An Elixir program is a sequence of module definitions. When a module definition contain a `@callback` declaration, this module is considered a *behaviour*. Behaviours are a kind of hybrid definitions since they may define both a module type and (in case they export functions) a module value. This ambiguity is resolved the compiler, which will generate a module type definition for the behaviour, and a module value definition for the implementation of the behaviour.

For the sake of simplicity, we will suppose that this distinction is already done in the program. In the rest of the document, then, we will suppose that a program is a sequence of module type definitions, followed by the `Main` module definition, containing just a list of bindings.

$$\overline{\mathsf{defmodtype}\ X\ \mathsf{do}\ \overline{P}\ \overline{D}\ \mathsf{end}}\ \mathsf{defmodule}\ \mathit{Main}\ \mathsf{do}\ \overline{B}\ \mathsf{end}$$

Intuitively, this list of bindings $\overline{B}$ will the list of the toplevel modules definitions of the program, of the form

$$\mathsf{defmodule}\ X\ \mathsf{do}\ \overline{P}\ \overline{S}\ \overline{B}\ \mathsf{end} \tag{1}$$

and every module in the program is then denoted by a unique path starting with `Main` module name.

The full syntax of the surface language is given in Figure 1. In Elixir there is a unique syntactic category for names, although it uses the naming convention of using capitalized names for modules and (thus) behaviours. To enhance clarity, we use x to range over expression variables, X for module names, X for type names, and $\mathcal{X}$ for behaviour names. Let us examine in detail the definition of a module, as given in (1).

The definition of a module X is a sequence of three blocks: the list $\overline{P}$ of type parameters of the form $\mathsf{\$param}\ \mathrm{X}$, the list $\overline{S}$ of behaviours declarations of the form $\mathsf{\$behaviour}\ \mathcal{X}$, and the definition of the module itself, which is a sequence of bindings $\overline{B}$. In Elixir, these three blocks may be interleaved, but for the sake of simplicity we will suppose that they are grouped together in the order given above. The module type parameters are used to define the type of the module and are instantiated when the module is used; the list of behaviours specifies the behaviours that this module implements. The bindings $\overline{B}$ are used to define the actual implementation of the module, which may include definitions of values, types, and modules.

A type t is bound to a name X which may be transparent or opaque, as defined by the syntax $\mathsf{\$type}\ \mathrm{X} = t$ and $\mathsf{\$opaque}\ \mathrm{X} = t$ respectively. An opaque type definition make the binding $\mathrm{X} = t$ visible only inside the module, while a transparent make this binding visible outside the module and is in general used to instantiate the parameters of the behaviours the module implements. Thus for instance if the `StackB` behaviour is parametrized in the type `elem` of its element, a module `StackInt` implementing the behaviour for integers will be defined as

```
1  defmodule StackInt do
2    $behaviour StackB
3    $type elem = integer()
4    $opaque stack = list(integer())
5
6    $ () -> stack
7    def new = fn() -> [] end
8
9    $ (stack, elem) -> stack
10   def push = fn(s, e) -> [e|s] end
11
12   $ (stack) -> {integer(),stack}
13   def pop = fn(s) -> {hd(s), tl(s)}
14 end
```

Of course we can also define the module stack to be parametric in the type of its elements, and then pass this parameter as argument of the behaviour, as in the following example, in which we replace `pop` by `top` and we add a higher-order function

```
15 defmodule StackModule do
16   $param elem
17   $behaviour StackB
18   $type elem = elem
19   $opaque stack = list(elem)
20
```

