A parsimonious module system

TODO

1 Introduction

TODO: BLABLA Intro

We propose an *expressive*, *syntactic* and *parsimonious* module system. By expressive, we mean that we support a rich subset of features that have so far not been formalized together and allow the user for rich manipulation of modules, notably transparent ascriptions, module aliases, enrichment constraints and applicative functors. Unlike work like **?**, our rules are defined *syntactically*, directly on the concrete syntax. This makes the rule easier to grasp, make deriving an inference algorithm almost immediate, and ensure that error messages and typing information are easier to surface to the user. Finally, this system is *parsimonious*, in that it does the minimum amount of expansion required by using ideas similar to explicit substitutions for module operations. Module types can be quite large, and limiting the expansion can simplify error messages and improve the speed of the typechecker.

2 Motivations

2.1 The aliasing problem

Several advanced module systems allow "module aliases". If we consider a declaration of the form:

```
module Alias = Original
```

In the usual presentation, `Alias` would have the module type `S`, which is the expanded module type of `Original`. With module aliases, `Alias` is exactly of type `(= Original)`, i.e., a singleton type **?** that uniquely identify it as an alias of `Original`. This works well with the applicative behavior of functor used notably by OCaml, allowing us to preserve `F(Alias) = F(Original)` for any applicative functor `F`.

Unfortunately, this also leads to subtle issues with functors and subtyping. Let us consider the following examples:

```
module F (X : S) = struct
  module A = X (* this is not an alias *)
end
```

What should be the type of `F(M).A`? If we keep the singleton type `A : (= X)`, we would have `F(M).A : (= M)`. This would mean that `F(M).A` is exactly the same module as `M`. This is however not the case: indeed, when given as argument to the functor, `M` was restricted, by subtyping, to the signature `S`. Inside the body of the functor, we only get a restricted view of `M` through the lense of `S` and `A` can only present this restricted view. In practice, this subtyping is implemented by copy: `X` is a new module which only contains the fields of `M` that are visible in `S`. If users outside the functor can observe the equality `F(M).A = M`, they might try to access to fields that were not copied and are thus not really present in the module. **TODO: Make all this a bit clearer, probably with a schema**

One solution to this problem is *transparent ascriptions*, a type of ascription that still hides values, but purposefully "leaks through" types and other static fields. We propose to add transparent ascriptions directly in module paths, allowing to use them in module aliases. This gives us an immediate solution to the identified unsafety while still preserving all the possible sharing. We obtain the following type for the example above:

```
module F (X : S) : sig
  module A = X (* This is a real alias *)
end
```

However, once we apply `F`, we obtain:

```
module Res = F(M)
> Module Res : sig
>   module A = (M <: S) (* X viewed through S *)
> end
```

Since we now have `F(M).A = (M <: S)`, the sharing between `A` and `M` is properly exposed, but it still restricts the access to dynamic fields that are not present in `S`.

2.2 The avoidance problem

TODO: Explain avoidance problem more

```
module F (X : S) = struct
  type a = X.t list
  type b = X.t * int
end
```

What's the type of:

```
module A = F(struct type t = A | B end)
```

The typechecker gives us the following un-helpful answer

```
module A : sig type a type b end
```

We propose to use local modules:

```
module A : let X : ... in sig
  type a = X.t list
  type b = X.t * int
end
```

2.3 Incrementality in module checking

Modern practical module systems **??** provide many additional operators not present in the original formulation. For instance in OCaml, "with constraints", noted `S with type t = int` allow to add an additional equality to a module type `S`. **TODO: Cite other operations**.

This operation are so far only syntactic: they are expanded immediately into a complete signature. For instance

```
module type S = sig
  val x : string
  module A : T
end
module M : S with type A.t = int
```

is immediately expanded as:

```
module M : sig  val x : string  module A : sig type t = int ... end  end
```

This yield a simple semantics, but causes numerous efficiency issues in practical systems that support large Ecosystems: indeed, the signature of the expanded version can be much larger than the original one. Our calculus, on the contrary, only expand when absolutely necessary, and keep the simplest form otherwise. This gives a presentation similar to explicit substitutions in more traditional type theories, with similar problematic and trade-offs: we want to minimize the amount of work, while still doing enough to ensure we do not delay errors.

