc.code` to generate the SystemC code,
- `make systemc.run` to generate the SystemC code and run the corresponding simulation (`make systemc.show` to display results),
- `make vhdl.code` to generate the VHDL code,
- `make vhdl.run` to generate the VHDL code and run the corresponding simulation (`make vhdl.show` to display results).

The description of the `caphmake` utility is provided in the appendices.

Application-specific customization is essentially done by providing a file named `<app>.proj`, where `<app>` is the name of the toplevel CAPH source file (without the `.cph` suffix),

The `.proj` file – which must be placed in the same directory than the compiled program – essentially contains the values of the options to be passed to the compiler for each possible target[27] (`.dot` generation, interpreter-based simulation, SystemC backend or VHDL backend). As an example, listing 10.4 gives a `.proj` file which can be used for the testing the edge extraction application described in chapter 3 of the CAPH tutorial ("Caph Primer").

Listing 10.4: A possible `.proj` file for the edge extraction application described in chapter 3 of the "Caph Primer"

```
GEN_OPTS = -D ifile=pcb.pgm -D threshold=80
DOT_OPTS = $(GEN_OPTS)
SIM_OPTS = $(GEN_OPTS) -abbrev_dc_ctors -dump_channel_stats
SC_OPTS = $(GEN_OPTS) -sc_abbrev_dc_ctors -sc_stop_when_idle 1000 -sc_dump_fifo_stats
VHDL_OPTS = $(GEN_OPTS) -vhdl_annot_file main_fifo_stats.dat
GHDL_RUN_OPTS = --stop-time=160000ns
```

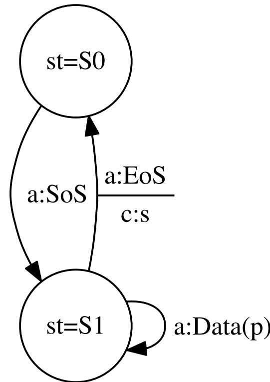
[27]Technically, this file is simply included at the beginning of each of the Makefiles generated from the toplevel Makefile.

```
actor suml
  in (a:signed<8> dc)
  out (c:signed<8>)
var st : {S0,S1} = S0
var s : signed<8>
rules
| (st:S0, a:'<) -> (st:S1, s:0)
| (st:S1, a:'p) -> (s:s+p)
| (st:S1, a:'>) -> (st:S0, c:s)
;

stream i:signed<8> dc from "sample.
    txt";
stream o:signed<8> to "result.txt";

net o = suml i;
```

Program



FSM of the `suml` actor

Figure 10.2: Example of FSM generation

In some (rare) cases, the default targets produced in the target specific Makefiles when invoking `make makefiles` after `caphmake` may not be adequate. A typical situation is when some extra arguments, which cannot be guessed by compiler from the source code, have to be passed to some utility programs. In this case, it is possible to override the corresponding rules by writing a file `<app>.rules` containing the new definitions[28]. As an example, listing 10.5 gives the `.rules` file which has been used for the testing the second version of the edge extraction application described in chapter 3 of the CAPH tutorial. Since this version makes use of centered convolution, *blanking* parameters must be passed to the passed to the `pgm2bin` program when generating the `.bin` input file from the input `.pgm` image. This cannot be guessed by the compiler and hence has to be specified in the `.rules` file[29].

Listing 10.5: A possible `.rules` file for the seconde version of the application described in chapter 3 of the tutorial

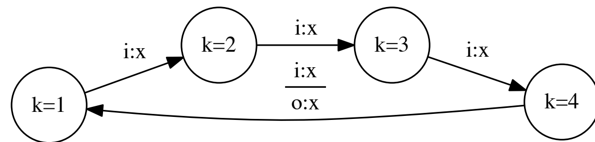
```
pcb.bin: pcb.pgm
    $(PGM2BIN) -hblank 4 -vblank 140 12 $< $@
```

[28]As for the `.proj` file, the `.rules` file must be located in the same directory than the compiled file.

[29]Technically this file is inserted at the end of the target specific Makefiles, so that overriding can take places. This explains why the corresponding definitions cannot be given in the `.proj` file.

```
actor sample (n: int)
  in (i: $t)
  out (o: $t)
var k : {1,..,n} = 1
rules
| i:x when k<n -> k:k+1
| i:x when k=n -> (k:1, o:x)
;
...
net o = sample 4 i;
```

Program



FSM of the `sample` actor

Figure 10.3: Example of FSM generation

Chapter 11

The CAPH standard libraries

The CAPH distribution comes with a small set of "libraries" containing definitions of actors and functions which are likely to be frequently used in signal or image processing applications :

- dc.cph contains the definition of the dc type (see Sec. 2.4.1),
- list_ops.cph contains the definitions of actors performing back and forward delays on lists of values,
- img_ops.cph contains the definitions of actors and wiring functions for manipulating images (horizontal and vertical delays, neighborhood generation, ...),
- convol.cph contains the definitions of actors performing convolutions on lists and images,
- stream_ops.cph contains the definitions of actors performing various operations on *unstructured* streams.

Most of the actors (resp. functions) are polymorphic and accept parameters making them sufficiently generic to be used in a large variety of contexts.

To use a library, one simply has to include it in the program, using the #include directive described in Sec. 10.6. For example, a program using the dc type will start with the following line :

```
#include "dc.cph"
```

The location of the standard libraries must be specified with the -I option (see Sec. 13).

The contents of the standard libraries is reproduced below.

Listing 11.1: The library dc.cph

```
-- The [dc] type is used for encoding structured streams of values.
--
-- The "SoS" and "EoS" constructors respectively encode the "Start of Structure" and
-- "End of Structure" control token.
-- The "Data" constructor encodes the data tokens (where the data has type $t).
-- For example, a list of boolean values can be represented with the following stream
-- of tokens, of type [bool dc]:
--     SoS true false true ... EoS
-- and a mxn image of 8-bit pixels, with the following stream, of type [unsigned<8> dc
    ]:
--     SoS SoS p11 p12 ... p1n EoS SoS p21 p22 ... p2n EoS ...
--      ... SoS pm1 pm2 ... pmn EoS EoS
--  where each inner list represents a line (row) of the image.
```

```
-- When reading / printing tokens of type [dc], the values "SoS", "EoS" and "Data v"
-- can be respectively abbreviated as "'<", "'>" and "'v" by invoking the compiler
-- with the [-abbrev_dc_ctors].
-- Note : the numeric encoding of the constructor tags (see LRM, chap 14) ultimately
-- depends on the hardware testbench; the value specified here (%1, %2, %3) match
-- the hardware specifications of the DreamCam platform. Do not change them if you
-- target this platform.
--
-- This file appeared in vers 2.6.2. Previously, the [dc] type was builtin.

type $t dc =
  SoS %1
| EoS %2
| Data %3 of $t
;

-- SMAP higher order actor operating on DC structured streams
-- smap(f) : < x1 x2 ... xn > = < f(x1) f(x2) ... f(xn) >
-- Ex : smap(inc) : < 1 2 3 .. > = < 2 3 4 .. >  if inc(x)=x+1

actor smap (f:$t1->$t2)
  in (i:$t1 dc)
 out (o:$t2 dc)
rules
| i:SoS      -> o:SoS
| i:Data x -> o:Data (f(x))
| i:EoS      -> o:EoS
;

-- SMAP2 higher order actor operating on DC structured streams
-- Generalisation of SMAP to two input streams
-- smap2(f) : (< x1 x2 ... xn >, <y1 y2 ... yn >) = < f(x1,y1) f(x2,y2) ... f(xn,yn) >
-- Ex : smap2(+) : (< 1 2 3 .. >, < 10 20 30 ... >) = < 11 22 33 .. >

actor smap2 (f:$t11*$t12->$t2)
  in (i1:$t11 dc, i2:$t12 dc)
 out (o:$t2 dc)
rules
| (i1:SoS, i2:SoS)           -> o:SoS
| (i1:Data x, i2:Data y) -> o:Data (f(x,y))
| (i1:EoS, i2:EoS)           -> o:EoS
;

-- SFOLD higher order actor operating on DC structured streams
-- sfold(f,z) : < x11 x12 ... x1n > < x21 ... x2n > ... = y1 y2 ...
--   where yi = f (f (f (f (z,xi1), xi2)) ..., xin)
-- Ex: sfold(+,0) : < 1 2 > < 3 4 5 > < 6 7 8 9 > ... = (1+2) (3+4+5) (6+7+8+9+) ... =
    3 12 30 ..

actor sfold (f:$t*$t->$t, z:$t)
  in (i:$t dc)
 out (o:$t)
var st : {S0, S1} = S0
var acc: $t = z
rules
| (st:S0, i:SoS) -> (acc:z, st:S1)
| (st:S1, i:EoS) -> (o:acc, st:S0)
| (st:S1, i:Data x) -> acc:f(acc,x)
;

-- SSFOLD higher order actor operating on DC structured streams
-- ssfold(f,z) : < l1 l2 ... ln > = < sfold(f,z)(l1) sfold(f,z)(l2) ... sfold(f,z)(ln)
    >
```

```
-- Ex: sffold(+,0) : < < 1 2 3 > < 4 5 6 > < 7 8 9 > > = < (1+2+3) (4+5+6) (7+8+9) > =
    < 6 15 24 >

actor ssfold (f:$t*$t->$t, z:$t)
  in (i:$t dc)
 out (o:$t dc)
var st : {S0, S1, S2} = S0
var acc: $t = z
rules
| (st:S0, i:SoS) -> (o:SoS, st:S1)
| (st:S1, i:EoS) -> (o:EoS, st:S0)
| (st:S1, i:SoS) -> (acc:z, st:S2)
| (st:S2, i:EoS) -> (o:Data acc, st:S1)
| (st:S2, i:Data x) -> acc:f(acc,x)
;
```

