HoCL v1.1
Semantics

J. Sérot

Chapter 1

Abstract syntax

This is the abstract syntax used to formalize the typing rules and static semantics in chapters 2 an 3.

$$
\begin{array}{rcl}
\langle\text{program}\rangle & ::= & \langle\text{type\_decl}\rangle^{*}\ \langle\text{val\_decl}\rangle^{*}\ \langle\text{node\_decl}\rangle^{*} \\[1ex]
\langle\text{type\_decl}\rangle & ::= & \textbf{type}\ \text{IDENT} \\[1ex]
\langle\text{node\_decl}\rangle & ::= & \textbf{node}\ \text{IDENT}\ \langle\text{node\_param\_decl}\rangle^{*}\ \textbf{in}\ \text{io\_decls}\ \textbf{out}\ \text{io\_decls}\ [\langle\text{node\_impl}\rangle] \\
& | & \textbf{graph}\ \text{IDENT}\ \langle\text{graph\_param\_decl}\rangle^{*}\ \textbf{in}\ \text{io\_decls}\ \textbf{out}\ \text{io\_decls}\ \langle\text{node\_impl}\rangle \\[1ex]
\langle\text{node\_param\_decl}\rangle & ::= & \text{IDENT}\ \mathbf{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{graph\_param\_decl}\rangle & ::= & \text{IDENT}\ \mathbf{:}\ \langle\text{type\_expr}\rangle\ \mathbf{=}\ \langle\text{const\_expr}\rangle \\[1ex]
\langle\text{io\_decl}\rangle & ::= & \text{IDENT}\ \mathbf{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{node\_impl}\rangle & ::= & \langle\text{val\_decl}\rangle^{*} \\[1ex]
\langle\text{binding}\rangle & ::= & \langle\text{pattern}\rangle\ \mathbf{=}\ \langle\text{expr}\rangle \\[1ex]
\langle\text{val\_decl}\rangle & ::= & \textbf{val}\ [\textbf{rec}]\ \langle\text{binding}\rangle^{+}_{\textbf{and}} \\[1ex]
\langle\text{expr}\rangle & ::= & \langle\text{const\_expr}\rangle \\
& | & \text{IDENT} \\
& | & \langle\text{expr}\rangle\ \langle\text{expr}\rangle \\
& | & \mathbf{(}\ \langle\text{expr}\rangle^{+}_{\mathbf{,}}\ \mathbf{)} \\
& | & \textbf{fun}\ \langle\text{pattern}\rangle \rightarrow \langle\text{expr}\rangle \\
& | & \textbf{let}\ [\textbf{rec}]\ \langle\text{binding}\rangle^{+}_{\textbf{and}}\ \textbf{in}\ \langle\text{expr}\rangle \\
& | & \mathbf{(\ )} \\[1ex]
\langle\text{const\_expr}\rangle & ::= & \text{INT} \\
& | & \textbf{true} \\
& | & \textbf{false} \\[1ex]
\langle\text{pattern}\rangle & ::= & \text{IDENT} \\
& | & \mathbf{(}\ \langle\text{pattern}\rangle^{+}_{\mathbf{,}}\ \mathbf{)} \\
& | & \mathbf{(\ )} \\[1ex]
\langle\text{type\_expr}\rangle & ::= & \text{IDENT}
\end{array}
$$

Chapter 2

Typing

The type language is fairly standard. A type τ is either :

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

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

- a type environment TE, recording type constructors,
- a variable environment VE, mapping identifiers to types[1].

The initial type environment TE_0 records the type of the *builtin* type constructors :
$\mathrm{TE}_0 \quad = \quad [\mathrm{int} \mapsto \mathsf{Int}, \quad \mathrm{bool} \mapsto \mathsf{Bool}, \quad \mathrm{unit} \mapsto \mathsf{Unit}]$
The initial variable environment VE_0 contains the types of the builtin primitives :
$\mathrm{VE}_0 \quad = \quad [+ \mapsto \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}, \quad = \mapsto \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}, \quad \ldots]$

2.1 Notations

Both type and variable *environments* are viewed as partial maps from identifiers to types and from type constructors to types respectively. If E is an environment, the domain of E is denoted by $\mathrm{dom}(E)$. The empty environment is written $\varnothing$. $[x \mapsto y]$ denotes the singleton environment mapping x to y and $E(x)$ the result of applying the underlying map to x (for ex. if E is $[x \mapsto y]$ then $E(x) = y$) and $E[x \mapsto y]$ 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'. Given two types τ and τ', we will note $\tau \cong \tau'$ if τ and τ' are equal modulo unification[2].

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

[1] More precisely, to *type schemes* $\sigma = \forall\alpha.\ \tau$; but, for simplicity, we do not distinguish types from type schemes in this presentation, *i.e.* the instanciation of a type scheme into a type and the generalisation of a (polymorphic) type into a type scheme are left implicit in the rules given above. The corresponding definitions are completely standard.

[2] If τ and τ' are monomorphic, this is structural equality.

Meta-variable	Meaning
TE	Type environment
VE	Variable environment
t	Type expression
τ	Type or type scheme
χ	Type constructor
id	Identifier
pat	Pattern
expr	Expression

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

2.2 Programs

$$\boxed{\vdash \text{Program} \Rightarrow \text{VE}'}$$

$$\frac{\begin{array}{c}\text{TE}_0 \vdash \textit{typedecls} \Rightarrow \text{TE}' \\ \text{TE}_0 \oplus \text{TE}', \text{VE}_0 \vdash \textit{valdecls} \Rightarrow \text{VE}' \\ \text{TE}_0 \oplus \text{TE}', \text{VE}' \vdash \textit{nodedecls} \Rightarrow \text{VE}\end{array}}{\text{TE}_0, \text{VE}_0 \vdash \textbf{program}\ \textit{typedecls valdecls nodedecls} \Rightarrow \text{VE}} \quad (\textsc{Program})$$

Typing a program consists in

- typing the type declarations, resulting in an augmented type environment,
- typing the global value declarations, resulting in an augmented value environment,
- typing the sequence of node declarations in these augmented environments.

The result is an environment containing the type of each declared node.

2.3 Type declarations

$$\boxed{\text{TE} \ \vdash \text{TypeDecls} \Rightarrow \text{TE}'}$$

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

$$\boxed{\text{TE} \ \vdash \text{TypeDecl} \Rightarrow \text{TE}'}$$

$$\frac{}{\text{TE}, \text{VE} \ \vdash \textbf{type}\ \text{id} \Rightarrow [\text{id} \mapsto \mathsf{Id}]} \quad (\textsc{TypeDecl})$$

An abstract type declaration simply adds the corresponding type constructor in the type environment.

2.4 Node declarations

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

$$\frac{\begin{array}{c}\mathrm{VE}_0 = \mathrm{VE} \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{VE}_{i-1} \vdash \mathit{nodedecl}_i \Rightarrow \mathrm{VE}_i\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{nodedecl}_1 \ \ldots \ \mathit{nodedecl}_n \Rightarrow \mathrm{VE}_n} \quad (\textsc{NodeDecls})$$

Node declarations are typed in the order of their declaration. A declaration can be used in the subsequent ones.

2.4.1 Parameter-less actors

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

$$\frac{\begin{array}{c}\mathrm{TE} \vdash \mathit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \mathit{outs} \Rightarrow \tau_o, \mathrm{VE}_o\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \textbf{node}\ \mathrm{id}\ \varnothing\ \mathit{ins}\ \mathit{outs} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_i \to \tau_o]} \quad (\textsc{NodeDeclA})$$

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

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE} \vdash \mathrm{id}_j{:}\mathrm{t}_j \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{TE} \vdash \mathrm{id}_1{:}\mathrm{t}_1, \ldots, \mathrm{id}_n{:}\mathrm{t}_n \Rightarrow \tau_1 \times \ldots \times \tau_n, \ \bigoplus_{i=1}^{n} \mathrm{VE}_i} \quad (\textsc{NodeIOs})$$

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

$$\frac{\mathrm{TE} \vdash \mathrm{t} \Rightarrow \tau}{\mathrm{TE} \vdash \mathrm{id}{:}\mathrm{t} \Rightarrow \mathsf{wire}\ \tau, \ [\mathrm{id} \mapsto \mathsf{wire}\ \tau]} \quad (\textsc{NodeIO})$$

Typing an *opaque* node (actor) declared as

$$\textbf{node}\ \mathrm{name}\ \textbf{ins}\ (\mathrm{i}_1{:}\mathrm{t}_1, \ldots, \mathrm{i}_m{:}\mathrm{t}_m)\ \textbf{outs}\ (\mathrm{o}_1{:}\mathrm{t}'_1, \ldots, \mathrm{o}_n{:}\mathrm{t}'_n)$$

assigns to it the type

$$\mathsf{wire}\ \tau_1 \times \ldots \times \mathsf{wire}\ \tau_m \to \mathsf{wire}\ \tau'_1 \times \ldots \times \mathsf{wire}\ \tau'_n$$

where τ_i (resp. τ'_i) is the type denoted by the type expression attached to the i^{th} input (resp. output). The wire type constructor ensures that only *wires* can be given as arguments to functions representing nodes (and not scalar values such as integers or booleans).