The application of this technique throughout our system lead to a novel presentation of parsimonious subtyping that only pushes ascriptions up to aliases, but not further, ensuring that we maximize the sharing described in the previous section.

2.4 Contributions

We propose a novel module systems which:

- Uses *syntactic* rules, making it easier to reason about and implement as part of production typechecker with inference and error reporting
- Support *rich operations* on modules such as strengthening (**?**), with constraints, ...
- Support *transparent ascriptions in path*, which safely and conveniently extend the applicative behavior of functors to a context with singletons.
- Computes types and subtyping *parsimoniously*, using an approach based on explicit substitutions and input/output judgements.

3 An ML module calculus

We now introduce our module calculus. First, let us defined some conventions: lowercase meta-variables such as x, e or t are used for the core language; capitalized meta-variables such as X, M, S represent modules; calligraphic meta-variables such as $\mathcal{X}$, $\mathcal{M}$, $\mathcal{S}$ represent module types.

3.1 Syntax

The syntax is defined in Figure 1. The module expression language contains paths P, which might be variables X or qualified accesses $P.X$, parameterized modules i.e. *functors* $((X : \mathcal{M}) \to M)$, module type annotations i.e. *ascriptions*, both opaque $(M : \mathcal{M})$ and transparent $(M <: \mathcal{M})$, and finally structures $\texttt{struct}_X\ S\ \texttt{end}$. Structures contains a list of declarations, noted D, which can be value, type, module or module type bindings. Structures are also annotated with a "Self" variable, which represent the object through which other field in the module should be accessed.

In our core calculus, module types are stratified. $\mathcal{M}$ represent module types with operations, while $\mathcal{M}^\circ$ are raw module types without initial operations (i.e., in head normal form). Raw module types can be (qualified) module type variables $\mathcal{X}$ or $\mathcal{P}.\mathcal{X}$, functor types $(\mathcal{X} : \mathcal{M}_1) \to \mathcal{M}_2$, signatures $\texttt{sig}_X\ \mathcal{S}\ \texttt{end}$, and singletons $(= \mathcal{P})$. As with structures, signatures are a list of declarations noted $\mathcal{D}$, and a "Self" variable. Allowed operations strengthening a module by a path, noted $\mathcal{M}/\mathcal{P}$, enriching a module with additional constraints, noted $\mathcal{M}_{[\mathcal{C}]}$, and locally binding a module name, noted $\texttt{let}\ X : \mathcal{M}\ \texttt{in}\ \mathcal{M}'$.

As described before, there are two kinds of paths. The first, P, is mostly used for *dynamic* components, and is composed of a list of modules of the form $A.B.C$. The second, $\mathcal{P}$, is mostly used for *static* components such as types, and can additionally contain functor applications and transparent ascriptions, for instance $F(X).Y$ or $(X <: \mathcal{M}).Y$

Finally, the core language is left largely undefined in our calculus, except for qualified accesses.

$\mathcal{M}[X \mapsto Y]$ denotes the substitution of variable X by variable Y in $\mathcal{M}$. We only substitute variables by other variables.

3.2 Notations

Let us now define a few notations that we will use in the rest of this article. We will then describe how these various construction interact on specific features.

We define the following type judgements on modules. Note that the subtyping judgement also *returns* a module type, which is the result of the transparent ascription between the two modules.

$\Gamma \vdash M : \mathcal{M}$: The module M is of type $\mathcal{M}$ in Γ. See Figure 2.
$\Gamma \vdash \mathcal{M} <: \mathcal{M}' \rightsquigarrow \mathcal{M}_r$: The module type $\mathcal{M}$ is a subtype of $\mathcal{M}'$ in environment Γ, and their ascription is $\mathcal{M}_r$. See Figure 3.
$(\!|\mathcal{M}|\!)_\Gamma = \mathcal{M}^\circ$: The module type with operations $\mathcal{M}$ can be reduced to a module type without operations $\mathcal{M}^\circ$ in environment Γ. See Figures 4 to 6. **TODO: Change bracket style ...**

