Contextual MetaML: Syntax and Full Abstraction

Haoxuan Yin
University of Oxford, UK

Andrzej S. Murawski
University of Oxford, UK

C.-H. Luke Ong
Nanyang Technological University, Singapore

Abstract

MetaML-style metaprogramming languages allow programmers to construct, manipulate and run code. In the presence of higher-order references for code, ensuring type safety is challenging, as free variables can escape their binders. In this paper, we present Contextual MetaML, *the first metaprogramming language that supports storing and running open code under a strong type safety guarantee*. The type system utilises contextual modal types to track and reason about free variables in code explicitly.

A crucial concern in metaprogramming-based program optimisations is whether the optimised program preserves the meaning of the original program. Addressing this question requires a notion of program equivalence and techniques to reason about it. In this paper, we provide a semantic model that captures contextual equivalence for Contextual MetaML, establishing *the first full abstraction result for an imperative MetaML-style language*. Our model is based on traces derived via operational game semantics, where the meaning of a program is modelled by its possible interactions with the environment. We also establish a novel closed instances of use theorem that accounts for both call-by-value and call-by-name closing substitutions.

2012 ACM Subject Classification Theory of computation → Program semantics; Software and its engineering → Imperative languages; Theory of computation → Type theory

Keywords and phrases Metaprogramming, operational game semantics, trace model, contextual modal type theory

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.83

Related Version *Full Version*: `https://arxiv.org/abs/2602.03033` [42]

Funding The work was partially supported by National Research Foundation, Singapore, under its RSS Scheme (NRF-RSS2022-009).
Haoxuan Yin: funded by Oxford-DeepMind Graduate Scholarship.

Acknowledgements We thank the reviewers for their helpful comments.

1 Introduction

1.1 Type-safe metaprogramming with higher-order 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 [36] 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 [34, 38] and Template Haskell [33, 4], 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 [22, 21] 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. This solution is unsatisfactory for two reasons. First, it breaks the type safety guarantee of MetaML, as such scope extrusions are detected at runtime rather than at compile time. Second, it limits the usability of the language, as storing and manipulating code with free variables is a useful tool in generative metaprogramming.

Recently, Hu and Pientka [12] developed a type theory for metaprogramming called *layered modal type theory*. They use contextual types [29] 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 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, $\Box(\Gamma \vdash T)$, 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;;
```

We would like a function `power_staged n` whose output is a function that takes x and computes x^n. In MetaOCaml, one would have to write the following program:

```
let rec power_staged_gen' n x =
    if n <= 0 then .< 1 >.
    else .< .~(power_staged_gen' (n - 1) x) * .~x >.;;
let power_staged' n =
    Runcode.run .< fun x -> .~(power_staged_gen' n .< x >.) >.;;
let f' = power_staged' 3;;
```

The function `f'` evaluates to `fun x -> 1 * x * x * x`, which is a more efficient program than the plain `power n` function as it no longer contains branching and recursion. However, the programmer has to write an auxiliary recursive function `power_staged_gen'`, and the pervasive use of Brackets and Escapes makes writing such programs laborious and error-prone.

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 [36]	×	N/A	×	✓	✓
Closed Types for MetaML [6]	✓	×	×	✓	✓
Refined Environment Classifiers [24]	✓	✓/×[1)]	×	✓	×
MetaOCaml [23]	✓	✓	✓	×	✓
Template Haskell [33]	×	N/A	×	✓/×[2)]	×
Typed Template Haskell [40]	×	N/A	×	✓	×
MacoCaml [41]	✓	×	×	✓	×
Mœbius [18]	×	N/A	✓	×	✓
Layered Modal Type Theory [12]	×	N/A	✓	✓	×
n-layered Modal Type Theory [11]	×	N/A	✓	✓	✓
Contextual MetaML (our work)	✓	✓	✓	✓	×

1) Refined Environment Classifiers allow storing open code temporarily, but such storage cannot go out of the range of the binder for the free variable.
2) In Template Haskell, if a program is executed, it is guaranteed not to go wrong. However, it requires typechecking any generated program before running it, so typechecking and program execution are interleaved.

In Contextual MetaML, one could write the program as[1]:

