Compile-time Computation for Caml

Tsung-Ju Chiang Maite Kramarz Michael Lee Ka Wing Li Javier Martinez Olivier Nicole Dima Szamozvancev Leo White Ningning Xie **Jeremy Yallop**

funded by Jane Street and Ahrefs

INRIA, December 2024

Language features

Running example

What? ● ○ ○ ○

Macros define compile-time functions:

```
Why?
```

Modules

Effects

Theory

How?

```
macro rec pow x n =
    if n = 0 then << 1 >>
    else << $x * $(pow x (n-1)) >>
```

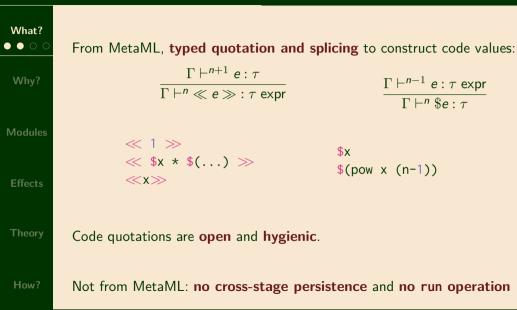
Code generated by macros can be spliced into programs:

```
let pow5 x = $(pow ≪x≫ 5)
```

Compilation runs the macros to generate program fragments:

let pow5 x = x * x * x * x * x * 1

Language features from MetaML



Basic differences with MetaML



Theory

How?

Compile-time code generation, not run-time code generation, via two features:

Macros are compile-time **let** bindings:

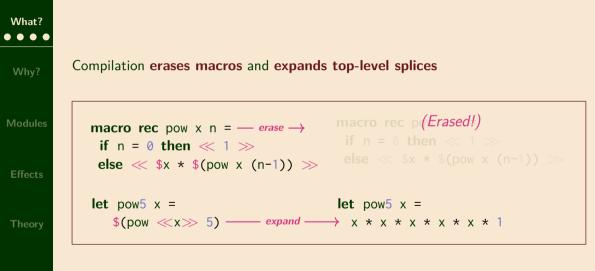
```
macro rec pow x n =
if n = 0 then ...
```

Top-level splices trigger compile-time code generation

let pow5 x = \$(pow ≪x≫ 5)

During compilation they are evaluated to code values that are inserted in place

Extras (not covered today): generating deep patterns, recursive bindings



Applications

Motivation: DSL optimizations (currently unoptimized code)

What?

Why? ● ○ ○ ○

Parsing libraries often involve abstraction overhead (table dispatch, closure application):

Modules

```
Effects
```

Theory

let sexp = fix (fun s \rightarrow char '(' >>> star s >>> char ')' <|> atom)

With staging: eliminate all overhead; generate code you'd write by hand

Motivation: make existing code-generating libraries safer

What?

Why? ● ● ○ ○

Modules

Effects

Theory

How?

Ctypes uses an untyped code representation & generates a functor + a match

let print_endline = foreign "puts"
 (string @→ returning int)

With staging: used a typed code representation and generate simple code

Motivation: avoid copy-and-paste optimizations

let for_all p a =

let rec loop i =

let n = length a in

if i = n then true



The standard library currently has lots of repetitive code:

let exists p a =

loop 0

let n = length a in
let rec loop i =

if i = n then false

Why? ● ● ● ⊂

Modules

Effects

Theory

then truethen loop (succ i)else loop (succ i) inelse false in

loop 0

With staging: safely generate this same code from templates.

Programming patterns (unrolling, bounds checks) become template parameters

else if p (unsafe_get a i) else if p (unsafe_get a i)

Motivation: very high-level programming



Why?

Modules

Effects

Theory

How?

Libraries like Scrap Your Boilerplate are 10-20× slower than hand-written code.

let rec listify {T:TYPEABLE} p {D:DATA} x =
 mkQ [] (single p) x @ concat (D.gmapQ (listify p) x)

With staging: eliminate overhead, make currently impractical programs practical

Integration with modules

Signatures and subtyping: the problem

What?

Whv?

Modules

 \odot

Theory

Macros are defined in modules, so must interact with functors, subtyping, etc.

```
module Pow : sig
   macro pow : int expr \rightarrow int \rightarrow int
 expr
 end = struct
   let sq x = x * x
   macro rec pow x n =
     if n = 0 then \ll 1 \gg
     else if n mod 2 = 0 then
      << sq $(pow x (n/2)) >>
     else << $x * $(pow x (n-1)) >>
 end
 let pow5 x = (Pow, pow \ll x \gg 5)
 \rightsquigarrow^*
 let pow5 x = (x * sq (sq x)) (* sq not bound! *)
Subject reduction failure!
```

Signatures and subtyping: the resolution

What?