In our calculus, environments are simply signature on which we can "query" fields. To properly model our module calculus, we need to carefully consider how these environments are accessed. For this purpose, we consider the following operations on environments, which are defined in Figure 7

$\Gamma(P)$: Lookup the path P in Γ to a module or module type.
$\text{normalize}_\Gamma(\mathcal{P})$: Normalizes the path $\mathcal{P}$ in Γ to a canonical path.
$\text{resolve}_\Gamma(\mathcal{P})$: Resolve the path $\mathcal{P}$ in Γ to its shape (i.e., either an arrow or a signature).
$\text{shape}_\Gamma(\mathcal{M})$: Resolve the module $\mathcal{M}$ in Γ to its shape (i.e., either an arrow or a signature).

Path

$P ::= X \mid P.X$
$\mathcal{P} ::= X \mid \mathcal{P}.X \mid \mathcal{P}_1(\mathcal{P}_2) \mid (\mathcal{P} <: \mathcal{M})$

Module Expressions

$M ::= P$ (Variables)
$\mid (M : \mathcal{M})$ (Opaque Ascription)
$\mid (M <: \mathcal{M})$ (Transparent Ascription)
$\mid M_1(M_2)$ (Functor application)
$\mid (X : \mathcal{M}) \to M$ (Functor)
$\mid \texttt{struct}_X\ S\ \texttt{end}$ (Structure)

Structures

$S ::= \varepsilon \mid D; S$
$D ::= \texttt{let}\ x = e$ (Values)
$\mid \texttt{type}\ t = \tau$ (Types)
$\mid \texttt{module}\ X = M$ (Modules)
$\mid \texttt{module type}\ \mathcal{X} = \mathcal{M}$ (Module types)

Core language

$e ::= P.x$ (Qualified variable)
$\mid \ldots$ (Other expressions)
$\tau ::= \mathcal{P}.t$ (Qualified type)
$\mid \ldots$ (Other types)

Module types

$\mathcal{M}^\circ ::= \mathcal{X} \mid \mathcal{P}.\mathcal{X}$ (Variables)
$\mid (= \mathcal{P})$ (Alias)
$\mid (X : \mathcal{M}_1) \to \mathcal{M}_2$ (Functor)
$\mid \texttt{sig}_X\ \mathcal{S}\ \texttt{end}$ (Signature)
$\mathcal{M} ::= \mathcal{M}/\mathcal{P}$ (Strengthening)
$\mid \texttt{let}\ X : \mathcal{M}\ \texttt{in}\ \mathcal{M}'$ (Let)
$\mid \mathcal{M}_{[\mathcal{C}]}$ (Enrichment)
$\mid \mathcal{M}^\circ$

Enrichment

$\mathcal{C} ::= P.t = \tau$
$\mid P : \mathcal{M}$

Signatures

$\mathcal{S} ::= \varepsilon \mid \mathcal{D}; \mathcal{S}$
$\mathcal{D} ::= \texttt{val}\ x : \tau$ (Values)
$\mid \texttt{type}\ t = \tau$ (Types)
$\mid \texttt{type}\ t$ (Abstract types)
$\mid \texttt{module}\ X : \mathcal{M}$ (Modules)
$\mid \texttt{module type}\ \mathcal{X} = \mathcal{M}$ (Module types)

Environments

$\Gamma ::= \mathcal{S}$

Figure 1: Module language

$$\textsc{ModVar}\ \frac{P \in \Gamma}{\Gamma \vdash P : (= P)} \qquad \frac{\Gamma \vdash M : \mathcal{M}' \quad \Gamma \vdash \mathcal{M}' <: \mathcal{M} \rightsquigarrow \_}{\Gamma \vdash (M : \mathcal{M}) : \mathcal{M}} \qquad \frac{\Gamma \vdash M : \mathcal{M}' \quad \Gamma \vdash \mathcal{M}' <: \mathcal{M} \rightsquigarrow \mathcal{M}_r}{\Gamma \vdash (M <: \mathcal{M}) : \mathcal{M}_r}$$