```
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;;
```

Here, the code is constructed in a way that is more intuitive and closer to the original imperative `power` function. The reference `y` stores the open code `box (1 * x * ...)` and is dynamically updated during the for-loop.

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

1.2 Closed instances of use with both call-by-name and call-by-value substitutions

After introducing Contextual MetaML and proving its type safety, we move on to discuss program equivalence in this language. 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?*

Capturing this problem denotationally is commonly known as the *full abstraction* problem [28], where one seeks a sound and complete model for *contextual equivalence*. Two programs M_1 and M_2 are said to be contextually equivalent if their observable behaviours (in our case, termination) are the same in all suitable contexts.

[1] Here we use syntactic sugar, omitting the explicit substitution δ if it is the identity, i.e. $[x/x]$.

The behaviour of a program in an arbitrary context is difficult to analyze for two reasons. First, the program might be erased or duplicated before it is executed. Second, the context can bind free variables in the program. To overcome these difficulties, programming language researchers utilise *closed instances of use* (CIU) [10, 37] 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

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

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. For local variables, we consider all closing substitutions that map them to values, while for global variables, we consider all closing substitutions that map them to arbitrary terms. In short, *local variables represent closed values, global variables represent open terms*[2].

As an example, consider the following two terms with free local variable x

$$x : \textbf{Int} \vdash_0 x : \textbf{Int}$$
$$x : \textbf{Int} \vdash_0 x; x : \textbf{Int}$$

where $M_1; M_2$ should be read as syntactic sugar for $(\lambda y.M_2)M_1$ for some fresh y. All call-by-value substitutions will map x to an integer value $\widehat{n}$. Under such substitutions, $\widehat{n}$ and $\widehat{n}; \widehat{n}$ are always equivalent as the latter β-reduces to the former in one step. This is consistent with the fact that the two terms are not distinguishable in any context, i.e., they are contextually equivalent.

On the other hand, consider the following two terms with free global variable u

$$u : (\cdot \vdash \textbf{Int}) \vdash_0 u : \textbf{Int}$$
$$u : (\cdot \vdash \textbf{Int}) \vdash_0 u; u : \textbf{Int}$$

where the typing context $u : (\cdot \vdash \textbf{Int})$ 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

$$\ell :=!\ell + 1; !\ell$$

where ℓ is location of type **ref** Int. Under this substitution, the two terms are not equivalent. Supposing the initial heap maps location ℓ to 0, then the first term evaluates to 1 as ℓ is only updated once, while the second term evaluates to 2 as ℓ is updated twice. Indeed, the two terms are not contextually equivalent as they can be distinguished by the context

$$\textbf{let}\ x \leftarrow \textbf{ref}\ 0\ \textbf{in letbox}\ u \leftarrow \textbf{box}\ (x :=!x + 1; !x)\ \textbf{in}\ \bullet$$

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

1.3 Operational game semantics for imperative metaprogramming

Although CIU approximations suffice for small examples like above, we need a more powerful tool to reason about contextual equivalences in general. *Operational game semantics* [25, 17] 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. This approach is valued for its intuitive, operational nature and is particularly well-suited to higher-order imperative programming languages. Its prominent feature is the use of names to represent in abstract terms the functions that are being called and communicated between the program and the context when they interact.

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, and will be made precise in Theorem 16 (Correctness) and Lemma 19 (Definability).

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 `power` $\approx$ `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 [25]. 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: $\Box(p \to q) \to \Box p \to \Box q$.

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

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 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. Again, the function value f_3 is represented by a fresh name f_4, while the location value ℓ_1 is passed directly. After obtaining the function name f_5 and location value ℓ_1 as answers from the context (return), the program can finally compute the result by applying f_5 to ℓ_1 (call). The context then asks for the result of applying f_4 to ℓ_1 (call), which the program can only answer after knowing the result of applying f_3 to ℓ_1 (call). The function f_3 modifies the value stored in location ℓ_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

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

Our trace model is the first fully abstract model for an imperative MetaML-style meta-programming language. A comparison with similar results can be found in Table 2.

Table 2 Models of contextual equivalence for quotation-based metaprogramming languages.

Paper	Approach	Fully abstract	Imperative	Typed
Taha and Sheard [36]	Equational rules	×	×	✓
Berger and Tratt [3]	Hoare logic	✓	×	✓
Inoue and Taha [15]	Applicative bisimulation	✓	×	×
Our work	Operational game semantics	✓	✓	✓

Outline. The paper is organized as follows. In Section 2, we introduce the syntax, type system, and operational semantics of Contextual MetaML, and prove its type safety. In Section 3, we define closed instances of use approximation for Contextual MetaML, and prove that it coincides with the contextual preorder. In Section 4, we define the trace model for Contextual MetaML based on operational game semantics. In Section 5, we prove that the trace model is fully abstract with respect to contextual equivalence. In Section 6, we use examples to illustrate how our trace model can be used to prove useful program equivalences in metaprogramming. In Sections 7 and 8, we discuss related and future work respectively.

2 Contextual MetaML

In this section, we present the syntax, operational semantics, and type safety of Contextual MetaML.

2.1 Syntax

The syntax of Contextual MetaML is given in Figure 1. The language can be thought of as the result of adding layered modal type theory [12] to ML with higher-order references.

There are two types of variables in Contextual MetaML. Local variables, ranging over $x, y, z, \ldots$, are ordinary variables that can be found in any call-by-value lambda calculus. An occurrence of x in a term is either free or bound. In the former case, it shall appear in the local variable context Γ as $x : T$, which reads "x is of type T". In the latter case, it shall be bound by a lambda abstraction $\lambda x.M$ or recursive definition $\mathbf{rec}\ y(x).M$.

Global variables, ranging over $u, v, w, \ldots$, are a special type of variables that are used in contextual type theories to represent open terms. A global variable u can never occur alone in a term; it must always occur as u^{δ}, associated with an explicit local substitution δ that provides values for free variables in the term represented by u. An occurrence of u in a term is also either free or bound. In the former case, it shall appear in the global variable context Ψ as $u : (\Gamma \vdash T)$, which reads "u is of type T under local variable context Γ". In the latter case, it shall be bound by a Letbox construct $\mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2$, which we shall explain now.

Contextual MetaML features two metaprogramming constructs. The Box operator $\mathbf{box}\ M$ converts a term M into a value that reads "a piece of code whose content is M". The Letbox operator $\mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2$ binds the occurrence of u in M_2, and reads "In M_2, let u be the term contained in M_1".

To avoid confusion, we shall use the word *term* as defined in Figure 1, and the word *code* to refer to a term of the form $\mathbf{box}\ M$. The word *program* is reserved for use in the operational game semantics.

Readers familiar with the original MetaML [36] might find it helpful to understand $\mathbf{box}\ M$ as MetaML's Bracket $\langle M \rangle$, and $\mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2$ as MetaML's Escape $\tilde{}M$ and Run $\mathbf{run}\ M$. Whether Letbox should be viewed as Escape or Run depends on whether

Types: $T \triangleq \mathbf{Unit} \mid \mathbf{Int} \mid T_1 \to T_2 \mid \mathbf{ref}\ T \mid \Box(\Gamma \vdash T)$

Store contexts: $\Sigma \triangleq \cdot \mid \Sigma, \ell : \mathbf{ref}\ T$

Local variable contexts: $\Gamma \triangleq \cdot \mid \Gamma, x : T$

Global variable contexts: $\Psi \triangleq \cdot \mid \Psi, u : (\Gamma \vdash T)$

Terms:

$$\begin{aligned} M \triangleq\ & () \mid \widehat{n} \mid x \mid u^{\delta} \mid \ell \mid \lambda x.M \mid M_1 M_2 \mid \mathbf{rec}\ y(x).M \mid M_1 \oplus M_2 \\ & \mid \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \mid \mathbf{ref}\ M \mid !M \mid M_1 := M_2 \mid \\ & \mathbf{box}\ M \mid \mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2 \end{aligned}$$

Primitive values: $a \triangleq () \mid \widehat{n} \mid \ell$

Values: $V \triangleq a \mid x \mid \lambda x.M \mid \mathbf{rec}\ y(x).M \mid \mathbf{box}\ M$

Local substitutions: $\delta \triangleq \cdot \mid \delta, V/x$

Global substitutions: $\sigma \triangleq \cdot \mid \sigma, M/u$

Heaps: $h \triangleq \cdot \mid h, \ell \mapsto V$

Contexts:

$$\begin{aligned} C \triangleq\ & \bullet \mid \lambda x.C \mid CM \mid MC \mid \mathbf{rec}\ y(x).C \mid C \oplus M \mid M \oplus C \mid \\ & \mathbf{if}\ C\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \mathbf{if}\ M_1\ \mathbf{then}\ C\ \mathbf{else}\ M_2 \mid \\ & \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ C \mid \mathbf{ref}\ C \mid !C \mid C := M \mid M := C \mid \\ & \mathbf{box}\ C \mid \mathbf{letbox}\ u \leftarrow C\ \mathbf{in}\ M \mid \mathbf{letbox}\ u \leftarrow M\ \mathbf{in}\ C \end{aligned}$$

Evaluation contexts:

$$\begin{aligned} K \triangleq\ & \bullet \mid KM \mid VK \mid K \oplus M \mid V \oplus K \mid \mathbf{if}\ K\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \\ & \mathbf{ref}\ K \mid !K \mid K := M \mid V := K \mid \mathbf{letbox}\ u \leftarrow K\ \mathbf{in}\ M \mid \\ & \mathbf{letbox}\ u \leftarrow V\ \mathbf{in}\ K \end{aligned}$$

Throughout the Figure, we assume $x \in \mathit{LVar}$, $u \in \mathit{GVar}$, $\ell \in \mathit{Loc}$, $n \in \mathbb{N}$, and $\oplus \in \{+, -, *, <, =\}$.

We write $\mathbf{let}\ x \leftarrow M_1\ \mathbf{in}\ M_2$ for $(\lambda x.M_2)M_1$. If x does not occur in M_2, we also write $M_1; M_2$.

Figure 1 Syntax.