2.4.2 Parameterized actors

$$\frac{\begin{array}{c} \mathit{params} \neq \emptyset \\ \mathrm{TE} \vdash \mathit{params} \Rightarrow \tau_p, \mathrm{VE}_p \\ \mathrm{TE} \vdash \mathit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \mathit{outs} \Rightarrow \tau_o, \mathrm{VE}_o \end{array}}{\mathrm{TE}, \mathrm{VE} \ \vdash \mathbf{node}\ \mathrm{id}\ \mathit{params}\ \mathit{ins}\ \mathit{outs} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_p \to \tau_i \to \tau_o]} \quad (\textsc{PNodeDeclA})$$

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

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE} \ \vdash \mathrm{id}_j\text{:}\mathrm{t}_j \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{TE} \ \vdash \mathrm{id}_1\text{:}\mathrm{t}_1 \ldots \mathrm{id}_n\text{:}\mathrm{t}_n \Rightarrow \tau_1 \times \ldots \times \tau_n, \ \bigoplus_{i=1}^{n} \mathrm{VE}_i} \quad (\textsc{NodeParams})$$

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

$$\frac{\mathrm{TE} \ \vdash \mathrm{t} \Rightarrow \tau}{\mathrm{TE} \ \vdash \mathrm{id}\text{:}\mathrm{t} \Rightarrow \tau', \ [\mathrm{id} \mapsto \tau]} \quad (\textsc{NodeParam})$$

Typing an *opaque* node (actor) declared as

$$\mathbf{node}\ \mathrm{name}\ \mathbf{params}\ (\mathrm{p}_1\text{:}\mathrm{t}_1, \ldots, \mathrm{p}_p\text{:}\mathrm{t}_p)\ \mathbf{ins}\ (\mathrm{i}_1\text{:}\mathrm{t}'_1, \ldots, \mathrm{i}_m\text{:}\mathrm{t}'_m)\ \mathbf{outs}\ (\mathrm{o}_1\text{:}\mathrm{t}''_1, \ldots, \mathrm{o}_n\text{:}\mathrm{t}''_n)$$

assigns to it the type

$$\tau_1 \times \ldots \times \tau_p \to \mathsf{wire}\ \tau'_1 \times \ldots \times \mathsf{wire}\ \tau'_m \to \mathsf{wire}\ \tau''_1 \times \ldots \times \mathsf{wire}\ \tau''_n$$

In other words, parameterized nodes are viewed as *curried* fonctions[3].

2.4.3 Refined nodes

This concerns "tranparent" nodes, *i.e.* nodes defined by a set of value declarations.

$$\frac{\begin{array}{c} \mathrm{TE} \vdash \mathit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \mathit{outs} \Rightarrow \tau_o, \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_i \vdash \mathit{valdecls} \Rightarrow \mathrm{VE}' \\ \mathrm{VE}' \subset\!\!\cdot\ \mathrm{VE}_o \end{array}}{\mathrm{TE}, \mathrm{VE} \ \vdash \mathbf{node}\ \mathrm{id}\ \varnothing\ \mathit{ins}\ \mathit{outs}\ \mathit{valdecls} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_i \to \tau_o]} \quad (\textsc{NodeDeclG})$$

In this case, we also check that the type assigned to outputs by these declarations are compatible with the type assigned to the corresponding node output. This condition is here expressed using the $\subset\!\!\cdot$ predicate. Given two typing environments VE and VE′, VE $\subset\!\!\cdot$ VE′ holds *iff* $\forall x \in \mathrm{dom}(\mathrm{VE}) \cap \mathrm{dom}(\mathrm{VE}')$, $\mathrm{VE}(x) \cong \mathrm{VE}'(x)$, *i.e.* *iff* for each symbol occuring both in VE and VE′, the related types are equals modulo unification.

[3] And the actual parameters will be supplied by partial application.

$$\frac{\begin{array}{c} \textit{params} \neq \varnothing \\ \mathrm{TE} \vdash \textit{params} \Rightarrow \tau_p, \mathrm{VE}_p \\ \mathrm{TE} \vdash \textit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \textit{outs} \Rightarrow \tau_o, \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_i \oplus \mathrm{VE}_p \vdash \textit{valdecls} \Rightarrow \mathrm{VE}' \\ \mathrm{VE}' \subseteq \mathrm{VE}_o \end{array}}{\mathrm{TE}, \mathrm{VE}\ \vdash \textbf{node}\ \mathrm{id}\ \textit{param ins outs valdecls} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_p \to \tau_i \to \tau_o]} \quad (\textsc{PNodeDeclG})$$

For parameterized actors, the declared parameters are also added to the typing environment when typing the definitions.

2.5 Graph declarations

Graph declarations are handled exactly as node declarations except that the value supplied with a parameter is type checked against the declared type. Note that graph declarations always have a *valdecls* section (there's no such thing as an opaque graph declaration).

$$\frac{\begin{array}{c} \mathrm{TE} \vdash \textit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \textit{outs} \Rightarrow \tau_o, \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_i \vdash \textit{valdecls} \Rightarrow \mathrm{VE}' \\ \mathrm{VE}' \subseteq \mathrm{VE}_o \end{array}}{\mathrm{TE}, \mathrm{VE}\ \vdash \textbf{graph}\ \mathrm{id}\ \varnothing\ \textit{ins outs valdecls} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_i \to \tau_o]} \quad (\textsc{GraphDecl})$$

$$\frac{\begin{array}{c} \textit{params} \neq \varnothing \\ \mathrm{TE} \vdash_g \textit{params} \Rightarrow \tau_p, \mathrm{VE}_p \\ \mathrm{TE} \vdash \textit{ins} \Rightarrow \tau_i, \mathrm{VE}_i \\ \mathrm{TE} \vdash \textit{outs} \Rightarrow \tau_o, \mathrm{VE}_o \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}_i \oplus \mathrm{VE}_p \vdash \textit{valdecl} \Rightarrow \mathrm{VE}' \\ \mathrm{VE}' \subseteq \mathrm{VE}_o \end{array}}{\mathrm{TE}, \mathrm{VE}\ \vdash \textbf{node}\ \mathrm{id}\ \textit{param ins outs valdecls} \Rightarrow \mathrm{VE} \oplus [\mathrm{id} \mapsto \tau_p \to \tau_i \to \tau_o]} \quad (\textsc{PGraphDecl})$$

$$\boxed{\mathrm{TE}\ \vdash_g \mathrm{GraphParams} \Rightarrow \tau, \mathrm{VE}}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\quad \mathrm{TE}\ \vdash_g \mathrm{id}_i\mathrm{:t}_i\mathrm{=}\textit{expr}_i \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{TE}\ \vdash_g \mathrm{id}_1\mathrm{:t}_1\mathrm{=}\textit{expr}_1 \ldots \mathrm{id}_n\mathrm{:t}_n\mathrm{=}\textit{expr}_n \Rightarrow \tau_1 \times \ldots \times \tau_n,\ \bigoplus_{i=1}^{n} \mathrm{VE}_i} \quad (\textsc{GraphParams})$$

$$\boxed{\mathrm{TE}\ \vdash_g \mathrm{GraphParam} \Rightarrow \tau, \mathrm{VE}}$$

$$\frac{\begin{array}{c} \mathrm{TE}\ \vdash \mathrm{t} \Rightarrow \tau \\ \mathrm{TE}, \varnothing \vdash \textit{expr} \Rightarrow \tau' \\ \tau \cong \tau' \end{array}}{\mathrm{TE}\ \vdash_g \mathrm{id{:}t}\mathrm{=}\textit{expr} \Rightarrow \tau',\ [\mathrm{id} \mapsto \tau]} \quad (\textsc{GraphParam})$$

