Type-safe Multi-Staged Programming

Junyoung Jang

February 10, 2022

1 Introduction

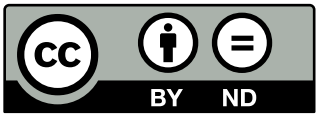
Metaprogramming is the art of writing programs, called metaprograms, that produce or manipulate other programs. This technique opens a way to reduce boilerplate code and exploit domain-specific knowledge to build high-performance programs.

This technique is widely supported in programming languages. For example, there is reflection in Java, metaclasses in Python, PPX syntax extension in OCaml, and TemplateHaskell[Sheard and Jones 2002] in Haskell. Moreover, several popular libraries and tools using these constructs show the practicality of metaprogramming. For example, in web development, the two most prevalent web frameworks in Python, the Django and Flask frameworks[JetBrains 2021] use metaclasses. A leading web framework in Haskell, the Yesod framework[Snoyman 2015] takes full advantage of TemplateHaskell to provide concise API. Metaprogramming is also wide-spread in mobile application development. For instance, the most popular dependency injection library for Android development, Google's Dagger2 library, is based on Java reflection.

However, writing and maintaining a reliable metaprogram is challenging. One reason is that testing, a usual methodology to ensure that programs work as intended, does not work well for most metaprograms. In testing, we compare the output of a program with the correct known output. We use this approach instead of the manual examination on program text because it is hard to guarantee the correctness of a program with such a manual examination. Unfortunately, for a metaprogram, the output is also a program. Therefore, it is hard to check the correctness of that output program, so is writing the tests of a metaprogram. This makes testing an unsuitable solution for metaprogramming.

If the testing is not an option for checking the reliability of metaprograms, what would be an alternative way to ensure the safety of a program? One naturally arising approach is type-checking. Types give strong guarantees on the output of a program based on its static structure. Thus, in its nature, this approach is more appropriate for metaprograms than testing.

The type-checking approach providing safety guarantees of metaprograms has been actively researched for more than three decades since Nielson and Nielson [1986]. This survey shows the development of typed metaprogramming and discusses some key challenges in the area: program-code separation, open code fragment characterization, and cross-stage dependency handling.



2 Untyped Quotation and Representation

One wide-spread approach to metaprogramming that goes back to Lisp/Scheme is using *quasiquotation*[McDermott and Sussman 1972]. It allows programmers to generate and compose code fragments. Just as we use quasiquotation to distinguish the use and mention of a linguistic expression[Quine 1940], we use quasiquotation in metaprogramming to distinguish a code fragment from a program with the same syntactic form. In this section, we will explain the main ideas of quasiquotation and metaprogramming using the functional programming language Scheme, since its design has influenced many subsequent approaches to metaprogramming.

In Scheme, we can use the keyword `quasiquote` to distinguish the code fragment `(quasiquote (+ 5 4))` from the program `(+ 5 4)`. In Figure 1, `prog` returns number `9` once executed, while `frag` returns the code fragment whose head is the symbol `+` followed by the code `5` and `4`. We can add a number to `prog` and retrieve the head of `frag`, but not the other way around.

```
1 (define prog (+ 5 4))
2 (define frag (quasiquote (+ 5 4)))
3
4 (+ prog 3) ; Adding 3 to prog. This gives number 12.
5 (car frag) ; Retrieving the head of frag. This gives symbol +
6
7 (eval frag) ; 9
```

Figure 1: Scheme Example of Quasiquotation

We can perform two kinds of operations on code fragments once we construct those. We can evaluate the fragment to execute the program that it represents. In the last line of Figure 1, we show an example of evaluating the fragment representing `(+ 5 4)` which results in `9`. We can also compose code fragments by splicing one into another. Figure 2 shows an example of code splicing using a construct **unquote**. In the example, we splice `frag1` into `frag2` and get a code fragment for `(+ 7 (* 2 2))`.

```
1 (define frag1 (quasiquote (* 2 2)))
2 (define frag2 (quasiquote (+ 7 (unquote frag1)))) ; fragment for (+ 7 (* 2 2))
3
4 (eval frag2) ; This gives number 11 as a result of (+ 7 (* 2 2))
```

Figure 2: Scheme Example of Code Fragment Splicing

Along with these constructs and the help of recursion, we can build a more practical metaprogram (Figure 3). We define a function `meta-nth`, which takes two arguments, a natural number `n` and a quoted list `xs`, and generates a code fragment to access the `n`-th element of `xs`.

If `n` is equal to or smaller than `0`, the recursion is finished, so we return a code fragment for the head (**car**) of the unquoted `xs`. Otherwise, we need to traverse the list represented by `xs` further, so we recursively call `meta-nth` with `n - 1` and the tail (**cdr**) of the unquoted `xs`. In both cases, we need to unquote `xs` because it is a variable for a quoted list, not a list value.