Well-formed types: $\vdash_i T : \mathbf{typ}$

$$\text{(Typ-Unit)}\ \frac{}{\vdash_i \mathbf{Unit} : \mathbf{typ}} \qquad \text{(Typ-Int)}\ \frac{}{\vdash_i \mathbf{Int} : \mathbf{typ}} \qquad \text{(Typ-Ref)}\ \frac{\vdash_i T : \mathbf{typ}}{\vdash_i \mathbf{ref}\ T : \mathbf{typ}}$$

$$\text{(Typ-Arrow)}\ \frac{\vdash_i T_1 : \mathbf{typ} \qquad \vdash_i T_2 : \mathbf{typ}}{\vdash_i T_1 \to T_2 : \mathbf{typ}} \qquad \text{(Typ-Box)}\ \frac{\vdash_0 \Gamma : \mathbf{lctx} \qquad \vdash_0 T : \mathbf{typ}}{\vdash_1 \Box(\Gamma \vdash T) : \mathbf{typ}}$$

Well-formed store contexts: $\vdash_i \Sigma : \mathbf{sctx}$

$$\text{(SCtx-Empty)}\ \frac{}{\vdash_i \cdot : \mathbf{sctx}} \qquad \text{(SCtx-Cons)}\ \frac{\vdash_i \Sigma : \mathbf{sctx} \qquad \vdash_i T : \mathbf{typ} \qquad \ell \notin \mathrm{dom}(\Sigma)}{\vdash_i \Sigma, \ell : \mathbf{ref}\ T : \mathbf{sctx}}$$

Well-formed local variable contexts: $\vdash_i \Gamma : \mathbf{lctx}$

$$\text{(LCtx-Empty)}\ \frac{}{\vdash_i \cdot : \mathbf{lctx}} \qquad \text{(LCtx-Cons)}\ \frac{\vdash_i \Gamma : \mathbf{lctx} \qquad \vdash_i T : \mathbf{typ} \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash_i \Gamma, x : T : \mathbf{lctx}}$$

Well-formed global variable contexts: $\vdash \Psi : \mathbf{gctx}$

$$\text{(GCtx-Empty)}\ \frac{}{\vdash \cdot : \mathbf{gctx}} \qquad \text{(GCtx-Cons)}\ \frac{\vdash \Psi : \mathbf{gctx} \qquad \vdash_0 T : \mathbf{typ} \qquad \vdash_0 \Gamma : \mathbf{lctx} \qquad u \notin \mathrm{dom}(\Psi)}{\vdash \Psi, u : (\Gamma \vdash T) : \mathbf{gctx}}$$

Throughout the figure, we assume $i \in \{0, 1\}$.

Figure 2 Well-formed types and typing contexts.

u occurs under a Box. For example, $\mathbf{letbox}\ u \leftarrow M\ \mathbf{in}\ u$ corresponds to MetaML's $\mathbf{run}\ M$, while $\mathbf{letbox}\ u \leftarrow M\ \mathbf{in}\ \mathbf{box}\ (1 + u)$ corresponds to MetaML's $\langle 1 + \tilde{} M \rangle$. Compared with the original MetaML, Contextual MetaML takes a more uniform and elegant approach to unboxing code, and avoids the problem that *evaluations can go under binders*, as noted by Inoue and Taha [15].

2.2 Type System

The typing rules of Contextual MetaML are presented in Figures 2–4.

The way global variables are typed in the rule (Tm-GVar) should be read as follows: if u represents a term of type T under local variable context Γ', and δ is a type-preserving local substitution that maps variables in Γ' to values under the local variable context Γ, then u^δ is a term of type T under Γ.

The new Box type $\Box(\Gamma \vdash T)$ shall be read as "a piece of code whose content is a term of type T under local variable context Γ". The rules (Tm-Box) and (Tm-Letbox) are the introduction and elimination rules for this type respectively.

In the rule (Tm-Box), the $\mathbf{box}\ M$ operator binds all the free local variables in M. The typing context required to type M, i.e. Γ', is encoded in the type of $\mathbf{box}\ M$, i.e., $\Box(\Gamma' \vdash T)$. The resulting term $\mathbf{box}\ M$ can therefore be typed under any, possibly empty, local variable context Γ. For example, we have $\vdash_1 \mathbf{box}\ x : \Box(x : \mathbf{Int} \vdash \mathbf{Int})$.

Well-typed terms: $\Sigma; \Psi; \Gamma \vdash_i M : T$

$$\text{(Tm-Unit)}\ \frac{\vdash_i \Gamma : \mathbf{sctx} \qquad \vdash \Psi : \mathbf{gctx} \qquad \vdash_i \Gamma : \mathbf{lctx}}{\Sigma; \Psi; \Gamma \vdash_i () : \mathbf{Unit}}$$

$$\text{(Tm-Int)}\ \frac{\vdash_i \Gamma : \mathbf{sctx} \qquad \vdash \Psi : \mathbf{gctx} \qquad \vdash_i \Gamma : \mathbf{lctx}}{\Sigma; \Psi; \Gamma \vdash_i \widehat{n} : \mathbf{Int}}$$

$$\text{(Tm-LVar)}\ \frac{\begin{array}{cc}\vdash_i \Sigma : \mathbf{sctx} & \vdash_i \Gamma : \mathbf{lctx} \\ \vdash \Psi : \mathbf{gctx} & x : T \in \Gamma\end{array}}{\Sigma; \Psi; \Gamma \vdash_i x : T}$$

$$\text{(Tm-GVar)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i \delta : \Gamma' \qquad u : (\Gamma' \vdash T) \in \Psi}{\Sigma; \Psi; \Gamma \vdash_i u^{\delta} : T}$$

$$\text{(Tm-Loc)}\ \frac{\vdash_i \Sigma : \mathbf{sctx} \qquad \vdash \Psi : \mathbf{gctx} \qquad \vdash_i \Gamma : \mathbf{lctx} \qquad \ell : \mathbf{ref}\ T \in \Sigma}{\Sigma; \Psi; \Gamma \vdash_i \ell : \mathbf{ref}\ T}$$

$$\text{(Tm-Abs)}\ \frac{\Sigma; \Psi; \Gamma, x : T_1 \vdash_i M : T_2}{\Sigma; \Psi; \Gamma \vdash_i \lambda x.M : T_1 \to T_2}$$

$$\text{(Tm-App)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M_1 : T_1 \to T_2 \qquad \Sigma; \Psi; \Gamma \vdash_i M_2 : T_1}{\Sigma; \Psi; \Gamma \vdash_i M_1 M_2 : T_2}$$

$$\text{(Tm-Rec)}\ \frac{\Sigma; \Psi; \Gamma, y : T_1 \to T_2, x : T_1 \vdash_i M : T_2}{\Sigma; \Psi; \Gamma \vdash_i \mathbf{rec}\ y(x).M : T_1 \to T_2}$$

$$\text{(Tm-Arith)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M_1 : \mathbf{Int} \qquad \Sigma; \Psi; \Gamma \vdash_i M_2 : \mathbf{Int}}{\Sigma; \Psi; \Gamma \vdash_i M_1 \oplus M_2 : \mathbf{Int}}$$

$$\text{(Tm-If)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M_1 : \mathbf{Int} \qquad \Sigma; \Psi; \Gamma \vdash_i M_2 : T \qquad \Sigma; \Psi; \Gamma \vdash_i M_3 : T}{\Sigma; \Psi; \Gamma \vdash_i \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : T}$$

$$\text{(Tm-Ref)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M : T}{\Sigma; \Psi; \Gamma \vdash_i \mathbf{ref}\ M : \mathbf{ref}\ T} \qquad \text{(Tm-Deref)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M : \mathbf{ref}\ T}{\Sigma; \Psi; \Gamma \vdash_i !M : T}$$

$$\text{(Tm-Assign)}\ \frac{\Sigma; \Psi; \Gamma \vdash_i M_1 : \mathbf{ref}\ T \qquad \Sigma; \Psi; \Gamma \vdash_i M_2 : T}{\Sigma; \Psi; \Gamma \vdash_i M_1 := M_2 : \mathbf{Unit}}$$

$$\text{(Tm-Box)}\ \frac{\vdash_1 \Gamma : \mathbf{lctx} \qquad \Sigma; \Psi; \Gamma' \vdash_0 M : T}{\Sigma; \Psi; \Gamma \vdash_1 \mathbf{box}\ M : \Box(\Gamma' \vdash T)}$$

$$\text{(Tm-LetBox)}\ \frac{\Sigma; \Psi; \Gamma \vdash_1 M_1 : \Box(\Gamma' \vdash T) \qquad \Sigma; \Psi, u : (\Gamma' \vdash T); \Gamma \vdash_1 M_2 : T'}{\Sigma; \Psi; \Gamma \vdash_1 \mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2 : T'}$$

Throughout the figure, we assume $i \in \{0, 1\}$.

Figure 3 Well-typed terms.

Well-typed heaps: $\Sigma; \Psi; \Gamma \vdash h : \mathbf{heap}$

$$(\text{Heap})\ \frac{\mathrm{dom}(\Sigma) = \mathrm{dom}(h) \qquad \Sigma; \Psi; \Gamma \vdash_i h(\ell) : T \text{ for all } \ell : \mathbf{ref}\ T \in \Sigma}{\Sigma; \Psi; \Gamma \vdash h : \mathbf{heap}}$$

Well-typed contexts: $\Sigma; \cdot; \cdot \vdash C : (\Psi'; \Gamma'; i'; T') \to (\Psi; \Gamma; i; T)$

$$(\text{Ctx-Hole})\ \frac{\Psi' \subseteq \Psi \qquad \Gamma' \subseteq \Gamma \qquad i' \leq i}{\Sigma; \cdot; \cdot \vdash \bullet : (\Psi'; \Gamma'; i'; T) \to (\Psi; \Gamma; i; T)}$$

Other rules are omitted for brevity.

Well-typed local substitutions: $\Sigma; \Psi; \Gamma \vdash_i \delta : \Gamma'$

$$(\text{LSubst})\ \frac{\mathrm{dom}(\delta) = \mathrm{dom}(\Gamma') \qquad \Sigma; \Psi; \Gamma \vdash_i \delta(x) : T \text{ for all } x : T \in \Gamma'}{\Sigma; \Psi; \Gamma \vdash_i \delta : \Gamma'}$$

Well-typed global substitutions: $\Sigma; \Psi \vdash \sigma : \Psi'$

$$(\text{GSubst})\ \frac{\mathrm{dom}(\sigma) = \mathrm{dom}(\Psi') \qquad \Sigma; \Psi; \Gamma \vdash_0 \sigma(u) : T \text{ for all } u : (\Gamma \vdash T) \in \Psi'}{\Sigma; \Psi \vdash \sigma : \Psi'}$$

Throughout the Figure, we assume $i \in \{0, 1\}$.

Figure 4 Well-typed heaps, contexts and substitutions.

The rule (Tm-Letbox) links two usages of the contextual type $\Gamma \vdash T$ in the type system: in a global variable context as $u : (\Gamma \vdash T)$, and in a Box type as $\Box(\Gamma \vdash T)$. The rule says that if u is used as a term of type T under Γ' in M_2, then the code that it is substituted for, i.e. M_1, must indeed be a piece of code whose content has type T under Γ'.

Most of the typing judgments are annotated with a layer i that tracks the maximum depth of nested Boxes in all the associated types. For example, we have $x : \Box(\cdot \vdash \mathbf{Int}) \vdash_1 x : \Box(\cdot \vdash \mathbf{Int})$, and this judgment is not valid for layer $i = 0$, even if the term x itself does not contain any Box. In this paper, we follow Hu and Pientka [12] and restrict the language to contain two layers only. In other words, we forbid nesting of Box types at all. This restriction greatly simplifies the metatheory of the language while still allowing most of the common usages of metaprogramming as we shall show in various examples in Section 6. We discuss the possible extension to multiple layers in Section 8.

2.3 Operational Semantics

The small-step operational semantics of Contextual MetaML is presented in Figure 5. The rule (Letbox) does two things simultaneously. First, it extracts the content M_1 from the code $\mathbf{box}\ M_1$, reminiscent of MetaML's Escape and Run. Second, it substitutes M_1 for all the occurrences of the global variable u in M_2, reminiscent of call-by-name β-reductions.

Applying the substitutions created by the (β), (Rec), and (Letbox) rules needs extra care. $\lambda x.M$ and $\mathbf{rec}\ y(x).M$ bind the mentioned local variables, and $\mathbf{box}\ M$ binds all local variables. Similarly, $\mathbf{letbox}\ u \leftarrow M_1\ \mathbf{in}\ M_2$ binds the global variable u in M_2. When a substitution goes under these binders, the bound variables are removed from the substitution, and all other variables are preserved.

Reduction rules: $(M, h) \to (M', h')$

$$(\beta)\ \frac{}{((\lambda x.M)V, h) \to (M[V/x], h)} \qquad (\text{Arith})\ \frac{}{(\widehat{n_1} \oplus \widehat{n_2}, h) \to (\widehat{n_1 \oplus n_2}, h)}$$

$$(\text{If true})\ \frac{n \neq 0}{(\textbf{if } \widehat{n} \textbf{ then } M_1 \textbf{ else } M_2, h) \to (M_1, h)}$$

$$(\text{If false})\ \frac{}{(\textbf{if } \widehat{0} \textbf{ then } M_1 \textbf{ else } M_2, h) \to (M_2, h)}$$

$$(\text{Ref})\ \frac{\ell \notin \text{dom}(h)}{(\textbf{ref } V, h) \to (\ell, h \uplus [\ell \mapsto V])} \qquad (\text{Deref})\ \frac{\ell \in \text{dom}(h)}{(!\ell, h) \to (h(\ell), h)}$$

$$(\text{Assign})\ \frac{\ell \in \text{dom}(h)}{(\ell := V, h) \to ((), h[\ell \mapsto V])}$$

$$(\text{Rec})\ \frac{}{((\textbf{rec } y(x).M)V, h) \to (M[V/x, \textbf{rec } y(x).M/y], h)}$$

$$(\text{Letbox})\ \frac{}{(\textbf{letbox } u \leftarrow \textbf{box } M_1 \textbf{ in } M_2, h) \to (M_2[M_1/u], h)}$$

$$(\text{Ktx})\ \frac{(M_1, h_1) \to (M_2, h_2)}{(K[M_1], h_1) \to (K[M_2], h_2)}$$

We write $\to^*$ for the reflexive transitive closure of $\to$. We write $(M, h) \downarrow$ if there exists value V and heap h' such that $(M, h) \to^* (V, h')$.

Applying local substitutions: $M[\delta]$

$$\begin{array}{rcl@{\qquad}rcl}
()[\delta] & = & () & u^{\delta}[\delta'] & = & u^{\delta[\delta']} \\
(\widehat{n})[\delta] & = & \widehat{n} & \ell[\delta] & = & \ell \\
x[\delta] & = & \delta(x) & (\lambda x.M)[\delta] & = & \lambda x.M[\delta, x/x]
\end{array}$$

$$\begin{array}{rcl}
