# Contextual Modal MetaML: Syntax and Full Abstraction

Haoxuan Yin[1], Andrzej Murawski[1], and Luke Ong[2]

[1]University of Oxford
[2]Nanyang Technological University

## 1 Type-safe metaprogramming with references for open code

Generative metaprogramming languages enable programmers to write programs that generate programs. The distinctive feature of quotation-based metaprogramming languages such as MetaML [16] is their once-and-for-all type safety guarantee: well-typed metaprograms always generate well-typed object programs. This stands in contrast to AST-based metaprogramming languages such as C++ Templates [15, 17] and Template Haskell [14, 1], where the generated object program needs to undergo an additional typechecking phase.

MetaML's strong safety guarantee is difficult to achieve, especially in the presence of higher-order references that can store code values. For example, consider the following MetaOCaml [9, 8] program:

```
let r = ref .< 1 >.;;
let f = .< fun x -> .~(r := .< x >.; .<0>.) >.;;
let t = Runcode.run (!r);;
```

The Bracket operator `.< >.` converts a term into a static piece of code, and the Escape operator `.~` allows terms to be evaluated inside a Bracket. In this example, the term `r := .< x >.; .<0>.` is evaluated, even if it occurs under the binder `fun x ->` . As a result, although the program appears to be well-scoped, the reference `r` stores the code `.< x >.` which contains a free variable. It is therefore unsafe to run the code stored in `r`. Currently, MetaOCaml raises a runtime exception whenever there is an attempt to use a piece of code that contains free variables.

Recently, Hu and Pientka [4] developed a type theory for metaprogramming called *layered modal type theory*. They use contextual modal types [13] to track and reason about free variables in code values explicitly, and provide a strong safety guarantee on the generation and manipulation of such code values. However, their result is based on an equational type theory that lacks general recursion and references.

In this paper, we present Contextual Modal MetaML, *the first metaprogramming language that allows storing and running open code under a strong type safety guarantee*. The language has two key operators. The Box operator, **box** $M$, converts any term $\Gamma \vdash M : T$ into a piece of code. The resulting type, $\square(\Gamma \vdash T)$, records both the type and the typing context of $M$. The Letbox operator, **letbox** $u \leftarrow M_1$ **in** $M_2$, "unboxes" the code $M_1$, and substitutes the extracted term for $u$ in $M_2$. For such unboxing to be type-safe, all occurrences of $u$ in $M_2$ must be associated with an explicit substitution $\delta$ that provides values for all the free variables in $M_1$. This guarantees that however a piece of code is used or passed around in references, its free variables can never be leaked in an undesirable context.

Consider the classical example of *staging* the power function:

```
let power n x =
    let y = ref 1 in
    for i = 1 to n do
        y := !y * x
    done;
    !y;;
```

In Contextual Modal MetaML, one could write the program

```
let power_staged n =
    let y = ref box 1 in
    for i = 1 to n do
        y := letbox u = !y in box (u * x)
    done;
    letbox u = !y in fun x -> u;;
let f = power_staged 3;;
```

Table 1: Type systems for quotation-based metaprogramming languages

| Type system | Imperative | Allows storing open code | Allows running open code | Type safety | Multi-layered |
|---|---|---|---|---|---|
| MetaML [16] | × | N/A | × | ✓ | ✓ |
| Closed Types for MetaML [2] | ✓ | × | × | ✓ | ✓ |
| Refined Environment Classifiers [11] | ✓ | ✓/× | × | ✓ | × |
| MetaOCaml [10] | ✓ | ✓ | ✓ | × | ✓ |
| Template Haskell [14] | × | N/A | × | ✓/× | × |
| Typed Template Haskell [18] | × | N/A | × | ✓ | × |
| MacoCaml [19] | ✓ | × | × | ✓ | × |
| Mœbius [7] | × | N/A | ✓ | × | ✓ |
| Layered Modal Type Theory [4] | × | N/A | ✓ | ✓ | × |
| $n$-layered Modal Type Theory [5] | × | N/A | ✓ | ✓ | ✓ |
| $\lambda^{\bigcirc\triangleright}$ [3] | × | N/A | × | ✓ | ✓ |
| Contextual Modal MetaML (our work) | ✓ | ✓ | ✓ | ✓ | × |