Listing 11.2: The library list_ops.cph

```
--------------------------------------
-- Basic operations on lists
-- 2014-11-05, JS
--------------------------------------

#include "dc.cph"  -- the [dc] type

-----------------------------------------------------
-- DL - One-pixel delay on lists
-- DL(v):<x1,x2,...,xn> = <v,x1,...,xn-1>
-----------------------------------------------------

actor dl (v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1} = S0
var z : $t
rules
| (s:S0, a:'<) -> (s:S1, c:'<, z:v)
| (s:S1, a:'p) -> (s:S1, c:'z, z:p)
| (s:S1, a:'>) -> (s:S0, c:'>)
;

-----------------------------------------------------
-- DkL - k-pixel delay on lists
-- DkL(k,v):<x1,x2,...,xn> = <v,...,v,x1,...,xn-k>
--                            \--k--/
-----------------------------------------------------

actor dkl (k:int, v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2} = S0
var z : $t array[k] = [ v | i = 0 to k-1 ]
var i : int
rules
| (s:S0, a:'<)            -> (s:S1, c:'<, i:0)
| (s:S1, a:'p) when i<k-1 -> (s:S1, c:'v, i:i+1, z[i]:p)
| (s:S1, a:'p)            -> (s:S2, c:'v, i:0, z[i]:p)
| (s:S2, a:'p)            -> (s:S2, c:'z[i], i:if i<k-1 then i+1 else 0, z[i]:p)
| (s:S2, a:'>)            -> (s:S0, c:'>)
;

-----------------------------------------------------
```

```
-- FL - One-pixel forward on lists
-- FL(s:v):<x1,x2,...,xn> = <x2,...,xn,v>
--------------------------------------------------------------

actor fl (v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2,S3} = S0
rules
| (s:S0, a:'<) -> (s:S1, c:'<)
| (s:S1, a:'p) -> (s:S2      )
| (s:S2, a:'p) -> (s:S2, c:'p)
| (s:S2, a:'>) -> (s:S3, c:'v)
| (s:S3      ) -> (s:S0, c:'>)
;

--------------------------------------------------------------
-- FkL - k-pixel forward on list
-- FkL(k,v):<x1,x2,...,xn> = <xk,...,xn-k,v,....,v>
--                                        \--k--/
--------------------------------------------------------------

actor fkl (k:int, v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2,S3} = S0
var i : int
rules
| (s:S0, a:'<)      -> (s:S1, c:'<, i:k)
| (s:S1, a:'p, i:0) -> (s:S2, c:'p)
| (s:S1, a:'p)      -> (s:S1, i:i-1)
| (s:S2, a:'>)      -> (s:S3, c:'v, i:k-1)
| (s:S2, a:'p)      -> (s:S2, c:'p)
| (s:S3, i:0)       -> (s:S0, c:'>)
| (s:S3)            -> (s:S3, c:'v, i:i-1)
;
```

Listing 11.3: The library img_ops.cph

```
-------------------------------------------------------
-- Basic operations on images (lists of lists)
-- 2014-11-05, JS
-------------------------------------------------------

#include "dc.cph"  -- the [dc] type

--------------------------------------------------------------
-- D1P - 1-pixel delay on images
-- D1P(v):<L1,L2,...,Ln>=<f:L1,f:L2,...,f:Ln>
--      where : f:<x1,x2,...,xn> = <v,x1,...,xn-1>
--------------------------------------------------------------

actor d1p (v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2} = S0
var z : $t
rules
| (s:S0, a:'<) -> (s:S1, c:'<)
| (s:S1, a:'>) -> (s:S0, c:'>)
| (s:S1, a:'<) -> (s:S2, c:'<, z:v)
| (s:S2, a:'p) -> (s:S2, c:'z, z:p)
```

```
| (s:S2, a:'>) -> (s:S1, c:'>)
;

------------------------------------------------------------
-- F1P - One-pixel forward on images
-- F1P(v):<L1,L2,...,Ln>=<f:L1,f:L2,...,f:Ln>
--      where : f:<x1,x2,...,xn> = <x2,...,xn,v>
------------------------------------------------------------

actor f1p (v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2,S3,S4} = S0
rules
| (s:S0, a:'<) -> (s:S1, c:'<)
| (s:S1, a:'>) -> (s:S0, c:'>)
| (s:S1, a:'<) -> (s:S2, c:'<)
| (s:S2, a:'p) -> (s:S3       )
| (s:S3, a:'p) -> (s:S3, c:'p)
| (s:S3, a:'>) -> (s:S4, c:'v)
| (s:S4       ) -> (s:S1, c:'>)
;

-----------------------------------------------------------------
-- DkP - k-pixel delay on images
-- DkP(k,v):<L1,L2,...,Ln>=<f:L1,f:L2,...,f:Ln>
--      where : f:<x1,x2,...,xn> = <v,...,v,x1,...,xn-k>
--                                   \--k--/
-----------------------------------------------------------------

actor dkp (k:int, v:$t)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2,S3} = S0
var i : int
var z : $t array[k] = [ v | i = 0 to k-1 ]
rules
| (s:S0, a:'<)            -> (s:S1, c:'<)
| (s:S1, a:'>)            -> (s:S0, c:'>)
| (s:S1, a:'<)            -> (s:S2, c:'<, i:0)
| (s:S2, a:'p) when i<k-1 -> (s:S2, c:'v, z[i]:p, i:i+1)
| (s:S2, a:'p)            -> (s:S3, c:'v, z[i]:p, i:0)
| (s:S3, a:'p)            -> (s:S3, c:'z[i], z[i]:p, i:if i<k-1 then i+1 else 0)
| (s:S3, a:'>)            -> (s:S1, c:'>)
;

--
    --------------------------------------------------------------------------------------------------------

-- FkP - k-pixel forward on images
-- FkP(k,v):<L1,L2,...,Ln>=<f:L1,f:L2,...,f:Ln>
--    where f:<x1,x2,...,xn> = <xk,...,xn-k,v,....,v>
--                                          \--k--/
-- WARNING: unless the input device insert some inter-line delay between lines of the
    input image,
-- this actor requires a FIFO with min capacity [k*nb_lignes] on input..
-- To understand why, just run the simulator in trace mode on a small (4x4) example..
--
    --------------------------------------------------------------------------------------------------------


actor fkp (k:int, v:$t)
  in (a:$t dc)
```

```
  out (c:$t dc)
var s : {S0,S1,S2,S3,S4} = S0
var i : int
rules
| (s:S0, a:'<)           -> (s:S1, c:'<)
| (s:S1, a:'>)           -> (s:S0, c:'>)
| (s:S1, a:'<)           -> (s:S2, c:'<, i:k)
| (s:S2, a:'p) when i>0  -> (s:S2, i:i-1)
| (s:S2, a:'p)           -> (s:S3, c:'p)
| (s:S3, a:'p)           -> (s:S3, c:'p)
| (s:S3, a:'>)           -> (s:S4, c:'v, i:k-1)
| (s:S4) when i>0        -> (s:S4, c:'v, i:i-1)
| (s:S4)                 -> (s:S1, c:'>)
;

-------------------------------------------------------------------
-- D1Li - 1-line delay on images (with local storage)
-- D1Li(v):<L1,L2,...,Ln>=<L0,L1,...,Ln-1>
--      where L0=<v,v,...,v>
-------------------------------------------------------------------

actor d1li (v:$t, maxwidth:int)
  in (a:$t dc)
  out (c:$t dc)
var s : {S0,S1,S2,S3,S4}=S0
var z : $t array[maxwidth] = [ v | i = 0 to maxwidth-1 ]
var i : int
rules
| (s:S0, a:'<) -> (s:S1, c:'<)
| (s:S1, a:'>) -> (s:S0, c:'>)
| (s:S1, a:'<) -> (s:S2, c:'<, i:0)
| (s:S2, a:'>) -> (s:S3, c:'>)
| (s:S2, a:'p) -> (s:S2, c:'v, z[i]:p, i:i+1)
| (s:S3, a:'>) -> (s:S0, c:'>)
| (s:S3, a:'<) -> (s:S4, c:'<, i:0)
| (s:S4, a:'p) -> (s:S4, c:'z[i], z[i]:p, i:i+1)
| (s:S4, a:'>) -> (s:S3, c:'>)
;

-------------------------------------------------------------------------------------------------------
-- D1L - 1-line delay on images (recursive version, with feedback via external FIFO)
-- D1L(v):<L1,L2,...,Ln>=<L0,L1,...,Ln-1>
--      where L0=<v,v,...,v>
-------------------------------------------------------------------------------------------------------

actor d1lr (v:$t)
  in (a:$t dc, z:$t)
  out (c:$t dc, zz:$t)
var s : {S0,S1,S2,S3,S4}=S0
rules
  ( s,  a, z) -> ( s,  c, zz)
| (S0,  _, z) -> (S0,  _,  _)  -- empty the feedback wire
| (S0, '<, _) -> (S1, '<,  _)  -- start of frame
| (S1, '>, _) -> (S0, '>,  _)  -- end of frame
| (S1, '<, _) -> (S2, '<,  _)  -- start of first line
| (S2, '>, _) -> (S3, '>,  _)  -- end of first line
| (S2, 'p, _) -> (S2, 'v,  p)  -- first line
| (S3, '>, _) -> (S0, '>,  _)  -- end of last line
| (S3, '<, _) -> (S4, '<,  _)  -- start of line
| (S4, 'p, z) -> (S4, 'z,  p)  -- line
| (S4, '>, _) -> (S3, '>,  _)  -- end of line
;
```

```
net d1l z i = let rec (o,l) = d1lr z (i,l) in o;
```