Solution: **closure conversion**. Export an environment alongside the macro:

```
Why?
```

Modules

Effects

Theory

How?

```
module Pow : sig (* slightly simplified *)
module type CLO = sig val square : int → int end
module %Clo : CLO
module %M(C: CLO) : sig macro pow : int expr → int → int expr end
end = ...
```

and pass in the environment when invoking the macro

```
let pow5 x = $(Pow.%M(≪Pow.%Clo≫).pow ≪x≫ 5)
```

 \rightsquigarrow

let pow5 x = x * Pow.%Clo.sq (Pow.%Clo.sq x) (* no extrusion! *)

Interactions with effects

Pure code and quotation

Why?

```
Modules
```

Effects

Theory

In staged pure programs generated code shape follows the evaluation trace:

 $[pow \ll x \gg 2]$ $\Rightarrow if [2 = 0] then \ll 1 \gg$ $else \ll x * (pow x (2-1)) \gg$ $\Rightarrow [if false then \ll 1 \gg$ $else \ll x * (pow x (2-1)) \gg]$ $\Rightarrow \ll x * (pow x (2-1)) \gg$ $\Rightarrow \ll x * (pow x (2-1)) \gg$ $\Rightarrow \ll x * (pow x (2-1)) \gg$ $\Rightarrow \ll x * (pow x (1-1)) \gg$ $\Rightarrow \ll x * (if [1 = 0] then \ll 1 \gg$ $else \ll x * (pow x (1-1)) \gg) \gg$ (slightly simplified)

Code generated in a splice cannot escape the surrounding quotation.

Effects and quotation

What?

Why?

Modules

```
try \ll x +  (perform (Chuck \ll y \gg)) \gg with
effect Chuck v, k \rightarrow \ll  $v + $(continue k \ll 3 \gg) \gg
\rightsquigarrow^*
\ll v + (x + 3) \gg
```

With effects, code generating programs can **reorder quotations**:

Effects

Theory

- This is good for code motion transformations, but problematic for safety: match \ll fun x \rightarrow \$(perform (Chuck \ll x \gg)) \gg with $\mid _ \rightarrow \ll \emptyset \gg$ \mid effect Chuck v, k \rightarrow v $\rightsquigarrow^* \ll$ x \gg (* x free! *)
- Plan: use effects to **detect scope extrusion** (inspired by MetaOCaml).

Metatheoretical properties



What?

Run-time soundness: well-typed programs don't go wrong

Elaboration soundness: well-typed programs elaborate to well-typed programs (Elaboration includes *compile-time evaluation*)

For typed quotations, soundness implies scope-safety

Theory \bullet \circ

Calculus: basic modules, ground-type references, quotation, macros, top-level splices Mechansiation for module-free fragment extended with run-time code generation

What?

Why?

Modules

Effects

Theory

How?

Phase separation: compile-time computations not needed for run-time evaluation

Implication: we can discard the compile-time heap

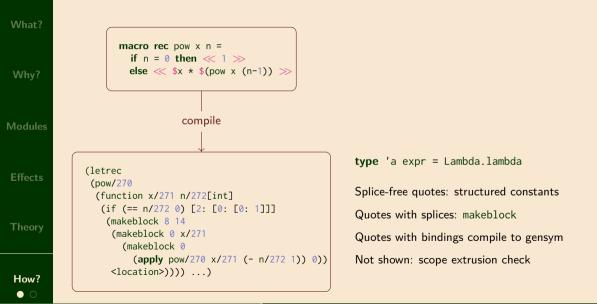
Implication: we can erase macros when running programs

For programmers: separation between the generating code and the generated code

Calculus: basic modules, ground-type references, quotation, macros, top-level splices Mechansiation for module-free fragment extended with run-time code generation

Practical matters

Implementation (based on OCaml 5.3)





		design	theory	2	
Why?	quote	s 🗸	1	1	1
odules	macro	s 🗸	 Image: A start of the start of	1	1
	elaboratio	n 🖌	\checkmark	1	m
ffects	module	s 🚧	X	X	X
	scope extrusion	n 🪧	in the second	X	i de la compañía de la compa
heory	pattern generation	n 🪧	X	X	ini,
	let rec generation	n 🗸	1	X	ini,

How? ●●