(\textbf{rec } y(x).M)[\delta] & = & \textbf{rec } y(x).M[\delta, y/y, x/x] \\
(\textbf{box } M)[\delta] & = & \textbf{box } M \\
(\textbf{letbox } u \leftarrow M_1 \textbf{ in } M_2)[\delta] & = & \textbf{letbox } u \leftarrow M_1[\delta] \textbf{ in } M_2[\delta]
\end{array}$$

where $\delta[\delta'](x) = \delta(x)[\delta']$, and $[\delta]$ commutes with all other term constructors.

Applying global substitutions: $M[\sigma]$

$$\begin{array}{rcl@{\qquad}rcl}
()[\sigma] & = & () & u^{\delta}[\sigma] & = & \sigma(u)[\delta[\sigma]] \\
(\widehat{n})[\sigma] & = & \widehat{n} & \ell[\sigma] & = & \ell \\
x[\sigma] & = & x & (\lambda x.M)[\sigma] & = & \lambda x.M[\sigma]
\end{array}$$

$$\begin{array}{rcl}
(\textbf{rec } y(x).M)[\sigma] & = & \textbf{rec } y(x).M[\sigma] \\
(\textbf{box } M)[\sigma] & = & \textbf{box } (M[\sigma]) \\
(\textbf{letbox } u \leftarrow M_1 \textbf{ in } M_2)[\sigma] & = & \textbf{letbox } u \leftarrow M_1[\sigma] \textbf{ in } M_2[\sigma, u/u]
\end{array}$$

where $\delta[\sigma](x) = \delta(x)[\sigma]$, and $[\sigma]$ commutes with all other term constructors.

Figure 5 Operational Semantics.

When we write a term $M[\delta]$ or $M[\sigma]$, the mentioned substitution is to be viewed as an implicit substitution. It is performed immediately, and is gone in the resulting term. When we write a term u^{δ}, the mentioned substitution is to be viewed as an explicit substitution, and the term stays as it is. When applying an implicit substitution σ or δ' to the term u^{δ}, the two substitutions are composed, yielding a new explicit substitution $\sigma[\delta]$ or $\delta'[\delta]$.

▶ Remark 1. One might have the intuition that a box can be mimicked by a function taking a unit argument, and unboxing can be mimicked by applying such a function to the unit value. Although this intuition can be helpful in establishing Lemma 8 below, there does not exist a translation from Contextual MetaML to ML. Consider the term $\lambda x.\mathbf{letbox}\ u \leftarrow x\ \mathbf{in}\ \mathbf{box}\ (u^{y/y} + y)$. It takes as input some code containing free variable y, and produces the code with "$+y$" appended to it. For example, it transforms $\mathbf{box}\ (y * y)$ to $\mathbf{box}\ (y{*}y{+}y)$. There is no corresponding ML function that can transform the function $\lambda z.y{*}y$ to $\lambda z.y * y + y$. Similarly, there is no ML counterpart to $\lambda x.\mathbf{letbox}\ u \leftarrow x\ \mathbf{in}\ \mathbf{box}\ (u + 1)$ that takes a closed function and appends "+1" to it.

2.4 Type Safety

▶ **Theorem 2** (Preservation). *If* $\Sigma; \Psi; \Gamma \vdash_i M : T$, $\Sigma; \Psi; \Gamma \vdash h : \mathbf{heap}$, *and* $(M, h) \to (M', h')$, *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma'; \Psi; \Gamma \vdash_i M' : T$, *and* $\Sigma'; \Psi; \Gamma \vdash h' : \mathbf{heap}$.

▶ **Lemma 3** (Canonical forms). *If* $\Sigma; \cdot; \cdot \vdash_i V : T$, *then one of the following cases holds:*

- $T = \mathbf{Unit}$ *and* $V = ()$.
- $T = \mathbf{Int}$ *and* $V = \widehat{n}$ *for some* $n \in \mathbb{N}$.
- $T = \mathbf{ref}\ T'$ *and* $V = \ell$ *for some* $\ell \in Loc$.
- $T = T_1 \to T_2$ *and* $V = \lambda x.M$ *or* $V = \mathbf{rec}\ y(x).M$.
- $T = \Box(\Gamma \vdash T')$ *and* $V = \mathbf{box}\ M$.

▶ **Theorem 4** (Progress). *If* $\Sigma; \cdot; \cdot \vdash_i M : T$ *and* $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$, *then either* M *is a value, or there exist* M', h' *such that* $(M, h) \to (M', h')$.

In the rest of this paper, we shall consider the following notion of equivalence for Contextual MetaML.

▶ **Definition 5** (Contextual preorder). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$, *we define* $M_1 \lesssim M_2$ *to hold if, for any context* $\Sigma'; \cdot; \cdot \vdash C : (\Psi; \Gamma; i; T) \to (\cdot; \cdot; i'; T')$ *and heap* $\Sigma'; \cdot; \cdot \vdash h : \mathbf{heap}$ *such that* $\Sigma' \supseteq \Sigma$, *if* $(C[M_1], h) \Downarrow$ *then* $(C[M_2], h) \Downarrow$.

We write $M_1 \approx M_2$ if $M_1 \lesssim M_2$ and $M_2 \lesssim M_1$.

3 Closed Instances of Use

Closed Instances of Use approximation [37] is widely used in reasoning about program equivalence in higher-order call-by-value languages. It says that two terms are contextually equivalent if and only if they are equivalent under all suitable evaluation contexts, heaps, and closing substitutions.

Contextual MetaML has two kinds of variables. Local variables are similar to normal variables in call-by-value languages, so for every free variable declared as $x : T$ in the local variable context, the closing substitution shall substitute a closed value of type T.

Global variables, on the other hand, are declared as $u : \Gamma \vdash T$ in the global variable context, which says that u represents a term of type T under local variable context Γ. Also, when a substitution for u is created by the (Letbox) rule, what gets substituted is exactly a term of type T under local variable context Γ. Therefore, for every such global variable u, the closing substitution shall substitute a term of type T under local variable context Γ.

The distinction between local and global variables can now be summarised as: "local variables represent closed values, global variables represent open terms." Note that the word "open" here means that the term might contain free variables, but does not have to.

▶ **Definition 6** (Closed Instances of Use). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*, we define* $M_1 \lesssim^{\text{CIU}} M_2$ *to hold if, for any global substitution* $\Sigma'; \cdot \vdash \sigma : \Psi$*, local substitution* $\Sigma'; \cdot; \cdot \vdash_i \delta : \Gamma$*, evaluation context* $\Sigma'; \cdot; \cdot \vdash K : (\cdot; \cdot; i; T) \to (\cdot; \cdot; i'; T')$*, and heap* $\Sigma'; \cdot; \cdot \vdash h : \mathbf{heap}$ *such that* $\Sigma' \supseteq \Sigma$*, if* $(K[M_1[\sigma][\delta]], h) \Downarrow$ *then* $(K[M_2[\sigma][\delta]], h) \Downarrow$*.*

Note that although we write the global substitution as $\Sigma'; \cdot \vdash \sigma : \Psi$, we are not saying that $\sigma(u)$ must be a closed term. By definition, this notation means $\Sigma'; \cdot; \Gamma \vdash \sigma(u) : T$ for all $u : \Gamma \vdash T$ in Ψ. In other words, $\sigma(u)$ can contain free local variables as declared in Ψ, but should not introduce any new free global variables.

▶ **Lemma 7** (CIU Basics). *Given* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*, if* $M_1 \lesssim^{\text{CIU}} M_2$*, then* $C[M_1] \lesssim^{\text{CIU}} C[M_2]$ *whenever the terms are typable and* C *is one of the following forms:* $\bullet M$*,* $\bullet \oplus M$*,* $\mathbf{if}\ \bullet\ \mathbf{then}\ M\ \mathbf{else}\ M'$*,* $\mathbf{ref}\ \bullet$*,* $!\bullet$*,* $\bullet := M$*,* $\mathbf{letbox}\ u \leftarrow \bullet\ \mathbf{in}\ M$*,* $M\bullet$*,* $M \oplus \bullet$*,* $\mathbf{if}\ M\ \mathbf{then}\ \bullet\ \mathbf{else}\ M'$*,* $\mathbf{if}\ M\ \mathbf{then}\ M'\ \mathbf{else}\ \bullet$*,* $\lambda x.\bullet$*,* $\mathbf{rec}\ y(x).\bullet$*,* $\mathbf{letbox}\ u \leftarrow M\ \mathbf{in}\ \bullet$*.*

▶ **Lemma 8** (CIU Box). *Given* $\Sigma; \Psi; \Gamma \vdash_0 M_1, M_2 : T$*, if* $M_1 \lesssim^{\text{CIU}} M_2$*, then* $\mathbf{box}\ M_1 \lesssim^{\text{CIU}} \mathbf{box}\ M_2$*.*

Proof Sketch. The "elimination rule" for $\mathbf{box}\ M_i$, i.e., $(\mathbf{letbox}\ u \leftarrow \mathbf{box}\ M_i\ \mathbf{in}\ M, h) \to (M[M_i/u], h)$, ends up in a term where M_i does not appear in an evaluation context, so we cannot use the assumption that $M_1 \lesssim^{\text{CIU}} M_2$ directly. To solve this problem, we utilise the (imperfect) intuition that a function taking unit argument mimics a box, and applying such function to unit mimics unboxing. For each occurrence of M_i in the resulting term, we can convert the term to an equivalent form, where $\lambda y.M_i$ appears in an evaluation context. Then we can use Lemma 7 to establish the relation. ◀

▶ **Theorem 9** (CIU). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*, we have* $M_1 \lesssim M_2$ *if and only if* $M_1 \lesssim^{\text{CIU}} M_2$*.*

4 Trace Model

This section presents the trace model of Contextual MetaML. To this end, we shall introduce a labelled transition system that will generate traces for a given term. Intuitively, they can be viewed as an abstract record of interaction between the term and a context. In particular, whenever functions or pieces of code are exchanged between the term and the context, the values will be represented symbolically using names, so that syntactically different but semantically equivalent values will not be distinguishable. In a similar spirit, a location in the program heap will be visible to the context only if it has been revealed.

4.1 Names

Names are a representation of closed higher-order values. They expose the type of a value but encapsulate the underlying syntax. We will rely on an infinite set of names $\text{Names} = \text{FNames} \uplus \text{BNames}$, partitioned into *function names* (FNames) and *box names* (BNames). The two sets will be used to represent function and box (code) values respectively. In line with Lemma 3, we assume that FNames and BNames are further partitioned by $\text{FNames} = \biguplus_{(T_1,T_2,i)\in \text{FTyp}} \text{FNames}_{T_1 \to T_2, i}$ where $\text{FTyp} = \{(T_1, T_2, i) \mid \vdash_i T_1 \to T_2 : \mathbf{typ}\}$,

and $\mathrm{BNames} = \biguplus_{(\Gamma,T)\in\mathrm{BTyp}} \mathrm{BNames}_{\Box(\Gamma\vdash T),1}$ where $\mathrm{BTyp} = \{(\Gamma, T) \mid \vdash_1 \Box(\Gamma \vdash T) : \mathbf{typ}\}$, to reflect the possible types of values. We assume that all the constituent sets are countably infinite. We will use f, b and n to range over FNames, BNames and Names respectively. We have typing rule

$$\text{(Tm-Name)}\ \frac{n \in \mathrm{Names}_{T,i} \qquad i' \geq i}{\Sigma; \Psi; \Gamma \vdash_{i'} n : T}$$

A name n is a closed value in the operational semantics, and subsumes any substitutions, i.e., $n[\delta] = n[\sigma] = n$.

To handle box names operationally, we add one special reduction rule

$$\text{(Letbox name)}\ \frac{}{(\mathbf{letbox}\ u \leftarrow b\ \mathbf{in}\ M, h) \to (M[\#b/u], h)}$$

where the symbol $\#b$ denotes the result of removing the outermost box of b. Like u, all the occurrences of $\#b$ should be associated with a local substitution that closes all the free local variables in it. Accordingly, we have typing rule

$$\text{(Tm-Unbox)}\ \frac{b \in \mathrm{Names}_{\Box(\Gamma'\vdash T),1} \qquad \Sigma; \Psi; \Gamma \vdash_i \delta : \Gamma'}{\Sigma; \Psi; \Gamma \vdash_i \#b^{\delta} : T}$$

Substitutions on $\#b^{\delta}$ are defined as $(\#b^{\delta})[\delta'] = \#b^{\delta[\delta']}$ and $(\#b^{\delta})[\sigma] = \#b^{\delta[\sigma]}$. The term $\#b^{\delta}$ is a redex in the operational semantics.

For a unified treatment of values of box type, we define $\#(\mathbf{box}\ M) = M$. One can verify that the above statements also hold if we change all occurrences of b to $\mathbf{box}\ M$.