It is worthwhile to mention that the metaprogram `meta-nth` does some computation to generate a code fragment. To distinguish this computation from the future computation performed by a generated code fragment, we introduce the concept of *stage*. The program that runs currently

```
1  (define (meta-nth n xs)
2    (if (<= n 0)
3        (quasiquote (car (unquote xs)))
4      (meta-nth (- n 1) (quasiquote (cdr (unquote xs))))))
5
6  (meta-nth 2 (quasiquote (list 0 1 2 3))) ; (car (cdr (cdr (list 0 1 2 3))))
7
8  (eval (meta-nth 0 (quasiquote (list 5 6 7 8 9)))) ; 5
9  (eval (meta-nth 1 (quasiquote (list 5 6 7 8 9)))) ; 6
10
11 (define open-frag (meta-nth 1 (quasiquote as))) ; (car (cdr as))
12
13 (let ((as  (list 10 11 12))  ; defining local as
14       (car (lambda (bs) 0))) ; overriding car
15   (eval open-frag))
16
17 (define as (list 10 11 12)) ; defining global as
18 (eval open-frag)             ; 11
```

Figure 3: Scheme `meta-nth`

without any **eval** is referred as *being in the current stage*, and a code fragment that does not perform any computation right now or performs computation only by using **eval** is referred as *being in the next stage*. Furthermore, we can keep enumerating stages to describe a situation where a code fragment represents a program that again generates another code fragment.

Other than the stage distinction, this program exposes another important issue in metaprogramming. If we execute `(meta-nth 1 (quasiquote as))` as in `open-frag`, it returns `(car (cdr as))` with a free variable `as`. This kind of code fragment, called *open code fragment*, poses the following question: which binding should we use for variables appearing in a code fragment when we **eval** it? If we bind free variables in an open code fragment to the local environment of evaluation, the same behaviour can also affect variables that already have global bindings, i.e. **car** in line 14. This makes a metaprogram extremely context-sensitive in the sense that users of a metaprogram should be careful not to override any global definitions used in it. On the other hand, if we bind free variables only to bindings in the global environment, this limits the potential of metaprogramming and will not allow using code fragment `(car (cdr as))` as-is with any local values like in line 15. Among languages with untyped quasiquotation, Scheme, Common Lisp, and Clojure use the latter approach and allows **eval** in line 18 only, while Haskell[Sheard and Jones 2002] provides both tools for the former and latter approaches.

2.1 Limitation of Untyped Representation

There are two main limitations of untyped representations: one from a practical perspective and the other from a theoretical perspective.

First, from a practical perspective, it is hard to check the correctness of metaprograms based on untyped representation. As the representation is untyped, static checking of the metaprograms is solely based on the text representation of an instance of generated code, and we use testing because it is challenging to give correctness guarantees to the text representations of programs on their own. This difficulty of checking is not restricted to the full correctness of a program. For example, it is also hard to guarantee whether a generated program is well-scoped or not. This

possibility of an ill-scoped program is also critical, as a programmer is not able to even run a test on the generated program if it is ill-scoped. Untyped representation is often vulnerable to ill-formed AST as well. In Scheme, one can define a fragment `(let)` using `(quasiquote (let))`, which emits a syntax error only when it is **eval**ed, not when it is defined.

From a theoretical perspective, if we employ an untyped representation in interpreters, it blocks the possibility of self-interpretation for core calculus of strongly typed functional languages, e.g. System F and System Fw[Brown and Palsberg 2016]. This limitation applies to any strongly normalizing confluent lambda calculus, i.e., a lambda calculus that always terminates with a unique normal form. Therefore, this also restrains most of the dependently typed systems, which are strongly normalizing and confluent, from self-interpretation using an untyped representation. This results from the following theorem:

Theorem 1 ([Brown and Palsberg 2016], rephrased)**.**
There is no self interpreter u using an untyped representation in a lambda calculus that terminates with a unique normal form.

Proof. Suppose that we have a self interpreter u using an untyped representation in such a lambda calculus.

Let $\overline{f}$ be an untyped representation of f that u can interpret and give back the normal form of f as the result of the interpretation, and let $p_u := \lambda x.\lambda y.((ux)x)$ where x and y are fresh variables that do not appear in u.

Then $p_u\ \overline{p_u}$ is well typed as u accepts an untyped representation, i.e., a representation of any expressions. Let v be the normal form of $p_u\ \overline{p_u}$, then, by the unique normal term property, $v =_\alpha \lambda y.v$ is satisfied, since $p_u\ \overline{p_u}$ can also be reduced to $\lambda y.v$. However, $v =_\alpha \lambda y.v$ is false, and thus, there is no self interpreter u using an untyped representation. □

These two limitations can be dealt with by typed representations. First, types statically provide strong correctness guarantees, and thus help programmers to write a correct metaprogram more easily. Second, as Brown and Palsberg [2016] also showed in their paper, if we employ some typed representations, there are self interpreters of System F and System F_ω lambda calculi and justifying the result by showing that $p_u\ \overline{p_u}$ from the above proof is ill-typed in those representations. This motivates us to look into type systems for metaprogramming.

