Exploring the limitations of Contextual Modal Type Theory for Multi-Stage Programming (Extended Abstract)*

Submission to the POPL 2024 Student Research Competition (Graduate category)

THEO WANG†, University of Oxford, United Kingdom

1 INTRODUCTION

Multi-stage programming (MSP) is a well-known metaprogramming technique using runtime code generation to reduce the overhead of pristine abstractions.

There are mainly two approaches to MSP. On the one hand, languages based on linear temporal logic like $\lambda^{\bigcirc}$ [3] and MetaOCaml [11] give an elegantly compositional calculus of open code, but fail to allow safe code execution [8, 12] and safe interactions with imperative effects (cf. scope extrusion, [1]). On the other hand, the second approach based on contextual modal (S4) types (CMTT, [4, 10]) gives a calculus of open code with explicit context. It does not present MetaOCaml's problems, but is known to be less expressive. Ample theoretical works have proposed extensions such as System F polymorphism (Moebius, [5]) and context polymorphism ($\lambda_{\forall[]}$, [9]), which significantly reduced the gap. The latter even demonstrates an embedding of $\lambda^{\bigcirc}$ into their modal types.

However, these works leave a number of questions open. The relationship between Moebius and $\lambda_{\forall[]}$ is unclear, and it remains a question whether the context polymorphic extension is *necessary* to match the expressiveness of $\lambda^{\bigcirc}$, and whether it is still *sufficient* to express the more elaborate, practical applications in MetaOCaml. In this work, we explore these questions under the lenses of a practical metaprogrammer. We implement Lys, a practical CMTT-based MSP language with a restricted form Moebius-like polymorphism, and study its expressiveness with novel (attempts of) reimplementations of MetaOCaml programs. With this, we provide empirical evidence of the limitations of System F polymorphism, and argue for the practical necessity of *first class contexts*.

2 BACKGROUND AND APPROACH

2.1 CMTT and System F polymorphism [5]

CMTT [10] is characterised by a code generation construct of the form `box(Ψ ⊢ e): [Ψ ⊢ A]`, best interpreted as an expression e with holes specified by the context Ψ. Unboxing (`let box u = box (Ψ ⊢ e) in e'`) takes the expression out of the box binds it to metavariable `u`, and syntactically splices it into `e'` at `u`'s occurrences. We require that each such occurrence *closes* the holes with an explicit substitution.

Conceptually, we can think of the boxed type `[Ψ ⊢ A]` like a reified typing derivation. Allowing more types of holes then amounts to including the corresponding context in the box type. In this spirit, Jang et al. [5] extend CMTT with multi-level System F style polymorphism, which corresponds to extending the box context to bind type variables and meta (or higher) level variables as well. For the purpose of this work, we only consider the 2-level restriction, i.e. types of the form `[Φ; Ψ ⊢ A]` where Φ is the type variable context and Ψ the intuitionistic context, for we believe the multi-level extension is orthogonal to our exploration. We leave this question to future work.

*Work realised as an undergraduate student in the University of Cambridge, as part of the author's undergraduate thesis, under the supervision of Jeremy Yallop and Alan Mycroft.

†ACM membership number: 5275675. The author is currently a master's student at the University of Oxford.

Author's address: Theo Wang, University of Oxford, Dept. of Computer Science, United Kingdom, theo.wang@cs.ox.ac.uk.

2.2 MetaOCaml and context polymorphic CMTT [9]

On the other hand, MetaOCaml [6, 11] achieves staging by dividing the program into different *stages*, indicated using quotes `.<e>.` and splices `.~(e)`. Quoting code *delays* its execution to the next stage; splicing *advances* it to the previous stage. Importantly, each stage has its shared implicit context, and each stage-n variable can only be bound to a stage-n binder[1]. With the implicit stage-specific context, MetaOCaml binds its free variables directly when the next-stage code is created. This is in contrast to CMTT boxes, where free variables are explicitly carried in their types and are only instantiated when used.

This difference makes *symbolic evaluation*, a key feature in MetaOCaml, hard in CMTT. In MetaOCaml, symbolic evaluation is enabled by functions of type `A code -> B code`. Because the input has all its free variables statically bound when it was constructed, all free variables the output `B code` inherits from the `A code` will also be bound in that same context. In other words, the output implicitly depends on the context of the input. CMTT, on the other hand, cannot express this dependence in general. Intuitively, the MetaOCaml `A code -> B code` would have corresponded to $\forall\Psi.[\Psi \vdash A] \to [\Psi \vdash B]$ in CMTT, which is not expressible in CMTT, thus motivating Murase et al.'s [9] context polymorphic extension.