$$\frac{\Gamma \vdash M_f : (X : \mathcal{M}_a) \to \mathcal{M}_r \quad Y \text{ fresh} \quad \Gamma \vdash M : \mathcal{M} \quad \Gamma \vdash \mathcal{M} <: \mathcal{M}_a \rightsquigarrow \mathcal{M}_c}{\Gamma \vdash M_f(M) : \texttt{let } Y : \mathcal{M}_c \texttt{ in } \mathcal{M}_r[X \mapsto Y]}$$

$$\frac{\Gamma \vDash \mathcal{M} \quad X \notin \Gamma \quad \Gamma; \texttt{module } X : \mathcal{M} \vdash M : \mathcal{M}'}{\Gamma \vdash (X : \mathcal{M}) \to M : (X : \mathcal{M}) \to \mathcal{M}'} \qquad \frac{\Gamma \vdash_X S : \mathcal{S}}{\Gamma \vdash \texttt{struct}_X\ S\ \texttt{end} : \texttt{sig}_X\ \mathcal{S}\ \texttt{end}} \qquad \frac{}{\Gamma \vdash_Y \varepsilon : \varepsilon}$$

$$\frac{\Gamma \vdash e : \tau \quad x \notin \Gamma(Y) \quad \Gamma; \texttt{val } Y.x : \tau \vdash_Y S : \mathcal{S}}{\Gamma \vdash_Y (\texttt{let } x = e; S) : (\texttt{val } x : \tau; \mathcal{S})} \qquad \frac{\Gamma \vdash M : \mathcal{M} \quad X \notin \Gamma(Y) \quad \Gamma; \texttt{module } Y.X : \mathcal{M} \vdash_Y S : \mathcal{S}}{\Gamma \vdash_Y (\texttt{module } X = M; S) : (\texttt{module } X : \mathcal{M}; \mathcal{S})}$$

$$\frac{\Gamma \vDash \tau \quad t \notin \Gamma(Y) \quad \Gamma; \texttt{type } Y.t = \tau \vdash_Y S : \mathcal{S}}{\Gamma \vdash_Y (\texttt{type } t = \tau; S) : (\texttt{type } t = \tau; \mathcal{S})} \qquad \frac{\Gamma \vDash \mathcal{M} \quad \mathcal{X} \notin \Gamma(Y) \quad \Gamma; \texttt{module type } Y.\mathcal{X} = \mathcal{M} \vdash_Y S : \mathcal{S}}{\Gamma \vdash_Y (\texttt{module type } \mathcal{X} = \mathcal{M}; S) : (\texttt{module type } \mathcal{X} = \mathcal{M}; \mathcal{S})}$$

Figure 2: Module typing rules – $\Gamma \vdash M : \mathcal{M}$

3.3 Structures and qualified accesses

3.4 Subtyping and transparent ascriptions

3.5 Functors

3.6 Module operations

$$\frac{\text{normalize}_\Gamma(\mathcal{P}) = \text{normalize}_\Gamma(\mathcal{P}')}{\Gamma \vdash \mathcal{P} = \mathcal{P}'} \qquad \frac{\Gamma \vdash (\!|\mathcal{M}|\!)_\Gamma <: (\!|\mathcal{M}'|\!)_\Gamma \rightsquigarrow \mathcal{M}''}{\Gamma \vdash \mathcal{M} <: \mathcal{M}' \rightsquigarrow \mathcal{M}''} \qquad \frac{\Gamma \vdash \Gamma(\mathcal{P}) <: \mathcal{M}^\circ \rightsquigarrow \mathcal{M}}{\Gamma \vdash \mathcal{P} <: \mathcal{M}^\circ \rightsquigarrow \mathcal{M}}$$