3 Typed Quotation and Representation

There are two major traditions of researches on typed quotations: one started from operational consideration, and the other started from a relation between lambda calculi and logic systems. We will explore each tradition in order.

3.1 Approach Based on Operational Consideration

One of the first approaches that provides a typed quotation is MetaML [Taha and Sheard 1997, 2000]. MetaML is a Standard-ML-like language extended with metaprogramming constructs for a typed representation. In MetaML, the type of a code fragment is defined as follows: If a code fragment produces a value of a type τ when evaluated, the code fragment is of the type <τ>. For example,

```
val prog  = 5 + 4         (* prog  : int   *)
val frag1 = <5 + 4>       (* frag1 : <int> *)
val frag2 = <7 + ~frag1>  (* frag2 : <int> *)
val res   = run frag2     (* res   : int   *)

(* metaNth : int -> <int list> -> <int> *)
fun metaNth n xs = if n = 0
                   then <head ~xs>
                   else metaNth (n - 1) <tail ~xs>
```

In this example, each of `<e>`, `~e`, and **run** `e` is MetaML syntax for (**quasiquote** `e`), (**unquote** `e`), and (**eval** `e`), respectively.

The design of MetaML is guided by two criteria that manage the scope of variables in a metaprogram:

- *Cross-stage safety*: A variable that is first available in a stage cannot be used in any previous stages.
- *Cross-stage persistence*: A variable that is available in a stage should be available in later stages.

Cross-stage safety prohibits, for example, the following expression:

```
(* invalid : (int -> int) -> <int -> int> *)
let fun invalid f = <fn b => ~(f b)>
```

In this example, the variable `b` becomes first available in the next stage, but we need to use `b` in the current stage to splice in `f b`. This usage of `b` should be forbidden, since there is no way to know the value of future `b` when we call the function `invalid`.

On the other hand, cross-stage persistence allows programmers to write

```
let val as = [1, 2, 3] in metaNth 2 <as>
```

without the ambiguity occurring in the case of Scheme. This is because the local `as` in its lexical scope should also be available in the next stage according to cross-stage persistence. Thus, `as` in `metaNth 2 <as>` is not a mere syntactic symbol `as`, but a variable with a clear binding location (i.e. **let val** `as`).

However, it should be mentioned that even with its cross-stage persistence, MetaML does not support all open code generation. Its typing rules allow only lexically bound variables, and thus one cannot generate an expression like `<head (tail (tail as))>` without a binding for the variable `as` in the current level.

Another issue in the type-checking of MetaML is that it allows **run** only for limited cases. Specifically, it does not allow a function like **fn** `x` -> **run** `x`. It is also noteworthy that the type system of MetaML is simply typed and cannot express polymorphism as well as type level functions.

3.2 Modal Logic and Metaprogramming

Davies and Pfenning [1996]; Davies [1996] first relate two modal logic systems to metaprogramming. One logic system is necessity modal logic, and the other is linear temporal logic.

Necessity Modal Logic Necessity modal logic is a logic system with a modality $\Box$ that expresses the necessary truth of some propositions. For example, if we show $\Box(1 + 1 = 2)$, it means that the proposition $1 + 1 = 2$ is necessarily true.

Davies and Pfenning [1996, 2001] observed that this necessity modality $\Box$ in a formulation of logic system called *S4* is well suited to distinguish a closed code fragment (of type $\Box\tau$) from a program (of type τ). First, the authors suggest describing S4 modal logic with the following inference rules (described with more detail in [Pfenning and Davies 2001]):

$$\frac{(x:\tau)\in\Delta}{\Delta;\Gamma\vdash x:\tau}\ \mathrm{var}_{\Delta} \qquad \frac{\Delta;\cdot\vdash e:\tau}{\Delta;\Gamma\vdash \mathsf{box}(e):\Box\tau}\ I_{e,\Box} \qquad \frac{\Delta;\Gamma\vdash e_1:\Box\tau \quad \Delta,x:\tau;\Gamma\vdash e_2:\tau'}{\Delta;\Gamma\vdash \mathsf{let\ box}\ x=e_1\ \mathsf{in}\ e_2:\tau'}\ E_{e,\Box}$$

where $\mathsf{box}(e)$ roughly corresponds to `(quasiquote e)`, and let box is an alternative of **unquote** and **eval**. In the above rules, Δ is the context which contains variables available in the current and future stages, while Γ is the usual context only for the current stage.

These rules explain how we check types of box and let box. When we check whether a quoted expression $\mathsf{box}(e)$ has a type $\Box\tau$, we check whether e has a type τ in an empty context. We need the empty context here because this e is in a future stage and should not access any variable in Γ that is available only in the current stage. When we check let box, we check whether e_1 is of type $\Box\tau$ for some τ, and then check e_2 under the context Δ extended with the variable x of the type τ which, in runtime, is bound to the syntactic expression inside the box of e_1.