2.3 Our approach

In contrast to Murase et al., we choose to study CMTT's difficulties with MetaOCaml style symbolic evaluation with a slight detour.

MetaOCaml inherits from LTL the isomorphism `(A -> B) code` $\cong$ `A code -> B code`[2], where `code` corresponds to the $\bigcirc$ modality, meaning 'true at the next stage'. In contrast, Lys can only express the left-to-right implication: $[x: A \vdash B] \to ([\Psi \vdash A] \to [\Psi \vdash B])$. Conveniently, this implication holds for any Ψ on the right-hand side. This means that $[x: A \vdash B]$ becomes an adequate translation for `A code -> B code`. Intuitively, regardless of what context the `A code` and `B code` both depend on, all we need is a context-agnostic *template* guiding where we should splice the `A` value in the resulting `B` code.

This insight enables us to start from Lys and attempt to reproduce practical MetaOCaml programs. We shall see that the roadblocks we encounter provide valuable insight on the expressiveness gaps between polymorphic CMTT, MetaOCaml, and – unsurprisingly – context polymorphic CMTT.

3 CASE STUDIES AND FINDINGS

We present two case studies in [unsuccessful] reproductions of MetaOCaml programs with Lys.

3.1 Reproducing a Stream Fusion Library

We first study the Strymonas [7] stream fusion library in MetaOCaml. The goal of such a library is to expose a functional stream processing API, while using staging to perform deforestation and compile it to efficient imperative code. For example, squaring all elements of an array and summing it is expressed as as `of_arr .<arr>. |> map (fun x -> .<~x * ~x>.) |> sum`, which then compiles to an imperative for loop without any abstraction overhead.

To achieve this, Kiselyov et al. first chose to use pull streams (α `stream =` $\exists\alpha.$ σ `*` `(`σ `->` `(1+`α`*`σ`))`), then crucially used continuation passing style (CPS) to *pull the process of unwrapping known data structures to an earlier stage.* This is a key technique in MetaOCaml, and as we shall see, this is also where Lys encounters difficulties.

[1] Or a stage-m binder with $m \leq n$ thanks cross-stage persistence [12], which we do not yet discuss to avoid confusion.
[2] In modal logic, the left-to-right implication is the K axiom, and the converse the K^{-1} axiom, proper to the linear time assumption.

In MetaOCaml, the power of symbolic evaluation comes when a composite type `A` contains statically known structures. This is because MetaOCaml allows us to pull all of the known structures to an earlier stage. Consider a MetaOCaml value `v: A code`. Since can only access a value if we are at the same stage as that value, we have to unwrap `v` it at the exact stage where it lies (*stage 1*), no matter what we prior knowledge we possess. Now suppose that we know that `v` is always a tuple, i.e. `A`=$\tau_1 \times \tau_2$, but do not know the value of its components. Naively transforming `A code` to `A'` = τ_1 `code` $\times$ τ_2 `code` is not helpful, as such a value lives at an earlier stage (*stage 0*) and so is not interchangeable with `A code`. So how can we exploit this knowledge? The crucial insight is that in MetaOCaml, while `A code` and `A'` are not interchangeable, `A code -> B code` and `A' -> B code` are, because they are both stage 0 values, and because every component of the input unknown at stage 0 lives at stage 1. Importantly, in the latter case, symbolic evaluation allows us to unwrap the input tuple at stage 0! Thus, we can naturally pull the deconstruction of `v` to stage 0 via CPS, by representing `v` instead as a function of type $\forall\omega.(\tau_1$ `code` $\times$ τ_2 `code` $\rightarrow \omega) \rightarrow \omega$.

Unfortunately, expressing this in Lys is rather awkward. Naively translating a value of type `A' -> B code` using the previous insight gives the type $[x : \mathtt{A} \vdash \mathtt{B}]$, which is clearly inefficient because we have to unwrap the data structure at stage 1. Alternatively, we can flatten the `A'` data structure, extract all the dynamic components, and explicitly specify the corresponding types in the result's context: $[x_1 : \tau_1, x_2 : \tau_2 \vdash \mathtt{B}]$. The problem with this second approach, is that we are committing to one particular structure of `A`. But suppose we wanted to quantify over `A`'s; what will happen?

In Strymonas, one of the most important savings (in [7] §5.2) was achieved through exploiting a known structure hidden under the existential quantification, where we transform our existential type from $\exists\sigma.\ \sigma$ `code * ...` to $\exists\sigma'.\ \forall\omega.\ \sigma'$ `->` ω `code ->` ω `code * ...`. The hidden state σ' now has its stage-0 structure pulled out, like `A'` in our previous example. It is possible to translate this using the second formulation, but this breaks the data abstraction and exposes the internals of a particular implementation of `'a`, which is not acceptable. Clearly, to circumvent this, we would need first class contexts, and more specifically, existential quantification over contexts.

