Mœbius: Metaprogramming using Contextual Types

The Stage Where System F Can Pattern Match on Itself

JUNYOUNG JANG, McGill University, Canada
SAMUEL GÉLINEAU, Simspace, Canada
STEFAN MONNIER, Université de Montréal, Canada
BRIGITTE PIENTKA, McGill University, Canada

We describe the foundation of the metaprogramming language, Mœbius, which supports the generation of polymorphic code and, more importantly, the analysis of polymorphic code via pattern matching.

Mœbius has two main ingredients: 1) we exploit contextual modal types to describe open code together with the context in which it is meaningful. In Mœbius, open code can depend on type and term variables (level 0) whose values are supplied at a later stage, as well as code variables (level 1) that stand for code templates supplied at a later stage. This leads to a multi-level modal lambda-calculus that supports System-F style polymorphism and forms the basis for polymorphic code generation. 2) we extend the multi-level modal lambda-calculus to support pattern matching on code. As pattern matching on polymorphic code may refine polymorphic type variables, we extend our type-theoretic foundation to generate and track typing constraints that arise. We also give an operational semantics and prove type preservation.

Our multi-level modal foundation for Mœbius provides the appropriate abstractions for both generating and pattern matching on open code without committing to a concrete representation of variable binding and contexts. Hence, our work is a step towards building a general type-theoretic foundation for multi-staged metaprogramming that, on the one hand, enforces strong type guarantees and, on the other hand, makes it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing the reliability of and trust in the code we are producing and running.

CCS Concepts: • **Software and its engineering → Functional languages**; **Formal language definitions**; • **Theory of computation → Type theory**; **Modal and temporal logics**.

Additional Key Words and Phrases: Metaprogramming, Type Systems, Contextual Types, Polymorphism

ACM Reference Format:
Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: Metaprogramming using Contextual Types: The Stage Where System F Can Pattern Match on Itself. *Proc. ACM Program. Lang.* 6, POPL, Article 39 (January 2022), 27 pages. https://doi.org/10.1145/3498700

1 INTRODUCTION

Metaprogramming is the art of writing programs that produce or manipulate other programs. This opens the possibility to eliminate boilerplate code and exploit domain-specific knowledge to build high-performance programs. Unfortunately, designing a language extension to support type-safe, multi-staged metaprogramming remains very challenging.

One widely used approach to metaprogramming going back to Lisp/Scheme is using *quasiquotation* which allows programmers to generate and compose code fragments. This provides a simple

Authors' addresses: Junyoung Jang, School of Computer Science, McGill University, Canada, junyoung.jang@mail.mcgill.ca; Samuel Gélineau, , Simspace, Canada; Stefan Monnier, Computer Science, Université de Montréal, Canada, monnier@iro.umontreal.ca; Brigitte Pientka, School of Computer Science, McGill University, Canada, bpientka@cs.mcgill.ca.

2475-1421/2022/1-ART39
https://doi.org/10.1145/3498700

and flexible mechanism for generating and subsequently splicing in a code fragment. However, it has been challenging to provide type safety guarantees about the generated code, and although many statically typed programming languages such as Haskell [Sheard and Jones 2022], and even dependently-typed languages such as Coq [Anand et al. 2018] or Agda [van der Walt and Swierstra. 2012] support a form of quasiquotation, they generate untyped code. Fundamentally, we lack a type-theoretic foundation for quasiquotation that provides rich static type guarantees about the generated code and allows programmers to analyze and manipulate it.