$$\frac{\Gamma \vdash \mathcal{M}^\circ <: \Gamma(\mathcal{P}) \rightsquigarrow \mathcal{M}}{\Gamma \vdash \mathcal{M}^\circ <: \mathcal{P} \rightsquigarrow \mathcal{M}} \qquad \frac{\Gamma \vdash \mathcal{P} = \mathcal{P}' \qquad \Gamma \vdash \text{resolve}_\Gamma(\mathcal{P}) <: \mathcal{M} \rightsquigarrow \mathcal{M}'' \qquad \Gamma \vdash \mathcal{M}'' <: \mathcal{M}' \rightsquigarrow \_}{\Gamma \vdash (= (\mathcal{P} <: \mathcal{M})) <: (= (\mathcal{P}' <: \mathcal{M}')) \rightsquigarrow (= (\mathcal{P}' <: \mathcal{M}'))}$$

$$\frac{\Gamma \vdash \mathcal{P} = \mathcal{P}' \qquad \Gamma \vdash \text{resolve}_\Gamma(\mathcal{P}) <: \mathcal{M} \rightsquigarrow \mathcal{M}'' \qquad \Gamma \vdash \mathcal{M}'' <: \text{resolve}_\Gamma(\mathcal{P}) \rightsquigarrow \_}{\Gamma \vdash (= (\mathcal{P} <: \mathcal{M})) <: (= \mathcal{P}') \rightsquigarrow (= \mathcal{P}')}$$

$$\frac{\Gamma \vdash \mathcal{P} = \mathcal{P}'}{\Gamma \vdash (= \mathcal{P}) <: (= (\mathcal{P}' <: \mathcal{M}')) \rightsquigarrow (= (\mathcal{P}' <: \mathcal{M}'))} \qquad \frac{\Gamma \vdash \mathcal{P} = \mathcal{P}'}{\Gamma \vdash (= \mathcal{P}) <: (= \mathcal{P}') \rightsquigarrow (= \mathcal{P}')}$$

$$\frac{\Gamma \vdash \text{resolve}_\Gamma(\mathcal{P}) <: \mathcal{M}^\circ \rightsquigarrow \_}{\Gamma \vdash (= \mathcal{P}) <: \mathcal{M}^\circ \rightsquigarrow (= (\mathcal{P} <: \mathcal{M}^\circ))} \qquad \frac{\Gamma \vdash \mathcal{M}'_a <: \mathcal{M}_a \rightsquigarrow \_ \qquad \Gamma, \texttt{module}\ X : \mathcal{M}'_a \vdash \mathcal{M}_r <: \mathcal{M}'_r \rightsquigarrow \mathcal{M}''_r}{\Gamma \vdash (X : \mathcal{M}_a) \to \mathcal{M}_r <: (X : \mathcal{M}'_a) \to \mathcal{M}'_r \rightsquigarrow (X : \mathcal{M}'_a) \to \mathcal{M}''_r}$$

$$\frac{\mathcal{B} = \mathcal{S} \times \mathcal{S}'[X' \mapsto X] \qquad \forall \mathcal{D}_x, \mathcal{D}'_x \in \mathcal{B}, \quad \Gamma; (\texttt{module}\ X = \texttt{sig}_X\ \mathcal{S}\ \texttt{end}) \vdash \mathcal{D}_x <: \mathcal{D}'_x \rightsquigarrow \mathcal{D}''_x}{\Gamma \vdash \texttt{sig}_X\ \mathcal{S}\ \texttt{end} <: \texttt{sig}_{X'}\ \mathcal{S}'\ \texttt{end} \rightsquigarrow \texttt{sig}_X\ \overline{\mathcal{D}''_x}\ \texttt{end}}$$

$$\frac{\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2 \rightsquigarrow \mathcal{M}}{\Gamma \vdash (\texttt{module}\ X : \mathcal{M}_1) <: (\texttt{module}\ X : \mathcal{M}_2) \rightsquigarrow (\texttt{module}\ X : \mathcal{M})}$$

$$\frac{\Gamma \vdash \mathcal{M}_1 <: (\texttt{sig}\ \ \texttt{end}) \rightsquigarrow \mathcal{M}}{\Gamma \vdash (\texttt{module}\ X : \mathcal{M}_1) <: \emptyset \rightsquigarrow (\texttt{module}\ X : \mathcal{M})}$$

$$\frac{\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2 \rightsquigarrow \mathcal{M}}{\Gamma \vdash (\texttt{module type}\ \mathcal{X} = \mathcal{M}_1) <: (\texttt{module type}\ \mathcal{X} = \mathcal{M}_2) \rightsquigarrow (\texttt{module type}\ \mathcal{X} = \mathcal{M})}$$

