Metaprogramming without time travel in OxCaml

Jakub Bachurski <jbachurski@janestreet.com>

Jane Street

SPLS 2026, 10 June 2026

This work

In this talk

I will present our *draft* **type system design** for **runtime metaprogramming** using **staging** in **OxCaml**, Jane Street's open-source dialect of OCaml.

Primarily in collaboration with Leo White & inspired by the work of Kovács (2022).
Shout-outs: Chris Casinghino, Richard Eisenberg, Andrej Ivaskovic, Jack Rickard.

Code snippets are OCaml (OxCaml).
Sorry about the postfix notation in types – `int list`, not `List Int`.

Introduction: Staging expressions

Metaprogramming with staging

- Metaprogramming is the manipulation of program fragments as data.

Metaprogramming with staging

- Metaprogramming is the manipulation of program fragments as data.
- We use MetaML-style staging: we separate the metaprogram and generated program fragments using **staging annotations**.

Metaprogramming with staging

- Metaprogramming is the manipulation of program fragments as data.
- We use MetaML-style staging: we separate the metaprogram and generated program fragments using **staging annotations**.

Staging annotations

- Quotes `<[ – ]>` yield the program fragment inside them as a value (of type `expr`).
- Splices `$( – )` insert a program fragment (`expr`) at the given point in a quote.

Metaprogramming with staging

- Metaprogramming is the manipulation of program fragments as data.
- We use MetaML-style staging: we separate the metaprogram and generated program fragments using **staging annotations**.

Staging annotations

- Quotes `<[ − ]>` yield the program fragment inside them as a value (of type `expr`).
- Splices `$( − )` insert a program fragment (`expr`) at the given point in a quote.

Intuition: in quotes we go into the "future", and in splices into the "past".

$$\texttt{<[\$(}e\texttt{)]>} \equiv e \quad \text{and} \quad \texttt{\$(<[}e\texttt{]>)} \equiv e$$

Metaprogramming with staging

- Metaprogramming is the manipulation of program fragments as data.
- We use MetaML-style staging: we separate the metaprogram and generated program fragments using **staging annotations**.

Staging annotations

- Quotes `<[ − ]>` yield the program fragment inside them as a value (of type `expr`).
- Splices `$( − )` insert a program fragment (`expr`) at the given point in a quote.

Intuition: in quotes we go into the "future", and in splices into the "past".

$$\texttt{<[\$(}e\texttt{)]>} \equiv e \quad \text{and} \quad \texttt{\$(<[}e\texttt{]>)} \equiv e$$

```
  let x = <[ 2 ]> in <[ $x + $x ]>
⇝*  <[2 + 2]>
```

Hello, world!

(nothing out of the ordinary here:)

```
let rec power (n : int) (b : int) : int =
  if n = 0 then 1
  else b * power' (n - 1) b
```

So:

$$\texttt{power 3 5} \rightsquigarrow^* 5^3 = 125$$

Hello, metaworld!

```
let rec power' (n : int) (b : expr) : expr =
  if n = 0 then <[1]>
  else <[$b * $(power' (n - 1) b)]>

let power (n : int) : expr = <[ fun x -> $(power' n <[x]>) ]>
```

For example:

$$\texttt{power' 3 <[5]>} \rightsquigarrow^* \texttt{<[5 * (5 * (5 * 1))]>}$$
$$\texttt{power 3} \rightsquigarrow^* \texttt{<[fun x -> x * (x * (x * 1))]>}$$

Generated programs are values – we can just `print` them to see them.

```
val print : expr -> string
```

Why is staging...

Why is staging...

Useful?

- Convenient and safe* way to do metaprogramming – *no more strings!*
- For $\underbrace{\mathrm{Config}}_{\text{few}} \to \underbrace{\mathrm{Input}}_{\text{many}} \to \mathrm{Output}$, specialise for a Config with many Inputs.
- For $\mathrm{Config} = \mathrm{Program}$, derives compilers from interpreters!