3.2 Reproducing Tagless-Final Encodings in CMTT

Another elegant application of MetaOCaml-like MSP languages is the staging of tagless-final interpreters [2]. We discover that embedding the simply typed lambda calculus (STLC) using De Bruijn indices can be very elegantly done with the representation type `('a, 'h) repr = [h: 'h ⊢ 'a]`, where `'h` is the polymorphic context type (corresponding to an STLC typing context) and `h` its instantiation. A De Bruijn index representing a binder is then encoded by the procedure of unwrapping the context and finding its corresponding value.

However, this procedure cannot be done at stage 0, for an exactly analogous reason as in section 3.1: the context being polymorphic, we cannot commit to one shape of the context. For example, the function which doubles its integer input, `lam (add (z) (z))` reduces to the following:

```
1 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}))))
```

This once again evidences that the System F extension alone seems to fail to match the expressiveness of MetaOCaml. Naturally, we would require first class contexts, and more specifically, type families indexed by context types, to elegantly express this tagless final embedding of STLC.

Additionally, we see a generic way of potentially encoding a polymporphic context with a polymorphic variable `h` in the context, and unwrapping the context with De Bruijn indices. We hypothesise that this generalisation does hold and that System F polymorphic CMTT can express any term in context polymorphic CMTT with only an overhead of unwrapping the context. We leave its verification to future work.

REFERENCES

[1] C. Calcagno, E. Moggi, and T. Sheard. 2003. Closed types for a safe imperative MetaML. *Journal of Functional Programming* 13, 3 (2003), 545–571. https://doi.org/10.1017/S0956796802004598

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238.

[3] R. Davies. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 184–195. https://doi.org/10.1109/LICS.1996.561317

[4] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (may 2001), 555–604. https://doi.org/10.1145/382780.382785

[5] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2021. Moebius: Metaprogramming using Contextual Types - The stage where System F can pattern match on itself (Long Version). *CoRR* abs/2111.08099 (2021). arXiv:2111.08099 https://arxiv.org/abs/2111.08099

[6] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

[7] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2016. Stream Fusion, to Completeness. *CoRR* abs/1612.06668 (2016). arXiv:1612.06668 http://arxiv.org/abs/1612.06668

[8] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. 1999. An Idealized MetaML: Simpler, and More Expressive. In *Programming Languages and Systems*, S. Doaitse Swierstra (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

[9] Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 281–308.

[10] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (jun 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

[11] Walid Taha, Cristiano Calcagno, Xavier Leroy, Ed Pizzi, Emir Pasalic, Jason Eckhardt, Roumen Kaiabachev, and Oleg Kiselyov. 2004. MetaOCaml: A compiled, type-safe multi-stage programming language. (01 2004).

[12] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

A A LITTLE BIT MORE CONTEXT

A.1 Staging the power function in CMTT – a first taste of MSP

Consider the function pow computing the nth power of x.

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

In a way, pow provides an *abstraction*. For any n, it gives a function `int -> int` which can compute the nth power of its input. However, if we knew n in advance, this abstraction would be cumbersome: at each step, instead of just multiplying, we unroll the fixpoint, check if $n = 0$, and do a recursive call.

MSP eliminates exactly this overhead. The concept is simple: we divide this program into two *stages* and use type-theoretical constructs to force the program to be reduced exactly in that order.

Stage 0 produces a piece of *compiled* code with respect to the first input n: `box (x:int |- x * x * ... x * 1)` (n times), of type `[x:int ⊢ int]` (meaning *piece of code of type int depending on a free variable x of type int*). Stage 1 can then efficiently compute x^n without any abstraction overhead.