Listing 11.4: The library convol.cph

```
-- -- ==============================================================
-- -- SINGLE-ACTOR CONVOLUTIONS
-- -- Generic name : [c]conv<d><k>
-- -- where
-- --   [c] is for centered convolution (default is shifted)
-- --   <d> is the signal dimension ("1" or "2")
-- --   <k> is the kernel dimension ("13", "33", ...)
-- -- Examples :
-- --   conv113 implements a (shifted) 1x3 convolution on a 1D signal (<x1,...,xn>)
-- --   cconv233 implements a centered 3x3 convolution on a 2D image (<<x11,..,x1n
    >,...,<xm1,...,xmn>>)
-- -- ==============================================================

#include "dc.cph" -- the [dc] type

--------------------------------------------------------------------------------
-- -- CONV11NA : 1xN shifted convolution on signals implemented as a single actor
-- -- CONV11N(k,n,v) :<x_1,x_2,...,x_i,...> =<f x_1,f x_2,...,f x_i,...>
-- -- where
-- --   f(x_i) = v if i<N
-- --   f(x_i) = (x_{i-N+1}*k[0] + ... + x_{i_1}*k[N-2] + ... + x_{i}*k[N-1]) / 2^n
--------------------------------------------------------------------------------

actor conv113 (k:int<s,m> array[3], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
var s : {S0,S1,S2,S3} = S0
var z : int<s,m> array[2]
rules
| (s:S0, i:'<) -> (s:S1, o:'<)
| (s:S1, i:'>) -> (s:S0, o:'>)
| (s:S1, i:'p) -> (s:S2, o:'v, z[0]:p)
| (s:S2, i:'p) -> (s:S3, o:'v, z[0]:p, z[1]:z[0])
| (s:S3, i:'p) -> (s:S3, o:'((p*k[2]+z[0]*k[1]+z[1]*k[0])>>n), z[0]:p,  z[1]:z[0])
| (s:S3, i:'>) -> (s:S0, o:'>)
;

actor conv115 (k:int<s,m> array[5], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
var s : {S0,S1,S2,S3,S4,S5} = S0
var z : int<s,m> array[4]
rules
| (s:S0, i:'<) -> (s:S1, o:'<)
| (s:S1, i:'>) -> (s:S0, o:'>)
| (s:S1, i:'p) -> (s:S2, o:'v, z[0]:p)
| (s:S2, i:'p) -> (s:S3, o:'v, z[0]:p, z[1]:z[0])
| (s:S3, i:'p) -> (s:S4, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1])
| (s:S4, i:'p) -> (s:S5, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'p) -> (s:S5, o:'((p*k[4]+z[0]*k[3]+z[1]*k[2]+z[2]*k[1]+z[3]*k[0])>>n), z
    [0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'>) -> (s:S0, o:'>)
;

--------------------------------------------------------------------------------
-- -- CCONV11NA : 1xN centered convolution on signals implemented as a single actor
-- -- !! Only defined for N=2*M+1
-- -- CCONV113A(k,n,v) :<x_1,x_2,...,x_i,...> =<f x_1,f x_2,...,f x_i,...>
```

```
-- -- where
-- --    f(x_i) = v if i<M
-- --    f(x_i) = (x_{i-M}*k[0] + ... + x_{i}*k[M] + ... + x_{i+M}*k[N-1]) / 2^n
---------------------------------------------------------------------------------------

actor cconv113 (k:int<s,m> array[3], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
var s : {S0,S1,S2,S3,S4} = S0
var z : int<s,m> array[2]
rules
| (s:S0, i:'<) -> (s:S1, o:'<)
| (s:S1, i:'>) -> (s:S0, o:'>)
| (s:S1, i:'p) -> (s:S2, z[0]:p)
| (s:S2, i:'p) -> (s:S3, o:'v, z[0]:p, z[1]:z[0])
| (s:S3, i:'p) -> (s:S3, o:'((p*k[2]+z[0]*k[1]+z[1]*k[0])>>n), z[0]:p,  z[1]:z[0])
| (s:S3, i:'>) -> (s:S4, o:'v)
| (s:S4)       -> (s:S0, o:'>)
;

actor cconv115 (k:int<s,m> array[5], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
var s : {S0,S1,S2,S3,S4,S5,S6,S7} = S0
var z : int<s,m> array[4]
rules
| (s:S0, i:'<) -> (s:S1, o:'<)
| (s:S1, i:'>) -> (s:S0, o:'>)
| (s:S1, i:'p) -> (s:S2, z[0]:p)
| (s:S2, i:'p) -> (s:S3, z[0]:p, z[1]:z[0])
| (s:S3, i:'p) -> (s:S4, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1])
| (s:S4, i:'p) -> (s:S5, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'p) -> (s:S5, o:'((p*k[4]+z[0]*k[3]+z[1]*k[2]+z[2]*k[1]+z[3]*k[0])>>n), z
    [0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'>) -> (s:S6, o:'v)
| (s:S6)       -> (s:S7, o:'v)
| (s:S7)       -> (s:S0, o:'>)
;

-- ------------------------------------------------------------------------------------
-- CONV21<n>a : shifted 1xn convolutions on 2D images implemented as single actors
-- Generalisation of CONV11<n>a to 2D signals
-- ------------------------------------------------------------------------------------

actor conv213 (k:int<s,m> array[3], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
var s : {S00,S0,S1,S2,S3} = S00
var z : int<s,m> array[2]
rules
| (s:S00, i:'<) -> ( s:S0, o:'<)
| ( s:S0, i:'>) -> (s:S00, o:'>)
| ( s:S0, i:'<) -> ( s:S1, o:'<)
| ( s:S1, i:'>) -> ( s:S0, o:'>)
| ( s:S1, i:'p) -> ( s:S2, o:'v, z[0]:p)
| ( s:S2, i:'p) -> ( s:S3, o:'v, z[0]:p, z[1]:z[0])
| ( s:S3, i:'p) -> ( s:S3, o:'((p*k[2]+z[0]*k[1]+z[1]*k[0])>>n), z[0]:p,  z[1]:z[0])
| ( s:S3, i:'>) -> ( s:S0, o:'>)
;

actor conv215 (k:int<s,m> array[5], n:int, v:int<s,m>)
   in (i:int<s,m> dc)
  out (o:int<s,m> dc)
```

```
var s : {S00,S0,S1,S2,S3,S4,S5} = S00
var z : int<s,m> array[4]
rules
| (s:S00, i:'<) -> ( s:S0, o:'<)
| (s:S0, i:'>) -> (s:S00, o:'>)
| (s:S0, i:'<) -> (s:S1, o:'<)
| (s:S1, i:'>) -> (s:S0, o:'>)
| (s:S1, i:'p) -> (s:S2, o:'v, z[0]:p)
| (s:S2, i:'p) -> (s:S3, o:'v, z[0]:p, z[1]:z[0])
| (s:S3, i:'p) -> (s:S4, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1])
| (s:S4, i:'p) -> (s:S5, o:'v, z[0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'p) -> (s:S5, o:'((p*k[4]+z[0]*k[3]+z[1]*k[2]+z[2]*k[1]+z[3]*k[0])>>n), z
    [0]:p, z[1]:z[0], z[2]:z[1], z[3]:z[2])
| (s:S5, i:'>) -> (s:S0, o:'>)
;

--
    ---------------------------------------------------------------------------------------------------------------

-- CONV233 : shifted 3x3 convolution on 2D images implemented as a
-- single actor (with external fifos for line delays)
-- CONV233(k,n,v):<<x_11,...,x_1n>,...<x_m1,...x_mn>> = <<f(x_11),...,f(x_1n)>,...<f(
    x_m1),...f(x_mn)>>
-- where
--    f(x_{i,j}) = (k_{0,0}*x_{i-2,j-2} + k_{0,1}*x_{i_2,j-1} + k_{0,2}*x_{i-2,j}
--                + k_{1,0}*x_{i-1,j-2} + k_{1,1}*x_{i_1,j-1} + k_{1,2}*x_{i-1,j}
--                + k_{2,0}*x_{i,j-2}   + k_{2,1}*x_{i,j-1}   + k_{2,2}*x_{i,j}) >> n
--  (with x_{-2,j}=x{-1,j}=x{i,-2}=x{i,-1}=v)
-- where :
--  +-----+-----+-----+   +-------------+-------------+-----------+
--  | k00 | k01 | k02 |   | x_{i-2,j-2} | x_{i-2,j-1} | x_{i-2,j} |
--  +-----+-----+-----+   +-------------+-------------+-----------+
--  | k10 | k11 | k12 | * | x_{i-1,j-2} | x_{i-1,j-1} | x_{i-1,j} |
--  +-----+-----+-----+   +-------------+-------------+-----------+
--  | k20 | k21 | k22 |   |  x_{i,j-2}  |  x_{i,j-1}  |  x_{i,j}  | <- current pixel
--  +-----+-----+-----+   +-------------+-------------+-----------+
--
    ---------------------------------------------------------------------------------------------------------------


actor conv233a (k:int<s,m> array[3][3], n:int, v:int<s,m>)
  in (i:int<s,m> dc,     -- input
      z0:int<s,m>,       -- previous line  (fed back through an external link)
      z1:int<s,m>)       -- pre-previous line  (fed back through an external link)
  out (o:int<s,m> dc,    -- output
       oz0:int<s,m>,     -- previous line  (fed back through an external link)
       oz1:int<s,m>)     -- previous line  (fed back through an external link)
var s : {S0,S1,S2,S3,S4,S5,S6,S7,S8} = S0
var x : int<s,m> array[3][2]
--  +-----+-----+-----+   +---------+---------+-------+
--  | k00 | k01 | k02 |   | x[2][1] | x[2][0] | x6=p2 |
--  +-----+-----+-----+   +---------+---------+-------+
--  | k10 | k11 | k12 | * | x[1][1] | x[1][0] | x3=p1 |
--  +-----+-----+-----+   +---------+---------+-------+
--  | k20 | k21 | k22 |   | x[0][1] | x[0][0] | x0=p0 |
--  +-----+-----+-----+   +---------+---------+-------+
rules
| (s:S0, i:'<)             -> (s:S1, o:'<)                -- Start of Image
| (s:S1, i:'>)             -> (s:S0, o:'>)                -- End of image
| (s:S1, i:'<)             -> (s:S2, o:'<)                -- Start of first line
| (s:S2, i:'p0)            -> (s:S2, o:'v, oz0:p0)        -- Read first line and
    store in fifo z0
| (s:S2, i:'>)             -> (s:S3, o:'>)                -- End of first line
```

```
| (s:S3, i:'<)             -> (s:S4, o:'<)                     -- Start of second line
| (s:S4, i:'p0, z0:p1)     -> (s:S4, o:'v, oz1:p1, oz0:p0)    -- Read second line,
    store in fifo z0 while moving z0 to z1
| (s:S4, i:'>)             -> (s:S5, o:'>)                     -- End of second line
| (s:S5, i:'>)             -> (s:S0, o:'>)                     -- End of image
| (s:S5, i:'<)             -> (s:S6, o:'<)                     -- Start of third (and
    subsequent) line(s)
                                                               -- The previous and pre
                                                                  -previous lines are
                                                                   avail from z0 and
                                                                  z1 resp.
| (s:S6, i:'p0, z0:p1, z1:p2) -> (s:S7, o:'v,                  -- First pixel of line
                                 x[2][0]:p2, x[1][0]:p1, x[0][0]:p0,
                                 oz1:p1, oz0:p0)
| (s:S7, i:'p0, z0:p1, z1:p2) -> (s:S8, o:'v,                  -- Second pixel
                                 x[2][1]:x[2][0], x[2][0]:p2, x[1][1]:x[1][0], x
                                     [1][0]:p1, x[0][1]:x[0][0], x[0][0]:p0,
                                 oz1:p1, oz0:p0)
| (s:S8, i:'p0, z0:p1, z1:p2) -> (s:S8,                        -- Third and subsequent
     pixels of line
                                 o:'((k[0][0]*x[2][1] + k[0][1]*x[2][0] + k[0][2]*p2
                                   + k[1][0]*x[1][1] + k[1][1]*x[1][0] + k[1][2]*p1
                                   + k[2][0]*x[0][1] + k[2][1]*x[0][0] +  k[2][2]*p0)
                                       >>n),
                                 x[2][1]:x[2][0], x[2][0]:p2, x[1][1]:x[1][0], x
                                     [1][0]:p1, x[0][1]:x[0][0], x[0][0]:p0,
                                 oz1:p1, oz0:p0)
| (s:S8, i:'>)              -> (s:S5, o:'>)
;

net conv233 (kernel,norm,pad) i = let rec (o,z0,z1) = conv233a (kernel,norm,pad) (i,z0,
    z1) in o;

actor conv255a (k:int<s,m> array[5][5], n:int, v:int<s,m>)
  in (i:int<s,m> dc,      -- input
      z0:int<s,m>,        -- previous lines  (fed back through an external link)
      z1:int<s,m>,
      z2:int<s,m>,
      z3:int<s,m>)
  out (o:int<s,m> dc,     -- output
       oz0:int<s,m>,      -- previous lines  (fed back through an external link)
       oz1:int<s,m>,
       oz2:int<s,m>,
       oz3:int<s,m>)
var s : {S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14} = S0
var x : int<s,m> array[5][4]
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k00 | k01 | k02 | k03 | k04 |   | x[4][3] | x[4][2] | x[4][1] | x[4][0] |    p4
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k10 | k11 | k12 | k13 | k14 |   | x[3][3] | x[3][2] | x[3][1] | x[3][0] |    p3
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k20 | k21 | k22 | k23 | k24 | * | x[2][3] | x[2][2] | x[2][1] | x[2][0] |    p2
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k30 | k31 | k32 | k33 | k34 |   | x[1][3] | x[1][2] | x[1][1] | x[1][0] |    p1
    |
```

```
--  +----+----+----+----+----+
    +----------+----------+----------+----------+----------+
--  | k40 | k41 | k42 | k43 | k44 |    | x[0][3] | x[0][2] | x[0][1] | x[0][0] |      p0
      |
--  +----+----+----+----+----+
    +----------+----------+----------+----------+----------+
rules
| (s:S0, i:'<)                 -> (s:S1, o:'<)                     -- Start of Image
| (s:S1, i:'>)                 -> (s:S0, o:'>)                     -- End of image

| (s:S1, i:'<)                 -> (s:S2, o:'<)                     -- Start of first line
| (s:S2, i:'p0)                -> (s:S2, o:'v, oz0:p0)             -- Read first line and
    store in fifo z0
| (s:S2, i:'>)                 -> (s:S3, o:'>)                     -- End of first line

| (s:S3, i:'<)                 -> (s:S4, o:'<)                     -- Start of second line
| (s:S4, i:'p0, z0:p1)         -> (s:S4, o:'v, oz1:p1, oz0:p0)     -- Read second line,
    store in fifo z0 while moving z0 to z1
| (s:S4, i:'>)                 -> (s:S5, o:'>)                     -- End of second line

| (s:S5, i:'<)                 -> (s:S6, o:'<)                     -- Start of third line
| (s:S6, i:'p0, z0:p1, z1:p2) -> (s:S6, o:'v, oz2:p2, oz1:p1, oz0:p0)
                                                                   -- Read third line,
                                                                      store in fifo z0
                                                                      while moving z0 to
                                                                      z1 and z1 to z2
| (s:S6, i:'>)                 -> (s:S7, o:'>)                     -- End of third line

| (s:S7, i:'<)                 -> (s:S8, o:'<)                     -- Start of 4th line
| (s:S8, i:'p0, z0:p1, z1:p2, z2:p3) -> (s:S8, o:'v, oz3: p3, oz2:p2, oz1:p1, oz0:p0)
                                                                   -- Read 4th line, store
                                                                      in fifo z0 while
                                                                      moving z0 to z1, z1
                                                                      to z2 and z2 to z3
| (s:S8, i:'>)                 -> (s:S9, o:'>)                     -- End of 4th line

| (s:S9, i:'>)                 -> (s:S0, o:'>)                     -- End of image
| (s:S9, i:'<)                 -> (s:S10, o:'<)                    -- Start of 5th (and
    subsequent) line(s)
                                                                   -- The previous lines
                                                                      are avail from z0,
                                                                      z1, z2 and z3 resp.

| (s:S10, i:'p0, z0:p1, z1:p2, z2:p3, z3:p4) -> (s:S11, o:'v,     -- First pixel of line
                                 x[4][0]:p4,
                                 x[3][0]:p3,
                                 x[2][0]:p2,
                                 x[1][0]:p1,
                                 x[0][0]:p0,
                                 oz3:p3, oz2:p2, oz1:p1, oz0:p0)

| (s:S11, i:'p0, z0:p1, z1:p2, z2:p3, z3:p4) -> (s:S12, o:'v,     -- Second pixel
                                 x[4][1]:x[4][0], x[4][0]:p4,
                                 x[3][1]:x[3][0], x[3][0]:p3,
                                 x[2][1]:x[2][0], x[2][0]:p2,
                                 x[1][1]:x[1][0], x[1][0]:p1,
                                 x[0][1]:x[0][0], x[0][0]:p0,
                                 oz3:p3, oz2:p2, oz1:p1, oz0:p0)

| (s:S12, i:'p0, z0:p1, z1:p2, z2:p3, z3:p4) -> (s:S13, o:'v,     -- Third pixel
                                 x[4][2]:x[4][1], x[4][1]:x[4][0], x[4][0]:p4,
                                 x[3][2]:x[3][1], x[3][1]:x[3][0], x[3][0]:p3,
                                 x[2][2]:x[2][1], x[2][1]:x[2][0], x[2][0]:p2,
```

```
                                  x[1][2]:x[1][1], x[1][1]:x[1][0], x[1][0]:p1,
                                  x[0][2]:x[0][1], x[0][1]:x[0][0], x[0][0]:p0,
                                  oz3:p3, oz2:p2, oz1:p1, oz0:p0)

| (s:S13, i:'p0, z0:p1, z1:p2, z2:p3, z3:p4) -> (s:S14, o:'v,     -- 4th pixel
                                  x[4][3]:x[4][2], x[4][2]:x[4][1], x[4][1]:x[4][0], x
                                      [4][0]:p4,
                                  x[3][3]:x[3][2], x[3][2]:x[3][1], x[3][1]:x[3][0], x
                                      [3][0]:p3,
                                  x[2][3]:x[2][2], x[2][2]:x[2][1], x[2][1]:x[2][0], x
                                      [2][0]:p2,
                                  x[1][3]:x[1][2], x[1][2]:x[1][1], x[1][1]:x[1][0], x
                                      [1][0]:p1,
                                  x[0][3]:x[0][2], x[0][2]:x[0][1], x[0][1]:x[0][0], x
                                      [0][0]:p0,
                                  oz3:p3, oz2:p2, oz1:p1, oz0:p0)

| (s:S14, i:'p0, z0:p1, z1:p2, z2:p3, z3:p4) -> (s:S14,                       -- 5th
    and subsequent pixels of line
                                  o:'((k[0][0]*x[4][3] + k[0][1]*x[4][2] + k[0][2]*x
                                      [4][1] + k[0][3]*x[4][0] + k[0][4]*p4
                                    + k[1][0]*x[3][3] + k[1][1]*x[3][2] + k[1][2]*x
                                        [3][1] + k[1][3]*x[3][0] + k[1][4]*p3
                                    + k[2][0]*x[2][3] + k[2][1]*x[2][2] + k[2][2]*x