2.6 Value declarations

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

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

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

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{VE}'}{\mathrm{TE}, \mathrm{VE} \vdash \mathbf{val}\ \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \;\Rightarrow \mathrm{VE} \oplus \mathrm{VE}'} \quad (\textsc{ValDecl})$$

$$\boxed{\mathrm{TE}, \mathrm{VE} \;\vdash_r \mathrm{ValDecl} \Rightarrow \mathrm{VE}'}$$

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash_r \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{VE}'}{\mathrm{TE}, \mathrm{VE} \vdash \mathbf{val\ rec}\ \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \;\Rightarrow \mathrm{VE} \oplus \mathrm{VE}'} \quad (\textsc{RecValDecl})$$

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

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n, \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{pat}_i = \mathit{expr}_i \Rightarrow \mathrm{VE}'_i \\ \mathrm{VE}' = \bigoplus_{i=1}^{n} \mathrm{VE}'_i\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \;\Rightarrow \mathrm{VE}'} \quad (\textsc{Bindings})$$

The rule Bindings deals with multiple bindings, as found in `val p1=e1 and ... and pn=en` declarations or `let p1=e1 and ... and pn=en` expressions.

$$\boxed{\mathrm{TE}, \mathrm{VE} \;\vdash_r \mathrm{PatExprs} \Rightarrow \mathrm{VE}'}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n, \quad \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}' \vdash \mathit{pat}_i = \mathit{expr}_i \Rightarrow \mathrm{VE}'_i \\ \mathrm{VE}' = \bigoplus_{i=1}^{n} \mathrm{VE}'_i\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash_r \mathit{pat}_1 = \mathit{expr}_1 \;\ldots\; \mathit{pat}_n = \mathit{expr}_n \;\Rightarrow \mathrm{VE}'} \quad (\textsc{RecBindings})$$

The rule RecBindings deals with multiple recursive bindings. Note that in this case, the recursively defined symbols are available when typing the RHS expressions.

$$\boxed{\mathrm{TE}, \mathrm{VE} \ \vdash \mathrm{Pat{=}Expr} \Rightarrow \mathrm{VE}'}$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \mathit{expr} \Rightarrow \tau \\ \vdash_p \mathit{pat}, \tau \Rightarrow \mathrm{VE}'\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{pat} = \mathit{expr} \ \Rightarrow \mathrm{VE}'} \qquad (\textsc{Binding})$$

where

$$\vdash_p \mathit{pat}, \tau \Rightarrow \mathrm{VE}$$

means that declaring *pat* with type τ creates the variable environment VE, as described in Sec. 2.6.1.

2.6.1 Patterns

$$\boxed{\vdash_p \mathrm{Pat}, \tau \Rightarrow \mathrm{VE}}$$

$$\frac{}{\vdash_p \mathrm{id}, \tau \Rightarrow [\mathrm{id} \mapsto \tau]} \qquad (\textsc{PatVar})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash_p \mathit{pat}\ _i, \tau_i \Rightarrow \mathrm{VE}_i}{\vdash_p \ (\mathit{pat}\ _1, \ldots \mathit{pat}\ _n), \tau_1 \times \ldots \times \tau_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{VE}_i} \qquad (\textsc{PatTuple})$$

$$\frac{}{\vdash_p (), \mathsf{Unit} \Rightarrow \varnothing} \qquad (\textsc{PatUnit})$$

$$\frac{}{\vdash_p \_, \tau \Rightarrow \varnothing} \qquad (\textsc{PatIgnore})$$

2.6.2 Expressions

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

$$\frac{\mathrm{VE}(\mathit{id}) = \tau}{\mathrm{TE}, \mathrm{VE} \vdash \mathrm{id} \Rightarrow \tau} \qquad (\textsc{EVar})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{TE}, \mathrm{VE} \vdash \ \mathit{expr}_i \Rightarrow \tau_i}{\mathrm{TE}, \mathrm{VE} \vdash (\mathit{expr}_1, \ \ldots \ \mathit{expr}_n) \Rightarrow \tau_1 \times \ldots \times \tau_n} \qquad (\textsc{ETuple})$$

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_1 \Rightarrow \tau \rightarrow \tau' \qquad \mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_2 \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_1 \ \mathit{expr}_2 \Rightarrow \tau'} \qquad (\textsc{EApp})$$

$$\frac{\vdash_p \mathit{pat}\ , \tau \Rightarrow \mathrm{VE}' \qquad \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}' \vdash \mathit{expr} \Rightarrow \tau'}{\mathrm{TE}, \mathrm{VE} \vdash \mathbf{fun}\ \mathit{pat}\ \rightarrow \mathit{expr} \Rightarrow \tau \rightarrow \tau'} \qquad (\textsc{EFun})$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \mathit{pat}_1 \ = \mathit{expr}_1 \ \ldots \ \mathit{pat}_n \ = \mathit{expr}_n \Rightarrow \mathrm{VE}' \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}' \vdash \mathit{expr}' \Rightarrow \tau\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathbf{let}\ \mathit{pat}_1 \ = \mathit{expr}_1 \ \ldots \ \mathit{pat}_n \ = \mathit{expr}_n \ \mathbf{in}\ \mathit{expr}' \Rightarrow \tau} \qquad (\textsc{ELet})$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash_r \mathit{pat}_1 = \mathit{expr}_1 \ \ldots \ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{VE}' \\ \mathrm{TE}, \mathrm{VE} \oplus \mathrm{VE}' \vdash \mathit{expr}' \Rightarrow \tau\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \textbf{let rec } \mathit{pat}_1 = \mathit{expr}_1 \ \ldots \ \mathit{pat}_n = \mathit{expr}_n \textbf{ in } \mathit{expr}' \Rightarrow \tau} \quad (\text{ELetRec})$$

$$\frac{}{\mathrm{TE}, \mathrm{VE} \vdash () \Rightarrow \mathsf{Unit}} \quad (\text{EUnit})$$

$$\frac{}{\mathrm{TE}, \mathrm{VE} \vdash \mathrm{int} \Rightarrow \mathsf{Int}} \quad (\text{EInt})$$

$$\frac{}{\mathrm{TE}, \mathrm{VE} \vdash \mathrm{bool} \Rightarrow \mathsf{Bool}} \quad (\text{EBool})$$

$$\frac{\begin{array}{c}\mathrm{VE}(\mathrm{op}) = \tau_1 \times \tau_2 \to \tau_3 \\ \mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_1 \Rightarrow \tau_1 \\ \mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_2 \Rightarrow \tau_2\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_1 \ \mathrm{op} \ \mathit{expr}_2 \Rightarrow \tau_3} \quad (\text{EBinop})$$

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \mathit{expr} \Rightarrow \mathsf{Bool} \\ \mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_1 \Rightarrow \tau \\ \mathrm{TE}, \mathrm{VE} \vdash \mathit{expr}_2 \Rightarrow \tau\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \textbf{if } \mathit{expr} \textbf{ then } \mathit{expr}_1 \textbf{ else } \mathit{expr}_2 \Rightarrow \tau} \quad (\text{EIf})$$

2.7 Type expressions

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

$$\frac{\mathrm{TE}(\mathrm{id}) = \tau}{\mathrm{TE} \vdash \mathrm{id} \Rightarrow \tau} \quad (\text{TyCon})$$

Type expressions, at the syntax level, are limited to type names.

Chapter 3

Static semantics

The static semantics gives the interpretation of HoCL programs, described with the abstract syntax given in chapter 1, as a set of (dataflow) *graphs*, where each graph is defined as a set of *boxes* connected by *wires*. The formulation given here assumes that the program has been successfully type checked.