This is not to say that no advances have been made in the area of typed metaprogramming. Two decades ago, MetaML[Taha and Nielsen 2003; Taha and Sheard 2000] pioneered the type-safe code generation from a practical perspective. At the same time, Davies and Pfenning [2001] observed that the *necessity (box) modality* allows us to distinguish the generated code from the programs that are generating it providing a logical foundation for metaprogramming. For example, code fragments like `box(2 + 2)` have boxed types such as `⌈nat⌉`, while programs such as `6 * 3`, which will evaluate to 18 and have type `nat`. Nanevski et al. [2008] extend this idea to describe open code with respect to a context in which it is meaningful using contextual types. For example, the code fragment `box(x. x + 2)` has the contextual box type `⌈x:nat ⊢ nat⌉`. To use a piece of code such as `box(x. x + 2)`, we bind `x.x+2` to the contextual variable `U` of type `(x:nat ⊢ nat)` using a **`let-box`** expression. At the site where we use `U` to eventually splice in `(x.x+2)`, we associate `U` with a delayed substitution giving a value to `x`. As soon as we know what `U` stands for, we can apply the substitution. For example, **`let box`** `(x.U) =` **`box`**`(x.x+2)` **`in box`**`(U` **`with`** `3)` will bind `U` to the code `(x.x+2)` during runtime, and produce **`box`**`(3 + 2)` as a result where `x` in the code `x+2` has been replaced by `3`. We hence treat contextual variables as closures consisting of a variable and a delayed substitution (written using the keyword **`with`**, in this example). This allows us not only to instantiate open code fragments, but also ensures that variables in code fragments are properly renamed when we splice them in. Using closures and contextual types is in contrast to using functions and function types where we could write a program **`let box`** `U =` **`box`**`(`**`fn`** `x → x + 2)` **`in box`**`(U 3)`. Here, evaluation would produce the code **`box`** `((`**`fn`** `x → x + 2) 3)` containing an administrative redex, which is particularly undesirable in Mœbius, where the program can inspect the code. Arguably, contextual types lead to more compact generated code which is more in line with the programmer's intention.

This line of work cleanly separates local variables (such as `x`) and global variables (i.e. variables such as `U`, accessible at every stage) into two zones. This provides an alternative to the Kripke-style view where a stack of contexts models the different stages of computation. While the latter is appealing, since **`box`** (quote) and **`unbox`** (unquote) match common metaprogramming practice, reasoning about context stacks and variable dependencies is difficult. The **`let box`** formulation of the modal necessity leads to an arguably much simpler formulation of the static and operational semantics, which we see as an advantage.

In this paper, we introduce a core metaprogramming language, Mœbius, which allows us to generate and, importantly, analyze polymorphically-typed open code fragments using pattern matching. Our starting point is the lambda-calculus presented by Nanevski et al. [2008] where they distinguish between local and global variables. This formulation makes evaluation order explicit using **`let box`**-expressions. This allows us to more clearly understand the behavior of multi-staged metaprograms that work with open code. We extend their work in three main directions:

1) Generating polymorphic code. First, we generalize their language to support System F style polymorphism and polymorphic code generation. In particular, `⌈'a:*, x:'a, f: 'a → 'a ⊢'a⌉` describes a polymorphic code fragment such as **`box`**`('a, x, f. f (f x))`. A context in our setting keeps track of both term and type variables, hence treating both assumptions uniformly. This allows us to support code generation for polymorphic data structures such as polymorphic lists.

2) Generating code that depends on other code and type templates. Second, we generalize contextual types to characterize code that depends itself on other code fragments and support a composition of code fragments which avoids creating administrative redexes due to boxing. This is achieved by generalizing the two-zone formulation by Davies and Pfenning [2001] and Nanevski et al. [2008], which distinguishes between two levels, to an n-ary zone formulation and leads us to a multi-level contextual modal lambda-calculus. The two-zone formulation by Davies, Pfenning, et. al. is then a special case of our multi-level calculus. For example, `⌈c:(x:nat ⊢ nat), x:nat ⊢ nat⌉` can describe a piece of code like **box**`(c, x. x + c with x)` that eventually computes a natural number, but depends on the variable `x:nat` and on another piece of open code `c:(x:nat ⊢ nat)`. Note that `c` simply stands for a piece of open code and any reference to it will have to provide an `x` with which to close it – this is in contrast to an assumption of type `⌈x:nat ⊢ nat⌉` which stands for the boxed version of a piece of code. This allows us to elegantly and concisely combine code fragments:

```
let box (y. R) = box (y. y + 2) in
let box (c,x. U) = box (c,x. 3 * x + (c with (2 * x))) in
  box (y. U with ((y. R with y), y))
```