The reference `y` stores the open code `box (1 * x * ...)` and is dynamically updated during the for-loop. The function `f` evaluates to `fun x -> x * x * x`, which is a more efficient program than the plain `power n` function as it no longer contains for-loops.

In Table 1, we summarise the design space of type systems for quotation-based metaprogramming.

## 2 Operational game semantics for contextual modal types

When performing metaprogramming-based program optimisations, a critical concern is whether the optimisation is faithful. In the above power example, we expect that `power` and `power_staged` are equivalent. To address this concern, we need to develop a theoretical foundation for the following question: *When are two Contextual Modal MetaML programs equivalent?* Operational game semantics [12, 6] has been successfully used to construct fully abstract models for various programming languages. It represents the behaviour of a program in a context as a sequence of interactions between the program and the context. An interaction is represented as an *action*, and a sequence of actions is called a *trace*. The semantics of a program is then given by a set of all the traces that the program can generate according to a set of transition rules.

For example, consider a simplified trace

$$\overline{f_1} \quad f_1(3) \quad \overline{f_2} \quad f_2(2) \quad \overline{8}$$

Each action represents either a call or a return. Actions performed by the program are overlined, while those without an overline correspond to the context. The program and the context alternate in taking actions as if it were a dialogue. The symbols $f_1$ and $f_2$ are used in the dialogue between the Player and the Opponent and are intended as abstract representations of function values, in this case functions that are passed to the Opponent by the Player. Since these are merely names, the Opponent gains no knowledge of the underlying function beyond its type. In contrast, the Player will maintain a record mapping these names to the corresponding function values.

The trace above can be generated by the `power_staged` program we mentioned earlier. In this trace, the program first announces itself as a function represented by the name $f_1$ (return). Then the context asks for the result of applying the function $f_1$ to $n = 3$ (call), and the program responds with another function name $f_2$ (return). The name $f_2$ denotes the result of computing `power_staged 3`, which is essentially the function `fun x -> 1 * x * x * x`, but this underlying value is hidden from the context. The context further supplies the argument $x = 2$ to $f_2$ (call), and the program responds that the result of the computation is 8. On the one hand, the trace can be generated from `power_staged` alone according to our transition rules. On the other hand, it also corresponds to the evaluation context $K = \bullet\ 3\ 2$. This correspondence between traces and contexts is key to our full abstraction result.

The trace above can also be generated by `power`. For `power`, the underlying value of $f_2$ is

```
fun x ->
    let y = ref 1 in
    for i = 1 to 3 do
        y := !y * x
```

```
done;
!y;;
```

Again, this underlying value is not reflected in the action, so the difference from `power_staged` is not observable to the context.

Writing $\mathrm{Tr}(M)$ for the set of traces corresponding to $M$ according to our model, we shall have $\mathrm{Tr}(\texttt{power}) = \mathrm{Tr}(\texttt{power\_staged})$. Given that the trace model is fully abstract, we can then deduce $\texttt{power} \approx \texttt{power\_staged}$ where $\approx$ stands for contextual equivalence. This equivalence confirms the faithfulness of the staging transformation.