                                        [2][1] + k[2][3]*x[2][0] + k[2][4]*p2
                                    + k[3][0]*x[1][3] + k[3][1]*x[1][2] + k[3][2]*x
                                        [1][1] + k[3][3]*x[1][0] + k[3][4]*p1
                                    + k[4][0]*x[0][3] + k[4][1]*x[0][2] + k[4][2]*x
                                        [0][1] + k[4][3]*x[0][0] + k[4][4]*p0)>>n),
                                  x[4][3]:x[4][2], x[4][2]:x[4][1], x[4][1]:x[4][0], x
                                      [4][0]:p4,
                                  x[3][3]:x[3][2], x[3][2]:x[3][1], x[3][1]:x[3][0], x
                                      [3][0]:p3,
                                  x[2][3]:x[2][2], x[2][2]:x[2][1], x[2][1]:x[2][0], x
                                      [2][0]:p2,
                                  x[1][3]:x[1][2], x[1][2]:x[1][1], x[1][1]:x[1][0], x
                                      [1][0]:p1,
                                  x[0][3]:x[0][2], x[0][2]:x[0][1], x[0][1]:x[0][0], x
                                      [0][0]:p0,
                                  oz3:p3, oz2:p2, oz1:p1, oz0:p0)
| (s:S14, i:'>)               -> (s:S9, o:'>)
;

net conv255 (kernel,norm,pad) i = let rec (o,z0,z1,z2,z3) = conv255a (kernel,norm,pad)
    (i,z0,z1,z2,z3) in o;

-- ---------------------------------------------------------------------------
-- CCONV233 : centered 3x3 convolution on 2D images implemented as a single
--  actor (with external fifos for line delays)
-- CCONV233(k,n,v):<<x_11,...,x_1n>,...<x_m1,...x_mn>> = <<f(x_11),...,f(x_1n)>,...<f(
    x_m1),...f(x_mn)>>
-- where
--    f(x_{i,j}) = (k_{0,0}*x_{i-1,j-1} + k_{0,1}*x_{i_1,j} + k_{0,2}*x_{i-1,j+1}
--                + k_{1,0}*x_{i,j-1} + k_{1,1}*x_{i,j} + k_{1,2}*x_{i,j+1}
--                + k_{2,0}*x_{i+1,j-1} + k_{2,1}*x_{i+1,j} + k_{2,2}*x_{i+1,j+1}) >> n
--   (with x_{0,j}=x{m+1,j}=x{i,0}=x{i,n+1}=v)
-- where :
--   +-----+-----+-----+   +---------------+-------------+--------------+
--   | k00 | k01 | k02 |   | x_{i-1,j-1}   | x_{i-1,j}   | x_{i-1,j+1}  |
--   +-----+-----+-----+   +---------------+-------------+--------------+
--   | k10 | k11 | k12 | * | x_{i,j-1}     | x_{i,j} <---|-x_{i,j+1}----|-- current pixel
--   +-----+-----+-----+   +---------------+-------------+--------------+
--   | k20 | k21 | k22 |   | x_{i+1,j-1}   | x_{i+1,j}   | x{i+1,j+1}   |
```

```
-- +------+------+------+   +-----------+----------+-----------+
-- Warning : this actor requires horizontal and vertical blanking (>3 pixels between
    rows,
--          >line_width between images)
--
   ------------------------------------------------------------------------------------------


actor cconv233a (k:int<s,m> array[3][3], n:int, v:int<s,m>)
  in (i:int<s,m> dc,          -- input
      z0:int<s,m> dc,         -- previous line  (fed back through an external link)
      z1:int<s,m> dc)         -- pre-previous line  (fed back through an external link)
  out (o:int<s,m> dc,         -- output
       oz0:int<s,m> dc,       -- previous line  (fed back through an external link)
       oz1:int<s,m> dc)       -- previous line  (fed back through an external link)
var s : {SoF,SoL1,L1,SoL2,L2,SoL3,P1,P2,P3,P4,D1,D2,EoF} = SoF
var x : int<s,m> array[3][2]
-- +-----+-----+-----+   +---------+---------+-------+
-- | k00 | k01 | k02 |   | x[2][1] | x[2][0] | x6=p2 |
-- +-----+-----+-----+   +---------+---------+-------+
-- | k10 | k11 | k12 | * | x[1][1] | x[1][0] | x3=p1 |
-- +-----+-----+-----+   +---------+---------+-------+
-- | k20 | k21 | k22 |   | x[0][1] | x[0][0] | x0=p0 |
-- +-----+-----+-----+   +---------+---------+-------+
rules
| (s:SoF , i:'<)                    -> (s:SoL1, o:'<)
| (s:SoL1, i:'>)                    -> (s:SoF, o:'>)
| (s:SoL1, i:'<)                    -> (s:L1, oz0:'<)
| (s:L1, i:'p)                      -> (s:L1, oz0:'p)
| (s:L1, i:'>)                      -> (s:SoL2, oz0:'>)
| (s:SoL2, i:'<, z0:'<)             -> (s:L2, o:'<, oz1:'<, oz0:'<)
| (s:L2, i:'p0, z0:'p1)             -> (s:L2, o:'v, oz1:'p1, oz0:'p0)
| (s:L2, i:'>, z0:'>)               -> (s:SoL3, o:'>, oz1:'>, oz0:'>)
| (s:SoL3, i:'<, z0:'<, z1:'<)      -> (s:P1, o:'<, oz1:'<, oz0:'<)
| (s:SoL3, i:'>)                    -> (s:D1, oz1:'>, oz0:'>)
| (s:P1, i:'p0, z0:'p1, z1:'p2)  -> (s:P2, x[2][0]:p2, x[1][0]:p1, x[0][0]:p0, oz1:'p1,
     oz0:'p0)
| (s:P2, i:'p0, z0:'p1, z1:'p2)  -> (s:P3,
                                      o:'v,
                                      x[2][1]:x[2][0], x[2][0]:p2, x[1][1]:x[1][0],
                                          x[1][0]:p1, x[0][1]:x[0][0], x[0][0]:p0,
                                      oz1:'p1, oz0:'p0)
| (s:P3, i:'p0, z0:'p1, z1:'p2)  -> (s:P3,
                                      o:'((k[0][0]*x[2][1] + k[0][1]*x[2][0] + k
                                          [0][2]*p2
                                        + k[1][0]*x[1][1] + k[1][1]*x[1][0] + k
                                            [1][2]*p1
                                        + k[2][0]*x[0][1] + k[2][1]*x[0][0] +  k
                                            [2][2]*p0)>>n),
                                      x[2][1]:x[2][0], x[2][0]:p2, x[1][1]:x[1][0],
                                          x[1][0]:p1, x[0][1]:x[0][0], x[0][0]:p0,
                                      oz1:'p1, oz0:'p0)
| (s:P3, i:'>, z0:'>, z1:'>)        -> (s:P4, o:'v, oz1:'>, oz0:'>)
| (s:P4 )                           -> (s:SoL3, o:'>)        -- this requires horizontal
    blanking
| (s:D1, z0:'<, z1:'< )             -> (s:D2, o:'<)          -- this requires vertical
    blanking (at least > image width)
| (s:D2, z0:'x, z1:'y)              -> (s:D2, o:'v)
| (s:D2, z0:'>, z1:'>)              -> (s:EoF, o:'>)
| (s:EoF, z0:'>, z1:'>)             -> (s:SoF, o:'>)
;
```

```
net cconv233 (kernel,norm,pad) i = let rec (o,z0,z1) = cconv233a (kernel,norm,pad) (i,
    z0,z1) in o;

-- ---------------------------------------------------------------------------
-- CCONV255 : centered 5x5 convolution on 2D images implemented as a single
--   actor (with external fifos for line delays)
-- CCONV255(k,n,v):<<x_11,...,x_1n>,...<x_m1,...x_mn>> = <<f(x_11),...,f(x_1n)>,...<f(
    x_m1),...f(x_mn)>>
-- where
--   f(x_{i,j}) = (k_{0,0}*x_{i-2,j-2} + k_{0,1}*x_{i-2,j-1} + k_{0,2}*x_{i-2,j} + k_
    {0,3}*x_{i-2,j+1} + k_{0,4}*x_{i-2,j+2}
--               + k_{1,0}*x_{i-1,j-2} + k_{1,1}*x_{i-1,j-1} + k_{1,2}*x_{i-1,j} + k_
    {1,3}*x_{i-1,j+1} + k_{1,4}*x_{i-1,j+2}
--               + k_{2,0}*x_{i,j-2} + k_{2,1}*x_{i,j-1} + k_{2,2}*x_{i,j} + k_{2,3}*x_
    {i,j+1} + k_{2,4}*x_{i,j+2}
--               + k_{3,0}*x_{i+1,j-2} + k_{3,1}*x_{i+1,j-1} + k_{3,2}*x_{i+1,j} + k_
    {3,3}*x_{i+1,j+1} + k_{3,4}*x_{i+1,j+2}
--               + k_{4,0}*x_{i+2,j-2} + k_{4,1}*x_{i+2,j-1} + k_{4,2}*x_{i+2,j} + k_
    {4,3}*x_{i+2,j+1} + k_{4,4}*x_{i+2,j+2})>>n
--   (with x_{0,j}=x_{-1,j}=x_{m+1,j}=x_{m+2,j}=x_{i,0}=x_{i,-1}=x_{i,n+1}=x_{i,n+2}=v)
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
--   | k00 | k01 | k02 | k03 | k04 |   | x{i-2,j-2} | x{i-2,j-1} | x{i-2,j} | x{i-2,j+1}
     | x{i-2,j+2} |
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
--   | k10 | k11 | k12 | k13 | k14 |   | x{i-1,j-2} | x{i-1,j-1} | x{i-1,j} | x{i-1,j+1}
     | x{i-1,j+2} |
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
--   | k20 | k21 | k22 | k23 | k24 | * |  x{i,j-2}  |  x{i,j-1}  |  x{i,j}  |  x{i,j+1}
     |  x{i,j+2}  |
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
--   | k30 | k31 | k32 | k33 | k34 |   | x{i+1,j-2} | x{i+1,j-1} | x{i+1,j} | x{i+1,j+1}
     | x{i+1,j+2} |
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
--   | k40 | k41 | k42 | k43 | k44 |   | x{i+2,j-2} | x{i+2,j-1} | x{i+2,j} | x{i+2,j+1}
     | x{i+2,j+2} |
--   +-----+-----+-----+-----+-----+
    +-----------+-----------+----------+-----------+-----------+
-- Warning : this actor requires horizontal and vertical blanking (>3 pixels between
    rows,
--           >line_width between images)
--
    ---------------------------------------------------------------------------------------


actor cconv255a (k:int<s,m> array[5][5], n:int, v:int<s,m>)
  in (i:int<s,m> dc,        -- input
      z0:int<s,m> dc,       -- previous lines  (fed back through an external link)
      z1:int<s,m> dc,
      z2:int<s,m> dc,
      z3:int<s,m> dc)
  out (o:int<s,m> dc,       -- output
       oz0:int<s,m> dc,     -- previous lines  (fed back through an external link)
       oz1:int<s,m> dc,
       oz2:int<s,m> dc,
       oz3:int<s,m> dc)
var s : {SoF,SoL1,L1,SoL2,L2,SoL3,L3,SoL4,L4,SoL5,P1,P2,P3,P4,P5,E1,E2,F11,F12,F21,F22,
    EoF} = SoF
var x : int<s,m> array[5][4]
```

```
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k00 | k01 | k02 | k03 | k04 |    | x[4][3] | x[4][2] | x[4][1] | x[4][0] |     p4
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k10 | k11 | k12 | k13 | k14 |    | x[3][3] | x[3][2] | x[3][1] | x[3][0] |     p3
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k20 | k21 | k22 | k23 | k24 | *  | x[2][3] | x[2][2] | x[2][1] | x[2][0] |     p2
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k30 | k31 | k32 | k33 | k34 |    | x[1][3] | x[1][2] | x[1][1] | x[1][0] |     p1
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
--  | k40 | k41 | k42 | k43 | k44 |    | x[0][3] | x[0][2] | x[0][1] | x[0][0] |     p0
    |
--  +-----+-----+-----+-----+-----+
    +---------+---------+---------+---------+---------+
rules
| (s:SoF, i:'<)               -> (s:SoL1, o:'<)               -- Start of Image
| (s:SoL1, i:'>)              -> (s:SoF, o:'>)                -- End of image

| (s:SoL1, i:'<)              -> (s:L1, oz0:'<)               -- Start of first
   line
| (s:L1, i:'p0)               -> (s:L1, oz0:'p0)              -- Read first line
   and store in fifo z0
| (s:L1, i:'>)                -> (s:SoL2, oz0:'>)             -- End of first line

| (s:SoL2, i:'<, z0:'<)       -> (s:L2, oz1:'<, oz0:'<)       -- Start of second
   line
| (s:L2, i:'p0, z0:'p1)       -> (s:L2, oz1:'p1, oz0:'p0)     -- Read second line,
   store in fifo z0 while moving z0 to z1
| (s:L2, i:'>, z0:'>)         -> (s:SoL3, oz1:'>, oz0:'>)     -- End of second line

| (s:SoL3, i:'<, z0:'<, z1:'<)  -> (s:L3, o:'<, oz2:'<, oz1:'<, oz0:'<)     -- Start of
    third line
| (s:L3, i:'p0, z0:'p1, z1:'p2) -> (s:L3, o:'v, oz2:'p2, oz1:'p1, oz0:'p0)
                                                          -- Read third line,
                                                             store in fifo z0
                                                             while moving z0 to
                                                             z1 and z1 to z2
| (s:L3, i:'>, z0:'>, z1:'>)    -> (s:SoL4, o:'>, oz2:'>, oz1:'>, oz0:'>)  -- End of
   third line

| (s:SoL4, i:'<, z0:'<, z1:'<, z2:'<)     -> (s:L4, o:'<, oz3:'<, oz2:'<, oz1:'<, oz0
   :'<)  -- Start of 4th line
| (s:L4, i:'p0, z0:'p1, z1:'p2, z2:'p3)   -> (s:L4, o:'v, oz3:'p3, oz2:'p2, oz1:'p1,
   oz0:'p0)
                                                          -- Read 4th line, store
                                                             in fifo z0 while
                                                             moving z0 to z1, z1
                                                             to z2 and z2 to z3
| (s:L4, i:'>, z0:'>, z1:'>, z2:'>)       -> (s:SoL5, o:'>, oz3:'>, oz2:'>, oz1:'>, oz0
   :'>)  -- End of 4th line

| (s:SoL5, i:'<, z0:'<, z1:'<, z2:'<, z3:'<)  -> (s:P1, o:'<, oz3:'<, oz2:'<, oz1:'<,
   oz0:'<)
| (s:SoL5, i:'>)                              -> (s:F11)
```

```
| (s:P1, i:'p0, z0:'p1, z1:'p2, z2:'p3, z3:'p4) -> (s:P2,             -- First pixel of
   line
                                 x[4][0]:p4,
                                 x[3][0]:p3,
                                 x[2][0]:p2,
                                 x[1][0]:p1,
                                 x[0][0]:p0,
                                 oz3:'p3, oz2:'p2, oz1:'p1, oz0:'p0)

| (s:P2, i:'p0, z0:'p1, z1:'p2, z2:'p3, z3:'p4) -> (s:P3,             -- Second pixel
                                 x[4][1]:x[4][0], x[4][0]:p4,
                                 x[3][1]:x[3][0], x[3][0]:p3,
                                 x[2][1]:x[2][0], x[2][0]:p2,
                                 x[1][1]:x[1][0], x[1][0]:p1,
                                 x[0][1]:x[0][0], x[0][0]:p0,
                                 oz3:'p3, oz2:'p2, oz1:'p1, oz0:'p0)

| (s:P3, i:'p0, z0:'p1, z1:'p2, z2:'p3, z3:'p4) -> (s:P4, o:'v,      -- Third pixel
                                 x[4][2]:x[4][1], x[4][1]:x[4][0], x[4][0]:p4,
                                 x[3][2]:x[3][1], x[3][1]:x[3][0], x[3][0]:p3,
                                 x[2][2]:x[2][1], x[2][1]:x[2][0], x[2][0]:p2,
                                 x[1][2]:x[1][1], x[1][1]:x[1][0], x[1][0]:p1,
                                 x[0][2]:x[0][1], x[0][1]:x[0][0], x[0][0]:p0,
                                 oz3:'p3, oz2:'p2, oz1:'p1, oz0:'p0)

| (s:P4, i:'p0, z0:'p1, z1:'p2, z2:'p3, z3:'p4) -> (s:P5, o:'v,     -- 4th pixel
                                 x[4][3]:x[4][2], x[4][2]:x[4][1], x[4][1]:x[4][0], x
                                     [4][0]:p4,
                                 x[3][3]:x[3][2], x[3][2]:x[3][1], x[3][1]:x[3][0], x
                                     [3][0]:p3,
                                 x[2][3]:x[2][2], x[2][2]:x[2][1], x[2][1]:x[2][0], x
                                     [2][0]:p2,
                                 x[1][3]:x[1][2], x[1][2]:x[1][1], x[1][1]:x[1][0], x
                                     [1][0]:p1,
                                 x[0][3]:x[0][2], x[0][2]:x[0][1], x[0][1]:x[0][0], x
                                     [0][0]:p0,
                                 oz3:'p3, oz2:'p2, oz1:'p1, oz0:'p0)

| (s:P5, i:'p0, z0:'p1, z1:'p2, z2:'p3, z3:'p4) -> (s:P5,                            -- 5
   th and subsequent pixels of line
                                 o:'((k[0][0]*x[4][3] + k[0][1]*x[4][2] + k[0][2]*x
                                      [4][1] + k[0][3]*x[4][0] + k[0][4]*p4
                                   + k[1][0]*x[3][3] + k[1][1]*x[3][2] + k[1][2]*x
                                        [3][1] + k[1][3]*x[3][0] + k[1][4]*p3
                                   + k[2][0]*x[2][3] + k[2][1]*x[2][2] + k[2][2]*x
                                        [2][1] + k[2][3]*x[2][0] + k[2][4]*p2
                                   + k[3][0]*x[1][3] + k[3][1]*x[1][2] + k[3][2]*x
                                        [1][1] + k[3][3]*x[1][0] + k[3][4]*p1
                                   + k[4][0]*x[0][3] + k[4][1]*x[0][2] + k[4][2]*x
                                        [0][1] + k[4][3]*x[0][0] + k[4][4]*p0)>>n),
                                 x[4][3]:x[4][2], x[4][2]:x[4][1], x[4][1]:x[4][0], x
                                     [4][0]:p4,
                                 x[3][3]:x[3][2], x[3][2]:x[3][1], x[3][1]:x[3][0], x
                                     [3][0]:p3,
                                 x[2][3]:x[2][2], x[2][2]:x[2][1], x[2][1]:x[2][0], x
                                     [2][0]:p2,
                                 x[1][3]:x[1][2], x[1][2]:x[1][1], x[1][1]:x[1][0], x
                                     [1][0]:p1,
                                 x[0][3]:x[0][2], x[0][2]:x[0][1], x[0][1]:x[0][0], x
                                     [0][0]:p0,
                                 oz3:'p3, oz2:'p2, oz1:'p1, oz0:'p0)
```

```
| (s:P5, i:'>, z0:'>, z1:'>, z2:'>, z3:'>) -> (s:E1, o:'v, oz3:'>, oz2:'>, oz1:'>, oz0
   :'>)

| (s:E1)                                   -> (s:E2, o:'v)          -- Filling end of
   line (two padding pixels)
| (s:E2)                                   -> (s:SoL5, o:'>)

| (s:F11, z0:'<, z1:'<, z2:'<, z3:'< )     -> (s:F12, o:'<, oz0:'<)          -- Filling
   end of frame (two padding lines)
| (s:F12, z0:'p1, z1:'p2, z2:'p3, z3:'p4)  -> (s:F12, o:'v, oz0:'p1)
| (s:F12, z0:'>, z1:'>, z2:'>, z3:'>)      -> (s:F21, o:'>, oz0:'>)
| (s:F21, z0:'<)                           -> (s:F22, o:'<)
| (s:F22, z0:'p0)                          -> (s:F22, o:'v)
| (s:F22, z0:'>)                           -> (s:EoF, o:'>)

| (s:EoF)      -> (s:SoF, o:'>)        -- End of frame
;

net cconv255 (kernel,norm,pad) i = let rec (o,z0,z1,z2,z3) = cconv255a (kernel,norm,pad
   ) (i,z0,z1,z2,z3) in o;
```