which results in the code **box**`(y. 3 * y + (2 * y + 2))`. We say that `⌈c:(x:nat ⊢ nat), x:nat ⊢ nat⌉` is a level 2 contextual type, as it depends on `x:nat` (level 0) and `c:(x:nat ⊢ nat)` (level 1) assumptions. This view is in contrast to the two-level modal lambda-calculus described in Davies and Pfenning [2001] or Nanevski et al. [2008], where we would need to characterize code depending on other code using the type `⌈c:⌈x:nat ⊢ nat⌉, x:nat ⊢ nat⌉`. However, this would not allow us to splice in `(y. y + 2)` for `c` in the code bound to `U`; instead, we would need to splice in **box**`(y. y + 2)` for `c` and retrieve the code `(y. y + 2)` using a **let-box**-expression. The code bound to `U` in line 2 would need to be written as: **box** `(c,x. let box (y. R') = c in 3 * x + (R' with (2 * x)))`. This would then generate the code **box**`(y. let box (y. R') = box(y. y + 2) in 3 * y + (R' with (2 * y)))` which contains an administrative redex. Our work avoids the generation of any administrative redexes and hence generates code as intended and envisioned by the programmer.

We further extend this idea of code templates uniformly to type templates, i.e. we are able to describe the skeleton and shape not only of code, but also of types themselves. As a consequence, Mœbius sits between System F and System F_ω where we support term and type-level computation.

3) Pattern matching on code. Third, we extend the multi-level contextual modal lambda-calculus with support for pattern matching on code. This follows ideas in Beluga [Pientka 2008; Pientka and Dunfield 2008] where pattern matching on higher-order abstract syntax (HOAS) trees is supported. However, in Beluga, we separate the language in which we write programs from the language that describes syntax. In Mœbius, such a distinction does not exist and meta-programs can pattern match on code that represents another (meta-)program in the same language. We view the syntax of code through the lenses of higher-order abstract syntax treating variables abstractly and characterize pattern variables using multi-level contextual types. In fact, multi-level contextual types are the key to characterizing pattern variables in code – especially code that may itself contain **box**-expressions and **let box** expressions!

Pattern matching on polymorphic code may refine type variables, as is typical in indexed or dependently typed systems. For example, when we pattern match on a piece of code of type `⌈x:'a ⊢ 'a⌉`, then one of the branches may have the pattern **box**`(x. 0)`, which has type `⌈x:int ⊢ int⌉`. Hence, we need to use the constraint `'a = int` to type check the body of the branch. We, therefore, extend our multi-level contextual modal lambda calculus to generate and track type constraints, and type-check a given expression modulo constraints. Despite the delicate issues that arise in supporting System F style polymorphism and pattern matching on code, our operational semantics and the accompanying type preservation proof is surprisingly compact and clean.

We view our work as a step towards building a general type-theoretic foundation for multi-staged metaprogramming, which enforces strong type guarantees and makes it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing the reliability of and trust in the code we are producing and running.

2 MOTIVATION

To illustrate the design and capabilities of Mœbius, we discuss several examples below.

2.1 Example: Generating Open Polymorphic Code

First, we implement the function `nth` which generates the code to look up the `i`-th element in a polymorphic list `v` where `v` is supplied at a later (next) stage.

```
nth : int → ⌈'a:*, v:'a list ⊢ 'a⌉
nth n = if n <= 0 then
          box('a,v. hd v )
        else
          let box ('a,v. X) = nth (n - 1) in box('a,v. X with 'a, tl v) )
```

The result of the computation has the contextual type `⌈'a:*, v:'a list ⊢'a⌉`. This type describes open code at level 1 that has type `'a` and depends on variables from level 0, namely `'a:*, v:'a list`. In general, a contextual type $\lceil \Psi \vdash^n T \rceil$ characterizes a code template at level n of type `T`. This code template may depend on locally bound variables in Ψ which contains variables at levels strictly lower than n. The code template may also refer to outer variables that are greater or equal to n and whose values are computed during run-time. This principle is what underlies the design of the multi-level modal lambda-calculus and is worth highlighting:

> *Mantra: a code template* $\lceil \Psi \vdash^n T \rceil$ *at level n, can depend on locally bound variables* Ψ *from levels less than n and outer variables that have levels greater or equal to n.*

In the above example, the result that we return in the recursive case is **box**`('a,v. X with 'a, tl v)`. The code inside the **box**, depends on the locally bound variables `'a` and `v` (all of which have level 0) and uses the outer variable `X` (which has level 1). During runtime, `X` will be bound to the result of the recursive call `nth (n-1)`. Given an `n`, the program `nth` will recursively build up the code

$$\textbf{box}(\texttt{'b, v. hd } \underbrace{(\texttt{tl } (.. \ (\texttt{tl v})))}_{n})$$

Ultimately producing a code template that depends only on variables at level 0.