Why is staging...

Useful?

- Convenient and safe* way to do metaprogramming – *no more strings!*
- For $\underbrace{\mathrm{Config}}_{\text{few}} \to \underbrace{\mathrm{Input}}_{\text{many}} \to \mathrm{Output}$, specialise for a Config with many Inputs.
- For $\mathrm{Config} = \mathrm{Program}$, derives compilers from interpreters!

Interesting?

- **Information is specific to one stage.**
- The language reasons about itself.

Why is staging...

Useful?

- Convenient and safe* way to do metaprogramming – *no more strings!*
- For $\underbrace{\mathrm{Config}}_{\text{few}} \to \underbrace{\mathrm{Input}}_{\text{many}} \to \mathrm{Output}$, specialise for a Config with many Inputs.
- For $\mathrm{Config} = \mathrm{Program}$, derives compilers from interpreters!

Interesting?

- **Information is specific to one stage.**
- The language reasons about itself.

Important?

- Subsumes rewriting-based optimisation – *out with the sufficiently smart compiler!*
  - Inlining (Kovács, 2024)
  - Partial evaluation
  - JIT compilation

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

- What is the type signature (more loosely: typing rule) of `eval`?

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

- What is the type signature (more loosely: typing rule) of `eval`?
- How do we know (without running) the generated program is valid?

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

- What is the type signature (more loosely: typing rule) of `eval`?
- How do we know (without running) the generated program is valid?
  - Well-formed?

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

- What is the type signature (more loosely: typing rule) of `eval`?
- How do we know (without running) the generated program is valid?
  - Well-formed?
  - Well-scoped?

Printing is one thing, evaluating is another

Many applications require us to run generated programs.
`eval` is a primitive which executes the program in an `expr` under an **empty* context** and returns the result.

`eval (power 5) : int -> int` is a *function* which for a given x computes x^5.
(not just a *program representation* anymore!)

- What is the type signature (more loosely: typing rule) of `eval`?
- How do we know (without running) the generated program is valid?
  - Well-formed?
  - Well-scoped?
  - **Well-typed**? (focus of this talk!)

Classic solution à la MetaML

We follow MetaML (Taha and Sheard, 2000) $\rightsquigarrow$ BER MetaOCaml (Kiselyov, 2014).

Classic solution à la MetaML

We follow MetaML (Taha and Sheard, 2000) $\rightsquigarrow$ BER MetaOCaml (Kiselyov, 2014).

- Well-formed: Ensured by using quotes of the same language, not arbitrary strings.

Classic solution à la MetaML

We follow MetaML (Taha and Sheard, 2000) $\rightsquigarrow$ BER MetaOCaml (Kiselyov, 2014).

- Well-formed: Ensured by using quotes of the same language, not arbitrary strings.
- Well-scoped: A context Γ for every stage (i.e. outside and inside quotes).
  - In `let x = 42 in <[ let y = 1337 in ... ]>`, `x` is only in-scope outside quotes, and `y` only inside.
  - *Scope extrusion* is still possible in the presence of side-effects (cf. Lee et al. (2026)).

Classic solution à la MetaML

We follow MetaML (Taha and Sheard, 2000) $\rightsquigarrow$ BER MetaOCaml (Kiselyov, 2014).

- Well-formed: Ensured by using quotes of the same language, not arbitrary strings.
- Well-scoped: A context Γ for every stage (i.e. outside and inside quotes).
  - In `let x = 42 in <[ let y = 1337 in ... ]>`, `x` is only in-scope outside quotes, and `y` only inside.
  - *Scope extrusion* is still possible in the presence of side-effects (cf. Lee et al. (2026)).
- **Well-typed:** α `expr` represents a program fragment yielding α when evaluated.
  - `<[ "abc" ]> : string expr`
  - `<[fun x -> x + 1]> : (int -> int) expr`
  - `eval : 'a expr -> 'a`

power in types

```
let rec power' (n : int) (b : int expr) : int expr =
  if n = 0 then <[1]>
  else <[$b * $(power' (n - 1) b)]>

let power (n : int) : (int -> int) expr =
  <[ fun x -> $(power' n <[x]>) ]>
```