Listing 11.5: The library stream_ops.cph

```
------------------------------------------
-- Basic operations on unstructured streams
-- 2015-07-17, JS
------------------------------------------

----------------------------------------------------------
-- D - One-sample delay
-- D(v):x1,x2,... = v,x1,x2,...
----------------------------------------------------------

actor d (v:$t)
  in (a:$t)
  out (c:$t)
var z : $t = v
rules
| a:x -> (c:z, z:x)
;

----------------------------------------------------------
-- Dk - k-sample delay on unstructured streams
-- Dk(k,v):x1,x2,..., = v,...,v,x1,x2,...
--                      \--k--/
----------------------------------------------------------

actor dka (k:int, v:$t)
  in (a:$t, b:$t)
  out (c:$t)
var s : {S0,S1} = S0
var i : int = 0
rules
| (s:S0, a:x) when i<k-1 -> (c:v, i:i+1)
| (s:S0, a:x)            -> (c:v, s:S1)
| (s:S1, a:x, b:y)       -> c:y
;

net dk k v i = dka (k,v) (i,i);

----------------------------------------------------------
-- sample - n->1 subsampling on unstructured streams
```

```
-- sample(n):x_1,x_2,..., = x_n,x_2n,...
--------------------------------------------------------------

actor sample (n: int)
  in (i:$t)
  out (o:$t)
var k : int = 0
rules
| i:x when k<n-1 -> k:k+1
| i:x when k=n-1 -> (o:x, k:0)
;
```