$$
\begin{array}{rcl}
\Sigma & ::= & \mathsf{defmodtype}\ \mathcal{X}\ \mathsf{do}\ \overline{P}\ \overline{D}\ \mathsf{end} \\
N & ::= & x \\
 & | & X \\
S & ::= & \$\mathsf{behaviour}\ \mathcal{X} \\
P & ::= & \$\mathsf{param}\ \mathrm{X} \\
B & ::= & \mathsf{defmodule}\ X\ \mathsf{do}\ \overline{P}\ \overline{S}\ \overline{B}\ \mathsf{end} \\
 & | & x = v \\
 & | & \$\mathsf{private}\ x = v \\
 & | & \$\mathsf{type}\ \mathrm{X} = t \\
 & | & \$\mathsf{opaque}\ \mathrm{X} = t \\
M & ::= & X[\overline{\mathrm{X} = t}] \\
 & | & M.M \\
E & ::= & v \\
 & | & x \\
 & | & \mathsf{let}\ N = E\ \mathsf{in}\ E \\
 & | & E(\overline{E}) \\
 & | & \%\{\overline{\ell = E}\} \\
 & | & E.\ell \\
 & | & (E \in t)?E : E \\
 & | & M.x \\
 & | & M \\
v & ::= & c \\
 & | & \%\{\overline{\ell = v}\} \\
 & | & \$ \bigwedge \overline{(\ \overline{t}\ ) \to t}\ \mathsf{fn}\ \overline{x} \to E \\
 & | & \$ \bigcap \overline{(\overline{N : T}) \to T}\ \mathsf{fn}\ \overline{N} \to E \\
T & ::= & t \\
 & | & (\overline{N : T}) \to T \\
 & | & [\overline{\mathrm{X} : \star}] \to T \\
 & | & \{\overline{S}; \overline{D}\} \\
t & ::= & \mathsf{int} \\
 & | & (\overline{t}) \to t \\
 & | & \%\{\overline{f}\} \\
 & | & t \vee t \\
 & | & t \wedge t \\
 & | & \neg t \\
 & | & \alpha \\
 & | & \mathbb{0} \\
 & | & M.\mathrm{X} \\
 & | & \mathcal{X}\ [\overline{\mathrm{X} = t}] \\
D & ::= & \$\mathsf{module}\ X : T \\
 & | & \$\mathsf{callback}\ x : \bigcap \overline{T} \\
 & | & \$\mathsf{opaque}\ \mathrm{X} \\
 & | & \$\mathsf{type}\ \mathrm{X} = t
\end{array}
$$

do we want this syntax or remove it and have $\{\overline{P}; \overline{S}; \overline{D}\}$

Figure 1: Syntax of the surface language

```
  $ () -> stack
  def new = fn() -> [] end

  $ (stack, elem) -> stack
  def push = fn(s, e) -> [e|s] end

  $ (stack) -> elem
  def top = fn(s) -> hd(s) end

  $ (X: StackB[elem=elem], sx: X.stack, Y: StackB[elem=elem], sy: Y.stack) -> Y.stack
  def move (X, sx, Y, sy) do Y.push(sy,X.top(sx)) end
```

A value v is bound to a name x which may be exported or private, as defined by the syntax $x = v$ and $\$\mathsf{private}\, x = v$ respectively. In general a value v is either a constant or a function. In the latter case the function is defined in Elixir by the syntax `def` and `defp` for exported and private functions respectively, and prefixed by their type annotations as in the following example:

```
defmodule X do
  $ (t1 -> t2) and (t3 -> t4)
  def f(x,y) do E1 end

  $ (X : StackModule[elem=integer()], s: X.stack) -> integer()
  defp g(X,s) do E2 end
end
```

The above examples are considered as syntactic sugar and in the syntax of Figure 1 are rendered the following definitions

```
defmodule X do
  f = $(t1 → t2) ∧ (t3 → t4) fn x, y -> E1
  g = private $(X : Stack[elem=integer()], s: X.stack) → integer() fn X, s -> E2
end
```