4.2 Value abstraction

As already mentioned, function and box values will be abstracted by names in our traces, while ground values and locations will be represented literally. To keep track of abstractions, we need a notation that links values to the set of their possible abstractions. When the abstraction is a name, we will also need to record the correspondence between the name and the underlying value.

▶ **Definition 10** (value abstraction). *Given a closed value $\Sigma; \cdot; \cdot \vdash_i V : T$, we define its abstraction set $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i V : T)$ by cases from Lemma 3.*

- $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i () : \mathbf{Unit}) \triangleq \{((), \emptyset)\}$.
- $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i \widehat{n} : \mathbf{Int}) \triangleq \{(\widehat{n}, \emptyset)\}$.
- $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i \ell : \mathbf{ref}\ T) \triangleq \{(\ell, \emptyset)\}$.
- $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i V : T_1 \to T_2) \triangleq \{(f, f \mapsto V) \mid f \in \mathrm{FNames}_{T_1\to T_2,i}\}$.
- $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_1 V : \Box(\Gamma \vdash T)) \triangleq \{(b, b \mapsto V) \mid b \in \mathrm{BNames}_{\Box(\Gamma\vdash T),1}\}$.

We use A to refer to any abstraction, i.e. A ranges over unit, numerals, locations and names. We use $\nu(A)$ to denote the set of names occurring in A, and $\mu(A)$ for the set of locations occurring in A. Both $\nu(A)$ and $\mu(A)$ are either empty or singleton.

Having defined what it means to abstract values, we now lift the process to local substitutions and heaps, where we want to replace all the participating values with corresponding abstractions. We use ρ and χ to refer to abstract local substitutions and heaps respectively.

Given a local substitution $\Sigma; \cdot; \cdot \vdash_i \delta : \Gamma$, we let $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i \delta : \Gamma) \triangleq \{(\rho, \bigcup_x \gamma_x) \mid \mathrm{dom}(\rho) = \mathrm{dom}(\delta), (\rho(x), \gamma_x) \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i \delta(x) : \Gamma(x)), \cap_{x\in\mathrm{dom}(\rho)}\nu(\rho(x)) = \emptyset\}$. Given a heap $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$, we let $\mathbf{AVal}(\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}) \triangleq \{(\chi, \bigcup_\ell \gamma_\ell) \mid \mathrm{dom}(\chi) = \mathrm{dom}(h), (\chi(\ell), \gamma_\ell) \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i h(\ell) : \Sigma(\ell)), \cap_{\ell\in\mathrm{dom}(\chi)}\nu(\chi(\ell)) = \emptyset\}$. We define $\nu(\chi) = \biguplus_\ell \nu(\chi(\ell))$, i.e. $\nu(\chi)$ contains all names in χ. We define $\mu(\rho) = \biguplus_x \mu(\rho(x))$, i.e. $\mu(\rho)$ contains all locations in ρ.

4.3 Actions

In our model, the meaning of a program will be interpreted as a set of traces. A trace will be a sequence of actions alternating between the program and the context. Traditionally in game semantics, the program is referred to as P (Player, Proponent) and the context as O (Opponent). We will stick to this terminology below for compatibility with earlier works in the area. Intuitively, the actions should be viewed as abstractions of calls and returns made by the program or context. Traditionally, the former are called questions and the latter are answers. Note that actions will also feature abstract heaps $\Sigma; \cdot; \cdot \vdash \chi : \mathbf{heap}$, which are an abstraction of the part of the heap that was exposed during the interaction.

Our semantics will be based on the following kinds of actions.

- Player Answer (PA) $(\overline{A}, \Sigma, \chi)$. Program replies to a previous question with the answer $\Sigma; \cdot; \cdot \vdash_i A : T$, setting the values of shared locations to $\Sigma; \cdot; \cdot \vdash \chi : \mathbf{heap}$.
- Player Question of Function (PQF) $(\overline{f}(A), \Sigma, \chi)$. Program asks for the result of applying $f \in \mathrm{FNames}_{T_1 \to T_2, i}$ to the abstract value $\Sigma; \cdot; \cdot \vdash_i A : T_1$, setting the values of shared locations to $\Sigma; \cdot; \cdot \vdash \chi : \mathbf{heap}$.
- Player Question of Box (PRB) $(\overline{\mathbf{run}\ b^{\delta}}, \Sigma, \chi)$. Program asks for the result of running (unboxing) the box name $b \in \mathrm{BNames}_{\Box(\Gamma \vdash T)}$ under abstract local substitution $\Sigma; \cdot; \cdot \vdash_i \rho : \Gamma$, setting the values of shared locations to $\Sigma; \cdot; \cdot \vdash \chi : \mathbf{heap}$.
- Opponent Answer (OA) (A, Σ, χ), Opponent Question of Function (OQF) $(f(A), \Sigma, \chi)$, and Opponent Question of Box (OQB) $(\mathbf{run}\ b^{\rho}, \Sigma, \chi)$ are similar, except that they are initiated by the context.

An action is legitimate only if χ is closed with respect to reachable locations and disjoint names are used for abstraction: $\chi(\ell) = \ell'$ implies $\ell' \in \mathrm{dom}(\chi)$ and $\bigcap_{\ell \in \mathrm{dom}(\chi)} \nu(\chi(\ell)) = \emptyset$.

4.4 Configurations and transitions

Finally, we define the labelled transition system (LTS) that will generate traces from terms. We start off by discussing the shape of its configurations.

A *passive configuration* describes a moment when the context (Opponent) is about to initiate a computation. It is a tuple $(\Sigma, h, \gamma, \phi, \xi, \theta)$ where $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$ is the heap, γ maps function and box names to the value that they represent, $\phi \subseteq \mathrm{Names}$ collects all the names that have been introduced so far, $\xi := \cdot \mid \xi : (\Sigma_1 \vdash K : (\cdot; \cdot; T_1; i_1) \to (\cdot; \cdot; T_2; i_2))$ is a stack of evaluation contexts, and $\theta \subseteq \mathrm{dom}(h)$ is the set of locations currently shared by program and context.

Following an action by the context, passive configurations will evolve into active ones. *Active configurations* have the form $(M, i, T, \Sigma, h, \gamma, \phi, \xi, \theta)$ where $\Sigma; \cdot; \cdot \vdash_i M : T$ is a term and other components are as defined above. Active configurations correspond to computations by the program (Player). An active configuration can evolve into another active configuration through silent moves, or into a passive one after the program performs an action.

The admissible transitions in, what we call the Program LTS, are shown in Figure 6.

(Pτ) is a silent transition that embeds the operational semantics into the LTS. (PA) concerns cases when the LTS has reached a value. The value is then announced in abstracted form A through the action $(\overline{A}, \Sigma, \chi)$, where χ is the abstraction of the current heap h, restricted to locations that have been revealed to the context. (PQF) corresponds to reaching a β-redex where the function is an FName f. In other words, the program wants to call an unknown function provided earlier by the context. This is represented by the action $(\overline{f}(A), \Sigma, \chi)$, where A abstracts the argument. Meanwhile, the current evaluation context K is stored on the stack ξ for later use. (PQB) is analogous to the previous case, and handles

- (Pτ) $(M_1, i, T, \Sigma_1, h_1, \gamma, \phi, \xi, \theta) \xrightarrow{\tau} (M_2, i, T, \Sigma_2, h_2, \gamma, \phi, \xi, \theta)$
  where $(M_1, h_1) \to (M_2, h_2)$.
- (PA) $(V, i, T, \Sigma, h, \gamma, \phi, \xi, \theta) \xrightarrow{(\overline{A}, \Sigma', \chi)} (\Sigma, h, \gamma \uplus \gamma' \uplus \gamma'', \phi \uplus \nu(A) \uplus \nu(\chi), \xi, \mathrm{dom}(\chi))$
  where $(A, \gamma') \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i V : T)$, $(\Sigma', h') = (\Sigma, h) \restriction_{\theta \cup \mu(A)}$, $(\chi, \gamma'') \in \mathbf{AVal}(\Sigma'; \cdot; \cdot \vdash h' : \mathbf{heap})$
- (PQF) $(K[fV], i, T, \Sigma, h, \gamma, \phi, \xi, \theta) \xrightarrow{(\overline{f}(A), \Sigma', \chi)} (\Sigma, h, \gamma \uplus \gamma' \uplus \gamma'', \phi \uplus \nu(A) \uplus \nu(\chi), \xi : (\Sigma \vdash K : (\cdot; \cdot; T_2; i_3) \to (\cdot; \cdot; T; i)), \mathrm{dom}(\chi))$
  where $f \in \mathrm{FNames}_{T_1 \to T_2, i_1}$, $(A, \gamma') \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_{i_2} V : T_1)$, $(\Sigma', h') = (\Sigma, h) \restriction_{\theta \cup \mu(A)}$, $(\chi, \gamma'') \in \mathbf{AVal}(\Sigma'; \cdot; \cdot \vdash h' : \mathbf{heap})$
- (PQB) $(K[\#b^\delta], i, T, \Sigma, h, \gamma, \phi, \xi, \theta) \xrightarrow{(\overline{\mathbf{run}\ b^\rho}, \Sigma', \chi)} (\Sigma, h, \gamma \uplus \gamma' \uplus \gamma'', \phi \uplus \nu(\chi), (\Sigma \vdash K : (\cdot; \cdot; T_1; i_1) \to (\cdot; \cdot; T; i)) : \xi, \mathrm{dom}(\chi))$
  where $b \in \mathrm{BNames}_{\square(\Gamma \vdash T_1), 1}$, $(\rho, \gamma') \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_{i_1} \delta : \Gamma)$, $(\Sigma', h') = (\Sigma, h) \restriction_{\theta \cup \mu(\rho)}$, $(\chi, \gamma'') \in \mathbf{AVal}(\Sigma'; \cdot; \cdot \vdash h' : \mathbf{heap})$
- (OA) $(\Sigma, h, \gamma, \phi, \xi : (\Sigma'' \vdash K : (\cdot; \cdot; T_1; i_1) \to (\cdot; \cdot; T_2; i_2)), \theta) \xrightarrow{(A, \Sigma', \chi)} (K[A], i_2, T_2, \Sigma \cup \Sigma', \chi + h, \gamma, \phi \uplus \nu(A) \uplus \nu(\chi), \xi, \mathrm{dom}(\chi))$
  where $\Sigma'; \cdot; \cdot \vdash_{i_1} A : T_1$, $\mathrm{dom}(\chi) \cap \mathrm{dom}(h) = \theta$, $\chi = \chi \restriction_{\theta \cup \mu(A)}$
- (OQF) $(\Sigma, h, \gamma, \phi, \xi, \theta) \xrightarrow{(f(A), \Sigma', \chi)} (V\,A, i_3, T_2, \Sigma \cup \Sigma', \chi + h, \gamma, \phi \uplus \nu(A) \uplus \nu(\chi), \xi, \mathrm{dom}(\chi))$
  where $f \in \mathrm{FNames}_{T_1 \to T_2, i_1}$, $\gamma(f) = V$, $\Sigma'; \cdot; \cdot \vdash_{i_2} A : T_1$, $i_3 = \max(i_1, i_2)$, $\mathrm{dom}(\chi) \cap \mathrm{dom}(h) = \theta$, $\chi = \chi \restriction_{\theta \cup \mu(A)}$
- (OQB) $(\Sigma, h, \gamma, \phi, \xi, \theta) \xrightarrow{(\mathbf{run}\ b^\rho, \Sigma', \chi)} ((\#V)[\rho], i, T, \Sigma \cup \Sigma', \chi + h, \gamma, \phi \uplus \nu(\chi), \xi, \mathrm{dom}(\chi))$
  where $b \in \mathrm{BNames}_{\square(\Gamma \vdash T),}$, $\gamma(b) = V$, $\Sigma'; \cdot; \cdot \vdash_i \rho : \Gamma$, $\mathrm{dom}(\chi) \cap \mathrm{dom}(h) = \theta$, $\chi = \chi \restriction_{\theta \cup \mu(A)}$

We use $\uplus$ to denote disjoint union.

Given two heaps h and h', we define $h + h'$ to be a heap with $\mathrm{dom}(h + h') = \mathrm{dom}(h) \cup \mathrm{dom}(h')$ such that $(h + h')(\ell) = h(\ell)$ if $\ell \in \mathrm{dom}(h)$, and $(h + h')(\ell) = h'(\ell)$ otherwise.

Given a set of locations θ and a heap $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$, We define $(\Sigma, h) \restriction_\theta = (\Sigma', h')$ where h' is the minimal heap such that (a) $h' \subseteq h$, (b) $\theta \subseteq \mathrm{dom}(h')$, and (c) $h'(\ell) = \ell'$ implies $\ell' \in \mathrm{dom}(h')$, and $\Sigma' \subseteq \Sigma$ is the corresponding store context such that $\Sigma'; \cdot; \cdot \vdash h' : \mathbf{heap}$. When store contexts are clear from the context, we sometimes simply write $h \restriction_\theta = h'$.