Subsequently, we write $\vdash^n$ in a contextual type, if we want to make explicit the level at which a term is well-typed, but we will mostly omit the levels when they can be easily inferred.

2.2 Example: Combining Code Templates

Next, we generate code that depends on two other code templates, `c` and `d`, both of which have the contextual type $(\texttt{x:int} \vdash^1 \texttt{int})$. The type of these templates is at level 1, as they depend on the variable `x:int`, which is at level 0. Hence, the overall generated code template lives at level 2.

In this simple example, we combine the two templates `c` and `d` in different ways depending on whether the input to the function `combine` is `true` or `false`. From a computational view, if the input evaluates to `true`, then we generate code that, during runtime, will first evaluate the template `d` and subsequently pass its result to the template `c`. If the input evaluates to `false`, then we do the opposite, i.e., we will first evaluate the template `c` and then pass its result to `d`.

```
combine : bool → ⌈c:(x:int ⊢¹ int) , d:(x:int ⊢¹ int), x:int ⊢² int⌉
combine p = if p then
```

```
    box(c,d,x. (fun y → c with y) (d with x))
  else
    box(c,d,x. (fun y → d with y) (c with x))
```

One might wonder whether we could have generated the code `box(c,d,x. c with (d with x))` instead of `box(c,d,x. (fun y → c with y) (d with x))`. To understand the difference in the runtime behaviour, we use the following instantiation: `(x. x+2*x)` for `c`, `(x. x*3)` for `d`, and `3` for `x`.

When we use the code `box(c,d,x. (fun y → c with y) (d with x))` with `(x. x+2*x), (x. x*3), 3`, then we run the code `(fun y → y + 2*y) (3*3)`, which will first evaluate `3*3` to `9` and then compute `9 + 2*9`, which returns 27. When we use `box(c,d,x. c with (d with x))` with the same instantiation, we will evaluate the code `3*3 + 2*(3*3)`, i.e. we will evaluate the code `3*3`, which we obtain by instantiating `d with x` twice! Effectively, we are using a call-by-name evaluation strategy, as the execution of `3*3` gets delayed. This example hence illustrates yet another difference between closures and function applications. Last, this example also highlights the difference between levels and stages. The result for `combine` is a contextual type at level 2, since it depends on other code templates. However, we are generating the code in one stage.

The difference between levels and stages can also be understood from a logical perspective: stages characterize *when code is generated*; as such they describe a property of a boxed type in a positive position. Levels express a property of the assumptions (i.e. negative occurrences) that are used in a boxed type, namely that *how code is used*. In particular, levels allow us to express directly and accurately the fact that code may depend not only on closed values, but also on open code. This allows us to describe code that may itself depend on other code.

2.3 Example: Type Templates

Mœbius supports System-F style polymorphism. Hence, we may not only want to describe open code, but also types that are open (i.e. type templates). For example, consider the contextual type

```
⌈'a:('c:* ⊢ *), f: ∀'b. 'b → ('a with 'b) → int ⊢ int⌉
```

This describes a piece of code that relies on the polymorphic function `f`, which computes an `int`. Here `'a` describes a type template – it stands for some type that has one free type variable. We again associate `'a` with a substitution which, in this example, renames the variable `'c` in the type template to be `'b`. As a consequence, `f` in fact stands for a family of functions! If we instantiate `'a` with the type template `'c.'c`, then `f` has type `∀ 'b. 'b → 'b → int`. If we instantiate it with `'c. 'c → c'`, then `f` stands for a function of type `∀ b'. b' → (b' → b') → int`. In System F_ω, we would have declared this type as: `⌈'a: * → *, f:∀ 'b. 'b → 'a 'b → int ⊢int⌉`

The support of type templates that have contextual kinds means that Mœbius sits between System F_ω where we have type level functions and System F. Using type variables that have a contextual kind brings a distinct advantage over System F_ω: as we apply the substitution associated with the type variable `'a` as soon as we know its instantiation, we can compare two types simply by structural equality, and we do not need to reason about type-level computation.