The trace above is not too different from what we can find in existing trace models for higher-order functions and references [12]. The distinctive feature of our model is that the traces can also involve names that represent code values, which we shall call *box values*. To illustrate this, let us consider the following program, reminiscent of the axiom $K$ for modal logic: $\Box(p \to q) \to \Box p \to \Box q$.

$$\begin{aligned}&\vdash_0 \lambda xy.\mathbf{letbox}\ u \leftarrow x\ \mathbf{in}\ \mathbf{letbox}\ v \leftarrow y\ \mathbf{in}\ \mathbf{box}\ u^{x_2/x_1}\ v^{y_2/y_1}\\ &: \Box(x_1 : \mathbf{ref\ Int} \to \mathbf{Int} \vdash \mathbf{ref\ Int} \to \mathbf{Int}) \to \Box(y_1 : \mathbf{ref\ Int} \vdash \mathbf{ref\ Int})\\ &\quad \to \Box(x_2 : \mathbf{ref\ Int} \to \mathbf{Int}, y_2 : \mathbf{ref\ Int} \vdash \mathbf{Int})\end{aligned}$$

The notation $V/x$ represents capture-avoiding substitution. For example, the term $u^{x_2/x_1}$ stands for the result of replacing all occurrences of $x_1$ with $x_2$ in the unboxed term represented by $u$.

This program can generate the trace

$$\begin{aligned}&(\overline{f_1}, \cdot, \cdot)\ (f_1(b_1), \cdot, \cdot)\ (\overline{f_2}, \cdot, \cdot)\ (f_2(b_2), \cdot, \cdot)\ (\overline{b_3}, \cdot, \cdot)\ (\mathbf{run}\ b_3^{f_3/x_2, \ell_1/y_2}, \Sigma, \chi_1)\\ &\overline{(\mathbf{run}\ b_1^{f_4/x_1}, \Sigma, \chi_1)}\ (f_5, \Sigma, \chi_1)\ \overline{(\mathbf{run}\ b_2^{\ell_1/y_1}, \Sigma, \chi_1)}\ (\ell_1, \Sigma, \chi_1)\ (\overline{f_5}(\ell_1), \Sigma, \chi_1)\\ &(f_4(\ell_1), \Sigma, \chi_1)\ (\overline{f_3}(\ell_1), \Sigma, \chi_1)\ (1, \Sigma, \chi_2)\ (\overline{1}, \Sigma, \chi_2)\ (1, \Sigma, \chi_2)\ (\overline{1}, \Sigma, \chi_2)\end{aligned}$$

where $\Sigma = \ell_1 : \mathbf{ref\ Int}$ is a typing context for references, and $\chi_1 = \ell_1 \mapsto 0$ and $\chi_2 = \ell_1 \mapsto 1$ are heaps.

Here, each action consists of three components. The first component is the questions and answers that we discussed above. The second and third components provide a heap $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$ that contains the references being shared by the program and the context. In the trace, the program first announces itself as a function represented by the name $f_1$ (return). Then the context asks for the result of applying $f_1$ to $x = b_1$ and $y = \ell_1$ sequentially (call), triggering $\overline{f_2}$ and $\overline{b_3}$ (return). The box name $b_3$ has type $\Box(x_2 : \mathbf{ref\ Int} \to \mathbf{Int}, y_2 : \mathbf{ref\ Int} \vdash \mathbf{Int})$, so in order to run it, the context needs to provide values for the variables $x_2$ and $y_2$, which are of type $\mathbf{ref\ Int} \to \mathbf{Int}$ and $\mathbf{ref\ Int}$ respectively. In the corresponding action $\mathbf{run}\ b_3^{f_3/x_2, \ell_1/y_2}$ (call), these values are represented by the function name $f_3$ and concrete location value $\ell_1$. Now the program needs to evaluate $\#b_1^{f_3/x_1}\ \#b_2^{\ell_1/y_1}$, where $\#b_i$ stands for the unboxed term extracted from box name $b_i$. To evaluate this term, the program asks for the result of running $b_1$ and $b_2$ (call) under corresponding substitutions. Again, the function value $f_3$ is represented by a fresh name $f_4$, while the location value $\ell_1$ is passed directly. After obtaining the function name $f_5$ and location value $\ell_1$ as answers from the context (return), the program can finally compute the result by applying $f_5$ to $\ell_1$ (call). The context then asks for the result of applying $f_4$ to $\ell_1$ (call), which the program can only answer after knowing the result of applying $f_3$ to $\ell_1$ (call). The function $f_3$ modifies the value stored in location $\ell_1$ from 0 to 1 and returns the updated value 1 (return). Finally, a sequence of answers respond to all the pending questions so far (return).