Chapter 12

Foreign function interfacing

This chapter describes how to make use of foreign functions in CAPH program using a simple example.

Consider the following program, in which the multiplication in the `scale` actor is performed by calling a foreign (external) function[1] :

Example :

```
function mult = extern "mult", "mult", "mult" :
  signed<s> * signed<s> -> signed<s>;

actor scale (k:signed<8>)
  in (a:signed<8>)
  out (c:signed<8>)
rules
| a:v -> c:mult(v,k)

stream i:signed<8> from "sample.txt";
stream o:signed<8> to "result.txt";

net o = scale [2] i;
```

Compiling this program with the **SystemC** back-end, with :

```
caph -systemc scale.cph
```

will produce (among others – see Chap. 10) a file `scale_act.cpp` containing the implementation of the SystemC module describing the `scale` actor and containing a call to a function `sc_int<8> mult(sc_int<8> x, sc_int<8> y)`. This function must be declared (resp. defined) in a file named `extfns.h` (resp. `extfns.cpp`). These two files must be accessible when compiling the SystemC executable. Here's a possible (and obvious) contents for these files :

File extfns.h

```
#ifndef _extfns_h
```

[1] The purpose of the example is only to demonstrate how to use the foreign function facilities; the `scale` can of course be written in a more straightforward manner using the builtin multiplication operator, like exemplified in Sec. 2.4.4 for example.

```
#define _extfns_h

sc_int<8> mult(sc_int<8> x, sc_int<8> y);

#endif
```

File extfns.cpp

```
#include <systemc.h>

sc_int<8> mult(sc_int<8> x, sc_int<8> y)
{
  sc_int<8> res = x*y;
  return res;
}
```

Things are similar for the **VHDL** backend. Here the generated file `scale_act.vhd` will refer to package named `work.extfns` which is supposed to contain the definition of the `mult` function. Here's a possible definition for the corresponding file :

File extfns.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package extfns is
  function mult(a, b: signed(7 downto 0)) return signed;
end extfns;

package body extfns is
  function mult(a, b: signed(7 downto 0)) return signed is
    variable res : signed(15 downto 0);
  begin
     res := a * b;
     return res(7 downto 0);
  end;
end package body extfns;
```

Programs making using of external functions have to provide a Caml implementation of the corresponding functions in order to be simulated[2]. Integration of this code within the simulator is done is done using the dynamic linking facilities offered by the Caml runtime[3]. For this a file `extfns.ml` must be written (and accessible from the directory from which the simulator is run), "registering" the corresponding functions. Here's the contents of this file for the `scale` example :

File extfns.ml