A module has a type

B	$::=$	$X = \mathsf{fn}(\overline{\mathrm{x} : \star}) \to \{\overline{B}\}_{\overline{\mathcal{X}}}$	module X with parameters $\overline{x}$ implementing $\overline{\mathcal{X}}$
	$\mid$	$x = v$	exported definition
	$\mid$	$x = \mathsf{private}\ v$	private definition
	$\mid$	$\mathrm{x} = \mathsf{type}\ t$	transparent type or behaviour argument
	$\mid$	$\mathrm{x} = \mathsf{opaque}\ t$	opaque type
M	$::=$	$X[\overline{\mathrm{x} = t}]$	local module
	$\mid$	$M.M$	imported module
N	$::=$	x	
	$\mid$	X	
E	$::=$	v	
	$\mid$	x	
	$\mid$	$\mathsf{let}\ N = E\ \mathsf{in}\ E$	
	$\mid$	$E(\overline{N})$	
	$\mid$	$\%\{\overline{\ell = E}\}$	
	$\mid$	$E.\ell$	
	$\mid$	$(E \in t)?E : E$	
	$\mid$	$M.x$	
	$\mid$	M	
v	$::=$	c	
	$\mid$	$\%\{\overline{\ell = v}\}$	
	$\mid$	$\$ \bigwedge \overline{(\,\overline{t}\,) \to t}\ \mathsf{fn}\ \overline{x} \to E$	
	$\mid$	$\$ \bigcap \overline{(\overline{N : T}) \to T}\ \mathsf{fn}\ \overline{N} \to E$	
T	$::=$	t	
	$\mid$	$(\overline{N : T}) \to T$	
	$\mid$	$\{\overline{D}\}_{\overline{\mathcal{X}}}$	
	$\mid$	$(\mathrm{x} : \star) \to T$	we can include this case in the above by adding $\star$ to T and x to N
	$\mid$	$\star$	I would remove it: we do not want to pass around modules unless they are fully instantiated $f(X : (\mathrm{x} : \star) \to ...) \to ...X[\mathrm{x} = \mathsf{int}]...$
t	$::=$	int	
	$\mid$	$(\overline{t}) \to t$	
	$\mid$	$\%\{\overline{f}\}$	
	$\mid$	$t \vee t$	
	$\mid$	$t \wedge t$	
	$\mid$	$\neg t$	
	$\mid$	α	
	$\mid$	$\mathbb{0}$	
	$\mid$	$M.\mathrm{x}$	the type x imported from module M
	$\mid$	$\mathcal{X}[\overline{\mathrm{x} = t}]$	a behaviour
D	$::=$	$X : T$	exports a module X of type T
	$\mid$	$x : \bigcap \overline{T}$	exports a value x of type $\bigcap \overline{T}$
	$\mid$	$\mathrm{x} : \star$	exports an opaque type x
	$\mid$	$\mathrm{x} : [= t]$	exports a transparent type $x = t$

where $\mathcal{X}$ ranges over behaviour names, X over module names, x over variable names, and ℓ over map keys.

Figure 2: Syntax of the surface language

$$
\begin{array}{rcl}
\tau & ::= & t \\
& | & \star \\
& | & \left(\overline{N : \tau}\right) \to \tau \\
& | & \mathsf{like}\left(\overline{X\ \left[\overline{x = t}\right]}.X\ \left[\overline{x = t}\right]\right) \\
& | & \cap \overline{\tau} \\
& | & X\ \left[\overline{x = t}\right]
\end{array}
$$

Figure 3: Syntax of surface module types

$$
\textsc{ElEnv-Empty}\ \frac{}{\Sigma; \Gamma \vdash \epsilon}
\qquad
\textsc{ElEnv-Expr}\ \frac{\Sigma; \Gamma \vdash t : \star}{\Sigma, \Gamma \vdash x : t, \Gamma}
\qquad
\textsc{ElEnv-Type}\ \frac{\Sigma; \Gamma \vdash t : \star}{\Sigma; \Gamma \vdash x = t, \Gamma}
$$

$$
\textsc{ModEnv-ModuleType}\ \frac{\Sigma; \Gamma, \overline{x : \star} \vdash \left\{\overline{D}\right\}}{\Sigma; \Gamma \vdash X = \overline{x} \mapsto \overline{D}, \Sigma}
\qquad
\textsc{ModEnv-Empty}\ \frac{}{\Sigma; \Gamma \vdash \epsilon}
\qquad
\textsc{ModEnv-Module}\ \frac{\Sigma; \Gamma, \overline{x : \star} \vdash \left\{; \overline{B}\right\}}{\Sigma; \Gamma \vdash X = \overline{x} \mapsto \overline{B}, \Sigma}
$$