The above trace corresponds to interacting with the evaluation context

$$\begin{aligned}&\mathbf{letbox}\ u \leftarrow \bullet\ (\mathbf{box}\ x_1)\ (\mathbf{box}\ y_1)\ \mathbf{in}\\ &\mathbf{let}\ x_3 \leftarrow \lambda z.z := !z + 1; !z\ \mathbf{in}\\ &\mathbf{let}\ y_3 \leftarrow \mathbf{ref}\ 0\ \mathbf{in}\\ &\quad u^{x_3/x_2, y_3/y_2}\end{aligned}$$

**Related work** Recently, Chiang and Xie [3] also presented a type system for metaprogramming based on modal types. Their system allows quoting and splicing code, but does not allow running code. This is because their system is based on temporal logic, where $\bigcirc A \to A$ is not valid. In contrast, LMTT [4] and our system are based on S4 modal logic, which permits the axiom $\Box A \to A$ that runs closed code. We follow LMTT in extending this axiom to contextual types $\Box(\Gamma \vdash A)$ and allow safely running open code.

# References


[1] Martin Berger, Laurence Tratt, and Christian Urban. Modelling Homogeneous Generative Meta-Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming*, volume 74 of *LIPIcs*, pages 5:1–5:23, 2017.

[2] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, May 2003.

[3] Tsung-Ju Chiang and Ningning Xie. Multi-stage Programming with Splice Variables. *Proc. ACM Program. Lang.*, 9(ICFP):400–434, 2025.

[4] Jason Z. S. Hu and Brigitte Pientka. Layered Modal Type Theory: Where Meta-programming Meets Intensional Analysis. In Stephanie Weirich, editor, *Programming Languages and Systems*, volume 14576, pages 52–82. Springer Nature Switzerland, Cham, 2024.

[5] Jason Z. S. Hu and Brigitte Pientka. A Layered Approach to Intensional Analysis in Type Theory. *ACM Trans. Program. Lang. Syst.*, 46(4):15:1–15:43, January 2025.

[6] Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 348–374. Springer, 2021.

[7] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. Møebius: Metaprogramming using contextual types: The stage where system f can pattern match on itself. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022.

[8] Oleg Kiselyov. The Design and Implementation of BER MetaOCaml: System Description. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, Michael Codish, and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475, pages 86–102. Springer International Publishing, Cham, 2014.

[9] Oleg Kiselyov. MetaOCaml Theory and Implementation. *CoRR*, abs/2309.08207, 2023.

[10] Oleg Kiselyov. MetaOCaml: Ten Years Later - System Description. In Jeremy Gibbons and Dale Miller, editors, *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15-17, 2024, Proceedings*, volume 14659 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2024.

[11] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers: Type- and scope-safe code generation with mutable cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Proceedings*, pages 271–291. Springer Verlag, 2016.

[12] James Laird. A Fully Abstract Trace Semantics for General References. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, volume 4596, pages 667–679. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[13] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008.

[14] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16, Pittsburgh Pennsylvania, October 2002. ACM.

[15] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

[16] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211–242, October 2000.

[17] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide (2nd Edition)*. Addison-Wesley Professional, 2nd edition, August 2017.

[18] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. Staging with class: A specification for typed template Haskell. *Proceedings of the ACM on Programming Languages*, 6(POPL):61:1–61:30, January 2022.

[19] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. MacoCaml: Staging Composable and Compilable Macros. *Proceedings of the ACM on Programming Languages*, 7(ICFP):209:604–209:648, August 2023.