The ability to characterize holes in types and terms is particularly important when we consider pattern matching on code. In our setting, polymorphic programs may contain explicit type applications, and hence, we not only pattern match on terms but also on types.

2.4 Example: Lift Polymorphic Data Structures

In Mœbius, as in other similar frameworks, we need to lift values explicitly to the next stage. Lifting integers is done by simply traversing the input and turning it into its syntax representation. This is straightforward. We write here `⌈int⌉` as an abbreviation for `⌈ ⊢¹ int⌉`, which describes a closed integer at level 1, i.e. the minimum level that generated code can have.

```
lift_int : int → ⌈int⌉
lift_int n = if n = 0 then box(0) else let box(X) = lift_int (n - 1) in box(X + 1)
```

To lift polymorphic lists, we need to lift values of type `'a` to their syntactic representations and then lift lists themselves. This generic lifting function for values of type `'a` will intuitively have the type `'a → ⌈'a⌉`. But how can we ensure that the type variable `'a` can be used inside a contextual type at level 1? – To put it differently, how can we guarantee that the type variable persists when we transition inside the contextual box type? We again use the intuition that *code and types at stage n, have locally bound variables from levels less than n, but they may depend on outer variables that have levels greater or equal to n.* For `⌈'a⌉` to be a well-formed type, the type variable `'a` must be declared at level 1 or higher.

Hence, we declare the type variable `'a` as a type template `'a: ( ⊢ *)`. We omit here again the level 1, since it can be inferred from the surrounding context.

```
lift_list : ('a:(  ⊢ *)) → 'a list → ('a → ⌈'a⌉) → ⌈'a list⌉
lift_list l lift = match l with
| []    → box([])
| x::xs → let box X = lift x in let box XS = lift_list xs lift in box(X::XS)
```

For terms, we have a natural interpretation of running and evaluating code. This rests on the idea that a term at stage 1 can always be used at stage 0.

```
eval_int : ⌈int⌉ → int
eval_int x = let box X = x in X

eval_list : ('a:( ⊢ *)) → ⌈'a list⌉ → 'a list
eval_list v = let box V = v in V
```

Similarly, we can run and evaluate polymorphic code. Again we rely on the idea that the type variable `a` is describing types at level 1 or below.

This concept also allows us to lift the result of a function `rev ('a:*) → 'a list → 'a list`. In the code below, we simply call `rev` with `'a` (or more precisely with `'a with .`).

```
lift_rev_list : ('a:(  ⊢ *)) → 'a list → ('a → ⌈'a⌉) → ⌈'a list⌉
lift_rev_list l = lift_list (rev l)
```

2.5 Example: Multi-Staged Polymorphic Code

We now give an example of multi-staged polymorphic code generation. Here, names `l` and `liftA` in the function type are to help the understanding, and are not part of our syntax. It is a multi-staged version of `map_reduce`. We annotate all the contextual types and kinds with their level, although the level can be inferred from where the type variable is used. It serves as another illustration that the level manages variable dependencies and cross-stage persistence[Taha and Sheard 2000], while the staging manages when code is generated. In the code below, we omit writing the identity substitution that is associated with variables writing for example simply `R` instead of `(R with 'b,f,liftB)`.

```
map_reduce : ('a:( ⊢² *)) → (l:'a list) → (liftA:'a →  ⌈ ⊢² 'a⌉)
          → ⌈'b:( ⊢¹ *), f:'a → 'b, liftB:'b → ⌈ ⊢² 'b⌉
                                    ⊢² ⌈'c:*, g:'b → 'c → 'c, base:'c  ⊢¹ 'c⌉⌉
map_reduce l liftA = match l with
| []    → box ('b,f,liftB. box('c,g,base. base) )
| x::xs →
  let box ('b,f,liftB. R) = map_reduce xs liftA in
  let box A = liftA x in box ('b,f,liftB.
    let box ('c,g,base. M) =  R in
```

```
let box X' = liftB (f A) in box ('c,g,base. M with 'c, g, (g X' base)))
```

Multi-stage code generation generates code incrementally in several stages. For example, `map_reduce` generates the final code in 2 stages. This is evident, since the return type has a boxed type nested inside another boxed type.