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

Variable	**Set ranged over**	**Definition**	**Meaning**
v	Val	$\mathsf{Loc} + \mathsf{Node} + \mathsf{Tuple} + \mathsf{Clos}$	Value
		$\mathsf{Unit} + \mathsf{Int} + \mathsf{Bool} + \mathsf{Prim}$	
ℓ	Loc	$\langle \mathsf{bid}, \mathsf{sel} \rangle$	Graph location
n	Node	$\langle \mathsf{NCat}, \{\mathsf{id} \mapsto \mathsf{Val}\}, \mathsf{Bool}, \mathsf{id}^+, \mathsf{id}^+, \mathsf{NImpl} \rangle$	Node description
κ	NCat	$\mathsf{node} + \mathsf{graph}$	Node category
vs	Tuple	Val^+	Tuple
cl	Clos	$\langle \mathit{pattern}, \mathit{expr}, \mathsf{Env} \rangle$	Closure
E	Env	$\{\mathsf{id} \mapsto \mathsf{Val}\}$	Value environment
η	NImpl	$\mathsf{actor} + \mathsf{Graph}$	Node implementation
g	Graph	$\langle \mathsf{Boxes}, \mathsf{Wires} \rangle$	Graph description
B	Boxes	$\{\mathsf{bid} \mapsto \mathsf{Box}\}$	Box environment
W	Wires	$\{\mathsf{wid} \mapsto \mathsf{Wire}\}$	Wire environment
L	Locs	Loc^*	Location set
b	Box	$\langle \mathsf{BCat}, \{\mathsf{sel} \mapsto \mathsf{wid}\}, \{\mathsf{sel} \mapsto \mathsf{wid}^*\}, \mathsf{Val} \rangle$	Box
c	BCat	$\mathsf{actor} + \mathsf{graph} + \mathsf{src} + \mathsf{snk} + \mathsf{rec}$	Box category
		$\mathsf{inParam} + \mathsf{localParam}$	
w	Wire	$\langle \langle \mathsf{bid}, \mathsf{sel} \rangle, \langle \mathsf{bid}, \mathsf{sel} \rangle \rangle$	Wire (src loc, dst loc)
l, l'	bid	$\{0, 1, 2, \ldots\}$	Box id
k, k'	wid	$\{0, 1, 2, \ldots\}$	Wire id
s, s'	sel	$\{0, 1, 2, \ldots\}$	Slot selector
	Int	$\{\ldots, -2, -1, 0, 1, \ldots\}$	Integer value
β	Bool	$\{\mathsf{true}, \mathsf{false}\}$	Boolean value
π	Prim	$\{\mathsf{Value} \mapsto \mathsf{Value}\}$	Primitive function

Values in the category Loc correspond to graph *locations*, where a location comprises a box index and and a selector. Selectors are used to distinguish inputs (resp. outputs when the box has several of them[1].

[1]Valid selectors start at 1. The selector value 0 is used for incomplete box definitions.

Nodes are described by

- a category, indicating whether the node is a toplevel graph or an ordinary node[2],
- a list a parameters, with associated values when available,
- a boolean flag β indicating whether the node still misses the value of its parameters[3],
- a list of inputs,
- a list of outputs,
- an implementation, which is either empty (in case of opaque actors) or given as a graph.

Boxes are described by

- a category,
- a input environment, mapping selector values (1,2,...) to wire identifiers,
- a output environment, mapping selector values to sets of wire identifiers[4],
- an optional value.

Box categories separate boxes

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

The category `rec` is used internally for building cyclic graphs (see Sec. 3.6.2).
The optional box value is only meaningful for local parameters bound to constants or for toplevel input parameters (giving in this case the constant value).

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

Closures correspond to functional values.

Primitives correspond to builtin functions operating on integer or boolean values (`+`, `=`, ...).

The environments E, B and W respectively bind

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

All *environments* are viewed as partial maps from keys to values. If E is an environment, the domain of E is denoted by $\mathrm{dom}(E)$. The empty environment is written $\varnothing$. $[x \mapsto y]$ denotes the singleton environment mapping x to y. $E(x)$ denotes the result of applying the underlying map to x (for ex. if E is $[x \mapsto y]$ then $E(x) = y$) and $E \oplus E'$ the environment obtained by adding the mappings of E' to those of E, assuming that E and E' are disjoints.

[2]This avoids having two distincts but almost identical semantic values for nodes and toplevel graphs.

[3]For parameter-less nodes and toplevel graphs, this flag will always be `false`; for nodes accepting parameters, it will be initially `true` and set to `false` when the corresponding values are provided by partial application of the corresponding function (see rules EAppN and EPAppN in Sec. 3.4.

[4]A box output can be broadcasted to several other boxes.

3.1 Programs

$$\boxed{\vdash \text{Program} \Rightarrow \text{E}}$$

$$\frac{\begin{array}{c}\text{E}_0, \varnothing \vdash \textit{valdecls} \Rightarrow \text{E}, \text{B}, \text{W} \\ \text{E}, \varnothing \vdash \textit{nodedecls} \Rightarrow \text{E}'\end{array}}{\vdash \textbf{program}\ \textit{typedecls}\ \textit{valdecls}\ \textit{nodedecls} \Rightarrow \text{E}'} \quad (\textsc{Program})$$

Global values are first evaluated to give a value environment (boxes and wires resulting from this evaluation are here ignored). Nodes declarations are evaluated in this environment. The result is an environment associating a node description to each defined node.

The initial environment E_0 contains, the value of the builtin primitives (+, =, ...).

3.2 Node and graph declarations

$$\boxed{\text{E}, \text{B} \vdash \text{GraphOrNodeDecls} \Rightarrow \text{E}', \text{B}'}$$

$$\frac{\begin{array}{c}\text{E}_0 = \text{E} \\ \text{B}_0 = \text{B} \\ \forall i.\ 1 \leq i \leq n, \quad \text{E}_{i-1}, \text{B}_{i-1} \vdash \textit{nodedecl}_i \Rightarrow \text{E}_i, \text{B}_i\end{array}}{\text{E}, \text{B} \vdash \textit{nodedecl}_1\ \ldots\ \textit{nodedecl}_n \Rightarrow \text{E}_n, \text{B}_n} \quad (\textsc{GraphOrNodeDecls})$$

Node declarations are interpreted in the order of their declaration. A declaration can be used in the subsequent ones.

$$\boxed{\text{E}, \text{B} \vdash \text{NodeDecl} \Rightarrow \text{E}', \text{B}'}$$

$$\frac{\begin{array}{c}\text{params} = [\text{id}_1, \ldots, \text{id}_p] \\ \beta = \text{params} \neq \emptyset \\ \texttt{n} = \mathsf{Node}\langle \mathsf{node}, [\text{id}_1 \mapsto \mathsf{Unit}, \ldots, \text{id}_p \mapsto \mathsf{Unit}], \beta, \text{ins}, \text{outs}, \mathsf{actor}\rangle\end{array}}{\text{E}, \text{B} \vdash \textbf{node}\ \text{id params ins outs} \Rightarrow \text{E} \oplus [\text{id} \mapsto \mathsf{Node}\ n], \text{B}} \quad (\textsc{NodeDeclA})$$

Nodes with no attached definition are mapped to opaque actors. For parameterized actors, parameter values are initially set to Unit (meaning "yet undefined in this case") and the corresponding boolean flag set to true. For parameter-less actors, the flag is set to false.

$$\frac{\begin{array}{c}\textit{valdecls} \neq \varnothing \\ \text{params} = [\text{id}_1, \ldots, \text{id}_p] \\ \text{B} \vdash_p \text{params} \Rightarrow \text{E}_p, \text{B}_p \\ \text{B}_p \vdash_i \text{ins} \Rightarrow \text{E}_i, \text{B}_i \\ \text{B}_p \oplus \text{B}_i \vdash_o \text{outs} \Rightarrow \text{E}_o, \text{B}_o \\ \text{E} \oplus \text{E}_p \oplus \text{E}_i \oplus \text{E}_o,\ \text{B} \oplus \text{B}_p \oplus \text{B}_i \oplus \text{B}_o \vdash \textit{valdecls} \Rightarrow \text{B},\ \text{W} \\ \beta = \text{params} \neq \emptyset \\ \texttt{n} = \mathsf{Node}\langle \mathsf{node}, [\text{id}_1 \mapsto \mathsf{Unit}, \ldots, \text{id}_p \mapsto \mathsf{Unit}], \beta, \text{ins}, \text{outs}, \mathsf{Graph}\langle \text{B}, \text{W}\rangle\rangle\end{array}}{\text{E}, \text{B} \vdash \textbf{node}\ \text{id params ins outs}\ \textit{valdecls} \Rightarrow \text{E} \oplus [\text{id} \mapsto \mathsf{Node}\ n], \text{B} \oplus \text{B}_p \oplus \text{B}_i \oplus \text{B}_o} \quad (\textsc{NodeDeclG})$$