How CMTT achieves this is shown in fig. 1, as presented by [10]. As we can see, the computation that can be performed given static input n are pulled outside boxes, and those which can't are pushed inside. Compositionality of the code boxes is enabled by the ability of unboxing a piece of code and splicing it in another box (`let box u = ... in ...`). When doing so, all the free variables have to be closed (`u with (σ)`), which happens by simultaneous substitution.

```
let rec pow: int -> [x:int |- int] = fun (n: int) ->
    if n = 0 then box (x: int |- 1)
    else
        let box pow_n_min_1 = pow (n-1) in
        box (x: int |- x * pow_n_min_1 with (x))
```

Fig. 1. Staged power function in CMTT

```
(*pow: int -> (int -> int) code*)
let pow n =
    (*aux: int -> int code -> int code*)
    let rec aux n x =
        if n = 0 then .<1>.
        else .<.~(x) * .~( aux (n-1) x )>.;;
    in
    .<fun x ->
        .~( aux n .<x>.)>.;;
10
```

Fig. 2. Staged power function in MetaOCaml

Yellow = stage 0; Green = stage 1; variables and corresponding bindings in the same colour

A.2 Staging the power function in MetaOCaml – intuition on 'symbolic evaluation'

MetaOCaml, based on $\lambda^{\bigcirc}$, hence linear temporal logic (LTL), achieves type-safe staging by dividing the program into different time steps, or *stages*. Stages are indicated using quotes `.<e>.` and splices `.~(e)`. Quoting code *delays* its execution to the next stage; splicing *advances* it to the previous stage. The current stage is the difference between the number of surrounding quotes and splices. The execution order is then exactly the stage order: execution only happens at stage 0. Importantly, each stage has its own independent context, and each stage-n variable can only be bound to a stage-n binder, or, specifically in MetaOCaml, a stage-m binder with $m \leq n$ thanks cross-stage persistence [12]. Figure 2 shows the staged power function, where constructs are highlighted by stage and variables coloured by their corresponding bindings.

With the implicit stage-specific context, MetaOCaml binds its free variables directly when the next-stage code is created. For example, stage-1 free variable `x` on line 9 is bound to the stage-1 binder on line 8. This is in contrast to Lys boxes, where free variables are explicitly carried in their types and are only instantiated when used.

This is the property that enables symbolic evaluation. It is expressed as functions of type `A code -> B code`, which constructs a `B code` within which the input `A code` is spliced into various places. Because the input has all its free variables statically bound when it was constructed, all free variables the output `B code` inherits will also be bound in that same context. In other words, the output implicitly depends on the context of the input. For example, given `n`, `aux n .<x>.` (line 9) reduces to an `int code` containing the variable `x`, which is bound to the same stage-1 binder on line 8 as the input.

B CASE STUDIES

B.1 Stream Fusion

```
(* Pull streams (3) *)
datatype ('a, 's) stream_shape = Nil | Cons of ('a * 's);;
datatype 'a stream = St of (exists 's. ('s * ('s -> ('a, 's) stream_shape)));;

(* Simple staging (4.2) *)
datatype 'a st_stream = St_staged of (exists 's. ([]'s * ([s: 's |- ('a, 's)
    stream_shape])));;

(* Fusing the stepper using CPS (5.1, the furthest we went) *)
datatype 'a st_stream_2 = St_staged_2 of (
    exists 's.
        ([]'s *
        ['fold_loop_type, 'fold_z_type, 'o;
        cons_cont:
            [fold_loop: 'fold_loop_type, fold_z: 'fold_z_type, a: 'a, t:'s |- 'o]
        |-
            [fold_loop: 'fold_loop_type, fold_z: 'fold_z_type,
                state:'s, nil_cont: 'o |- 'o]])
);;
```

Fig. 3. Stream types for reproducing Strymonas [7] (labeled with the corresponding section names)

B.2 Tagless final interpreter of STLC

We present the a code fragment of the tagless final embedding in fig. 4.

```
datatype ('a, 'h) repr = [h: 'h |- 'a];;

let lam: forall 'a. forall 'b. forall 'h. ('b, ('a * 'h)) repr -> ('a -> 'b, 'h)
    repr = fun (b: ('b, ('a * 'h)) repr) ->
    let box u = b in
    (box (h: 'h |- fun (x: 'a) -> u with ((x, h))));;
let app: forall 'a. forall 'b. forall 'h. ('a -> 'b, 'h) repr -> ('a, 'h) repr ->
    ('b, 'h) repr
= fun (f: ('a -> 'b, 'h) repr) -> fun (x: ('a, 'h) repr) ->
    let box ff = f in
    let box xx = x in
    (box (h: 'h |- (ff with (h)) (xx with (h))));;

(* De Bruijn indices *)
let z: forall 'a. forall 'h. ('a, ('a * 'h)) repr
    = (box (h: ('a * 'h) |- match h with (a, h) -> a));;
let s: forall 'a. forall 'b. forall 'h. ('a, 'h) repr -> ('a, ('b * 'h)) repr
    = fun (x: ('a, 'h) repr) ->
        let box u = x in
        (box (h: ('b * 'h) |- match h with (b, h) -> u with (h)));;
```

Fig. 4. Fragment of a staged tagless final interpreter of STLC in Lys