In the function declaration of `map_reduce`, we declare `('a:( ⊢² *))`, as we want to use `a` at any stage 2 or below. However, the other declarations `l` and `liftA` are declared at level 0 – they will be inaccessible inside a box-expression. Therefore, this forces us to lift the head of the list (`liftA x`) first before we build our result, as when we enter the scope of `'b:( ⊢¹ *); f:'a → 'b, liftB:'b → ⌈⊢² 'b⌉` we only keep the declaration `('a:( ⊢² *))`, but the outer declarations at level 0 get dropped as they become inaccessible. Similarly, when we enter the inner contextual box `⌈'c:*, g:'b → 'c → 'c, base:'c ⊢¹ 'c⌉`, we will drop all the outer assumptions at level 0 (as they get overwritten by the new local context `'c:*, g:'b → 'c → 'c, base:'c`, but assumptions at levels higher than 0 remain accessible. The levels, therefore, manage scope dependencies – especially as we cross different stages.

2.6 Example: Working with Church Encodings and Pattern Matching

The last example illustrates the benefits of generating code as intended by programmers and the use of pattern matching. We represent the type of natural numbers using Church encoding as `⌈'a:*, x:'a, f:'a → 'a ⊢'a⌉`. However, unlike the usual Church encoding where we use functions and function application, we exploit the power of contextual types instead. This then allows for elegant implementations of addition or other arithmetic operations. We will again omit writing the identity substitutions, i.e. for example, writing simply `N` instead of `(N with 'a,x,f)`.

```
gen_church: int → ⌈'a:*,x:'a, f:'a → 'a ⊢ 'a⌉
gen_church n = if n = 0 then box('a,x,f. x)
               else let box('a,x,f. N) = gen_church (n-1) in box('a,x,f. f N)

add: ⌈'a:*,x:'a, f:'a → 'a ⊢ 'a⌉ → ⌈'a:*,x:'a, f:'a → 'a ⊢ 'a⌉
  → ⌈'a:*,x:'a, f:'a → 'a ⊢ 'a⌉
add n m = let box ('a,x,f. N) = n in
          let box ('a,x,f. M) = m in box('a,x,f. N with 'a, M, f)
```

Working with the Church encoding directly as syntactic representations again allows us to generate code as the programmer intended. For example adding the Church encoding of 2 and the Church encoding of 3 will yield the Church encoding of 5 – not some program that will evaluate to 5 when we run the result. As we know the structure of the Church encoding for numbers, we can also inspect it via pattern matching. This opens new possibilities to implement the predecessor function directly via pattern matching instead of the usual more painful solution when we work with functions and function applications.

```
pred n = case n of
| box('a,x,f. x) → box('a,x,f. x)
| box('a,x,f. f X) → box('a,x,f. X)
```

The first case captures the fact that the input `n` is a representation of 0, and we simply return it. In the second case, we know that it is an application `f X` where `X` is the pattern variable denoting a code template of type `('a:*,x:'a,f:'a → 'a ⊢'a)`. We hence simply strip off the `f`.

We delay a more in-depth discussion of pattern matching on code to Sec. 4.

3 A MULTI-LEVEL MODAL LAMBDA-CALCULUS WITH POLYMORPHISM

We describe first a modal polymorphic lambda-calculus where we use multi-level contextual types to characterize open code. This calculus serves as a foundation to generate polymorphic open code. We then extend this calculus to support pattern matching on polymorphic code in Sec. 4.