For nodes with an attached definition, this definition is evaluated in an environment augmented with its input, parameter and output declarations and the resulting graph (a pair of boxes and wires) is attached to the node description.

$$\boxed{\mathrm{B} \vdash_{i/o/p} \mathrm{NodeIOs} \Rightarrow \mathrm{E}, \mathrm{B}'}$$

$$\frac{\begin{array}{c}\mathrm{B}_0 = \mathrm{B} \\ \forall i.\ 1 \leq i \leq n,\quad \mathrm{B}_{i-1} \vdash_{i/o/p} \mathrm{id}_i{:}\mathrm{t}_i \Rightarrow \mathrm{E}_i, \mathrm{B}_i\end{array}}{\mathrm{B} \vdash_{i/o/p} \mathrm{id}_1{:}\mathrm{t}_1\ \ldots\ \mathrm{id}_n{:}\mathrm{t}_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{E}_i,\ \mathrm{B}_n} \quad (\textsc{NodeIOs})$$

$$\boxed{\mathrm{B} \vdash_{i/o/p} \mathrm{NodeIO} \Rightarrow \mathrm{E}, \mathrm{B}'}$$

$$\frac{\mathrm{l} \notin \mathit{Dom}(\mathrm{B})}{\mathrm{B} \vdash_i \mathrm{id{:}t} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ \mathrm{B} \oplus [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{src}, \varnothing, [1 \mapsto \varnothing]\rangle]} \quad (\textsc{NodeInp})$$

$$\frac{\mathrm{l} \notin \mathit{Dom}(\mathrm{B})}{\mathrm{B} \vdash_o \mathrm{id{:}t} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ \mathrm{B} \oplus [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{snk}, [1 \mapsto 0], \varnothing\rangle]} \quad (\textsc{NodeOutp})$$

$$\frac{\mathrm{l} \notin \mathit{Dom}(\mathrm{B})}{\mathrm{B} \vdash_p \mathrm{id{:}t} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ \mathrm{B} \oplus [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{inParam}, \varnothing, [1 \mapsto \varnothing]\rangle]} \quad (\textsc{NodeParam})$$

Each parameter, input and output adds a box in the enclosing graph (with category inParam, src and snk respectively). These boxes have no input (resp. no output). The premise $\mathrm{l} \notin \mathit{Dom}(\mathrm{B})$ ensures that l is a "fresh" box index.

3.2.1 Graph declaration

$$\boxed{\mathrm{E}, \mathrm{B} \vdash \mathrm{GraphDecls} \Rightarrow \mathrm{E}', \mathrm{B}'}$$

$$\frac{\begin{array}{c}\mathrm{B} \vdash_g \mathrm{params} \Rightarrow \mathrm{E}_p, \mathrm{B}_p \\ \mathrm{B}_p \vdash_i \mathrm{ins} \Rightarrow \mathrm{E}_i, \mathrm{B}_i \\ \mathrm{B}_p \oplus \mathrm{B}_i \vdash_o \mathrm{outs} \Rightarrow \mathrm{E}_o, \mathrm{B}_o \\ \mathrm{E} \oplus \mathrm{E}_p \oplus \mathrm{E}_i \oplus \mathrm{E}_o,\ \mathrm{B} \oplus \mathrm{B}_p \oplus \mathrm{B}_i \oplus \mathrm{B}_o \vdash \mathit{valdecls} \Rightarrow \mathrm{B},\ \mathrm{W} \\ \beta = \mathrm{params} \neq \emptyset \\ \mathtt{n} = \mathsf{Node}\langle \mathsf{graph}, [\mathrm{id}_1 \mapsto \mathrm{E}_p(\mathrm{id}_1), \ldots, \mathrm{id}_p \mapsto \mathrm{E}_p(\mathrm{id}_p)], \mathsf{false}, \mathrm{ins}, \mathrm{outs}, \mathsf{Graph}\langle \mathrm{B}, \mathrm{W}\rangle\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash \textbf{graph}\ \mathrm{id\ params\ ins\ outs}\ \mathit{valdecls} \Rightarrow \mathrm{E} \oplus [\mathrm{id} \mapsto \mathsf{Node}\ n],\ \mathrm{B} \oplus \mathrm{B}_p \oplus \mathrm{B}_i \oplus \mathrm{B}_o} \quad (\textsc{GraphDecl})$$

Evaluation of toplevel graph declarations is similar to that of node declarations. The only difference is that the value of each parameter is evaluated and attached to the corresponding box.

$$\boxed{\mathrm{B} \vdash_g \mathrm{GraphParams} \Rightarrow \mathrm{E}, \mathrm{B}'}$$

$$\frac{\begin{array}{c}\mathrm{B}_0 = \mathrm{B} \\ \forall i.\ 1 \leq i \leq n,\quad \mathrm{B} \vdash_g \mathrm{id}_i{:}\mathrm{t}_i{=}\mathit{expr}_i \Rightarrow \mathrm{E}_i, \mathrm{B}_i\end{array}}{\mathrm{B}\ \vdash_g \mathrm{id}_1{:}\mathrm{t}_1{=}\mathit{expr}_1 \ldots \mathrm{id}_n{:}\mathrm{t}_n{=}\mathit{expr}_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{E}_i, \mathrm{B}_n} \quad (\textsc{GraphParams})$$

$$\boxed{\mathrm{B} \vdash_g \text{GraphParam} \Rightarrow \mathrm{E}, \mathrm{B}'}$$

$$\frac{\varnothing, \varnothing \vdash \mathit{expr} \Rightarrow \mathrm{v}, \varnothing, \varnothing \qquad \mathrm{l} \notin \mathit{Dom}(\mathrm{B})}{\mathrm{B} \vdash_g \mathrm{id{:}t}{=}\mathit{expr} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ \mathrm{B} \oplus [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{inParam}, \varnothing, [1 \mapsto \varnothing], \mathrm{v}\rangle]} \ (\textsc{GraphParam})$$

Boxes materialzing input parameters for toplevel graphs hold the value of the specified expression. Type checking ensures that this value is an integer or boolean constant[5].

3.3 Value declarations

$$\boxed{\mathrm{E}, \mathrm{B} \vdash \text{ValDecls} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}}$$

$$\frac{\mathrm{E}_0 = \mathrm{E}, \mathrm{B}_0 = \mathrm{B}, \mathrm{W}_0 = \varnothing \qquad \forall i.\ 1 \leq i \leq n,\quad \mathrm{E}_{i-1}, \mathrm{B}_{i-1}, \mathrm{W}_{i-1} \vdash \mathit{valdecl}_i \Rightarrow \mathrm{E}_i, \mathrm{B}_i, \mathrm{W}_i}{\mathrm{E}, \mathrm{B} \vdash \mathit{valdecl}_1 \ \ldots\ \mathit{valdecl}_n \Rightarrow \mathrm{E}_n, \mathrm{B}_n, \mathrm{W}_n} \ (\textsc{ValDecls})$$

Within a node definition, value declarations are interpreted in the order of their declaration. A declaration can be used in the subsequent ones. Each declaration updates the value, box and wire environments.

$$\boxed{\mathrm{E}, \mathrm{B}, \mathrm{W} \vdash \text{ValDecl} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\mathrm{E}, \mathrm{B} \vdash \mathit{pat}_1 = \mathit{expr}_1 \ \ldots\ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}{\mathrm{E}, \mathrm{B}, \mathrm{W} \vdash \textbf{val}\ \mathit{pat}_1 = \mathit{expr}_1 \ \ldots\ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{E} \oplus \mathrm{E}',\ \mathrm{B} \oplus \mathrm{B}',\ \mathrm{W} \oplus \mathrm{W}'} \ (\textsc{ValDecl})$$

$$\boxed{\mathrm{E}, \mathrm{B}, \mathrm{W} \vdash_{rec} \text{ValDecl} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_1 = \mathit{expr}_1 \ \ldots\ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}{\mathrm{E}, \mathrm{B}, \mathrm{W} \vdash \textbf{val rec}\ \mathit{pat}_1 = \mathit{expr}_1 \ \ldots\ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{E} \oplus \mathrm{E}',\ \mathrm{B} \oplus \mathrm{B}',\ \mathrm{W} \oplus \mathrm{W}'} \ (\textsc{RecValDecl})$$

$$\boxed{\mathrm{E}, \mathrm{B} \vdash \text{PatExprs} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\quad \mathrm{E}, \mathrm{B} \vdash \mathit{pat}_i = \mathit{expr}_i \Rightarrow \mathrm{E}'_i, \mathrm{B}'_i, \mathrm{W}_i}{\mathrm{E}, \mathrm{B} \vdash \mathit{pat}_1 = \mathit{expr}_1 \ \ldots\ \mathit{pat}_n = \mathit{expr}_n \ \Rightarrow \bigoplus_{i=1}^{n} \mathrm{E}'_i, \bigoplus_{i=1}^{n} \mathrm{B}'_i, \bigoplus_{i=1}^{n} \mathrm{W}'_i} \ (\textsc{Bindings})$$