$$\frac{}{\Gamma \vdash (\texttt{module type}\ \mathcal{X} = \mathcal{M}) <: \emptyset \rightsquigarrow (\texttt{module type}\ \mathcal{X} = \mathcal{M})}$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash (\texttt{val}\ x : \tau_1) <: (\texttt{val}\ x : \tau_2) \rightsquigarrow (\texttt{val}\ x : \tau_2)} \qquad \frac{}{\Gamma \vdash (\texttt{val}\ x : \tau_1) <: \emptyset \rightsquigarrow \emptyset}$$

$$\frac{}{\Gamma \vdash (\texttt{type}\ t) <: (\texttt{type}\ t) \rightsquigarrow (\texttt{type}\ t)} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash (\texttt{type}\ t = \tau_1) <: (\texttt{type}\ t = \tau_2) \rightsquigarrow (\texttt{type}\ t = \tau_1)}$$

$$\frac{\Gamma \vdash t <: \tau}{\Gamma \vdash (\texttt{type}\ t) <: (\texttt{type}\ t = \tau) \rightsquigarrow (\texttt{type}\ t = \tau)} \qquad \frac{}{\Gamma \vdash (\texttt{type}\ t = \tau) <: (\texttt{type}\ t) \rightsquigarrow (\texttt{type}\ t = \tau)}$$

$$\frac{}{\Gamma \vdash (\texttt{type}\ t = \tau) <: \emptyset \rightsquigarrow (\texttt{type}\ t = \tau)} \qquad \frac{}{\Gamma \vdash (\texttt{type}\ t) <: \emptyset \rightsquigarrow (\texttt{type}\ t)}$$

Figure 3: Module subtyping rules – $\Gamma \vdash \mathcal{M} <: \mathcal{M}' \rightsquigarrow \mathcal{M}_r$

$$
\begin{aligned}
(\!|\mathcal{X}/\mathcal{P}|\!)_\Gamma &\to (\!|\Gamma(\mathcal{X})/\mathcal{P}|\!)_\Gamma \\
(\!|\mathcal{P}'.\mathcal{X}/\mathcal{P}|\!)_\Gamma &\to (\!|\Gamma(\mathcal{P}'.\mathcal{X})/\mathcal{P}|\!)_\Gamma \\
(\!|(= \mathcal{P}')/\mathcal{P}|\!)_\Gamma &\to (= \mathcal{P}') \\
(\!|((X : \mathcal{M}) \to \mathcal{M}')/\mathcal{P}|\!)_\Gamma &\to (X : \mathcal{M}) \to \mathcal{M}'/\mathcal{P}(X) \\
(\!|\mathtt{sig}_X\ \mathcal{S}\ \mathtt{end}/\mathcal{P}|\!)_\Gamma &\to \mathtt{sig}_X\ (\!|\mathcal{S}/\mathcal{P}|\!)_\Gamma\ \mathtt{end}
\end{aligned}
$$

$$
\begin{aligned}
(\!|\mathtt{type}\ t = \tau; \mathcal{S}/\mathcal{P}|\!)_\Gamma &\to \mathtt{type}\ t = \mathcal{P}.t; (\!|\mathcal{S}/\mathcal{P}|\!)_\Gamma \\
(\!|\mathtt{type}\ t; \mathcal{S}/\mathcal{P}|\!)_\Gamma &\to \mathtt{type}\ t = \mathcal{P}.t; (\!|\mathcal{S}/\mathcal{P}|\!)_\Gamma \\
(\!|\mathtt{module}\ X : \mathcal{M}; \mathcal{S}/\mathcal{P}|\!)_\Gamma &\to \mathtt{module}\ X : (= \mathcal{P}.X); (\!|\mathcal{S}/\mathcal{P}|\!)_\Gamma \\
(\!|\mathtt{module\ type}\ \mathcal{X} = \mathcal{M}; \mathcal{S}/\mathcal{P}|\!)_\Gamma &\to \mathtt{module\ type}\ \mathcal{X} = \mathcal{M}; (\!|\mathcal{S}/\mathcal{P}|\!)_\Gamma
\end{aligned}
$$