Figure 4: Formation rules for environments

$$
\textsc{EqPath-Empty}\ \frac{}{\Sigma; \Gamma \vdash \epsilon \cong \epsilon}
\qquad
\textsc{EqPath-Add}\ \frac{\Sigma; \Gamma \vdash P_1 \cong P_2 \qquad \forall i. \Sigma, \Gamma \vdash t_i \cong t_i'}{\Sigma; \Gamma \vdash P_1.X\ \left[\overline{x_i = t_i}\right] \cong P_2.X\ \left[\overline{x_i = t_i'}\right]}
$$

Figure 5: Rules for path equivalence

$$
\textsc{WF-Behave}\ \frac{\Sigma; \Gamma \vdash X : \overline{x : \star} \to \{D\} \qquad \Sigma; \Gamma \vdash t_i : \star}{\Sigma; \Gamma \vdash X[\overline{x_i = t_i}]}
\qquad
\textsc{WF-Field}\ \frac{\Sigma; \Gamma \vdash P : \{\$\mathsf{type}\, x = t\} \cup \{\$\mathsf{opaque}\, x\}}{\Sigma; \Gamma \vdash P.x}
\qquad
\textsc{WF-Star}\ \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma \vdash \star}
$$

$$
\textsc{WF-Function}\ \frac{\Sigma; \Gamma \vdash \qquad \forall 0 \le j \le n. \Sigma; \Gamma, \overline{N_i : T_i}^{i=1,\ldots,j} \vdash T_{j+1}}{\Sigma; \Gamma \vdash \left(\overline{N_i : T_i}^{i=1,\ldots,n}\right) \to T_{n+1}}
$$

$$
\textsc{WF-ModuleTypeNames}\ \frac{\begin{array}{c}\forall S_i \in \overline{S}. \Sigma(S_i) = \overline{x : \star}^i \mapsto \overline{D}^i \\ \left\{\overline{D}\right\} \preccurlyeq \left\{\overline{\overline{x : [= \_]}^i\ \overline{D}^i}\right\} \qquad \forall i \ne j. (\mathsf{dom}\left(\overline{D}^i\right) \# \mathsf{dom}\left(\overline{D}^j\right) \text{ and } \overline{x}^i \# \overline{x}^j) \qquad \Sigma; \Gamma \vdash \left\{\overline{D}\right\}\end{array}}{\Sigma; \Gamma \vdash \left\{\overline{S}; \overline{D}\right\}}\ (S \ne \epsilon)
$$

$$
\textsc{WF-ModuleTypeModule}\ \frac{\Sigma; \Gamma \vdash T \qquad \Sigma; \Gamma, X : T \vdash \left\{\overline{D}\right\}}{\Sigma; \Gamma \vdash \left\{(\$\mathsf{module}\, X : T)\overline{D}\right\}}
\qquad
\textsc{WF-ModuleTypeOpaque}\ \frac{\Sigma; \Gamma, x : \star \vdash \left\{\overline{D}\right\}}{\Sigma; \Gamma \vdash \left\{(\$\mathsf{opaque}\, x)\overline{D}\right\}}
$$

$$
\textsc{WF-ModuleTypeType}\ \frac{\Sigma; \Gamma \vdash t \qquad \Sigma; \Gamma, x : [= t] \vdash \left\{\overline{D}\right\}}{\Sigma; \Gamma \vdash \left\{(\$\mathsf{type}\, x = t)\overline{D}\right\}}
\qquad
\textsc{WF-ModuleTypeCallback}\ \frac{\overline{\Sigma; \Gamma \vdash T} \qquad \Sigma; \Gamma, x : \bigcap \overline{T} \vdash \left\{\overline{D}\right\}}{\Sigma; \Gamma \vdash \left\{(\$\mathsf{callback}\, x : \bigcap \overline{T})\overline{D}\right\}}
$$