[5] So that the evaluation of the defining expression creates no box nor wire.

$$\boxed{\mathrm{E}, \mathrm{B} \vdash \mathrm{Pat{=}Expr} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{expr} \Rightarrow \mathrm{v}, \mathrm{B}', \mathrm{W}' \\ \mathrm{E}, \mathrm{B} \overleftarrow{\oplus} \mathrm{B}' \vdash_p \mathit{pat},\ \mathrm{v} \Rightarrow \mathrm{E}', \mathrm{B}'', \mathrm{W}''\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathit{pat} = \mathit{expr} \Rightarrow \mathrm{E}',\ \mathrm{B}' \overleftarrow{\oplus} \mathrm{B}'',\ \mathrm{W}' \oplus \mathrm{W}''} \quad (\textsc{Binding})$$

Evaluating a `val` declaration consists in evaluating the RHS expression and binding the result value to the LHS pattern.

The $\overleftarrow{\oplus}$ operator used in rule Binding merges box descriptors. If a box appears in both argument environments, the resulting environment contains a single occurrence of this box in which the respective input and output environments have been merged. For example

$$\begin{aligned}&[\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 0], [1 \mapsto \{2\}]\rangle] \overleftarrow{\oplus} [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 4], [1 \mapsto \{3\}]\rangle] \\ &\qquad = [\mathrm{l} \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 4], [1 \mapsto \{2, 3\}]\rangle]\end{aligned}$$

The semantics of *recursive* definitions ($\mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_1 = \mathit{expr}_1\ \ldots\ \mathit{pat}_n = \mathit{expr}_n$) is given in Sec. 3.6.

3.4 Expressions

$$\boxed{\mathrm{E}, \mathrm{B} \vdash \mathrm{Expr} \Rightarrow \mathrm{v}, \mathrm{B}', \mathrm{W}}$$

$$\frac{\mathrm{E}(\mathrm{id}) = \mathrm{v}}{\mathrm{E}, \mathrm{B} \vdash \mathrm{id} \Rightarrow \mathrm{v},\ \varnothing,\ \varnothing} \quad (\textsc{EVar})$$

The value of a variable is simply obtained from the value environment.

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \mathrm{E}, \mathrm{B} \vdash \mathit{expr}_i \Rightarrow \mathrm{v}_i, \mathrm{B}_i, \mathrm{W}_i}{\mathrm{E}, \mathrm{B} \vdash (\mathit{expr}_1, \ldots, \mathit{expr}_n) \Rightarrow \langle v_1, \ldots, v_n\rangle,\ \bigoplus_{i=1}^{n} \mathrm{B}_i,\ \bigoplus_{i=1}^{n} \mathrm{W}_i} \quad (\textsc{ETuple})$$

For tuples, each component is evaluated separately.

$$\frac{}{\mathrm{E}, \mathrm{B} \vdash \textbf{fun}\ \mathit{pat} \to \mathit{exp}\ \Rightarrow \mathsf{Clos}\langle \mathit{pat}, \mathit{exp}, \mathrm{E}\rangle,\ \varnothing,\ \varnothing} \quad (\textsc{EFun})$$

Functions are evaluated, classically, as closures, capturing the current value environment.

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{pat}_1 = \mathit{expr}_1\ \ldots\ \mathit{pat}_n = \mathit{expr}_n \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}' \\ \mathrm{E} \oplus \mathrm{E}', \mathrm{B} \overleftarrow{\oplus} \mathrm{B}' \vdash \mathit{exp}_2 \Rightarrow \mathrm{v}, \mathrm{B}'', \mathrm{W}''\end{array}}{\mathrm{E}, \mathrm{B} \vdash \textbf{let}\ \mathit{pat}_1 = \mathit{expr}_1\ \ldots\ \mathit{pat}_n = \mathit{expr}_n\ \textbf{in}\ \mathit{expr}' \Rightarrow \mathrm{v}, \mathrm{B} \overleftarrow{\oplus} \mathrm{B}', \mathrm{W} \oplus \mathrm{W}'} \quad (\textsc{ELet})$$

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathsf{Clos}\langle \mathit{pat}, \mathit{exp}, \mathrm{E}'\rangle, \mathrm{B}_f, \mathrm{W}_f \\ \mathrm{E}, \mathrm{B} \vdash \mathit{exp}_2 \Rightarrow \mathrm{v}, \mathrm{B}_a, \mathrm{W}_a \\ \varnothing, \varnothing \vdash_p \mathit{pat}, \mathrm{v} \Rightarrow \mathrm{E}_p, \mathrm{B}_p, \mathrm{W}_p \\ \mathrm{E}' \oplus \mathrm{E}_p, \mathrm{B} \vdash \mathit{exp} \Rightarrow \mathrm{v}', \mathrm{B}', \mathrm{W}'\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow \mathrm{v}', \mathrm{B}_f \overleftarrow{\oplus} \mathrm{B}_a \overleftarrow{\oplus} \mathrm{B}', \mathrm{W}_f \oplus \mathrm{W}_a \oplus \mathrm{W}'} \quad (\textsc{EAppC})$$

Rule EAppC deals with the application of closures and follows the classical call-by-value strategy (the closure body is evaluated in an environment augmented with the bindings resulting from binding its pattern to the argument).

$$\frac{\begin{array}{c}\mathrm{E},\mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathsf{Node}\langle\mathsf{node}, [\mathrm{p}_1 \mapsto \mathsf{Unit}, \ldots, \mathrm{p}_p \mapsto \mathsf{Unit}], \mathsf{true}, \mathrm{ins}, \mathrm{outs}, \eta\rangle, \mathrm{B}_f, \mathrm{W}_f \\ \mathrm{E},\mathrm{B} \vdash_{param} \mathit{exp}_2 \Rightarrow \langle \ell_1, \ldots, \ell_p\rangle, \mathrm{B}_p, \mathrm{W}_p \\ \mathtt{n} = \mathsf{Node}\langle\mathsf{node}, [\mathrm{p}_1 \mapsto \ell_1, \ldots, \mathrm{p}_p \mapsto \ell_p], \mathsf{false}, \mathrm{ins}, \mathrm{outs}, \eta\rangle\end{array}}{\mathrm{E},\mathrm{B} \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow \mathtt{n}, \mathrm{B}_f \overleftarrow{\oplus} \mathrm{B}_p, \mathrm{W}_f \oplus \mathrm{W}_p} \quad (\text{EAppNP})$$

Rule EAppNP deals with the partial application of nodes which supplies their actual parameters. This happens for nodes for which the *request* flag is true and the current parameter values set to Unit. The evaluation of parameter values is described, by rule $\mathrm{E},\mathrm{B} \vdash_{param} \mathit{expr}$ is described in Sec. 3.4.1. This results in a node for which these values are bound to the corresponding parameters.