$$
\begin{aligned}
(\!|\mathcal{M}/\mathcal{P}/\mathcal{P}'|\!)_\Gamma &\to (\!|\mathcal{M}/\mathcal{P}|\!)_\Gamma \\
(\!|\mathcal{M}_{[\mathcal{C}]}/\mathcal{P}|\!)_\Gamma &\to (\!|(\!|\mathcal{M}_{[\mathcal{C}]}|\!)_\Gamma/\mathcal{P}|\!)_\Gamma
\end{aligned}
$$

Figure 4: Module strengthening operation – $\mathcal{M}/\mathcal{P}$

$$
\begin{aligned}
(\!|\mathcal{X}_{[\mathcal{C}]}|\!)_\Gamma &\to (\!|\Gamma(\mathcal{X})_{[\mathcal{C}]}|\!)_\Gamma \\
(\!|\mathcal{P}.\mathcal{X}_{[\mathcal{C}]}|\!)_\Gamma &\to (\!|\Gamma(\mathcal{P}.\mathcal{X})_{[\mathcal{C}]}|\!)_\Gamma \\
(\!|(= \mathcal{P})_{[\mathcal{C}]}|\!)_\Gamma &\to (= \mathcal{P}) && \text{when } (\!|\mathrm{resolve}_\Gamma(\mathcal{P})_{[\mathcal{C}]}|\!)_\Gamma \to \_ \\
(\!|((X : \mathcal{M}) \to \mathcal{M}')_{[\mathcal{C}]}|\!)_\Gamma &\to \mathtt{fail} \\
(\!|\mathtt{sig}_X\ \mathcal{S}\ \mathtt{end}_{[\mathcal{C}]}|\!)_\Gamma &\to \mathtt{sig}_X\ (\!|\mathcal{S}_{[\mathcal{C}]}|\!)_\Gamma\ \mathtt{end} \\
\\
(\!|\mathtt{type}\ t = \tau; \mathcal{S}_{[t=\tau']}|\!)_\Gamma &\to \mathtt{type}\ t = \tau'; \mathcal{S} && \text{when } \Gamma \vdash \tau' <: \tau \\
(\!|\mathtt{module}\ X : \mathcal{M}; \mathcal{S}_{[X:\mathcal{M}']}|\!)_\Gamma &\to \mathtt{module}\ X : \mathcal{M}'; \mathcal{S} && \text{when } \Gamma \vdash \mathcal{M}' <: \mathcal{M} \rightsquigarrow \_ \\
(\!|\mathtt{module}\ X : \mathcal{M}; \mathcal{S}_{[X.P.t=\tau]}|\!)_\Gamma &\to \mathtt{module}\ X : (\!|\mathcal{M}_{[P.t=\tau]}|\!)_\Gamma; \mathcal{S} \\
(\!|\mathtt{module}\ X : \mathcal{M}; \mathcal{S}_{[X.P:\mathcal{M}']}|\!)_\Gamma &\to \mathtt{module}\ X : (\!|\mathcal{M}_{[P:\mathcal{M}']}|\!)_\Gamma; \mathcal{S} \\
(\!|(\mathcal{D}; \mathcal{S})_{[\mathcal{C}]}|\!)_\Gamma &\to \mathcal{D}; (\!|\mathcal{S}_{[\mathcal{C}]}|\!)_\Gamma && \text{when } id(\mathcal{D}) \text{ is not a prefix of } id(\mathcal{C}) \\
\\
(\!|(\mathcal{M}/P)_{[\mathcal{C}]}|\!)_\Gamma &\to (\!|(\!|\mathcal{M}/P|\!)_{\Gamma_{[\mathcal{C}]}}|\!)_\Gamma \\
(\!|\mathcal{M}_{[\mathcal{C}']_{[\mathcal{C}]}}|\!)_\Gamma &\to (\!|(\!|\mathcal{M}_{[\mathcal{C}']}|\!)_{\Gamma_{[\mathcal{C}]}}|\!)_\Gamma
\end{aligned}
$$