Figure 6: Well-formedness rules for types $\boxed{\Sigma; \Gamma \vdash T}$

$$\textsc{Bind-DefModule}\ \frac{\Sigma;\Gamma,\overline{P:\star}\vdash\overline{B}:\overline{D}\qquad\Sigma;\Gamma,X:\left(\overline{P:\star}\right)\to\left\{\overline{S};\overline{D}\right\}\vdash\overline{B_0}:\overline{D_0}}{\Sigma;\Gamma\vdash\left(\mathsf{defmodule}\,X\,\mathsf{do}\,\overline{PSB}\,\mathsf{end}\right)\overline{B_0}:\left(X:\left(\overline{P:\star}\right)\to\left\{\overline{S};\overline{D}\right\}\right)\overline{D_0}}$$

$$\textsc{Bind-Type}\ \frac{\Sigma;\Gamma\vdash t:\star\qquad\Sigma;\Gamma,x:[=t]\vdash\overline{B}:\overline{D}}{\Sigma;\Gamma\vdash(\$\mathsf{type}\,x=t)\,\overline{B}:(x:[=t])\,\overline{D}}$$

$$\textsc{Bind-Opaque}\ \frac{\Sigma;\Gamma\vdash t:\star\qquad\Sigma;\Gamma,x:[=t]\vdash\overline{B}:\overline{D}}{\Sigma;\Gamma\vdash(\$\mathsf{opaque}\,x=t)\,\overline{B}:(x:\star)\,\overline{D}}$$

$$\textsc{Bind-Empty}\ \frac{}{\Sigma;\Gamma\vdash\epsilon:\epsilon}$$

$$\textsc{Bind-Value}\ \frac{\Sigma;\Gamma\vdash v:\cap\overline{T}\qquad\Sigma;\Gamma,x:\cap\overline{T}\vdash\overline{B}:\overline{D}}{\Sigma;\Gamma\vdash(x=v)\,\overline{B}:\left(x:\cap\overline{T}\right)\overline{D}}$$

Figure 7: Typing rules for bindings $\boxed{\Sigma;\Gamma\vdash\overline{B}:\overline{D}}$

$$\textsc{Path-Empty}\ \frac{}{\Sigma;\Gamma\vdash\epsilon:\{\epsilon;\Gamma\}}$$

$$\textsc{Path-SubModule}\ \frac{\Sigma;\Gamma\vdash P:\left\{\_;\overline{D}\right\}\qquad\overline{D}=\ldots,\mathsf{defmodule}\,X\,\mathsf{do}\,\overline{PSB}\,\mathsf{end},\ldots\qquad\overline{P}=\overline{\$\mathsf{param}\,x}\qquad\Sigma;\Gamma\vdash\overline{t}}{\Sigma;\Gamma\vdash P.X\left[\overline{x=t}\right]:\left\{\overline{S};\overline{\$\mathsf{type}\,x=t}\,\overline{B}\right\}}$$

Figure 8: Well-formdness rules for paths

$$\textsc{Type-Variable}\ \frac{\Sigma;\Gamma\vdash P:\left\{\overline{S};\overline{D}(x:\cap T)\overline{D'}\right\}}{\Sigma;\Gamma\vdash P.x:\cap T}$$

$$\textsc{Type-Subsumption}\ \frac{\Sigma;\Gamma\vdash T\qquad\Sigma;\Gamma\vdash E:\cap\overline{T'}\qquad\Sigma;\Gamma\vdash\cap\overline{T'}\preccurlyeq T}{\Sigma;\Gamma\vdash E:T}$$

$$\textsc{Type-BigFunction}\ \frac{\Sigma;\Gamma\vdash\cap\overline{\overline{N:T}\to T'}\qquad\overline{\Sigma;\Gamma,\overline{N:T}\vdash E:T'}}{\Sigma;\Gamma\vdash\$\cap\overline{\forall\overline{\alpha}\left(\overline{N:T}\right)\to T'}\,\mathsf{fn}\,\overline{N}\to E:\cap\overline{\overline{N:T}\to T'}}$$

$$\textsc{Type-Module}\ \frac{\Sigma;\Gamma\vdash P:\left\{\overline{S_0};\overline{D_0}(X:\overline{y:\star}\to\left\{\overline{S};\overline{D}\right\})\overline{D_1}\right\}\qquad\Sigma;\Gamma\vdash\overline{t}\qquad\overline{x}\simeq\overline{y}}{\Sigma;\Gamma\vdash P.X\left[\overline{x=t}\right]:\left\{\overline{S};\overline{D}\right\}}$$