Typing, roughly

Locks `<[` and `]>` enter the next and previous stage, respectively.
Accessing a variable, we require the same number of `<[`s and `]>`s *crossed*.

$$\frac{x : \tau \in \Gamma \text{ at same stage}}{\Gamma \vdash x : \tau}\textsc{Var}$$

$$\frac{\Gamma, \texttt{<[} \vdash e : \tau}{\Gamma \vdash \texttt{<[}e\texttt{]>} : \tau\ \texttt{expr}}\textsc{Quote}$$

$$\frac{\Gamma, \texttt{]>} \vdash e : \tau\ \texttt{expr}}{\Gamma \vdash \texttt{\$}e : \tau}\textsc{Splice}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{expr}}{\Gamma \vdash \texttt{eval}\ \ e : \tau}\textsc{Eval}$$

Core: Staging types

Well-typed wishes

Well-typed metaprograms generated well-typed programs.

Well, do they?

With great types comes great responsibility

OCaml's expressive type system gets in the way – examples:

1. Module abstraction (**loss** of type information)
2. Type equality proofs (**gain** of type information) – GADTs/first-class modules

Why? **Type information travels between stages illegally.**

Module abstraction blinds the generated program (Example 1)

In OCaml, a module **implementation** can be abstracted using a module **interface**.

- The metaprogram knows the **implementation**: specific to the **meta-stage**.
- Generated programs only see the **interface**, as it is *persistent*: at **all stages**.

```
(* Interface *)
type t  (* some abstract type *)

(* === *)

(* Implementation *)
type t = int  (* alias for [int] *)
let f = eval <[ fun (x : t) -> (x : int) ]>
```

- Outside quotes, `t` = `int`.
- Inside quotes, `t` is abstract, but naïvely `<[x]> : t expr` = `int expr` *(inj.)*.

Easy to circumvent (MetaOCaml bans <[t]>), but glaring!

First-class type equalities (for Example 2)

GADTs let you define a type `(s, t) eq` whose values are proofs that $s = t$.

```
(* boilerplate *)
type ('a, 'b) eq = Refl : 'a. ('a, 'a) eq

(* val f : ('a, int) eq -> 'a -> int *)
let f (type a) (eq : (a, int) eq) (x : a) : int =
  match eq with
  | Refl ->
    (* the type checker now knows that [a = int] *)
    (x : int)
```

Side note

Yallop and Kiseylov (2010) have shown that OCaml's first-class modules – another powerful feature – can encode higher-kinds ⇒ Leibniz equality ⇒ GADTs.

Time travel tricks the metaprogram (Example 2a)

Variation on MetaOCaml example by Yallop (2016), for some `type t`:

```
let f (x : t) =
  <[
    (* ask a future where [t = int] for a favour *)
    fun (Equal : (t, int) eq) ->
      $(
        (* [t = int] in the past?! *)
        print_int x;  (* side effect coercing [t = int] *)
        <[()]>        (* trivial splice *)
      )
  ]>
```

Consequences

GADTs remain banned in MetaOCaml due to unsoundness (Kiselyov, 2026).
Since we like GADTs, we are sad.

Ill-staged equation confuses the generated program (Example 2b)

Even solving time travel, there are still problems with equality staging:

```
let f (Refl : (t, int) eq) =
  <[ fun (x : t) -> (x : int) ]>
 (* but type-checking this quote by itself, [t =/= int] *)
```

We cannot evaluate the expression `fun (x : t) -> (x : int)` – context is empty.