Figure 6 Transition rules for Program LTS.

running unknown pieces of code. Note that a local substitution δ is always provided for all its free variables, and this substitution is also converted into an abstract representation ρ. Such a question is represented by the action $(\overline{\mathbf{run}\ b^\rho}, \Sigma, \chi)$.

The above (PA), (PQF), and (PQB) actions lead from active to passive configurations. Below we discuss transitions from passive to active configurations, which correspond to actions by the context. In (OA) the context provides a value to a pending player question, which is represented by the action (A, Σ, χ). The stack ξ is popped and A is plugged into the top evaluation context. The $\mathrm{dom}(\chi) \cap \mathrm{dom}(h) = \theta$ condition ensures that the context cannot modify the program's private locations, while $\chi = \chi \restriction_{\sigma \cup \mu(A)}$ means that the context should only share relevant locations. (OQF) describes a situation where the context calls a function that was communicated to it previously as f, and applies it to an arbitrary abstract argument A, as illustrated by the action $(f(A), \Sigma, \chi)$. The transition fetches the

corresponding actual value $\gamma(f)$, which is then scheduled for execution in the successor active configuration. Finally, (OQB) covers the case when the context wants to run code b under some substitution ρ. This rule is analogous to the previous one and the action is $(\mathbf{run}\ b^{\rho}, \Sigma, \chi)$.

4.5 Trace semantics

To start defining the trace semantics, we first need to define the initial configuration induced by an arbitrary term M. To handle free global variables in M, we define abstraction over global substitutions as $\mathbf{AVal}(\Sigma; \cdot \vdash \sigma : \Psi) \triangleq \{(\eta, \bigcup_u \gamma_u) \mid \mathrm{dom}(\eta) = \mathrm{dom}(\sigma), \eta(u) = \#(\eta'(u)), (\eta'(u), \gamma_u) \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_1 \mathbf{box}\ \sigma(u) : \square(\Gamma \vdash T))$ for all $u : (\Gamma \vdash T) \in \Psi, \cap_{u \in \mathrm{dom}(\eta)} \nu(\eta(u)) = \emptyset\}$. We use η to refer to an *abstract global substitution* that maps global variables to $\#b$ of the corresponding type.

Reminiscent of Theorem 9, the initial configurations are parameterized by the term M, an abstract global substitution η, an abstract local substitution ρ, and an abstract heap χ.

▶ **Definition 11** (Initial configurations). *Given a term $\Sigma; \Psi; \Gamma \vdash_i M : T$, abstract global substitution $\Sigma'; \cdot \vdash \eta : \Psi$, abstract local substitution $\Sigma'; \cdot; \cdot \vdash_i \rho : \Gamma$, and abstract heap $\Sigma'; \cdot; \cdot \vdash \chi : \mathbf{heap}$ such that $\Sigma' \supseteq \Sigma$, the program configuration is defined as $\mathsf{C}_M^{\eta,\rho,\chi} = (M[\eta][\rho], i, T, \Sigma', \chi, \emptyset, \nu(\eta) \uplus \nu(\rho) \uplus \nu(\chi), \cdot, \mathrm{dom}(\chi))$.*

The semantics will only contain traces that empty the stack, as these represent the convergent interactions. Below we define the set of such traces $\mathrm{Tr}(\mathbf{C})$ that are generated from $\mathbf{C}$.

▶ **Definition 12** (Derived traces). *Given two configurations $\mathbf{C}$, $\mathbf{C}'$ and action $\mathbf{a}$, we write $\mathbf{C} \overset{\mathbf{a}}{\Longrightarrow} \mathbf{C}'$ if $\mathbf{C} \overset{\tau}{\rightarrow}{}^{*}\ \mathbf{C}'' \overset{\mathbf{a}}{\rightarrow} \mathbf{C}'$. Given a trace $\mathtt{t} = \mathbf{a}_1 \ldots \mathbf{a}_n$, we write $\mathbf{C} \overset{\mathtt{t}}{\Longrightarrow} \mathbf{C}'$ if $\mathbf{C} \overset{\mathbf{a}_1}{\Longrightarrow} \mathbf{C}'' \overset{\mathbf{a}_2}{\Longrightarrow} \cdots \overset{\mathbf{a}_n}{\Longrightarrow} \mathbf{C}'$. We write $\mathrm{Tr}(\mathbf{C}) = \{\mathtt{t} \mid \mathbf{C} \overset{\mathtt{t}}{\Longrightarrow} (\Sigma, h, \gamma, \phi, \cdot, \theta)\}$. Equivalently, $\mathtt{t} \in \mathrm{Tr}(\mathbf{C})$ iff $\mathbf{C} \overset{\mathtt{t}}{\Longrightarrow} \mathbf{C}'$ for some $\mathbf{C}'$ and the number of (OA) actions minus the number of (PQ) actions in $\mathtt{t}$ equals the size of ξ in $\mathbf{C}$.*

Finally, we define the intended trace semantics of Contextual MetaML terms.

▶ **Definition 13** (Trace semantics). *Given a term $\Sigma; \Psi; \Gamma \vdash_i M : T$, we define its trace semantics as*

$$\mathrm{Tr}(M) = \{(\eta, \rho, \chi, \mathtt{t}) \mid \eta, \rho, \chi \text{ are as specified in Definition 11,and } \mathtt{t} \in \mathrm{Tr}(\mathsf{C}_M^{\eta,\rho,\chi})\}$$

In the next section, this semantics will be shown fully abstract with respect to $\lesssim^{\mathrm{CIU}}$, and hence to $\lesssim$ by Theorem 9.

▶ **Example 14.** In this example, we consider some terms and the traces that they can generate under our LTS. We omit the typing context Σ and heap χ in the traces where possible for brevity.

- Term: $\cdot; \cdot; \cdot \vdash_1 \mathbf{box}\ x : \square(x : \mathbf{Int} \vdash \mathbf{Int})$
  Trace: $\overline{b_1}\quad \mathbf{run}\ b_1^{1/x}\quad \overline{1}$
- Term: $\cdot; \cdot; y : \square(x : \mathbf{Int} \vdash \mathbf{Int}) \to \mathbf{Int} \vdash_1 y\ \mathbf{box}\ x : \mathbf{Int}$
  Substitution: $y \mapsto f_1$
  Trace: $\overline{f_1(b_1)}\quad \mathbf{run}\ b_1^{1/x}\quad \overline{1}\quad 2\quad \overline{2}$
- Term: $\cdot; u : (x : \mathbf{Int} \vdash \mathbf{Int}); y : \mathbf{Int} \vdash_0 u^{y/x} + 3 : \mathbf{Int}$
  Substitution: $u \mapsto \#b_1,\ y \mapsto 1$
  Trace: $\overline{\mathbf{run}\ b_1^{1/x}}\quad 5\quad \overline{8}$

- Term: $\ell : \mathbf{ref}\ \mathbf{Int}; u : (\vdash \mathbf{Unit}); \cdot \vdash_0 u; \ell \mapsto !\ell + 2; u : \mathbf{Unit}$
  Substitution: $u \mapsto \#b_1$
  Initial heap: $\ell \mapsto 0$
  Trace: $(\overline{\mathbf{run}\ b_1}, \{\ell \mapsto 0\})$ $((), \{\ell \mapsto 1\})$ $(\overline{\mathbf{run}\ b_1}, \{\ell \mapsto 3\})$ $((), \{\ell \mapsto 4\})$ $(\overline{()}, \{\ell \mapsto 4\})$

5 Full Abstraction

In order to prove full abstraction we need to develop terminology and results about the interaction between active and passive configurations. In Definition 11, we showed how to assign *active* configurations to terms. Below we define *passive* configurations corresponding to the evaluation context K, substitutions σ, δ, and heap h as given in Definition 6.

▶ **Definition 15.** *Given an evaluation context* $\Sigma \vdash K : (\cdot; \cdot; T_1; i_1) \to (\cdot; \cdot; T_2; i_2)$*, global substitution* $\Sigma; \cdot \vdash \sigma : \Psi$*, local substitution* $\Sigma; \cdot; \cdot \vdash_i \delta : \Gamma$*, and heap* $\Sigma; \cdot; \cdot \vdash h : \mathbf{heap}$*, let* $(\eta, \gamma) \in \mathbf{AVal}(\Sigma; \cdot \vdash \sigma : \Psi)$*,* $(\rho, \gamma') \in \mathbf{AVal}(\Sigma; \cdot; \cdot \vdash_i \delta : \Gamma)$*,* $(\Sigma', h') = (\Sigma, h) \restriction_{\mu(\eta) \cup \mu(\rho)}$*, and* $(\chi, \gamma'') \in \mathbf{AVal}(\Sigma'; \cdot; \cdot \vdash h' : \mathbf{heap})$*, the context configuration is defined as* $\mathsf{C}^{\eta,\gamma,\rho,\gamma',\chi,\gamma''}_{K,\sigma,\delta,h} = (\Sigma, h, \gamma \uplus \gamma' \uplus \gamma'', \nu(\eta) \uplus \nu(\rho) \uplus \nu(\chi), \cdot : (\Sigma \vdash K : (\cdot; \cdot; T_1; i_1) \to (\cdot; \cdot; T_2; i_2)), \mathrm{dom}(\chi))$.

Given a sequence $\mathtt{t}$ of actions, we write $\overline{\mathtt{t}}$ for the sequence obtained by converting each action to one of the opposite polarity (i.e. context to program, program to context). We write $\mathbf{C}_1 | \mathbf{C}_2 \downarrow^{\mathtt{t}}$ if either (a) $\mathtt{t} \in \mathrm{Tr}(\mathbf{C}_1)$ and $\overline{\mathtt{t}}\ \mathbf{a} \in \mathbf{Tr}(\mathbf{C}_2)$, or (b) $\mathtt{t}\ \mathbf{a} \in \mathrm{Tr}(\mathbf{C}_1)$ and $\overline{\mathtt{t}} \in \mathbf{Tr}(\mathbf{C}_2)$ for some (PA) action $\mathbf{a}$.

To show soundness, we need to show that a program configuration and a context configuration share a trace iff the term obtained by combining them terminates.

▶ **Theorem 16** (Correctness). *For any term* $\Sigma; \Psi; \Gamma \vdash_i M : T$*, evaluation context* $\Sigma' \vdash K : (\cdot; \cdot; i; T) \to (\cdot; \cdot; i'; T')$*, global substitution* $\Sigma'; \cdot \vdash \sigma : \Psi$*, local substitution* $\Sigma'; \cdot; \cdot \vdash_i \delta : \Gamma$*, and heap* $\Sigma'; \cdot; \cdot \vdash h : \mathbf{heap}$ *such that* $\Sigma' \supseteq \Sigma$*, let* $(\eta, \gamma) \in \mathbf{AVal}(\Sigma'; \cdot \vdash \sigma : \Psi)$*,* $(\rho, \gamma') \in \mathbf{AVal}(\Sigma'; \cdot; \cdot \vdash_i \delta : \Gamma)$*,* $(\Sigma'', h') = (\Sigma', h) \restriction_{\mu(\eta) \cup \mu(\rho)}$*, and* $(\chi, \gamma'') \in \mathbf{AVal}(\Sigma''; \cdot; \cdot \vdash h' : \mathbf{heap})$*, then* $(K[M[\sigma][\delta]], h) \Downarrow$ *iff there exists trace* $\mathtt{t}$ *such that* $\mathsf{C}^{\eta,\gamma,\rho,\gamma',\chi,\gamma''}_{K,\sigma,\delta,h} \mid \mathsf{C}^{\eta,\rho,\chi}_{M} \downarrow^{\mathtt{t}}$.

Proof Sketch. We generalize the statement and prove that for any pair of "compatible" configurations, they share a trace iff their "combination" terminates. In particular, we show that such pairs cannot generate an infinite τ-free trace by giving an upper bound on the length of such traces. ◀

▶ **Example 17.** Here we revisit Example 14 and show the context configurations that correspond to those terms and traces.

- Term: $\cdot; \cdot; \cdot \vdash_1 \mathbf{box}\ x : \square(x : \mathbf{Int} \vdash \mathbf{Int})$
  Evaluation context: $\mathbf{letbox}\ u \leftarrow \bullet\ \mathbf{in}\ (u^{1/x} + 1)$
  Trace: $b_1 \quad \overline{\mathbf{run}\ b_1^{1/x}} \quad 1 \quad \overline{2}$
- Term: $\cdot; \cdot; y : \square(x : \mathbf{Int} \vdash \mathbf{Int}) \to \mathbf{Int} \vdash_1 y\ \mathbf{box}\ x : \mathbf{Int}$
  Evaluation context: $\bullet + 2$
  Substitution: $y \mapsto \lambda z.\mathbf{letbox}\ u \leftarrow z\ \mathbf{in}\ (u^{1/x} + 1)$
  Trace: $f_1(b_1) \quad \overline{\mathbf{run}\ b_1^{1/x}} \quad 1 \quad \overline{2} \quad 2 \quad \overline{4}$
- Term: $\cdot; u : (x : \mathbf{Int} \vdash \mathbf{Int}); y : \mathbf{Int} \vdash_0 u^{y/x} + 3 : \mathbf{Int}$
  Evaluation context: $\bullet + 2$
  Substitution: $u \mapsto x + 4$, $y \mapsto 1$
  Trace: $\mathbf{run}\ b_1^{1/x} \quad \overline{5} \quad 8 \quad \overline{10}$

- Term: $\ell : \mathbf{ref}\ \mathbf{Int}; u : (\vdash \mathbf{Unit}); \cdot \vdash_0 u; \ell \mapsto !\ell + 2; u : \mathbf{Unit}$
  Evaluation context: $\bullet$
  Substitution: $u \mapsto \ell := !\ell + 1$
  Initial heap: $\ell \mapsto 0$
  Trace: $(\mathbf{run}\ b_1, \{\ell \mapsto 0\})$ $(\overline{()}, \{\ell \mapsto 1\})$ $(\mathbf{run}\ b_1, \{\ell \mapsto 3\})$ $(\overline{()}, \{\ell \mapsto 4\})$ $((), \{\ell \mapsto 4\})$ $(\overline{()}, \{\ell \mapsto 4\})$