Figure 5: Enrichment operation – $\mathcal{M}_{[\mathcal{C}]}$

$$(\!|\texttt{let } X : (= \mathcal{P}) \texttt{ in } \mathcal{M}|\!)_\Gamma \to \mathcal{M}[X \mapsto \mathcal{P}]$$
$$(\!|\texttt{let } X : \mathcal{M} \texttt{ in } \mathcal{M}'|\!)_\Gamma \to \mathcal{M}' \qquad \texttt{when } X \notin FV(\mathcal{M}')$$

TODO: Provide something more elaborate

Figure 6: Subst operation – $\texttt{let } X : \mathcal{M} \texttt{ in } \mathcal{M}'$

$$\frac{(\texttt{module } X : \mathcal{M}) \in \Gamma}{\Gamma(X) = \mathcal{M}} \qquad \frac{\text{shape}_\Gamma(\Gamma(\mathcal{P})) = \texttt{sig}_Y\ \mathcal{S}_1; \texttt{module } X : \mathcal{M}; \mathcal{S}_2 \texttt{ end}}{\Gamma(\mathcal{P}.X) = \mathcal{M}[Y \mapsto \mathcal{P}]}$$

$$\frac{(\texttt{module type } \mathcal{X} = \mathcal{M}) \in \Gamma}{\Gamma(\mathcal{X}) = \mathcal{M}} \qquad \frac{\text{shape}_\Gamma(\Gamma(\mathcal{P})) = \texttt{sig}_Y\ \mathcal{S}_1; \texttt{module type } \mathcal{X} = \mathcal{M}; \mathcal{S}_2 \texttt{ end}}{\Gamma(\mathcal{P}.\mathcal{X}) = \mathcal{M}[Y \mapsto \mathcal{P}]}$$

$$\frac{\Gamma(\mathcal{P}) = \mathcal{M} \qquad \Gamma \vdash \mathcal{M} <: \mathcal{M}' \rightsquigarrow \mathcal{M}_r}{\Gamma((\mathcal{P} <: \mathcal{M}')) = \mathcal{M}_r} \qquad \frac{\text{shape}_\Gamma(\Gamma(\mathcal{P}_f)) = (X : \mathcal{M}_a) \to \mathcal{M}_r \qquad \Gamma \vdash (= \mathcal{P}_a) <: \mathcal{M}_a \rightsquigarrow \mathcal{M}}{\Gamma(\mathcal{P}_f(\mathcal{P}_a)) = \texttt{let } Y : \mathcal{M} \texttt{ in } \mathcal{M}_r[X \mapsto Y]}$$

(a) Lookup rules – $\Gamma(P) = \mathcal{M}$

$$\text{normalize}_\Gamma(\mathcal{P}) = \begin{cases} \text{normalize}_\Gamma(\mathcal{P}') & \text{when } \Gamma(\mathcal{P}) = (= \mathcal{P}') \\ \mathcal{P} & \text{otherwise} \end{cases}$$

$$\text{resolve}_\Gamma(\mathcal{P}) = \Gamma(\text{normalize}_\Gamma(\mathcal{P}))/\,\text{normalize}_\Gamma(\mathcal{P})$$

(b) Path normalization and resolution

$$\text{shape}_\Gamma(\mathcal{M}) = \text{shape}_\Gamma((\!|\mathcal{M}|\!)_\Gamma)$$
$$\text{shape}_\Gamma((= \mathcal{P})) = \text{shape}_\Gamma(\text{resolve}_\Gamma(\mathcal{P}))$$
$$\text{shape}_\Gamma(\texttt{sig}_X\ \mathcal{S}\ \texttt{end}) = \texttt{sig}_X\ \mathcal{S}\ \texttt{end}$$
$$\text{shape}_\Gamma((X : \mathcal{M}) \to \mathcal{M}') = (X : \mathcal{M}) \to \mathcal{M}'$$

(c) Shape resolution – $\text{shape}_\Gamma(\mathcal{M}) = \mathcal{M}'$

Figure 7: Environment access, normalization and resolution