Staging types

Our solution

We introduce staging in types:

$$\tau ::= \cdots \mid \texttt{<[}\tau\texttt{]>} \mid \texttt{\$(}\tau\texttt{)} \mid \texttt{<[}\tau\texttt{]> eval}$$

with definitional equalities:

$$\texttt{<[\$(}\tau\texttt{)]>} \equiv \tau \quad \text{and} \quad \texttt{\$(<[}\tau\texttt{]>)} \equiv \tau$$

$$\texttt{<[}\tau\texttt{]> eval} = \tau \quad \text{for } \textit{persistent } \tau$$

Staging types

Our solution

We introduce staging in types:

$$\tau ::= \cdots \mid \texttt{<[}\tau\texttt{]>} \mid \texttt{\$(}\tau\texttt{)} \mid \texttt{<[}\tau\texttt{]> eval}$$

with definitional equalities:

$$\texttt{<[\$(}\tau\texttt{)]>} \equiv \tau \quad \text{and} \quad \texttt{\$(<[}\tau\texttt{]>)} \equiv \tau$$

$$\texttt{<[}\tau\texttt{]> eval} = \tau \quad \text{for } \textit{persistent } \tau$$

Importantly, `t` *is not* `<[t]>`: we syntactically delineate the flow of type information.

```
    <[2 + 2]> : <[int]> expr
eval <[2 + 2]> : <[int]> eval = int
```

Core idea (quotes/splices in types) is from the type theory of Kovács (2022).

Typing, staged, roughly

Unification-based type inference still works.

$$\frac{\Gamma, \texttt{<[} \vdash e : \tau}{\Gamma \vdash \texttt{<[}e\texttt{]>} : \texttt{<[}\tau\texttt{]> expr}}\textsc{Quote}$$

$$\frac{\Gamma, \texttt{]>} \vdash e : \texttt{<[}\tau\texttt{]> expr}}{\Gamma \vdash \texttt{\$}e : \tau}\textsc{Splice}$$

$$\frac{\Gamma \vdash e : \texttt{<[}\tau\texttt{]> expr}}{\Gamma \vdash \texttt{eval } e : \texttt{<[}\tau\texttt{]> eval}}\textsc{Eval}$$

Stages of the context Γ*, expression* e *and type* τ *are now all consistent.*

Staging in Examples 2a and 2b

Example 2a is wrong for two independent reasons – equality staging and time travel:

```
let f (x : t) =
  <[
    fun (Equal : (t, int) eq) ->
      (* 1. We only know that <[t]> = <[int]>  *)
      $(
        (* 2. We forget this when going back in time *)
        print_int x;  (* error! [t =/= int] *)
        <[()]>
      )
  ]>
```

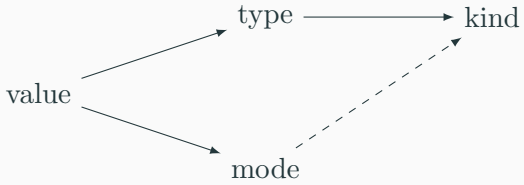
Example 2b is wrong because `t = int` does not imply `<[t]> = <[int]>`. Fixed:

```
let f (Refl : (<[t]>, <[int]>) eq) = (* [<[x]> = <[int]>] outside *)
  <[ fun (x : t) -> (x : int)         (* [x = int] inside; ok! *) ]>
```

Further work: Staging kinds & modes

OxCaml Judgment Zoo

$a \to b$ indicates that a has b, and $a \dashrightarrow b$ that a is influenced by b.



Kinds

- In OxCaml, kinds are a convenient mechanism for reasoning about:
  - Types with different memory representations.
  - Types for which we might have stronger guarantees about modes.

Kinds

- In OxCaml, kinds are a convenient mechanism for reasoning about:
  - Types with different memory representations.
  - Types for which we might have stronger guarantees about modes.