▶ **Theorem 18** (Soundness). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*, if* $\mathrm{Tr}(M_1) \subseteq \mathrm{Tr}(M_2)$*, then* $M_1 \lesssim M_2$.

To show completeness, we prove a definability property stating that, for any well-behaved sequence of actions, we can find a configuration that generates this trace only.

We say that a name is *introduced* in an action if it is used as an answer, as the argument in a (QF) action, as the codomain of the substitution in a (QB) action, or appears in χ. We say that a (PA) (resp. (OA)) action *is an answer to* an (OQ) (resp. (PQ)) action if it is the first (PA) (resp. (OA)) action such that the number of (PA) (resp. (OA)) actions between them equals the number of (OQ) (resp. (PQ)) actions between them.

Given $N_O, N_P \subseteq \mathrm{Names}$, an N_O, N_P*-trace* is a sequence of actions such that (i) Player and Opponent actions alternate; (ii) no name in $N_P \cup N_O$ is introduced; (iii) no name is introduced twice; (iv) if a name is used as a question, then it must either occur in the set $N_{\_}$ of opposite polarity or have been introduced in a previous action of opposite polarity; (v) an answer to a question must have the same type as the question is expecting; and (vi) if χ and χ' belong to two neighbouring actions $\mathbf{a}$ and $\mathbf{a}'$, then $\mathrm{dom}(\chi')$ should be the minimal set of locations satisfying the following three conditions: (a) $\mathrm{dom}(\chi) \subseteq \mathrm{dom}(\chi')$, (b) locations occurring in $\mathbf{a}'$ (if any) occur in $\mathrm{dom}(\chi')$, and (c) $\chi'(\ell) = \ell'$ implies $\ell' \in \mathrm{dom}(\chi')$.

We say an $\emptyset, N_P$-trace is *complete* if (i) it starts with an Opponent action, (ii) it ends with a (PA) action, (iii) the number of (OA) actions equals the number of (PQ) actions plus one, (iv) in any prefix of it the number of (OA) actions is at most the number of (PQ) actions plus one, and (v) in any proper prefix of it the number of (PA) actions is at most the number of (OQ) actions. Then the traces in $\mathrm{Tr}(\mathbf{C})$ are complete iff $\mathbf{C}$ is a passive configuration whose stack ξ contains exactly one evaluation context. In particular, if $\mathtt{t} \in \mathrm{Tr}(\mathsf{C}^{\eta,\gamma,\rho,\gamma',\chi,\gamma''}_{K,\sigma,\delta,h})$, then $\mathtt{t}$ must be a complete $\emptyset, \mathrm{dom}(\gamma) \uplus \mathrm{dom}(\gamma') \uplus \mathrm{dom}(\gamma'')$-trace.

▶ **Lemma 19** (Definability). *Given a complete* $\emptyset, N_P$*-trace* $\mathtt{t}$*, there exists* $\mathbf{C} = (\Sigma, h, \gamma, N_P, \cdot : (\Sigma \vdash K : (\cdot; \cdot; i_1; T_1) \to (\cdot; \cdot; i_2; T_2)), \theta)$ *such that* $\mathrm{Tr}(\mathbf{C}) = \{\mathtt{t}\}$ *up to renamings that preserve* N_P.

Proof Sketch. Suppose $\mathtt{t} = \mathbf{a}_1 \dots \mathbf{a}_n$, and let $\mathtt{t}_i = \mathbf{a}_i \dots \mathbf{a}_n$. We specify the configurations $\mathbf{C}_i$ such that $\mathrm{Tr}(\mathbf{C}_i) = \{\mathtt{t}_i\}$ up to renaming by backward induction on i. We create references for all the names, and set the references appropriately at each step so that only the designated actions can be taken. For P actions, M_i should set the locations according to χ_i and generate the corresponding player action. For O actions, the popped evaluation context should assert that the action is received as expected, assert that the shared locations are updated as expected, set the stored values to be the same as h_{i+1}, and return M_{i+1}. ◀

▶ **Theorem 20** (Completeness). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*, if* $M_1 \lesssim M_2$*, then* $\mathrm{Tr}(M_1) \subseteq \mathrm{Tr}(M_2)$.

▶ **Theorem 21** (Full Abstraction). *Given two terms* $\Sigma; \Psi; \Gamma \vdash_i M_1, M_2 : T$*,* $M_1 \lesssim M_2$ *iff* $\mathrm{Tr}(M_1) \subseteq \mathrm{Tr}(M_2)$.

If we consider the fragment of Contextual MetaML that does not contain global variables, box terms, or box types, the resulting language is essentially simply-typed ML with higher-order references, which we shall call HOS [8, 17] here. For an HOS term, its trace semantics does not involve BNames or (QB) rules, and coincides with its trace semantics in HOS as given by Laird [25]. Therefore, we have the following result.

▶ **Theorem 22** (Conservative extension). *Contextual MetaML is a conservative extension of HOS with respect to contextual equivalence. In other words, two HOS terms are equivalent in HOS contexts iff they are equivalent in Contextual MetaML contexts.*

6 Examples

▶ **Example 23** (Switching letbox order). Let typing context Γ_1 be

$$x : \Box(\cdot \vdash \mathbf{Int}), y : \Box(\cdot \vdash \mathbf{Int})$$

and terms M_1, M_2 be

$$\Gamma_1 \vdash_1 \mathbf{letbox}\ u \leftarrow x\ \mathbf{in}\ \mathbf{letbox}\ v \leftarrow y\ \mathbf{in}\ u; v : \mathbf{Int}$$

$$\Gamma_1 \vdash_1 \mathbf{letbox}\ v \leftarrow y\ \mathbf{in}\ \mathbf{letbox}\ u \leftarrow x\ \mathbf{in}\ u; v : \mathbf{Int}$$

respectively. Let $\rho = b_1/x, b_2/y$, $\eta = \cdot$, and $\chi = \cdot$, then

$$\begin{aligned} \mathbf{C}^{\eta,\rho,\chi}_{M_1} = &\ (\mathbf{letbox}\ u \leftarrow b_1\ \mathbf{in}\ \mathbf{letbox}\ v \leftarrow b_2\ \mathbf{in}\ u; v, 1, \mathbf{Unit}, \cdots) \\ \xrightarrow{\tau} &\ (\mathbf{letbox}\ v \leftarrow b_2\ \mathbf{in}\ \#b_1; v, 1, \mathbf{Unit}, \cdots) \\ \xrightarrow{\tau} &\ (\#b_1; \#b_2, 1, \mathbf{Unit}, \cdots) \end{aligned}$$

Similarly, $\mathbf{C}^{\eta,\rho,\chi}_{M_2} \xrightarrow{\tau}{}^{*} (\#b_1; \#b_2, 1, \mathbf{Unit}, \cdots)$. So $\mathrm{Tr}(M_1) = \mathrm{Tr}(M_2)$, and therefore, $M_1 \approx M_2$.

The above equivalence holds because of the call-by-name nature of the Letbox operation. This is not to be confused with the fact that the argument of letbox is still supplied in a call-by-value way. For example, let typing context Γ_2 be

$$x : \mathbf{Unit} \to \Box(\cdot \vdash \mathbf{Int}), y : \mathbf{Unit} \to \Box(\cdot \vdash \mathbf{Int})$$

and terms M_3, M_4 be

$$\Gamma_2 \vdash_1 \mathbf{letbox}\ u \leftarrow x()\ \mathbf{in}\ \mathbf{letbox}\ v \leftarrow y()\ \mathbf{in}\ u; v : \mathbf{Int}$$

$$\Gamma_2 \vdash_1 \mathbf{letbox}\ v \leftarrow y()\ \mathbf{in}\ \mathbf{letbox}\ u \leftarrow x()\ \mathbf{in}\ u; v : \mathbf{Int}$$

respectively. Let $\rho = f_1/x, f_2/y$, $\eta = \cdot$, and $\chi = \cdot$. Then $\mathbf{C}^{\eta,\rho,\chi}_{M_3} \xrightarrow{(\overline{f_1()},\cdot,\cdot)} \cdots$, and $\mathbf{C}^{\eta,\rho,\chi}_{M_4} \xrightarrow{(\overline{f_2()},\cdot,\cdot)} \cdots$. So M_3 and M_4 are incomparable.

▶ **Example 24** (Code duplication). Let terms M_1, M_2, M_3, M_4 be

$$x : \mathbf{Unit} \to \mathbf{Unit} \vdash_1 \mathbf{letbox}\ u \leftarrow \mathbf{box}\ x()\ \mathbf{in}\ u^{x/x}; u^{x/x}$$
$$x : \mathbf{Unit} \to \mathbf{Unit} \vdash_1 \mathbf{letbox}\ u \leftarrow \mathbf{box}\ z()\ \mathbf{in}\ u^{x/z}; u^{x/z}$$
$$x : \mathbf{Unit} \to \mathbf{Unit} \vdash_1 x(); x()$$
$$x : \mathbf{Unit} \to \mathbf{Unit} \vdash_1 \mathbf{let}\ y \leftarrow x()\ \mathbf{in}\ y; y$$

respectively. Then, intuitively, both M_1, M_2 and M_3 will call x twice, while M_4 will only call x once. Let $\rho = f_1/x$ and $\chi = \ell_1 \mapsto 0$, then both M_1, M_2, and M_3 (but not M_4) have the following trace (we omit Σ in the action for brevity):

$$(\overline{f_1}(), \ell_1 \mapsto 0)\ ((), \ell_1 \mapsto 1)\ (\overline{f_1}(), \ell_1 \mapsto 1)\ ((), \ell_1 \mapsto 3)\ (\overline{()}, \ell_1 \mapsto 3)$$

On the other hand, M_4 (but not M_1, M_2 or M_3) has the trace

$$(\overline{f_1}(), \ell_1 \mapsto 0)\ ((), \ell_1 \mapsto 1)\ (\overline{()}, \ell_1 \mapsto 1)$$

Therefore, $M_1 \approx M_2 \approx M_3 \not\approx M_4$.

When $x()$ is replaced with some long effectful program, **letbox** allows type-safe code duplication. This is one of the common usages of metaprogramming.

▶ **Example 25** (Box binds local variables). Let M_1, M_2, M_3, M_4 be

$$\vdash_1 \lambda x.\lambda y.\mathbf{box}\ x : \mathbf{Int} \to \mathbf{Int} \to \Box(x : \mathbf{Int}, y : \mathbf{Int} \vdash \mathbf{Int})$$
$$\vdash_1 \lambda y.\lambda x.\mathbf{box}\ x : \mathbf{Int} \to \mathbf{Int} \to \Box(x : \mathbf{Int}, y : \mathbf{Int} \vdash \mathbf{Int})$$
$$\vdash_1 \lambda z_1.\lambda z_2.\mathbf{box}\ x : \mathbf{Int} \to \mathbf{Int} \to \Box(x : \mathbf{Int}, y : \mathbf{Int} \vdash \mathbf{Int})$$
$$\vdash_1 \lambda x.\lambda y.\mathbf{box}\ y : \mathbf{Int} \to \mathbf{Int} \to \Box(x : \mathbf{Int}, y : \mathbf{Int} \vdash \mathbf{Int})$$

respectively. Then $M_1 \approx M_2 \approx M_3$, and they are incomparable to M_4. This is because the free variables inside **box** are bound by **box** and revealed to the context via the type of the term. They are not bound by the preceding lambda abstractions, and will not be α-converted when we rename those lambda abstractions. For example, the trace

$$(\overline{f_1}, \cdot, \cdot)\ (f_1(1), \cdot, \cdot)\ (\overline{f_2}, \cdot, \cdot)\ (f_2(2), \cdot, \cdot)\ (\overline{b_1}, \cdot, \cdot)\ (\mathbf{run}\ b_1^{3/x,4/y}, \cdot, \cdot)\ (\overline{3}, \cdot, \cdot)$$

belongs to $\mathrm{Tr}(M_1)$, $\mathrm{Tr}(M_2)$, and $\mathrm{Tr}(M_3)$, while the trace

$$(\overline{f_1}, \cdot, \cdot)\ (f_1(1), \cdot, \cdot)\ (\overline{f_2}, \cdot, \cdot)\ (f_2(2), \cdot, \cdot)\ (\overline{b_1}, \cdot, \cdot)\ (\mathbf{run}\ b_1^{3/x,4/y}, \cdot, \cdot)\ (\overline{4}, \cdot, \cdot)$$

belongs to $\mathrm{Tr}(M_4)$.

Below we use the imperative power function given by Calcagno et al. [6] to demonstrate our model's ability to reason about programs with effectful types. Here the transformed term is not equivalent to the original term as computation related to n has been brought forward. However, the η-expansion of the transformed term is equivalent to the original term, so the transformation is faithful in the sense that it never gives different results.

▶ **Example 26** (Correctness of staging the imperative power function). Let *power* be the following term:

$$\begin{aligned}
&\vdash_1 \mathbf{rec}\ f(n).\lambda xy.\mathbf{if}\ n = 0\ \mathbf{then}\ y := 1 \\
&\qquad\qquad \mathbf{else}\ (f\ (n-1)\ x\ y; y :=\!!y * x) \\
&: \mathbf{Int} \to \mathbf{Int} \to \mathbf{ref}\ \mathbf{Int} \to \mathbf{Unit}
\end{aligned}$$

Let *power_staged_gen* be the following term:

$$\begin{aligned}
&\vdash_1 \mathbf{rec}\ f(n).\lambda xy.\mathbf{if}\ n = 0\ \mathbf{then}\ \mathbf{letbox}\ u \leftarrow y\ \mathbf{in}\ \mathbf{box}\ (u^{y/y} := 1) \\
&\qquad\qquad \mathbf{else}\ \mathbf{letbox}\ u \leftarrow f\ (n-1)\ x\ y\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{letbox}\ v \leftarrow x\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{letbox}\ w \leftarrow y\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{box}\ (u^{x/x,y/y}; w^{y/y} :=\!!w^{y/y} * v^{x/x}) \\
&: \mathbf{Int} \to \Box(x : \mathbf{Int} \vdash \mathbf{Int}) \to \Box(y : \mathbf{ref}\ \mathbf{Int} \vdash \mathbf{ref}\ \mathbf{Int}) \\