```
let mult [x;y] = x * y
let _ = Foreign.register "mult" mult
```

[2]In theory, the simulator could use the C++ implementation but this is currently not implemented.
[3]simply because the simulator itself is written ... in Caml.

The first argument to the function `Foreign.register` is the name by which the `mult` function will be called in the CAPH program (same name here, but another name could be used in case of name conflicts).

Due to technical limitations, interfacing of foreign functions at the simulator level is limited to functions taking and returning integer type(s).

Chapter 13

Compiler options

Compiler usage : `caphc [options...] file`

General options

-prefix	set prefix output file names (default is main source file basename)
-I	add path to the list of dirs to search for include
-dump_tenv	dump builtin typing environment (for debug only)
-dump_typed	dump typed program (for debug only)
-phantom_types	print sized types using underlying representation (not for the casual user)
-dump_senv	dump static environment (for debug only)
-dump_boxes	dump static representation of boxes
-dump_fsms	dump a graphical representation (.dot) of stateful boxes
-dump_ir	dump intermediate representation (just before backends))
-dump_denv	dump dynamic environment (for debug only)
-suppress_cast_warnings	do not print warnings for dubious type casts
-show_signness	display integer constants with signness suffix (S,U)
-flat_variants	display constructed values without parens
-sim	run the program through the simulator
-dot	generate .dot representation of the program
-xdf	generate .xdf representation of the network and .cal descriptions of the actors
-dif	generate .dif representation of the program
-systemc	activate the SystemC backend
-vhdl	activate the VHDL backend
-target_dir	set target directory for generated files (default is current directory)
-D	define macro symbol
-make	generate makefile dependencies
-proj_file	set project file name (default: same as main source file)
-version	print version of the compiler
–v	print version of the compiler
-ignore_pragmas	ignore all pragma directives
-split_output_frames	generate separate files when outputing data structured as frames (simulation and systemc only)
-restrict_inputs	undocumented
-restrict_outputs	undocumented

DOT-specific options

-dot_unlabeled_edges	do not annotate graph edges
-dot_wire_annots	print wire annotations (phase/fifo_size) when available (implies [-dot_show_indexes])
-dot_unboxed_ios	do not outline io boxes with a triangle shape
-dot_show_indexes	print box and wire indexes
-dot_simple_boxes	print boxes without i/o slots

Simulation-specific options

-infer_mocs	compute model of computation for boxes
-dump_sdf_fifo_sizes	dump statically computed FIFO sizes for SDF graphs (implies [-infer_mocs])
-chan_cap	set default capacity for channel (default:256)
-warn_channels	emit a warning when channels are full
-dump_channel_stats	dump channel statictics (max occ,...) after run
-abbrev_dc_ctors	use abbreviated syntax when reading/writing values with of type [t dc] from/to file
-trace	run in trace mode
-trace_ports	trace change of values at IO ports (simulation and SystemC)
-stop_after	stop after n run cycles
-stdin	redirect stdin from file
-stdout	redirect stdout to file
-sim_extra	add file to the list of external modules for the interpreter
-absint	run the abstract interpreter on the given box
-ai_max_cycles	set the maximum number of cycles when performing abstract interpretation of boxes (default:32)

SystemC-specific options

-sc_stop_time	stop after n ns
-sc_io_monitor	dump i/o start and stop times to file
-sc_io_monitor_file	set file for dumping i/o start times (default: io_monitor.dat)
-sc_stop_when_idle	stop when outputs have been inactive for 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)
-sc_use_int	use int for representing signed and unsigned values
-sc_abbrev_dc_ctors	use abbreviated syntax when reading/writing values with of type [t dc] from/to file
-sc_istream_period	set pixel period for input streams in the testbench (in clock periods, default=2)
-sc_istream_hblank	set horizontal blanking for input streams (in clock cycles)
-sc_istream_vblank	set vertical blanking for input streams (in clock cycles)
-sc_extra	add file to the list of external modules for the SystemC backend

VHDL-specific options

-vhdl_num_lib	set library for handling numerical operations (default: ieee.numeric_std)
-vhdl_annot_file	give the name of the back-annotation file
-vhdl_fifo_offset	add offset to each FIFO size when read in back-annotation file (default value: 2)
-vhdl_default_int_size	set default size for ints (default: 8)
-vhdl_default_fifo_capacity	set default fifo capacity (default: 4)
-vhdl_small_fifo_model	set model for small fifos (default: fifo)
-vhdl_big_fifo_model	set model for big fifos (default: fifo)
-vhdl_fifo_model_threshold	set threshold for switching between fifo models
-vhdl_extra	add file to the list of external modules for the VHDL backend
-vhdl_reset_duration	set duration of the reset signal (ns)
-vhdl_clock_period	set clock period (ns) (default: 10)
-vhdl_seq_delay	set propagation time in sequentiel logic (ns) (default: 1)
-vhdl_istream_period	set pixel period for input streams in the testbench (in clock periods, default is 1)
-vhdl_istream_blanking	activate blanking when reading .bin source files for input streams
-vhdl_istream_skew	set clock skew when reading .bin source files for input streams (default: 0)
-vhdl_trace	add diag pins in vhdl code (actor states and fifo caps)
-vhdl_warn_on_unsized_consts	warn whenever the size of integer constants cannot be determined (default: false)
-vhdl_use_native_mult	use builtin operator for multiplication (warning: this may cause bound check failures)
-vhdl_init_array_at_decl	initialize arrays at declaration (warning: this may be not supported by the synthetizer)
-vhdl_float_support	enable float support
-vhdl_io_converters	generate C programs for reading/writing input/output files involving user-defined types
-vhdl_rename_io_wires	rename IO wires in the _net.vhd and _tb.vhd files according to the connected IO stream/po
-vhdl_tb_external_clock	make clock and reset input signals for the generated testbench (default: false)
-vhdl_tb_inline_io	use inline processes for I/O in the generated testbench instead of reading/writing files (defau

Chapter 14

Implementation of variant types

This chapter gives a very brief overview of how CAPH variant type declarations are translated into SystemC and VHDL code. The casual programmer is normally not concerned with this.

For this, we take the example of the (polymorphic) `option` type introduced in Sec. 2.4.1 :

```
type $t option =
  Absent
| Present of $t
```

The code generated by the SystemC backend when this type is instanciated with `t=signed<8>`, for example, is given in listing 14.1[1]. The underlying representation is basically a structure with one field for the tag and the other for the associated data. The inlined `iostream` operators are used to read (resp. write) input and output data to text files.

Listing 14.1: Code generated by the SystemC backend for the `option` type declaration

```
1  class t_option_int8 {
2  public:
3    enum t_tag { Absent, Present };
4    struct t_data { sc_int<8> present;  };
5    struct { t_tag tag; t_data data; } repr;
6    ~t_option() { };
7    t_option(void) { };
8    t_option(_tag<Absent>) { repr.tag = Absent; };
9    t_option(_tag<Present>, sc_int<8> v) { repr.tag = Present; repr.data.
         present = v; };
10   inline friend ::std::ostream& operator << ( ::std::ostream& os, const
         t_option<sc_int<8> >& v) {
11     switch ( v.repr.tag ) {
12       case Absent: os << "Absent "; return os;
13       case Present: os << "Present " <<  v.repr.data.present; return os;
14       }
15     }
16   inline friend ::std::istream& operator >> ( ::std::istream& is, t_option
         <sc_int<8> >& v) {
17     char tmp[64];
```

[1] This code is produced in file `xxx_globals.h`, where `xxx` is the name of the top module.

```
18       is >> tmp;
19       if ( is.good() ) {
20         if ( !strcmp(tmp,"Absent") ) { v.repr.tag=Absent; return is;}
21         if ( !strcmp(tmp,"Present") ) { v.repr.tag=Present; is >> v.repr.
             data.present; return is;}
22         }
23       }
24 };
```

The code generated by the VHDL backend[2] is given in listing 14.2. The generated package includes three kinds of functions :

- inspectors, for testing if the binary representation[3] of a value matches a given tag (`is_xxx`),
- injectors, for building the binary representation of an `option` value (`mk_xxx`),
- extractors, for extracting the data part of a constructed value (`get_xxx`).

As can be observed[4], variant types are encoded by concatenating a *tag* part and a *data* part.

The *tag* part (MSBs) contains an integer representing the tag of the encoded value. By default, this integer is 0 for the first tag, 1 for the second, *etc.* It is possible to specify a custom encoding when declaring the type[5]. For example, to have the `Absent` tag represented as 1 and the `Present` tag represented as 0, one could have defined the `option` type as :

```
type $t option =
  Absent %1
| Present %0 of $t
```

The data associated to each distinct tag are encoded in the *data part* (LSBs).

Listing 14.2: Code generated by the VHDL backend for the `option` type declaration

```
1  package option_int8 is
2    function is_absent(t: std_logic_vector) return boolean;
3    function is_present(t: std_logic_vector) return boolean;
4    constant mk_absent: std_logic_vector(8 downto 0) := "0" &
         to_std_logic_vector(to_signed(0,8),8);
5    function mk_present(d: signed(7 downto 0)) return std_logic_vector;
6    function get_present(t: std_logic_vector) return signed;
7  end option_int8;
8
9  package body option_int8 is
10
11   function is_absent(t: std_logic_vector) return boolean is
12   begin
13     return t(8 downto 8) = "0";
14   end;
```

[2]In file `xxx_types.vhd`, where `xxx` is the name of the top module.

[3]A `std_logic_vector`

[4]And detailed in Tab. 2.5

[5]This feature is typically used when the values of the variant type are read/written by dedicated processes at the VHDL level, which must know the actual bit-level encoding of values.

```
15
16    function is_present(t: std_logic_vector) return boolean is
17    begin
18      return t(8 downto 8) = "1";
19    end;
20
21    function mk_present(d: signed(7 downto 0)) return std_logic_vector is
22    begin
23      return "1" & to_std_logic_vector(d,8);
24    end;
25
26    function get_present(t: std_logic_vector) return signed is
27    begin
28      return from_std_logic_vector(t(7 downto 0),8);
29    end;
30
31  end package body option_int8;
```

Appendix A

Writing your own FIFO model

TBW

Appendix B

The `txt2bin` command

NAME

txt2bin - convert CAPH text input files to text-encoded binary format

SYNOPSIS

txt2bin [-eventf] [-dc] [-abbrev] [-hblank n] [-vblank n] [-out file] format bitwidth file(s)

DESCRIPTION

This program can be used to convert CAPH text input files to text-encoded binary format to be used by the testbenches generated by the VHDL backend of the CAPH compiler

ARGUMENTS

format

type of data tokens to be read in the input file: `uint`, `sint`, `float` or `bool`

bitwidth

size, in bits, of the data words to write in the output file

file(s)

name of the input file(s); if several files are specified, the results of converting each of them will be concatenated in the output file

OPTIONS

-eventf

process *event files*, describing timed sequence of events for input *ports* (default is to process *stream files*, describing untimed sequence of tokens for input *streams*)

-dc

encode tokens having type `t dc` with the following convention: `11xxxxxx` for `Data v`, `01xxxxxx` for `SoS` and `10xxxxxx` for `SoS`

-abbrev

use abbreviated syntax for tokens having type `t dc` in the input file (`v` for `Data v`, < for `SoS` and > for `EoS`)

-hblank n

insert *n* "no data" extra tokens (`00xxxxxx`) in the output file after each `EoS` token (end of line)

-vblank n

insert *n* "no data" extra tokens (`00xxxxxx`) in the output file after each pair of successive `EoS` token (end of frame)

-out file

write result in file *file* (default is to write on *stdout*)

EXAMPLES

The following command will convert the file *sample.txt*, containing the representation of an stream of type `signed<16>`, writting the result in file *sample.bin* :

```
txt2bin -out sample.bin sint 16 sample.txt
```

The following command will convert a sequence of file *im1.txt*, ..., *im8.txt*, each containing the representation of an image encoded with the type `unsigned<8 dc>`, inserting 4 blank tokens at the end of each line and 16 blank tokens at the end of each image, and writting the result in file *imgs.bin* :

