Pushing Contextual Modal Type Theory to its limits

Theo Wang

Department of Computer Science, University of Oxford

Multi-Stage Programming (MSP)

Consider the following `pow` function:

```
let rec pow n x = if n = 0 then 1 else x * pow (n-1) x
```

Here, `pow` is an *abstraction*: it works for any n, but is 'inefficient' for any fixed n.

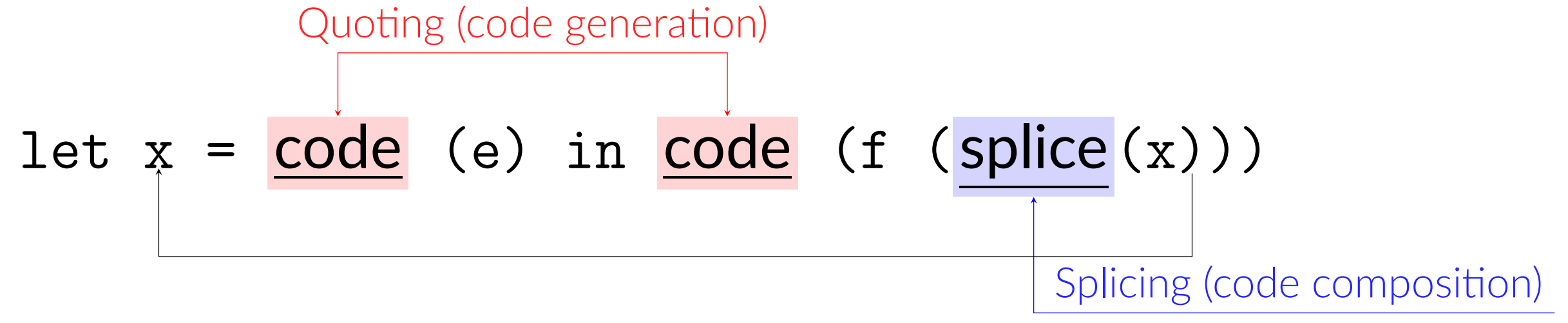
MSP goal: Recover the performance for fixed n via run-time code generation.

$$\texttt{pow\_staged}\ n \to^{*} \underline{\texttt{code}}\ (\lambda x.\ (x * \ldots * x))\ \text{(n times)}$$

This is by conceptually dividing the execution of `pow` into two stages: stage 0 ('compile time') takes input n, and generates more efficient code to be executed at stage 1 ('run time'), given x.

Mechanics of quasi-quoting

(In the style of Murase et al. 2023).



Quoting (code generation)

`let x = code (e) in code (f (splice(x)))`

Splicing (code composition)

Two appraoches to dealing with free variables: $\lambda^{\bigcirc}$ vs CMTT

Two of the most prominent approaches to MSP are $\lambda^{\bigcirc}$ (Davies 1996) and CMTT (Nanevski et al. 2008). $\lambda^{\bigcirc}$ binds its free variables statically at *quoting-time* with an *implicit* context shared across the stage; CMTT maintains a list of free variables (*explicit* context) and binds them dynamically at *splicing-time* via explicit substitution.

Property	$\lambda^{\bigcirc}$ (implicit context)	CMTT (explicit context)
`A code` Type	$\bigcirc A$	$[\Psi \vdash A]$
`A code` Term	$\dfrac{\Gamma \vdash^{n+1} e : A}{\Gamma \vdash^{n} < e >: \bigcirc A}$	$\dfrac{\Delta; \Psi \vdash e : A}{\Delta; \Gamma \vdash \textbf{box}\ (\Psi \vdash e) : [\Psi \vdash A]}$
`A code` splicing	$\dfrac{\Gamma \vdash^{n} e : \bigcirc A}{\Gamma \vdash^{n+1} \sim e : A}$	$\dfrac{\Delta; \Gamma \vdash e : [\Psi \vdash A] \quad \Delta, u :: A[\Psi]; \Gamma \vdash e' : C}{\Delta; \Gamma \vdash \textbf{let box}\ u = e \text{ in } e' : C}$ $\dfrac{\Delta, u :: A[\Psi], \Delta'; \Gamma \vdash \sigma : \Psi}{\Delta, u :: A[\Psi], \Delta'; \Gamma \vdash u\ \textbf{with}\ \sigma : A}$

Table 1. $\lambda^{\bigcirc}$ vs CMTT: a comparison

(Implementations of `pow` omitted due to lack of space.)