Based on these rules of S4 modal logic, the authors extend the logic system into a small language, MiniML$^{\Box}_{e}$:

```
1 let prog  = 5 + 4                              (* prog  : int *)
2 let frag1 = box(5 + 4)                         (* frag1 : □int *)
3 let frag2 = let box x = frag1 in box(7 + x)    (* frag2 : □int *)
4 let res   = let box y = frag2 in y             (* res   : int *)
5
6 (* metaNth : int -> □(int list) -> □int *)
7 let rec metaNth n xs = if n = 0
8                        then let box vs = xs in box(head vs)
9                        else metaNth (n - 1) (let box vs = xs in box(tail vs))
```

MiniML$^{\Box}_{e}$, which Davies and Pfenning [1996] refer as an *explicit* formulation, requires users to explicitly mark where a cross-stage computation happens with **let box**. For example, in the above definition of `metaNth`, we used the **let box** `vs = xs` **in** part in front of each **box**, but we can also place it at the beginning of the definition once. This explicit control of staging is tedious and hard to be applied to the pre-existing (untyped) metaprogramming systems. To mitigate this, the authors also provide a more implicit language based on a different formulation of S4 modal logic:

$$\frac{\Psi;\Gamma;\cdot\vdash e:\tau}{\Psi;\Gamma\vdash \mathsf{box}(e):\Box\tau}\ I_{\Box} \qquad \frac{\Psi;\Gamma\vdash e:\Box\tau}{\Psi;\Gamma;\Gamma_1;\ldots;\Gamma_n\vdash \mathsf{unbox}_n(e):\tau}\ E_{\Box}\ (n\geq 0)$$

Note that now each rule requires a stack of contexts, not only two contexts as in MiniML$^{\Box}_{e}$. Each context in the stack corresponds to a single stage, and the right-most context is the context of the current stage. Unlike the previous formulation, there is no context that is directly accessible throughout all future stages, and only unbox can access a context before the context of the current stage.

Under this setting, we check $\mathsf{box}(e)$ by checking e in the context stack extended with a new right-most context, instead of ignoring all entries in the previous right-most context (Γ). When we check $\mathsf{unbox}_n(e)$, we pop last n contexts out from the context stack, and check e under that shortened context stack. It is worth nothing that $\mathsf{unbox}_0(e)$ allows us to **eval** the expression e.

The authors extend these rules into an implicit version of the language, MiniML$^{\Box}$. In MiniML$^{\Box}$, `metaNth` can be written like this:

```
let rec metaNth n xs = if n = 0
                       then box(head unbox1(xs))
                       else metaNth (n - 1) box(tail unbox1(xs))
```

Davies and Pfenning [1996] prove that there is a correspondence between MiniML$^{\Box}_{e}$ and MiniML$^{\Box}$ by showing that there is a translation from each one to the other. In fact, the authors define the runtime semantics of MiniML$^{\Box}$ by translating it into MiniML$^{\Box}_{e}$ instead of defining the semantics directly.

However, both of their languages share the same limitation in their expressive powers. Those languages do not support open code fragments. This is because rules for $\mathsf{box}(e)$ in both languages check e in a context where no entries associated to the stage of e. For example, both systems support none of the following definitions:

```
let open = metaNth 2 box(as)

(* defer : (□int -> □int) -> □(int -> int) if typable *)
let defer f = box(fn x -> unbox1(f box(x)))
```

where in `defer`, the `box(x)` is considered as an open code fragment.

Linear Temporal Logic On the other hand, Davies [1996] proposes a correspondence between linear temporal logic and a type system that can characterize some open code fragments. Linear temporal logic is a logic system with a modality $\bigcirc$ that expresses the truth at the next time step. For example, if we show $\bigcirc$(A bus arrives), it means a bus will arrive at the next time step.

Instead of giving a type $\Box\tau$ to a closed code fragment using the modality $\Box$ as in MiniML$^{\Box}$, Davies [1996] gives a type $\bigcirc\tau$ to an open code fragment. Specifically, the typing is based on the following rules:

$$\frac{(x:\tau)^n \in \Gamma}{\Gamma \overset{n}{\vdash} x:\tau}\ \text{var} \qquad \frac{\Gamma \overset{n+1}{\vdash} e:\tau}{\Gamma \overset{n}{\vdash} \mathsf{next}(e):\bigcirc\tau}\ I_{\bigcirc} \qquad \frac{\Gamma \overset{n}{\vdash} e:\bigcirc\tau}{\Gamma \overset{n+1}{\vdash} \mathsf{prev}(e):\tau}\ E_{\bigcirc}$$

where the number on $\vdash$ means a time step, which is virtually a stage of interest. In this system, next and prev can be understood as **quasiquote** and **unquote** respectively.