&\quad \to \Box(x : \mathbf{Int}, y : \mathbf{ref}\ \mathbf{Int} \vdash \mathbf{Unit})
\end{aligned}$$

Let *power_staged* be the following term:

$$\vdash_1 \lambda n.\mathbf{letbox}\ u \leftarrow \mathit{power\_staged\_gen}\ n\ (\mathbf{box}\ x)\ (\mathbf{box}\ y)\ \mathbf{in}\ \lambda xy.u^{x/x,y/y}$$
$$: \mathbf{Int} \to \mathbf{Int} \to \mathbf{ref}\ \mathbf{Int} \to \mathbf{Unit}$$

Then $\mathit{power\_staged} \lessapprox \mathit{power} \approx \lambda nxy.\mathit{power\_staged}\ n\ x\ y$.

7 Related Work

Type systems for imperative MetaML. Purely functional MetaML as proposed by Taha and Sheard [36] was one of the first safe multi-stage programming languages with explicit annotations. Since then, many type systems for imperative MetaML have been developed, with a primary focus on designing a safe type system for higher-order references of code. Notable examples include closed types [6], refined environment classifiers [24, 16], Mint [39], and $\lambda^{\oslash}$ [19, 20]. These type systems view scope extrusion as undesirable, and prevent a piece of open code such as $\langle x \rangle$ (or, in our term, **box** x) from being stored in a reference cell at all. Another line of work such as MetaOCaml [22, 21] and MacoCaml [41, 7] gives up static type safety and relies on a dynamic scope extrusion detection when dereferencing a cell at runtime. In the former approach, open code cannot be stored. In the latter approach, open code can be stored but cannot be used. However, as we have shown in this paper, there are situations where the ability to store and use open code is useful. And scope extrusion does not necessarily lead to type unsafety, as long as the type system reasons about free variables in code values properly.

Contextual type theory. Early contextual modal type theories [29, 30, 31] were originally developed for recursively analyzing open terms via higher-order pattern matching. In those systems, there is a hard division between the meta language that analyzes and object language that is analyzed. Therefore, those type systems do not support running code, which is essentially promoting the object language into the meta language. More recently, Mœbius [18] is the first so-called homogeneous contextual type theory where there is no distinction between the meta language and the object language, and it supports running code. However, its operational semantics is not type safe as there is no guarantee of progress, i.e. a well-typed program might get stuck. Layered modal type theory [12] achieves type safety by restricting the depth of nested code, and is most similar to our work. There has been an extension of the layered approach to dependent types [13], but this is orthogonal to our concern. All of the above systems are purely functional; in contrast, our contribution is the design of such a (layered modal type) system for higher-order references of arbitrary open code. We drop the ability to pattern-match on code because this would make full abstraction trivial. Any syntactically different code (e.g. **box** $(1+1)$ and **box** 2) could be distinguished by pattern matching, and there would be no non-trivial equivalences between pieces of code.

Full abstraction for MetaML. Taha [35] presented a few equational rules for purely functional MetaML, which are further extended by Inoue and Taha [15]. Both equational systems are sound for contextual equivalence but not complete. Berger and Tratt [3] gave a Hoare-style program logic for typed purely functional MetaML that is fully abstract. Inoue

and Taha [15][3] gave a fully abstract characterization of contextual equivalence for untyped functional MetaML using the notion of applicative bisimulation [2, 9]. We are the first to give a fully abstract model for imperative MetaML.

Operational game semantics. Game semantics has been a powerful tool to obtain fully abstract models for a variety of programming languages. Many such models are categorical constructions [1, 14, 32, 27] in the sense that programs are interpreted denotationally as morphisms (typically strategies) and types as objects (typically games) of a suitable category. In contrast, the more recent operational game semantics [25, 17, 26, 5] provide a more intuitive operationally-driven framework. Our trace model can be thought of as an extension of Laird's model for higher-order references [25].

8 Future Work

Contextual MetaML only has two layers. Although this already covers most practical use cases of generative metaprogramming, there are scenarios where more than two layers are needed. Hu and Pientka [11] presented a type-safe layered modal type theory generalized to n layers. This generalization requires more work than just letting the i in the typing judgment range over natural numbers. For example, the distinction between local and global variables should be generalized so that all variables x^k carry an annotation $0 \leq k < n$ denoting the layer it is from. And the contextual types should be of the form $\Box_k(\Gamma \vdash_j T)$ that reads "a term from layer k that describes a code from layer j" for $0 \leq j < k < n$. We believe that if we generalize our system to n layers in this way, we can retain the type safety proof by following Hu and Pientka [11]. However, adapting the CIU and full abstraction proofs will require some more work. In CIU, for example, we need to define the eligible closing substitutions for a free variable x^k for any k.

References

1 Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full Abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. doi:10.1006/INCO.2000.2930.

2 Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Inf. Comput.*, 105(2):159–267, 1993. doi:10.1006/INCO.1993.1044.

3 Martin Berger and Laurence Tratt. Program Logics for Homogeneous Generative Run-Time Meta-Programming. *Log. Methods Comput. Sci.*, 11(1), 2015. doi:10.2168/LMCS-11(1:5)2015.

4 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. doi:10.4230/LIPIcs.ECOOP.2017.5.

5 Peio Borthelle, Guilhem Jaber, Tom Hirschowitz, and Yannick Zakowski. An abstract, certified account of operational game semantics. In *ESOP'25*, Hamilton, Canada, 2025.

[3] In that paper, they believed "imperative hygienic MSP (multi-stage programming) is not yet ready for an investigation like this ... The foundations for imperative hygienic MSP have not matured to the level of the functional theory that we build upon here." Our type system solves their concern.

6 Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, 2003. doi:10.1017/S0956796802004598.

7 Tsung-Ju Chiang, Jeremy Yallop, Leo White, and Ningning Xie. Staged Compilation with Module Functors. *Proc. ACM Program. Lang.*, 8(ICFP):693–727, 2024. doi:10.1145/3674649.

8 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012. doi:10.1017/S095679681200024X.

9 Andrew D. Gordon. Bisimilarity as a Theory of Functional Programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999. doi:10.1016/S0304-3975(98)00353-3.

10 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. doi:10.1006/INCO.1995.1077.

11 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, 2024. doi:10.1145/3707203.

12 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. doi:10.1007/978-3-031-57262-3_3.

13 Jason Z. S. Hu and Brigitte Pientka. A Dependent Type Theory for Meta-programming with Intensional Analysis. *Proc. ACM Program. Lang.*, 9(POPL):15:416–15:445, 2025. doi:10.1145/3704851.

14 J. M. E. Hyland and C.-H. Luke Ong. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. doi:10.1006/INCO.2000.2917.

15 Jun Inoue and Walid Taha. Reasoning about multi-stage programs. *Journal of Functional Programming*, 26:e22, 2016. doi:10.1017/S0956796816000253.

16 Kanaru Isoda, Ayato Yokoyama, and Yukiyoshi Kameyama. Type-Safe Code Generation with Algebraic Effects and Handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '24, pages 53–65, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3689484.3690731.

17 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. doi:10.1007/978-3-030-72019-3_13.

18 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. doi:10.1145/3498700.

19 Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage - Staging with delimited control. *J. Funct. Program.*, 21(6):617–662, 2011. doi:10.1017/S0956796811000256.

20 Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Combinators for impure yet hygienic code generation. *Sci. Comput. Program.*, 112:120–144, 2015. doi:10.1016/J.SCICO.2015.08.007.

21 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. doi:10.1007/978-3-319-07151-0_6.

22 Oleg Kiselyov. MetaOCaml Theory and Implementation. *CoRR*, abs/2309.08207, 2023. doi:10.48550/arXiv.2309.08207.

23 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. doi:10.1007/978-981-97-2300-3_12.

24 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. doi:10.1007/978-3-319-47958-3_15.

25 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. doi:10.1007/978-3-540-73420-8_58.

26 Soren B. Lassen and Paul Blain Levy. Typed Normal Form Bisimulation. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, pages 283–297, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74915-8_23.

27 Paul Blain Levy and Sam Staton. Transition systems over games. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 64:1–64:10. ACM, 2014. doi:10.1145/2603088.2603150.

28 Robin Milner. Fully Abstract Models of Typed λ-Calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977. doi:10.1016/0304-3975(77)90053-6.

29 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008. doi:10.1145/1352582.1352591.

30 Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 371–382. ACM, 2008. doi:10.1145/1328438.1328483.

31 Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. A Type Theory for Defining Logics and Proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785683.

32 Jurriaan Rot, Bart Jacobs, and Paul Blain Levy. Steps and traces. *J. Log. Comput.*, 31(6):1482–1525, 2021. doi:10.1093/LOGCOM/EXAB050.

33 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. doi:10.1145/581690.581691.

34 Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

35 Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

36 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211–242, 2000. doi:10.1016/S0304-3975(00)00053-0.

37 Carolyn Talcott. Reasoning about Programs With Effects. *Electronic Notes in Theoretical Computer Science*, 14:301–314, 1997. doi:10.1016/S1571-0661(05)80243-9.

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

39 Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. *ACM SIGPLAN Notices*, 45(6):400–411, 2010. doi:10.1145/1809028.1806642.

40 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. doi:10.1145/3498723.

41 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, 2023. doi:10.1145/3607851.

42 Haoxuan Yin, Andrzej S. Murawski, and C. H. Luke Ong. Layered modal ML: Syntax and full abstraction, 2026. doi:10.48550/arXiv.2602.03033.