The (closed?) expressivity gap

Compared to $\lambda^{\bigcirc}$, CMTT has been known to lack the expressivity to have multiple pieces of code always share the same context. The type $\bigcirc A \to \bigcirc B$ really means '*a function taking an* `A code` *and giving a* `B code` *implicitly under the same context,* ***for any such context***' – i.e. $\forall \Psi.[\Psi \vdash A] \to [\Psi \vdash B]$. This sort of **abstraction over contexts** is exactly Murase et al.'s extension to CMTT in $\lambda^{\forall[]}$, who then proved the following theorem:

Theorem (Murase et al. 2023): *There exists a sounds embedding of $\lambda^{\bigcirc}$ in $\lambda^{\forall[]}$.*

Problem statement & approach

Current state of the world:

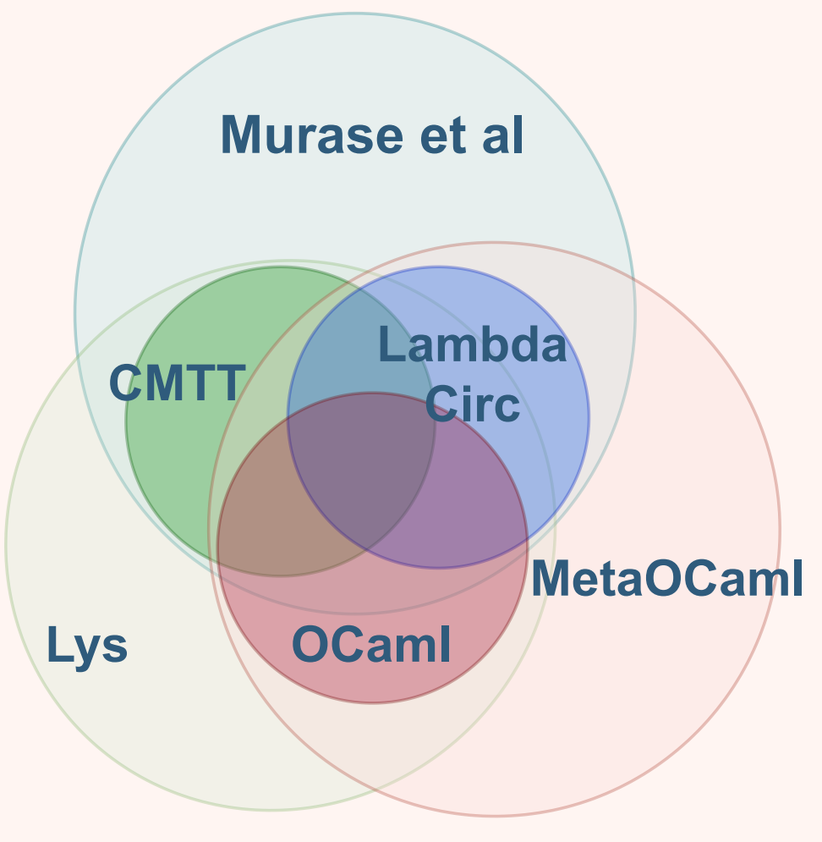


Figure 1. Expressivity landscape of MSP languages

*Are first-class contexts strictly necessary to get the expressivity of $\lambda^{\bigcirc}$? Are they still necessary for embedding derived **polymorphic** languages like MetaOCaml?*

Fair question: **PL features interact in surprising ways** (cf. Landin's knot).

For fair comparison with **MetaOCaml** $\approx \lambda^{\bigcirc} \cup$ **OCaml**, we:

1. Create a language **Lys** $\approx$ **CMTT** $\cup$ **OCaml**;
2. Include in Lys a restricted version of Jang et al. (2022)'s polymorphic code type (Φ, Θ contain System-F type variables):
$$\frac{\Phi \cap \Theta = \emptyset \qquad \Theta, \Phi \vdash \Psi \text{ ctx} \qquad \Theta, \Phi \vdash \tau \text{ type} \qquad \Theta, \Phi; \Delta; \Psi \vdash e : \tau}{\Theta; \Delta; \Gamma \vdash \textbf{box}\ (\Phi; \Psi \vdash e) : [\Phi; \Psi \vdash \tau]}$$
3. Attempt to translate complex MetaOCaml applications to Lys.

