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

Macros define compile-time functions:

```
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

From MetaML, **typed quotation and splicing** to construct code values:

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \ll e \gg : \tau \text{ expr}} \qquad \frac{\Gamma \vdash^{n-1} e : \tau \text{ expr}}{\Gamma \vdash^{n} \$e : \tau}$$

```
≪ 1 ≫
≪ $x * $(...) ≫
≪x≫
```

```
$x
$(pow x (n-1))
```

Code quotations are **open** and **hygienic**.

Not from MetaML: **no cross-stage persistence** and **no run operation**

Basic differences with MetaML

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

What?
Why?
Modules
Effects
Theory
How?

Compilation **erases macros** and **expands top-level splices**

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

let pow5 x =                         let pow5 x =
    $(pow <<x>> 5) —— expand ——>       x * x * x * x * x * 1
```

Applications

Motivation: DSL optimizations (currently unoptimized code)

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

```
let sexp = fix (fun s →
                  char '(' >>> star s >>> char ')'
               <|> atom)
```

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

Motivation: make existing code-generating libraries safer

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

The standard library currently has lots of repetitive code:

```
let exists p a =
  let n = length a in
  let rec loop i =
    if i = n then false
    else if p (unsafe_get a i)
         then true
    else loop (succ i) in
  loop 0
```

```
let for_all p a =
  let n = length a in
  let rec loop i =
    if i = n then true
    else if p (unsafe_get a i)
         then loop (succ i)
    else false in
  loop 0
```

With staging: safely generate this same code from templates.

Programming patterns (unrolling, bounds checks) become **template parameters**

Motivation: very high-level programming

Libraries like *Scrap Your Boilerplate* are **10-20$\times$ 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

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

```
module Pow : sig
  macro pow : int expr → int → int
expr
end = struct
  let sq x = x * x
  macro rec pow x n =
    if n = 0 then << 1 >>
    else if n mod 2 = 0 then
      << sq $(pow x (n/2)) >>
    else << $x * $(pow x (n-1)) >>
end
```

```
let pow5 x = $(Pow.pow <<x>> 5)
⇝*
let pow5 x = $(x * sq (sq x))  (* sq not bound! *)
```

Subject reduction failure!

Signatures and subtyping: the resolution

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

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

Interactions with effects

Pure code and quotation

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

```
[pow ≪x≫ 2]
⇝ if [2 = 0] then ≪1≫
  else ≪x * $(pow x (2-1))≫
⇝ [if false then ≪1≫
   else ≪x * $(pow x (2-1))≫]
⇝ ≪x * $(pow x ([2-1]))≫
⇝ ≪x * $([pow x 1])≫
⇝ ≪x * $(if [1 = 0] then ≪1≫
           else ≪x * $(pow x (1-1))≫)≫
⇝ ≪x * $[if false then ≪1≫
           else ≪x * $(pow x (1-1))≫])≫
```

```
⇝ ≪x * $≪x * $(pow x [1-1])≫≫
⇝ ≪x * $≪x * $([pow x 0])≫≫
⇝ ≪x * $≪x *
        $(if [0 = 0] then ≪1≫
         else ≪x * $(pow x (0-1))≫)≫≫
⇝ ≪x * $≪x *
        $([if true then ≪1≫
           else ≪x * $(pow x (0-1))≫])≫≫
⇝ ≪x * $≪x * [$≪1≫]≫≫
⇝ ≪x * [$≪x * 1≫]≫
⇝ ≪x * x * 1≫
```

(slightly simplified)

Code generated in a splice cannot escape the surrounding quotation.

What?
Why?
Modules
Effects
Theory
How?

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

```
try ≪ x + $(perform (Chuck ≪y≫)) ≫ with
effect Chuck v, k → ≪ $v + $(continue k ≪3≫) ≫
⇝*
≪ y + (x + 3) ≫
```

This is **good for code motion transformations**, but **problematic for safety**:

```
match ≪ fun x → $(perform (Chuck ≪x≫)) ≫ with
              | _ → ≪ 0 ≫
              | effect Chuck v, k → v
⇝* ≪x≫ (* x free! *)
```

Plan: use effects to **detect scope extrusion** (inspired by MetaOCaml).

Metatheoretical properties

Soundness

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**

Calculus: basic modules, ground-type references, quotation, macros, top-level splices

Mechansiation for module-free fragment extended with run-time code generation

Phase separation

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)


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

compile

```
(letrec
 (pow/270
  (function x/271 n/272[int]
   (if (== n/272 0) [2: [0: [0: 1]]]
    (makeblock 8 14
     (makeblock 0 x/271
       (makeblock 0
         (apply pow/270 x/271 (- n/272 1)) 0))
     <location>)))) ...)
```

```
type 'a expr = Lambda.lambda
```

Splice-free quotes: structured constants

Quotes with splices: `makeblock`

Quotes with bindings compile to gensym

Not shown: scope extrusion check

Progress

	design	theory		
quotes	✓	✓	✓	✓
macros	✓	✓	✓	✓
elaboration	✓	✓	✓	🚧
modules	🚧	✗	✗	✗
scope extrusion	🚧	🚧	✗	🚧
pattern generation	🚧	✗	✗	🚧
let rec generation	✓	✓	✗	🚧