It should be emphasized that these typing rules do not change the context Γ when changing the stage. This unchanged context allows this system to access any previously-bound variables in the same stage. Thus, in MiniML$^{\bigcirc}$, an extension of this linear temporal logic, we can write the following program:

```
(* defer : (○int -> ○int) -> ○(int -> int) *)
let defer f = next(fn x -> prev(f next(x + 1)))
```

While it provides a logical foundation describing how to generate and splice in an open code fragment, this language has two critical limitations. First, it does not support some valid open code fragments. For example, we cannot express `next(head (tail (tail as)))` where `as` is not in scope. In fact, this approach is more limited from a practical point of view than MetaML, since MiniML$^{\bigcirc}$ does not support all code fragments that are allowed in MetaML. Second, it does not support **eval**. This is because we cannot distinguish an open code fragment from a closed one, and it is not sound to evaluate an open expression before all of its free variables are bound. For example, suppose that we have a **eval** construct and call `defer (fn c -> if eval(c)= 0 then next(0)else next(1))`. Since `c` here is `next(x + 1)` where `x` is a future variable, we cannot get a **next** value from the evaluation of `f next(x + 1)`, and thus, cannot splice the result into `next(fn x -> _)`.

3.3 Limitation of Early Approaches

These early approaches share one common drawback: they support only a limited notion of openness if any. This limitation becomes more problematic if we want to support analysis of code fragments as well as their generation. Suppose that we pattern match on the code fragment `box(fn x -> x + 0)`. Since `x` (parameter) and `x + 0` (body) are distinct syntactic components, we want to match on each of these parts with a separate pattern variable like `box(fn X -> B)`. Now the problem is: what type should we give to `B`? If the system does not support typing for an open code fragment, we cannot give a type to `B` because it is an open code fragment that depends on a name given in `X`.

Another problem shared by those approaches with a limited open code support is restricted **run**/**eval**. MetaML supports **run**, but it cannot be used with a function argument, and MiniML$^{\bigcirc}$ does not support **eval** at all.

Both of these weaknesses are related to open code fragments, which can be evaluated only in certain contexts. Subsequently, the desire to overcome these limitations has inspired both traditions to give a context of an open code fragment a more explicit type.

4 Explicit Typing on Context

4.1 Environment Classifiers

Taha and Nielsen [2003] first introduce *environment classifiers* with a language λ^{α} that extends MetaML[Taha and Sheard 1997]. An environment classifier is a name that labels a part of the context in which a term is typed, and thus specify where a code fragment can be **run**. For example,

```
1 (* metaNth : int -> (α)<int list>^α -> (β)<int list>^β *)
2 fun metaNth n xs = if n = 0
3                    then (β)<head ~(xs[β])>^β
4                    else metaNth (n - 1) ((α)<tail ~(xs[α])>^α)
5
6 (* wrappedRun : (α)<int>^α -> (β)int *)
7 val wrappedRun = fn e -> (β)((run e)[β])
```

In λ^{α}, the authors extend the quoter `<e>` with an environment classifier annotation, `<e>`$^{\alpha}$, and add introduction (α) and instantiation $[\alpha]$ of environment classfiers. Essentially, a type $(\alpha)\tau$ means a type quantified with environment classifier, i.e., that τ is available in any context, and a type $\tau[\alpha]$ means an instantiation of a such quantified types.

`metaNth` example uses two environment classifiers, α and β. The environment classifier α refers to the context in which `xs` will be used, and the environment classifier β denotes the context in which the result code fragment will be used.

This environment classifier allows, unlike MetaML, to wrap **run** into a function. `wrappedRun` example shows how this work with environment classifiers. Moreover, environment classifiers rightfully reject the following `defer`, which evaluate a future variable as explained in MiniML$^{\bigcirc}$:

```
1 val defer = (β)<fn x -> ~(let val y = (run ((α)<x + 1>^α))[β] in <y>^β)>^β
```

This rejection is because `x` is available only in a context specified by β, not α.

This λ^{α} was once implemented in MetaOCaml but replaced with a dynamic scope checker[Kiselyov 2014], due to two issues: First, polymorphism, especially with the polymorphic version of `wrappedRun`, makes the core calculus non-terminating even without explicit recursions or loops. Second, in the

presence of effects, especially with mutable references, type checking based on environment classifiers does not guarantee that an open code fragment is well-scoped.

The second issue can be illustrated with this example:

```
1 val scope = (α)(let val r = ref <0>^α in
2                 let val _ = <fn x -> ~(r := <x + 1>^α; <x>^α)>^α in
3                 <fun y -> ~(!r)>^α)
4
5 val _ = run scope
```

where **ref** is a constructor of a mutable reference with an initial value, and `:=` and `!` are operators for the update and dereference of mutable reference, respectively. This code is accepted by λ^α extended with mutable references, but in runtime, the code spliced in by `~(!r)`, i.e., `<x + 1>`, cannot be evaluated, since `x` is not in scope.