```
txt2bin -dc -abbrev -out imgs.bin -hblank 4 -vblank 16 uint 8 im[1-8].txt
```

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

CAVEAT

The program can only convert files containing tokens with a scalar type (signed or unsigned int, float32 or boolean) or with type `t dc`, where `t` is a scalar type. Arrays and user-defined variant types are not supported.

When converting tokens of type `t dc`, the size in bits of individual words in the output file is *n+2*, where *n* is the specified bitwidth. The two extra bits are used to encode the tag.

Using the special value `00xxxxx` to handle blanking is a hack. It can also increase significantly the size of the output file.

When using the *float* format, the only accepted *bitwidth* is 32. Moreover, the program may not work correctly in this case on platforms on which the size of `unsigned int` and `float` (as reported by the C operator `sizeof` is not 4 bytes (32 bits).

The program cannot direcly read image file in PGM format. For this, use the *pgm2bin* program.

Appendix C

The bin2txt command

NAME

bin2txt - convert CAPH text-encoded binary format to text format

SYNOPSIS

txt2bin [-dc] [-abbrev] [-out <file>] [-split_frames] format bitwidth file

DESCRIPTION

This program can be used to convert the *.bin* files read or produced by the VHDL backend of the CAPH compiler to text format

ARGUMENTS

format

type of data tokens (`uint`, `sint`, `float` or `bool`)

bitwidth

size, in bits, of the data words to read in the input file

file

name of the input file

OPTIONS

-dc

decode tokens having type `t dc` with the following convention: `11xxxxxx` for `Data v`, `01xxxxxx` for `SoS` and `10xxxxxx` for `SoS`

-abbrev

for tokens having type `t dc`, write the result using the abbreviated syntax in the output file (`v` for `Data(v)`, "<" for `SoS` and ">" for `EoS`)

-out file

write result in file *file* (default is to write on *stdout*)

-split_frames

if the sequence of input tokens, having type `t dc`, represent a sequence of images (frames), write each result image in a separate file; the resulting files will be named *xxx_1*, *xxx_2*, etc.. where *xxx* is the name given under the *-out* option

EXAMPLES

The following command will convert the file *result.bin*, containing the representation of a stream of type `signed<16>`, writting the result in file *result.txt* :

```
bin2txt -out result.txt sint 16 result.bin
```

The following command will convert the file *result.bin*, containing the representation of a sequence of 8 images, each encoded with the type `unsigned<8>dc`, writting the result in files *result_1.txt*, ..., *result_8.txt* :

```
txt2bin -dc -abbrev -out result.txt -split_frames uint 8 result.bin
```

EXIT STATUS

The program returns a zero exit status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

CAVEAT

The program can only convert files containing tokens with a scalar type (signed or unsigned int, float32 or boolean) or with type `t dc`, where `t` is a scalar type. Arrays and user-defined variant types are not supported.

When converting tokens of type `t dc`, the size in bits of individual words in the input file should $n+2$, where n is the specified bitwidth. The two extra bits are used to encode the tag.

When using the *float* format, the only accepted *bitwidth* is 32. Moreover, the program may not work correctly in this case on platforms on which the size of `unsigned int` and `float` (as reported by the C operator `sizeof` is not 4 bytes (32 bits).

When using the *bool* format, the only accepted *bitwidth* is 1.

The program cannot direcly write image file(s) in PGM format. To have a VHDL testbench generate PGM files, the *txt2pgm* program has be used on each generated text file.

When using the *-split_frames* option, an extra, empty file is sometimes generated.

The result of using the *-split_frames* when the input file does not actually contains a sequence of dc-encoded images is undefined.

Appendix D

The `pgm2txt` command

NAME

pgm2txt - convert PGM image files to CAPH text input files

SYNOPSIS

pgm2txt [-abbrev] in_file out_file

DESCRIPTION

This program can be used to convert PGM files to text files which can be read by the CAPH interpreter and the testbench generated by the SystemC backend.

ARGUMENTS

in_file

name of the input file

out_file

name of the output file

OPTIONS

-abbrev

use abbreviated syntax for tokens in the ouput file (`v` for `Data v`, < for `SoS` and > for `EoS`)

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

Appendix E

The `txt2pgm` command

NAME

txt2pgm - convert CAPH text input/output files to PGM files

SYNOPSIS

txt2pgm [-abbrev] maxv in_file out_file

DESCRIPTION

This program can be used to convert text files to PGM files, to be viewed by image viewers.

ARGUMENTS

maxv

maximum pixel value, to be specified in the PGM file header

in_file

name of the input file

out_file

name of the output file

OPTIONS

-abbrev

use abbreviated syntax for tokens in the input file (`v` for `Data v`, < for `SoS` and > for `EoS`)

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

Appendix F

The pgm2bin command

NAME

pgm2bin - convert PGM image files to text-encoded binary format

SYNOPSIS

pgm2bin [-hblank n] [-vblank n] bitwidth in_file out_file

DESCRIPTION

This program can be used to convert PGM files to text-encoded binary format to be used by the testbenches generated by the VHDL backend of the CAPH compiler

ARGUMENTS

bitwidth

size, in bits, of the data words to write in the output file (including the two extra bits for tagging)

in_file

name of the input file

out_file

name of the output file

OPTIONS

-hblank n

insert *n* "no data" extra tokens (`00xxxxxx`) in the output file after each `EoS` token (end of line)

-vblank n

insert *n* "no data" extra tokens (`00xxxxxx`) in the output file after each pair of successive `EoS` token (end of frame)

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

CAVEAT

The size in bits of individual words in the output file is $n+2$, where n is the specified bitwidth. The two extra bits are used to encode the tag.

Using the special value `00xxxxx` to handle blanking is a hack. It can also significantly increase the size of the output file.

Appendix G

The `bin2pgm` command

NAME

bin2pgm - convert CAPH text-encoded binary format to PGM file

SYNOPSIS

bin2pgm bitwidth ifile ofile

DESCRIPTION

This program can be used to convert the *.bin* files read or produced by the VHDL backend of the CAPH compiler to to be viewed by image viewers.

ARGUMENTS

bitwidth

size, in bits, of the data words to read in the input file

ifile

name of the input file

ofile

name of the output file

EXIT STATUS

The program returns a zero exit status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

Appendix H

The `mkdcimg` command

NAME

mkdcimg - make text representation of test images using the dc format

SYNOPSIS

mkdcimg [-const val | -linear max | -grid width] [-out file] [-abbrev] nrows ncols

DESCRIPTION

This program can be used to create test images stored in the text format used by the CAPH interpreter and the backends generated by the SystemC backend.

ARGUMENTS

nrows

number of rows in the output image

ncols

number of columns in the output image

OPTIONS

-const val

the image is composed of pixels all having value *val*

-linear maxv

the ith pixel of the image (starting at column 0 of row 0) has value *i mod maxv*

-grid width

the pixel at column j of row i has value *((i+1)*width+(j+1)*

-out file

write result in file *file* (default is to write on *stdout*)

-abbrev

write the result image using the abbreviated syntax (`v` for `Data v`, < for `SoS` and > for `EoS`)

EXAMPLES

The following command

```
mkdcimg -const 1 -abbrev 4 4
```

produces this image :

```
< < 1 1 1 1 > < 1 1 1 1 > < 1 1 1 1 > < 1 1 1 1 > >
```

The following command

```
mkdcimg -linear 8 -abbrev 4 4
```

produces this image :

```
< < 1 2 3 4 > < 5 6 7 0 > < 1 2 3 4 > < 5 6 7 0 > >
```

The following command

```
./mkdcimg -grid 10 -abbrev -out foo.txt 4 4
```

produces this image, in file *foo.txt* :

```
< < 11 12 13 14 > < 21 22 23 24 > < 31 32 33 34 > < 41 42 43 44 > >
```

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

Appendix I

The mkconv command

NAME

mkdcimg - generate MxN convolution and neighborhood generation actors

SYNOPSIS

mkconv [-name name] [-o file] -dim d -mode (shifted|centered|neigh) -h nr -w nc"

DESCRIPTION

This program can be used to generate the description in CAPH of MxN convolution and neighborhood generation actors.

ARGUMENTS

-w nc

kernel / neighborhood width (number of columns)

OPTIONS

-o file

write result in file *file* (default is to write on *stdout*)

-name name

name of the generated actor (default is *(conv|cconv|neigh)*<*dim*<h><w>a>)

-mode m

kind of actor generated (*m=shifted* : shifted convolution, *m=centered* : centered convolution, *m=neigh* : neighborhood generator) (default: shifted convolution)

-dim d

input signal dimension (1 for 1D signals, 2 for 2D images) (default: 1)

-h nr

kernel / neighborhood height (number of rows) when *dim*=2

EXAMPLES

The following command

mkconv -dim 1 -mode shifted -w 3

generates (on stdout) the description of an actor *conv113* computing the shifted *1x3* convolution of 1D signals (*i.e.* lists)

The following command

mkconv -o myconv.cph -name conv55 -dim 2 -mode centered -h 5 -w 5

generates, in file *myconv.cph*, the description of an actor *conv55* computing the centered *5x3* convolution operating on images

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

CAVEAT

The generated convolution actors (shifted or centered) only operate on size and sign generic integers but the sign and size of the inputs, kernel coefficients and outputs must be the same. It is not passible, for example, to take a (structured) stream a signed 8-bit integers, convoluate it with a kernel made of unsigned 4-bit integers and produce a stream of 12-bit signed integers.

Some predefined versions of the actors which can be generated by this program can be found in the CAPH library, in files *convol.cph* and *neigh.cph*. A detailed description of the convolution and neighborhood generation operations is given in these files.

This program appeared in version 2.7.2 of the CAPH distribution.

Appendix J

The `caphmake` command

NAME

caphmake - Makefile generator for CAPH projects

SYNOPSIS

caphmake [-main name] [-caph_dir path] [-o file] [file]

DESCRIPTION

This program reads *.proj* and *.cph* files and generates top-level Makefiles for CAPH projects.

ARGUMENTS

file

name of the project file (default is *main.proj*)

OPTIONS

-main name

set name of the top-level CAPH source file (default is *main.cph*)

-caph_dir path

set path to the CAPH install directory (default: got from the CAPH environment variable)

-o file

write result in file *file* (default is to write on *Makefile*)

EXIT STATUS

The program returns a zero exist status if it succeeds and a non zero value in case of failure. In the latter case, an error message is printed to stderr.

Bibliography

[1] Graphviz - graph visualization software.

[2] The objective caml language.

[3] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982.

[4] T.M. Parks E.A. Lee. Dataflow process networks. *Proceedings of The IEEE*, 83:773–801, 1995.

[5] J. Eker and J. Janneck. Cal language report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003.

[6] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 37–56, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[7] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[8] S. North and E. Koutsofios. Applications of graph visualization. In *Graphics Interface*, Banff, Alberta, 1994.

[9] T.M. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, UCB, 1995.

[10] J. Sérot. The semantics of a purely functional graph notation system. In *9th Symposium on Trends in Functional Programming*, 2008.

[11] Jocelyn Sérot, Georges Quénot, and Bertrand Zavidovique. A visual dataflow programming environment for a real time parallel vision machine. *J. Vis. Lang. Comput.*, 6(4):327–347, 1995.

[12] Matthieu Wipliez and Mickaël Raulet. Classification of dataflow actors with satisfiability and abstract interpretation. *Int. J. Embed. Real-Time Commun. Syst.*, 3(1):49–69, January 2012.