$$\frac{\begin{array}{c}\mathrm{E},\mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathsf{Node}\langle\mathsf{node}, [\mathrm{p}_1 \mapsto \ell_1, \ldots, \mathrm{p}_p \mapsto \ell_p], \mathsf{false}, [i_1, \ldots, i_m], [o_1, \ldots, o_n], \eta\rangle, \mathrm{B}_f, \mathrm{W}_f \\ \mathrm{E},\mathrm{B} \vdash \mathit{exp}_2 \Rightarrow \langle \ell'_1, \ldots, \ell'_m\rangle, \mathrm{B}_a, \mathrm{W}_a \\ \mathrm{l} \notin \mathit{Dom}(\mathrm{B}) \\ \forall j.\ 1 \le j \le p,\quad \mathrm{k}_j \notin \mathit{Dom}(\mathrm{W}),\quad \mathrm{w}_j = \langle \ell_j, \mathsf{Loc}\langle l, j\rangle\rangle \\ \forall j.\ p+1 \le j \le p+m,\quad \mathrm{k}_j \notin \mathit{Dom}(\mathrm{W}),\quad \mathrm{w}_j = \langle \ell'_j, \mathsf{Loc}\langle l, j\rangle\rangle \\ \kappa = \mathsf{cat}(\eta) \\ \mathrm{b} = \mathsf{Box}\langle \kappa, [1 \mapsto k_1, \ldots, p \mapsto k_p, p+1 \mapsto k_{p+1}, \ldots, p+m \mapsto k_{p+m}], [1 \mapsto \varnothing, \ldots, n \mapsto \varnothing]\rangle \\ \mathrm{B}' = [l \mapsto b] \\ \mathrm{W}' = [k_1 \mapsto w_1, \ldots, k_p \mapsto w_p, k_{p+1} \mapsto w_{p+1}, \ldots, k_{p+m} \mapsto w_{p+m}] \\ \mathrm{v}' = \langle \mathsf{Loc}\langle \mathrm{l}, 1\rangle, \ldots, \mathsf{Loc}\langle \mathrm{l}, n\rangle\rangle\end{array}}{\mathrm{E},\mathrm{B} \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow \mathrm{v}', \mathrm{B}_f \overleftarrow{\oplus} \mathrm{B}_a \overleftarrow{\oplus} \mathrm{B}', \mathrm{W}_f \oplus \mathrm{W}_a \oplus \mathrm{W}'} \quad (\text{EAppN})$$

Rule EAppN deals with the application of nodes with supplied parameters or without parameters. It creates a new box and a set of wires connecting the parameters and arguments to the inputs of the inserted box (parameters first, then arguments). The outputs of the box will be connected by the binding step described in the next section. The function cat : NImpl → BCat is trivially defined as cat(actor) = actor and cat(Graph) = graph. Note that, for simplicity, the formulation of the rule assumes that single values and a tuples of size one are semantically equivalent[6].

$$\frac{}{\mathrm{E},\mathrm{B} \vdash () \Rightarrow \mathsf{Unit},\ \varnothing,\ \varnothing} \quad (\text{EUnit})$$

$$\frac{}{\mathrm{E},\mathrm{B} \vdash \mathrm{int} \Rightarrow \mathsf{Int},\ \varnothing,\ \varnothing} \quad (\text{EInt})$$

$$\frac{}{\mathrm{E},\mathrm{B} \vdash \mathrm{bool} \Rightarrow \mathsf{Bool},\ \varnothing,\ \varnothing} \quad (\text{EBool})$$

$$\frac{\begin{array}{c}\mathrm{E},\mathrm{B} \vdash \mathit{exp} \Rightarrow \mathsf{true}, \mathrm{B}', \mathrm{W}' \\ \mathrm{E},\mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathrm{v}, \mathrm{B}'', \mathrm{W}''\end{array}}{\mathrm{E},\mathrm{B} \vdash \mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{exp}_1\ \mathbf{else}\ \mathit{exp}_2 \Rightarrow \mathrm{v}, \mathrm{B}' \overleftarrow{\oplus} \mathrm{B}'', \mathrm{W}' \oplus \mathrm{W}''} \quad (\text{EIf0})$$

$$\frac{\begin{array}{c}\mathrm{E},\mathrm{B} \vdash \mathit{exp} \Rightarrow \mathsf{false}, \mathrm{B}', \mathrm{W}' \\ \mathrm{E},\mathrm{B} \vdash \mathit{exp}_2 \Rightarrow \mathrm{v}, \mathrm{B}'', \mathrm{W}''\end{array}}{\mathrm{E},\mathrm{B} \vdash \mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{exp}_1\ \mathbf{else}\ \mathit{exp}_2 \Rightarrow \mathrm{v}, \mathrm{B}' \overleftarrow{\oplus} \mathrm{B}'', \mathrm{W}' \oplus \mathrm{W}''} \quad (\text{EIf1})$$

[6] *I.e.* that $\langle \mathsf{Loc}\langle \mathrm{l}, 1\rangle\rangle = \mathsf{Loc}\langle \mathrm{l}, 1\rangle$).

3.4.1 Parameter expressions

These variant rules describe the evaluation of expressions giving values to node parameters. The set of accepted expressions is here limited to identifiers bound to input parameters, integer or boolean constants or any combination of the latter using builtin binary operators. Each case creates a single box in the graph, with inputs connected to the input parameters on which the expression depends.

$$\boxed{\mathrm{E}, \mathrm{B} \vdash_{param} \mathrm{Expr} \Rightarrow \mathrm{L}, \mathrm{B}', \mathrm{W}}$$

$$\frac{\begin{array}{c}\mathrm{l} \notin \mathit{Dom}(\mathrm{B}) \\ \mathrm{b} = \mathsf{Box}\langle\mathsf{localParam}, \varnothing, [1 \mapsto \varnothing], \mathsf{Int}\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{param} \mathrm{int} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ [\mathrm{l} \mapsto \mathrm{b}], \varnothing} \quad (\textsc{PInt})$$

$$\frac{\begin{array}{c}\mathrm{l} \notin \mathit{Dom}(\mathrm{B}) \\ \mathrm{b} = \mathsf{Box}\langle\mathsf{localParam}, \varnothing, [1 \mapsto \varnothing], \mathsf{Bool}\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{param} \mathrm{bool} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0\rangle],\ [\mathrm{l} \mapsto \mathrm{b}], \varnothing} \quad (\textsc{PBool})$$

For integer and boolean constants, a box is created, registering the corresponding value.

$$\frac{\begin{array}{c}\mathrm{E}(\mathrm{id}) = \ell = \mathsf{Loc}\langle \mathrm{l}, \mathrm{s}\rangle \\ \mathrm{B}(\mathrm{l}) = \mathsf{Box}\langle\mathsf{inParam}, ., .\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{param} \mathrm{id} \Rightarrow [\ell], \varnothing, \varnothing} \quad (\textsc{PVar})$$

Identifiers must refer to an input parameter of the graph.

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash_{param} \mathit{exp}_1 \Rightarrow \mathrm{L}_1, \mathrm{B}_1, \mathrm{W}_1 \\ \mathrm{E}, \mathrm{B} \vdash_{param} \mathit{exp}_2 \Rightarrow \mathrm{L}_2, \mathrm{B}_2, \mathrm{W}_2\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{param} \mathit{exp}_1 \ \mathrm{op}\ \mathit{exp}_2 \Rightarrow \mathrm{L}_1 \cup \mathrm{L}_2, \mathrm{B}_1 \oplus \mathrm{B}_2, \mathrm{W}_1 \oplus \mathrm{W}_2} \quad (\textsc{PBinop})$$

$$\frac{\forall i.\ 1 \le i \le n, \quad \mathrm{E}, \mathrm{B} \vdash_{param} \mathit{exp}_i \Rightarrow \mathrm{L}_i, \mathrm{B}_i, \mathrm{W}_i}{\mathrm{E}, \mathrm{B} \vdash_{param} (\mathit{exp}_1, \ldots, \mathit{exp}_n) \Rightarrow \bigcup_{i=1}^{n} \mathrm{L}_i,\ \bigoplus_{i=1}^{n} \mathrm{B}_i,\ \bigoplus_{i=1}^{n} \mathrm{W}_i} \quad (\textsc{PTuple})$$

3.5 Pattern matching

The following rules describe how variables are bound to values using pattern matching.

$$\boxed{\mathrm{E}, \mathrm{B} \vdash_{pat} \mathrm{pat}, \mathrm{v} \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}'}$$

$$\frac{\mathrm{id} \notin \mathit{Dom}(\mathrm{E})}{\mathrm{E}, \mathrm{B} \vdash_{pat} \mathrm{id}, \mathrm{v} \Rightarrow [\mathrm{id} \mapsto \mathrm{v}], \varnothing, \varnothing} \quad (\textsc{PatVar})$$

The previous rule concerns new local variables (introduced with `val` declaration). The created binding is just registered to be added to the value environment.