To overcome this issue, Kiselyov et al. [2016] design the calculus $\langle \mathrm{NJ} \rangle$, which has only two stages but allows well-scoped code generation with the power of mutable references. This calculus correctly rejects the `scope` example, while accepting the following variant of `metaNth` written with a mutable reference.

```
1 fun metaNth n xs = (α)(let val r = ref <~xs>^α
2                        let fun helper n =
3                          if n = 0
4                          then !r
5                          else (r := <tail ~(!r)>^α; helper (n - 1))
6                        in
7                        <head ~(helper n)>^α)
```

Nevertheless, none of these calculi supports an open code fragment without lexically bound variables, i.e., `metaNth 2 <as>` without a binding for `as`, and thus they are not extensible to pattern matching on metaprograms. Moreover, as an environment classifier denotes a context abstractly instead of considering each concrete entry in the context, it is not possible to, for example, swap two free variables appearing in a code fragment.

Calculus $\langle \mathrm{NJ} \rangle$ also has two other limitations. First, it allows only two stages; thus, it is not possible to generate a code fragment that generates another code fragment, unlike other previous approaches. Second, it does not explicitly explain how to type check **run**. Even though the authors mention that "Adding it (**run**) to $\langle \mathrm{NJ} \rangle$ is straightforward"[Kiselyov et al. 2016], but the actual typing rule is not given in their paper.

4.2 Contextual Types

To capture the notion of open code logically, Nanevski et al. [2008] extend MiniML$^{\Box}_{e}$ so that the necessity modality also carries a context Ψ in which the quoted term is typable. A code fragment for an expression of a type τ under a context Γ has the type $[\Gamma]\tau$, which corresponds to $\Box\tau$ if Γ is an empty context. To deal with this accompanying context, the authors change typing rules of MiniML$^{\Box}_{e}$ as follows:

$$\frac{(u : \tau[\Psi]) \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash \mathsf{clo}(u, \sigma) : \tau}\ \text{ctxvar}$$

$$\frac{\Delta; \Psi \vdash e : \tau}{\Delta; \Gamma \vdash \mathsf{box}(\Psi.e) : [\Psi]\tau}\ \Box I \qquad \frac{\Delta; \Gamma \vdash e_1 : [\Psi]\tau \qquad \Delta, x : \tau[\Psi]; \Gamma \vdash e_2 : \tau'}{\Delta; \Gamma \vdash \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 : \tau'}\ \Box E$$

In the contextual type theory, box comprises two parts, Ψ and e. As in MiniML$^{\Box}_{e}$, e is the expression we want to quote. However, unlike MiniML$^{\Box}_{e}$, we check the type of e in Ψ, not in an empty context. With this typing rule, box$(\Psi.e)$ can characterize open code fragments with free variables specified in Ψ. Since we allow an expression e in box$(\Psi.e)$ to access variables in Ψ, we need to provide values for variables in Ψ when we use names in Δ. The authors add a new construct clo and a *substitution* σ for this purpose. clo(u, σ) uses σ to get terms for context entries in Ψ and substitutes them in u to get closed code.

This contextual box and explicit substitution allow programmers more fine-grained control of an open code fragment. For example, we can type check the following code:

```
1  (* metaNth : int -> [ys : int list](int list) -> [ys : int list]int *)
2  let rec metaNth n xs =
3    if n = 0
4    then let box u = xs in box(ys . head clo(u, ys))
5    else metaNth (n - 1) (let box u = xs in box(ys . tail clo(u, ys)))
6
7  (* openFrag : [ys : int list]int *)
8  let openFrag = metaNth 2 box(ys . ys)
9
10 (* useOpenFrag : int *)
11 let useOpenFrag = let as = [1, 2, 3] in let box v = openFrag in clo(v, as)
12
13 (* swap : [xs:int list, ys:int list](int list) *)
14 let swap = let box append = box(xs, ys . xs ++ ys) in
15            box(xs, ys . clo(append, (ys, xs)))
```

As a matter of fact, in contextual types, the second parameter of `metaNth` is unnecessary. In previous systems, we need the parameter because we have to receive an identifier used in an open code fragment. With contextual types, `metaNth` can pick any identifier, and a caller of the function can substitute a desired identifier or value afterwards. By removing the redundant parameter, we get the following definition:

```
1  (* ctMetaNth : int -> [ys : int list]int *)
2  let rec ctMetaNth n =
3    if n = 0
4    then box(ys . head ys)
5    else (let box u = metaNth (n - 1) in box(ys . clo(u, tail ys)))
6
7  (* useCt0 : int *)
8  let useCt0 = let box v = ctMetaNth 2 in clo(v, [1, 2, 3])
9
10 (* useCt1 : int *)
11 let useCt1 = let as = [1, 2, 3] in let box v = ctMetaNth 2 in clo(v, as)
```

Here, we construct an open code fragment that uses a fixed name `ys`. However, substitutions allow us to use that open code fragment with a constant (in `useCt0`) or a variable with a different name (in `useCt1`).

