Contextual 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. In quotation-based metaprogramming languages such as MetaML [10], type safety can be difficult to achieve in the presence of higher-order references that can store code values. Currently, MetaOCaml [7, 6] raises a runtime exception whenever there is an attempt to use a piece of code that contains free variables.

In this paper, we present Contextual MetaML, *the first metaprogramming language that allows storing and running open code under a strong type safety guarantee.* The language is based on Hu and Pientka's layered modal type theory [3], where contextual types [9] are used to track and reason about free variables in code values explicitly. 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 δ 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 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;;
```

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.

2 Closed instances of use with both CBN and CBV substitutions

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 MetaML programs equivalent?* To answer this question, people frequently utilise *closed instances of use* (CIU) [2, 11] approximations, which say that equivalence under all suitable contexts coincides with equivalence under all suitable evaluation contexts, heaps, and substitutions that close all free variables.

In Contextual MetaML, there are two kinds of variables. Local variables x, y, z are bound by λ-abstractions and substituted in β-reductions. Such variables are meant to represent values, and are substituted in a call-by-value manner. Global variables u, v, w are bound by **letbox** and substituted in the reduction rule

$$(\textbf{letbox } u \leftarrow \textbf{box } M_1 \textbf{ in } M_2, h) \rightarrow (M_2[M_1/u], h)$$

Such variables are meant to represent unboxed code (M_1 in the above rule), and are substituted in a call-by-name manner. Therefore, our definition of closed instances of use needs to take into account both kinds of substitutions.

Table 1: Equivalence models for MetaML

Paper	Approach	Fully abstract	Imperative	Typed
Taha and Sheard [10]	Equational rules	×	×	✓
Berger and Tratt [1]	Hoare logic	✓	×	✓
Inoue and Taha [4]	Applicative bisimulation	✓	×	×
Our work	Operational game semantics	✓	✓	✓

For local variables, we consider all closing substitutions that map them to values, while for global variables, we consider all substitutions that map them to arbitrary terms. In short, *local variables represent closed values, global variables represent open terms*[1].

For example, consider the terms $x : \mathbf{Int} \vdash_0 x : \mathbf{Int}$ and $x : \mathbf{Int} \vdash_0 x; x : \mathbf{Int}$. All call-by-value substitutions will map x to an integer value $\widehat{n}$, and $\widehat{n}$ and $\widehat{n}; \widehat{n}$ are always equivalent. This is consistent with the fact that the two terms are contextually equivalent.

On the other hand, consider the terms $u : (\cdot \vdash \mathbf{Int}) \vdash_0 u : \mathbf{Int}$ and $u : (\cdot \vdash \mathbf{Int}) \vdash_0 u; u : \mathbf{Int}$, where the typing context $u : (\cdot \vdash \mathbf{Int})$ says that u stands for a closed term of type **Int**. A call-by-name substitution may map u to an arbitrary term M of type **Int**, for example $\ell :=!\ell + 1; !\ell$vwhere ℓ is location of type **ref** Int. Under this substitution, the two terms are not equivalent. Indeed, the two terms are not contextually equivalent as they can be distinguished by the context **let** $x \leftarrow$ **ref** 0 **in letbox** $u \leftarrow$ **box** $(x :=!x + 1; !x)$ **in** $\bullet$.

3 Operational game semantics for imperative metaprogramming

Operational game semantics [8, 5] is a more powerful tool to reason about contextual equivalences in general. Write $\mathrm{Tr}(M)$ for the set of traces corresponding to M. According to our model, we shall have $\mathrm{Tr}(\texttt{power}) = \mathrm{Tr}(\texttt{power\_staged})$. For example, the (simplified) trace $\overline{f_1}\quad f_1(3)\quad \overline{f_2}\quad f_2(2)\quad \overline{8}$ belongs to both of them. 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 [8]. 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 in code. 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.

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 **ref Int** $\to$ **Int** and **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 ℓ_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. The rest of the trace is self-explanatory.

The above trace corresponds to the evaluation context

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

Our trace model is the first fully abstract model for an imperative MetaML-style metaprogramming language. A comparison with similar results can be found in Table 1.

[1]The word "open" here means that the term *might* (but does not have to) contain free variables.

References

[1] Martin Berger and Laurence Tratt. Program Logics for Homogeneous Generative Run-Time Meta-Programming. *Log. Methods Comput. Sci.*, 11(1), 2015.

[2] Furio Honsell, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Variable Typed Logic of Effects. *Inf. Comput.*, 119(1):55–90, 1995.

[3] 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.

[4] Jun Inoue and Walid Taha. Reasoning about multi-stage programs. *Journal of Functional Programming*, 26:e22, January 2016.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] Carolyn Talcott. Reasoning about Programs With Effects. *Electronic Notes in Theoretical Computer Science*, 14:301–314, January 1998.