$$\frac{\begin{array}{c}\mathrm{E}(\mathrm{id}) = \mathsf{Loc}\langle \mathrm{l}', \mathrm{s}'\rangle \\ \mathrm{v} = \mathsf{Loc}\langle \mathrm{l}, \mathrm{s}\rangle \\ \mathrm{B}(\mathrm{l}') = \mathsf{Box}\langle \mathsf{snk}, [i \mapsto 0], \varnothing\rangle = \mathrm{b}_d \\ \mathrm{B}(\mathrm{l}) = \mathsf{Box}\langle \mathsf{actor}, \mathrm{bins}, \mathrm{bouts}\rangle = \mathrm{b}_s \\ \mathrm{k} \notin \mathit{Dom}(\mathrm{W}) \\ \mathrm{w} = \langle \mathsf{Loc}\langle \mathrm{l}, \mathrm{s}\rangle, \mathsf{Loc}\langle \mathrm{l}', \mathrm{s}'\rangle\rangle \\ \mathrm{b}'_s = \mathsf{Box}\langle \mathsf{actor}, \mathrm{bins}, \mathrm{bouts} \oplus [\mathrm{s}' \mapsto \mathrm{k}]\rangle \\ \mathrm{b}'_d = \mathsf{Box}\langle \mathsf{snk}, i \mapsto \mathrm{k}, \varnothing\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{pat} \mathrm{id}, \mathrm{v} \Rightarrow \varnothing, [\mathrm{l} \mapsto \mathrm{b}'_s, \mathrm{l}' \mapsto \mathrm{b}'_d], [\mathrm{k} \mapsto \mathrm{w}]} \quad (\textsc{PatOutput})$$

The previous rule describes the binding of graph outputs (which have been inserted as Sink boxes in the target graph). This creates a new wire, connecting the source box to the output box and updates the outputs (resp. inputs) of these boxes. As for boxes, the premise $\mathrm{l} \notin \mathit{Dom}(\mathrm{W})$ ensures that k is a "fresh" wire index.

$$\frac{\forall i.\ 1 \le i \le n, \quad \vdash_{pat} \mathit{pat}_i, \mathrm{v} \Rightarrow \mathrm{E}_i, \mathrm{B}_i, \mathrm{W}_i}{\mathrm{E}, \mathrm{B} \vdash_{pat} (\mathit{pat}_1, \dots, \mathit{pat}_n), \langle \mathrm{v}_1, \dots, \mathrm{v}_n\rangle \Rightarrow \bigoplus_{i=1}^{n} \mathrm{E}_i, \bigoplus_{i=1}^{n} \mathrm{B}_i, \bigoplus_{i=1}^{n} \mathrm{W}_i} \quad (\textsc{PatTuple})$$

$$\frac{}{\mathrm{E}, \mathrm{B} \vdash_{pat} \_, \mathrm{v} \Rightarrow \varnothing, \varnothing, \varnothing} \quad (\textsc{PatIgnore})$$

$$\frac{}{\mathrm{E}, \mathrm{B} \vdash_{pat} (), \mathsf{unit} \Rightarrow \varnothing, \varnothing, \varnothing} \quad (\textsc{PatUnit})$$

3.6 Recursive definitions

There are two kinds of recursive definitions. The first case is when the defined value (resp. set of values in case of *mutually* recursive definitions) is a *function* (resp. set of functions). The second case is when the defined values (resp. set of values) denotes a *wire* in the graph (resp. set of wires). In the first case, the result is a circular closure (resp. set of mutually recursive closures). In the second case, the recursively defined values correspond to *cycles* in the network. These related rules are given in Sec. 3.6.1 and 3.6.2 respectively.

3.6.1 Recursive functions

$$\boxed{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_1 = \mathit{exp}_1 \ \dots \ \mathit{pat}_n = \mathit{exp}_n \Rightarrow \mathrm{E}', \mathrm{B}', \mathrm{W}}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n, \quad \mathit{pat}_i = \mathrm{id}_i, \quad \mathit{exp}_i = \textbf{fun}\ \mathit{pat}'_i \to \mathit{exp}'_i \\ \forall i.\ 1 \le i \le n, \quad \mathrm{E}'_i = [\mathrm{id}_i \mapsto \mathsf{Clos}\langle \mathit{pat}'_i, \mathit{exp}'_i, \mathrm{E}'\rangle] \\ \mathrm{E}' = \bigoplus_{i=1}^{n} \mathrm{E}'_i\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_1 = \mathit{exp}_1 \ \dots \ \mathit{pat}_n = \mathit{exp}_n \Rightarrow \mathrm{E}', \varnothing, \varnothing} \quad (\textsc{RecBindingsF})$$

3.6.2 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, a pair of *recursive* value and box environments E_r and B_r are created by binding each identifier occuring in the LHS patterns, and not designating an output, to the location of a temporary, freshly created, box with the special tag rec :

$$\boxed{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathrm{Pat} \Rightarrow \mathrm{E}_r, \mathrm{B}_r}$$

$$\frac{\begin{array}{c}\mathrm{E}(\mathrm{id}) = \mathsf{Loc}\langle \mathrm{l}', \mathrm{s}' \rangle \\ \mathrm{B}(\mathrm{l}') \neq \mathsf{Box}\langle \mathsf{snk}, ., . \rangle \\ \mathrm{l} \notin \mathit{Dom}(\mathrm{B}) \\ \mathrm{b} = \mathsf{Box}\langle \mathsf{rec}, [1 \mapsto 0], [1 \mapsto \varnothing] \rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathrm{id} \Rightarrow [\mathrm{id} \mapsto \mathsf{Loc}\langle \mathrm{l}, 0 \rangle],\ [\mathrm{l} \mapsto \mathrm{b}]} \qquad (\textsc{RPatVar})$$

$$\frac{\forall i.\ 1 \leq i \leq n, \quad \vdash_{rec} \mathit{pat}_i \Rightarrow \mathrm{E}_i,\ \mathrm{B}_i}{\vdash_{rec} (\mathit{pat}_1, \ldots, \mathit{pat}_n) \Rightarrow \bigoplus_{i=1}^{n} \mathrm{E}_i,\ \bigoplus_{i=1}^{n} \mathrm{B}_i} \qquad (\textsc{RPatTuple})$$

2. Second, all the RHS expressions are evaluated in environments augmented with E_r and B_r, and the resulting values are bound, as in the non-recursive case, to the LHS patterns.

3. Third, the temporary rec boxes are removed from the resulting graph.

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n, \quad \mathit{exp}_i \neq \textbf{fun}\ \mathit{pat}'_i \rightarrow \mathit{exp}'_i \\ \forall i.\ 1 \leq i \leq n,\ \mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_i \Rightarrow \mathrm{E}_i, \mathrm{B}_i \\ \mathrm{E}_r = \bigoplus_{i=1}^{n} \mathrm{E}_i \quad \mathrm{B}_r = \bigoplus_{i=1}^{n} \mathrm{B}_i \\ \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \oplus \mathrm{E}_r, \mathrm{B} \oplus \mathrm{B}_r \vdash \mathit{exp}_i \Rightarrow \mathrm{v}_i, \mathrm{B}'_i, \mathrm{W}'_i \\ \mathrm{B}' = \bigoplus_{i=1}^{n} \mathrm{B}'_i \quad \mathrm{W}' = \bigoplus_{i=1}^{n} \mathrm{W}'_i \\ \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \oplus \mathrm{E}_r, \mathrm{B} \oplus \mathrm{B}_r \oplus \mathrm{B}' \vdash_{pat} \mathit{pat}_i, \mathrm{v}_i \Rightarrow \mathrm{E}'_i, \mathrm{B}''_i, \mathrm{W}'''_i \\ \mathrm{E}'' = \bigoplus_{i=1}^{n} \mathrm{E}'_i \quad \mathrm{B}'' = \bigoplus_{i=1}^{n} \mathrm{B}''_i \quad \mathrm{W}'' = \bigoplus_{i=1}^{n} \mathrm{W}''_i \\ \langle \mathrm{B}''', \mathrm{W}''' \rangle = \langle \mathrm{B}'', \mathrm{W}'' \rangle \ominus \mathrm{B}_r\end{array}}{\mathrm{E}, \mathrm{B} \vdash_{rec} \mathit{pat}_1 = \mathit{exp}_1\ \ldots\ \mathit{pat}_n = \mathit{exp}_n \Rightarrow \mathrm{E}', \mathrm{B}'', \mathrm{W}''} \qquad (\textsc{RecBindingsV})$$

where $\ominus$, when applied to a graph, represented as a pair of box and wires environments, and a a box environments, denotes the operation of removing all boxes occuring the latter from the former, shortening the corresponding paths of wires accordingly.