We straightforwardly get **quoted kinds** $\kappa ::= \cdots \mid \texttt{<[}\kappa\texttt{]>}$:

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \texttt{<[}\tau\texttt{]>} : \texttt{<[}\kappa\texttt{]>}} \qquad \frac{\Gamma \vdash \tau : \texttt{<[}\kappa\texttt{]>}}{\Gamma \vdash \texttt{\$(}\tau\texttt{)} : \kappa}$$

Useful for:

- Making `int expr` ill-kinded (vs `<[int]> expr`).
- Preventing quantification over splice types (there is no splice kind).

Modes

Modes describe how a value **has been used** *or* **can be used**. They thus describe how a value interacts with its **context**.
For more about OxCaml's mode system: Lorenzen et al. (2024), Georges et al. (2025).

Modes

Modes describe how a value **has been used** *or* **can be used**. They thus describe how a value interacts with its **context**.
For more about OxCaml's mode system: Lorenzen et al. (2024), Georges et al. (2025).

Questions

- How should substructural typing interact with staging?
- How does should context interaction be treated across stages?

Modes

Modes describe how a value **has been used** *or* **can be used**. They thus describe how a value interacts with its **context**.
For more about OxCaml's mode system: Lorenzen et al. (2024), Georges et al. (2025).

Questions

- How should substructural typing interact with staging?
- How does should context interaction be treated across stages?

A function's mode is given by the modes of variables it captures.
Analogously for a quote's mode – `<[τ]> expr` is kinda like `unit` $\to \tau$.
It turns out useful to consider accesses at different stages separately. For convenience, take separate contexts for stage-0 Γ and stage-1 Δ:

$$\frac{\Gamma, \square_\mu; \Delta, \square_\nu \vdash_1 e : \tau \mathbin{@} \omega}{\Gamma; \Delta \vdash_0 \texttt{<[}e\texttt{]>} : \texttt{<[}\tau \texttt{ @ } \omega\texttt{]> expr @ } \mu\texttt{<[}\nu\texttt{]>}}$$

sketch!! ω is the mode of the expression's result; μ of stage-0 context, ν of stage-1 context

Modes – closed and "closed"

A new mode axis relevant to functions:

A function is `closed` if it can be typed in an empty context, and otherwise it is `open`.

For example:

`fun x -> 2 * x` is `closed`

`let a = 3 in fun x -> a * x` is `open` (captures `a`)

For free, we get a mode axis relevant to quoted expressions and empty *staged* contexts:

`<[2 + 2]>` is `<[closed]>`

`<[x + 1]>` is `<[open]>` (captures `<[x]>`)

Conjecture: we can prevent scope extrusion with this technique.

Closed modality?

Conclusions: Staging it all

Conclusions

- OCaml's type system is powerful enough (*because of first-class type equalities?)* that we need staging annotations not just for **expressions**, but also **types**.
  - Recovers soundness by preventing type information from leaking between stages.
  - A sufficiently powerful language must carefully separate a metaprogram and the programs it generates.
- We apply results from type theory in practical language extension design. Thanks to the underlying foundations, further extensions – like staging **kinds** and **modes** – seem to follow naturally.

Conclusions

- OCaml's type system is powerful enough (*because of first-class type equalities?)* that we need staging annotations not just for **expressions**, but also **types**.
  - Recovers soundness by preventing type information from leaking between stages.
  - A sufficiently powerful language must carefully separate a metaprogram and the programs it generates.
- We apply results from type theory in practical language extension design. Thanks to the underlying foundations, further extensions – like staging **kinds** and **modes** – seem to follow naturally.

Thank you!

Appendix

Type inference

- Definitional equalities are eliminated with type β-reductions
  - Like abbreviation expansion: e.g for `type 'a t = 'a * 'a`, `int t` $\rightsquigarrow_\beta$ `int * int`).
- Quotes and splices form a group, so `<[`τ`]>` $\sim \pi \iff \tau \sim$ `$(`π`)`.
- **Conjecture:** `eval`-constraints `<[`τ`]> eval` $= \pi$ solved as τ kind-constraints.
- Non-Hindley-Milner type inference features in OCaml are type inspections, which can be resolved by inserting type annotations in the generated program.