For the purpose of showing that this language is type-safe and terminates, the authors investigate different types of substitutions and their properties. However, these substitutions are not first-class entities in this language as there is no way to abstract over substitutions nor bind names to them.

It is also important to mention that the authors extend the simple contextual type theory above to a dependent contextual type theory. The authors describe how contextual types could be applied to the Edinburgh Logical Framework (LF) [Harper et al. 1993], a simple form of a dependently

typed system. Specifically, they notice that contextual type can be used to logically characterize holes in a program. This characterization allows a systematic approach to the implementation of proof assistants based on LF, such as the Twelf system[Pfenning and Schürmann 1999]. This also provides the foundation under the Beluga Proof environment and other related research[Pientka and Dunfield 2008; Pientka 2010; Boespflug and Pientka 2011; Ferreira and Pientka 2017; Pientka et al. 2019].

Even though this paper gives a versatile description of open code fragments, it still has some limitations. One of them is pattern matching on code fragments.

5 Pattern Matching

Pientka [2008, 2010]; Pientka and Dunfield [2008] first observe that applying the contextual type on LF allows one to provide a language that facilitates manipulations on LF terms, not just an internal implementation. Their language, called Beluga, contains both LF and a functional language that generates and analyses LF terms based on the contextual type. For example,

```
1  LF nat : type = z : nat | s : nat -> nat;
2
3  LF exp : type = val : nat -> exp | add : exp -> exp -> exp;
4
5  rec opt-exp : [|- exp] -> [|- exp] = / total x (opt-exp x) /
6  fn x => case x of
7    | [|- val N] => [|- val N]
8    | [|- add E1 E2] => case opt-exp [|- E1] of
9      | [|- val z] => opt-exp [|- E2]
10     | [|- E1'] => let [|- E2'] = opt-exp [|- E2] in [|- add E1' E2'];
```

Here, the first `nat` and `exp` are LF data type definitions, and `opt-exp` is a Beluga function that pattern matches on LF terms. In the type of this function, `[Γ |- τ]` corresponds to `[Γ]`τ in Nanevski et al. [2008]'s language.

This example `opt-exp` shows the power of pattern matching in metaprogramming. It can optimize expressions of the form `add (val z)e` into `e`. This feature allows programmers to write metaprograms in a more modular way, one for program generation and one for optimization, as in compilers, which often handle those in separate phases.

Furthermore, Beluga treats both contexts and substitutions as first-class entity in the language, in the sense that it allows programmers to abstract over contexts and substitutions and to pass those to functions.

However, unlike other metaprogramming systems previously explained in this survey, Beluga is heterogeneous, i.e., the language of code fragments and the language for metaprograms are different. In Beluga, code fragments are in the logical framework LF, while the metaprograms are written in the functional language of Beluga. Thus, it is not possible to write a metaprogram that itself generates and manipulates the functional language of Beluga. Boespflug and Pientka [2011] extend Beluga so that it can manipulate itself but do not provide the operational semantics for their multi-staged language.

Another limitation of Beluga is that it does not allow its functions to appear in types. Pientka et al. [2019] resolved this by integrating Martin-Löf style Type Thoery with the contextual type on LF. This work provides theoretical foundation for a heterogeneous dependent type system. Unfortunately, no implementation has reported yet.

A similar approach to Beluga has also been tried outside of LF. Ferreira and Pientka [2017] provides a metaprogramming system with $\lambda^{\square}$-like code fragments and ML-like meta-language. As Beluga combines LF and a functional language using the contextual type, Ferreira and Pientka [2017]'s language, Core-ML$^{\text{gadt}}$, uses the contextual type to seamlessly integrate those two languages. However, their language for code fragments does not allow any dependent types unlike LF.

Recently Jang et al. [2022] give a foundation for type-safe, homogeneous metaprogramming, i.e., where the languages for code fragments and metaprogram coincide. The proposed system, called Mœbius, extends the contextual type theory[Nanevski et al. 2008] to polymorphic types as well as pattern matching on code fragments. The authors introduce a new concept, called levels, which gives fine-grained control of cross-stage persistence[Taha and Sheard 1997] of variables. This control allows Mœbius to characterize pattern variables with correct dependencies. However, Mœbius uses **box** and **let box**, and does not have a translation to **box** and **unbox**, and its implementation is not provided.

6 Conclusion

Metaprogramming is a useful tool in practice. However, its theoretical foundation is not yet well understood, and this lack of understanding prevents us from exploiting the full potential of metaprogramming. In this survey, we discussed different approaches to give this theoretical foundation. However, given state-of-the-art, there are three major open questions: effectful metaprogramming, metaprogramming with dependent types, and runtime implementation.

First, it is not yet known how to integrate effects (mutation, failure, IO, and others) with genuine open code fragments. Even though Kiselyov et al. [2016]; Oishi and Kameyama [2017] handle this issue in a restricted setting, their approach has not been extended to open code fragments in contextual type theory or to general computational effects other than mutations.