Translating symbolic evaluation

Stage-0 functions manipulating (potentially open) stage-1 code to form other pieces of code are referred to as *symbolic evaluation* in the literature. E.g. $\bigcirc A \to \bigcirc B$:

- $\lambda^{\forall[]}$ style translation: explicitly share the context. $\forall \Psi.[\Psi \vdash A] \to [\Psi \vdash B]$.
- Our Insight: for any context Ψ we have $[A \vdash B] \to ([\Psi \vdash A] \to [\Psi \vdash B])$, i.e. $[A \vdash B]$ incorporates the information needed. So our solution: $[A \vdash B]$.

Symbolic evaluation allows MetaOCaml to **exploit statically known structures**. Example: representing a stage-0 $f : (\tau_1 \times \tau_2)$ `code` $\to B$ `code`.

```
let f_inefficient (x: ◯(τ₁ × τ₂)) =
    (* tuple deconstructed at stage 1*)
    ... .<match .~x with (v1, v2) -> ... v1 ... v2 ...>.

let f_efficient (x: ◯τ₁ × ◯τ₂) =
    (* tuple deconstructed at stage 0*)
    match x with (c1, c2) -> ... .< ... .~c1 ... .~c2 ...>.
```

By using stage-0 data-structures of stage-1 values, we remove the overhead of destructing that data-structure at runtime.

- $\lambda^{\forall[]}$ style: $\forall \Psi.([\Psi \vdash \tau_1] \times [\Psi \vdash \tau_2] \to [\Psi \vdash B])$.
- Our soln 1: $[\tau_1, \tau_2 \vdash B]$, but hard-coding structure of data in the context. Idea: force any destructing of data-structures to have happened already.
- Our soln 2: $[\tau_1 \times \tau_2 \vdash B]$: but did not remove the overhead.

Case Study 1: Staged tagless final interpreters

Problem: embedding some object-language in MetaOCaml (Carette et al. 2007).

Concept: 'CPS of universal algebra' (from anonymous reviewer).

- Choose a 'representation' `type ('a,'h) repr` as the 'outcome' of the interpretation (`'a` is the type; `'h` is the context)
- Each node in the AST is represented as a function; and use De Bruijn indices to retrieve elements from the context. E.g. `Lam (x, e2)` becomes

```
lam: ∀'a. ∀'b. ∀'h. ('b, ('a * 'h)) repr -> ('a -> 'b, 'h) repr
```

Take the repr to be `'h ->` $\bigcirc$`'a` to get '**compilation**'.

Translation: (*failed*) Assume a fixed `'a`.

- Soln 1 `[... flattened_h_1...,... flattened_h_2 ... |- 'a]` does not work: it forces us to include the structure of *every* possible `'h` in our context, which breaks our abstraction and is completely intractable.
- Soln 2 `['h; 'h |- 'a]` does work, but unwraps the context at runtime.

```
val: (((int -> int), unit) repr) = (box (h: unit|- (fun (x: int) -> (match (x,
  h) with (a, h)-> a + match (x, h) with (a, h)-> a{0}))))
```

What we really need: `type ('a, Ψ) repr`.

Case Study 2: Staged stream fusion

Problem: deforestation of Java-like stream operations (Kiselyov et al. 2017).

Concept:

- Use pull-streams

```
type ('a, 's) stream_shape = Nil | Cons of ('a * 's);;
type 'a stream =
    St of (∃'s. ('s * ('s -> ('a, 's) stream_shape)));;
```

- Gradually pull the destruction of data structures to 'compile time'.

Translation: (*failed*) similarly to before, we get to a stage where we have some `type T = ∃'a. 'a ->` $\bigcirc$`C` for some fixed type `C`. Thus solution 1 breaks the abstraction; solution 2 is inefficient.

What we really need: $\exists \Psi.\ [\Psi \vdash C]$.

Discussion & future work

- Negative results $\implies$ strong evidence that **Murase et al.'s extension is necessary for MetaOCaml**. *TODO: prove it?*
- But problems mainly when we need the expressiveness of type-level abstraction which translate naturally to abstracting over contexts. Hence **unclear whether still minimal for $\lambda^{\bigcirc}$**. *TODO: what is slightly less expressive than $\lambda^{\forall[]}$?*
- **Interesting connection between polymorphic (Jang et al 2022) and context-polymorphic CMTT (Murase et al 2023)**: using De Bruijn indices, we conjecture being able to translate the latter to the former, with constant overhead every piece of generated code. *TODO: Prove it?*
- Criticism: instead of trying to reproduce MetaOCaml programs, perhaps try writing **native CMTT programs**? *TODO: try this out!*

Advised by Jeremy Yallop and Alan Mycroft (University of Cambridge)

POPL 2024 Student Research Competitition (Graduate)

theo.wang@cs.ox.ac.uk