Scope extrusion

Dynamic approaches modelled by Lee et al. (2026); static approaches exist but heavy.

```
(* Exceptions (or effects) *)
exception Extrude of <[int]> expr
let () =
  try ignore <[ let x = 42 in $(raise (Extrude <[x]>); <[()]>) ]>
  with Extrude x -> print_int (eval x)

(* Mutable state *)
let cell = ref <[0]> in
ignore <[ let x = 42 in $(cell := <[x]>; <[()]>) ]>;
print_int (eval !cell)
```

Let-polymorphism

Tricky example based on Kiselyov (2015):

```
val inject : 'a eval -> 'a expr

let push x xs = xs := x :: !xs
let bad = <[
  let f = fun () -> $(inject (ref [])) in
  push (f ()) 123;
  push (f ()) "abc"
]>
```

- Ill-staged evaluation-constraint?
- Splice kind? *(quantification over types in the past)*

MacoCaml

Is MacoCaml sound with respect to the typing issues here?

- I do not believe this is mentioned in the paper – and the maco calculus proven sound **does not include first-class equalities**.
- However, the implementation at least implement the time travel check. This might be enough – ill-staged equations might not appear as programs are only **spliced, and not evaluated** in an empty context.

References i

- Georges, Aïna Linn et al. (2025). 'Data Race Freedom à la Mode'. In: *Proc. ACM Program. Lang.* 9.POPL, pp. 656–686. DOI: 10.1145/3704859. URL: https://doi.org/10.1145/3704859.
- Kiselyov, Oleg (2014). 'The Design and Implementation of BER MetaOCaml - System Description'. In: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. Ed. by Michael Codish and Eijiro Sumii. Lecture Notes in Computer Science. Springer, pp. 86–102. DOI: 10.1007/978-3-319-07151-0_6. URL: https://doi.org/10.1007/978-3-319-07151-0_6.
- Kiselyov, Oleg (2015). 'Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing'. In: *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015*. Ed. by Jeremy Yallop and Damien Doligez. EPTCS, pp. 1–22. DOI: 10.4204/EPTCS.241.1. URL: https://doi.org/10.4204/EPTCS.241.1.

References ii

- Kiselyov, Oleg (2026). 'MetaOCaml: ten years later - System description'. In: *Sci. Comput. Program.* 250, p. 103397. DOI: 10.1016/J.SCICO.2025.103397. URL: https://doi.org/10.1016/j.scico.2025.103397.
- Kovács, András (2022). 'Staged compilation with two-level type theory'. In: *Proc. ACM Program. Lang.* 6.ICFP, pp. 540–569. DOI: 10.1145/3547641. URL: https://doi.org/10.1145/3547641.
- Kovács, András (2024). 'Closure-Free Functional Programming in a Two-Level Type Theory'. In: *Proc. ACM Program. Lang.* 8.ICFP, pp. 659–692. DOI: 10.1145/3674648. URL: https://doi.org/10.1145/3674648.
- Lee, Michael et al. (2026). 'Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks'. In: *Proc. ACM Program. Lang.* 10.POPL, pp. 1123–1152. DOI: 10.1145/3776681. URL: https://doi.org/10.1145/3776681.
- Lorenzen, Anton et al. (2024). 'Oxidizing OCaml with Modal Memory Management'. In: *Proc. ACM Program. Lang.* 8.ICFP, pp. 485–514. DOI: 10.1145/3674642. URL: https://doi.org/10.1145/3674642.

References iii

Taha, Walid and Tim Sheard (2000). 'MetaML and multi-stage programming with explicit annotations'. In: *Theor. Comput. Sci.* 248.1-2, pp. 211–242. DOI: 10.1016/S0304-3975(00)00053-0. URL: https://doi.org/10.1016/S0304-3975(00)00053-0.