Dependent type systems, as one of Coq or Agda, are also a long way off. There are efforts to provide metaprogramming machinery for those languages, such as MetaCoq[Sozeau et al. 2019; Anand et al. 2018], Agda reflection[van der Walt and Swierstra. 2012], and Agda library of Devriese and Piessens [2013]. However, MetaCoq and Agda reflection do not support typed representations, and Devriese and Piessens [2013]'s library does not allow open code fragments.

We also do not yet know how to implement a runtime of a multi-stage metaprogramming system that can deal with open code fragments. For typed metaprogramming in general, MetaML[Taha and Sheard 1997], MetaOCaml[Kiselyov 2014], and Typed TemplateHaskell has been implemented, but none of them supports open code fragments. Multi-stage systems that support the contextual type theory, including Boespflug and Pientka [2011]'s extension of Beluga and Mœbius[Jang et al. 2022], have no implementation yet.

These challenges give a better understanding on how to write and maintain safe metaprograms. Since metaprogramming is nearly universal in the field of programming, this better understanding has the potential to impact many programs that we write on a daily basis.

Bibliography

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *9th International Conference*

Interactive Theorem Proving (ITP'18) (Lecture Notes in Computer Science (LNCS 10895)). Springer, 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level Contextual Modal Type Theory. In *6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11) (Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 71)*, Gopalan Nadathur and Herman Geuvers (Eds.). 29–43. https://doi.org/10.4204/EPTCS.71.3

Matt Brown and Jens Palsberg. 2016. Breaking through the normalization barrier: a self-interpreter for F-Omega. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, 5–17. https://doi.org/10.1145/2837614.2837623

Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 184–195. https://doi.org/10.1109/LICS.1996.561317

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 258–270. https://doi.org/10.1145/237721.237788

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Dominique Devriese and Frank Piessens. 2013. Typed syntactic meta-programming. In *ACM SIGPLAN International Conference on Functional Programming, (ICFP'13)*. ACM, 73–86. https://doi.org/10.1145/2500365.2500575

Francisco Ferreira and Brigitte Pientka. 2017. Programs Using Syntax with First-Class Binders. In *26th European Symposium on Programming (ESOP 2017) (Lecture Notes in Computer Science (LNCS 20201))*, Hongseok Yang (Ed.). 504–529. https://doi.org/10.1007/978-3-662-54434-1_19

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *Journal of the ACM* 40, 1 (January 1993), 143–184.

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Moebius: Metaprogramming using Contextual Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, Pennsylvania, USA) *(POPL '22)*. Association for Computing Machinery, New York, NY, USA.

JetBrains. 2021. The State of Developer Ecosystem 2021. https://www.jetbrains.com/lp/devecosystem-2021/ Accessed: November 28th, 2021.

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

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *14th Asian Symposium on Programming Languages and Systems (APLAS'16) (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

Drew McDermott and Gerald Sussman. 1972. *CONNIVER reference manual*. Technical Report. http://hdl.handle.net/1721.1/6203

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 1–49.

Flemming Nielson and Hanne R. Nielson. 1986. Code generation from two-level denotational metalanguages. In *Programs as Data Objects*, Harald Ganzinger and Neil D. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–205.

Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with Control: Type-Safe Multi-Stage Programming with Control Operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/3136040.3136049

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540.

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction (CADE-16) (Lecture Notes in Artificial Intelligence (LNAI 1632))*, H. Ganzinger (Ed.). Springer, 202–206.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, 371–382. https://doi.org/10.1145/1328897.1328483

Brigitte Pientka. 2010. Beluga: programming with dependent types, contextual data, and contexts. In *10th International Symposium on Functional and Logic Programming (FLOPS'10) (Lecture Notes in Computer Science (LNCS 6009))*, German Vidal Matthias Blume, Naoki Kobayashi (Ed.). Springer, 1–12.

Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rebecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *34th IEEE/ ACM Symposium on Logic in Computer Science (LICS'19)*. IEEE Computer Society, 1–13.

Brigitte Pientka and Joshua Dunfield. 2008. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*. ACM Press, 163–173. https://doi.org/10.1145/1389449.1389469

Willard Quine. 1940. *Mathematical logic*. Harvard University Press.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *ACM SIGPLAN Workshop on Haskell (Haskell'02)*. ACM, 1–16. https://doi.org/10.1145/581690.581691

Michael Snoyman. 2015. *Developing web apps with Haskell and Yesod: safety-driven web development*. "O'Reilly Media, Inc.".

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2019. The MetaCoq Project. (June 2019). https://hal.inria.fr/hal-02167423 working paper or preprint.

Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, New Orleans, Louisisana, 26–37. https://doi.org/10.1145/640128.604134

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

Walid Taha and Tim Sheard. 2000. MetaML and Multi-stage Programming with Explicit Annotations. *Theoretical Computer Science* 248, 1-2 (Oct. 2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *24th Intern. Symp. on Implementation and Application of Functional Languages (IFL)*, Ralf Hinze (Ed.). Springer, 157–173.