Figure 9: Typing rules for the surface language

$$\textsc{Sub-StarRefl}\ \frac{}{\star \preccurlyeq \star} \qquad \textsc{Sub-Elixir}\ \frac{t \preccurlyeq t'}{\Sigma;\Gamma \vdash t \preccurlyeq t'} \qquad \textsc{Sub-Intersection}\ \frac{\exists i \in I, T_i \preccurlyeq T}{\Sigma;\Gamma \vdash \cap_I \overline{T_i} \preccurlyeq T}$$

$$\textsc{Sub-ModuleLeft}\ \frac{\Sigma;\Gamma \vdash P : \left\{\overline{S}; \overline{D}(x : [= t])\overline{D'}\right\} \qquad \Sigma;\Gamma \vdash t \preccurlyeq T}{\Sigma;\Gamma \vdash P.x \preccurlyeq T}$$

$$\textsc{Sub-ModuleRight}\ \frac{\Sigma;\Gamma \vdash P : \left\{\overline{S}; \overline{D}(x : [= t])\overline{D'}\right\} \qquad \Sigma;\Gamma \vdash \cap\overline{T} \preccurlyeq t}{\Sigma;\Gamma \vdash \cap\overline{T} \preccurlyeq P.x}$$

$$\textsc{Sub-Opaque}\ \frac{\Sigma;\Gamma \vdash P \cong P' \qquad \Sigma;\Gamma \vdash P : \left\{\overline{S}; \overline{D}(x : \star)\overline{D'}\right\}}{\Sigma;\Gamma \vdash P.x \preccurlyeq P'.x}$$

$$\textsc{Sub-BigFunction}\ \frac{\forall i. \Sigma;\Gamma, X_1 : R_1, \ldots, X_{i-1} : R_{i-1} \vdash T_i \succcurlyeq R_i \qquad \Sigma;\Gamma, \overline{X_i : R_i} \vdash T' \preccurlyeq R'}{\Sigma;\Gamma \vdash (X_i : T_i) \to T' \preccurlyeq (X_i : R_i) \to R'}$$

Figure 10: Subtyping rules $\boxed{\Sigma;\Gamma \vdash \cap\overline{T} \preccurlyeq T}$

$$\textsc{Sub-BehBeh}\ \frac{\Sigma;\Gamma \vdash t \cong t'}{\Sigma;\Gamma \vdash X\ \overline{[x = t]} \preccurlyeq X\ \overline{[x = t']}} \qquad \textsc{Sub-BehMod}\ \frac{}{\Sigma;\Gamma \vdash X\ \overline{[x = t]} \preccurlyeq \Sigma(X)\ (\overline{t})}$$

$$\textsc{Sub-ModBeh}\ \frac{t_i = \left\{\overline{D}\right\}_\xi.x_i \qquad X \in \xi}{\Sigma;\Gamma \vdash \left\{\overline{D}\right\}_\xi \preccurlyeq X\ \overline{[x_i = t_i]}} \qquad \textsc{Sub-TypeType}\ \frac{\Sigma;\Gamma \vdash t \cong t'}{[= t] \preccurlyeq [= t']} \qquad \textsc{Sub-TypeOpaque}\ \frac{}{\Sigma;\Gamma \vdash [= t] \preccurlyeq \star}$$

$$\textsc{Sub-ModMod}\ \frac{\xi' \subseteq \xi \qquad \left\{\overline{D}\right\} \preccurlyeq \left\{\overline{D'}\right\}}{\Sigma;\Gamma \vdash \left\{\overline{D}\right\}_\xi \preccurlyeq \left\{\overline{D'}\right\}_{\xi'}}$$

Figure 11: New subtyping rules

$$[\![[= t]]\!] = \text{type} \times ([\![t]\!] \times [\![\neg t]\!])$$
$$[\![\star]\!] = \text{type} \times ([\![\mathbb{1}]\!] \times [\![\mathbb{1}]\!])$$

Figure 12: Semantic interpretation of the new constructions