$$
\begin{array}{llll}
\text{Types} & T, S & ::= & \alpha[\sigma] \mid T_1 \to T_2 \mid (\alpha{:}(\Psi \vdash^n *)) \to T \mid \lceil \Psi \vdash^n T \rceil \\
\text{Terms} & e & ::= & x[\sigma] \mid \mathsf{fn}\ x \to e \mid e_1\ e_2 \mid \mathsf{Fn}\ \alpha^n \to e \mid e\ (\hat{\Psi}^n.T) \mid \\
 & & & \mathsf{box}(\hat{\Gamma}^n.\ e) \mid \mathsf{let\ box}\ (\hat{\Gamma}^n.u) = e_1\ \mathsf{in}\ e_2 \\
\hline
\text{Substitution} & \sigma & ::= & \cdot \mid \sigma, \hat{\Psi}^n.e \mid \sigma; x^n \mid \sigma, \hat{\Psi}^n.T \mid \sigma; \alpha^n \\
\text{Context} & \Gamma, \Psi, \Phi & ::= & \cdot \mid \Gamma, x{:}(\Psi \vdash^n T) \mid \Gamma, \alpha{:}(\Psi \vdash^n *) \\
\text{Erased context} & \hat{\Gamma}, \hat{\Psi}, \hat{\Phi} & ::= & \cdot \mid \hat{\Gamma}, x^n \mid \hat{\Gamma}, \alpha^n
\end{array}
$$

Fig. 1. Syntax of Multi-Level Modal Lambda-Calculus

3.1 Syntax

We describe the syntax of Mœbius in Fig. 1. Central to Mœbius are multi-level contextual types and kinds that describe code and type templates together with the context in which they are meaningful. A multi-level contextual type $(\Psi \vdash^n T)$ describes code of type T in a context Ψ where all assumptions in Ψ are themselves at levels below n. Variables at level n or above are viewed as global variables whose values will be computed during run-time. Similarly, a multi-level contextual kind $(\Psi \vdash^n *)$ describes a type fragment in a context Ψ.

We treat all variable declarations in our context uniformly; hence our typing context keeps track of variable declarations $x : (\Psi \vdash^n T)$ and type declarations $\alpha : (\Psi \vdash^n *)$. If $n = 0$, then we recover our ordinary bound variables of type T from $x : (\cdot \vdash^0 T)$. As there are no possible declarations at a level below 0, this context is necessarily empty (in fact it doesn't even exist). Similarly, a type declaration $\alpha : (\cdot \vdash^0 *)$ denotes simply a type variable α.

A contextual type or term variable is associated with a substitution, written here as $\alpha[\sigma]$ and $x[\sigma]$. Intuitively, if a contextual variable $x : (\Psi \vdash^n \_)$ is declared in some context Φ, then the substitution σ provides instantiations for variables in Ψ in terms of Φ. As soon as we know the instantiation for x, we apply the substitution σ. We previously wrote `(x with σ)` as concrete source syntax in our code examples in Sec. 2.

Mœbius is a generalization of the modal lambda-calculus described by Davies and Pfenning [2001] or Nanevski et al. [2008] which separates the global assumptions in a meta-context from the local assumptions. This gives us a two-zone representation of the modal lambda-calculus. Mœbius generalizes this work to an n-ary zone representation. If $n = 1$, then we obtain the two-zone representation from the previous work. Variables at level 1 correspond to the meta-variables which live in the global meta-context, and variables at level 0 correspond to the ordinary bound variables.

Our notion of multi-level contextual types is also similar to the work by Boespflug and Pientka [2011]. However, this work was concerned with developing a multi-level contextual logical framework. We adopt the ideas to polymorphic multi-staged programming.

Mœbius supports boxed contextual types, $\lceil \Psi \vdash^n T \rceil$ which describe a code template of type T in the context Ψ. The level n enforces that the given template can only depend locally on variables at levels below n. This will allow us to cleanly manage variable dependencies between stages and will serve as a central guide in the design of Mœbius. In addition, we support function types (written as $T_1 \to T_2$) and polymorphic function $(\alpha{:}(\Psi \vdash^n *)) \to T$ which abstract over type variables α. The

ordinary polymorphic function space is a special case where $n = 0$. As a type variable α stands in general for a type-level template, we associate it with a substitution σ, written as $\alpha[\sigma]$. As soon as we know what type variable α stands for, we splice it in and apply σ. This ensures that the type makes sense in the context where it is used. When $n = 1$, we can model types in System F_ω. For example, in System F_ω, we might define: $(\alpha{:}* \to *) \to (\beta : *) \to \alpha\ \beta$. In Mœbius, we declare it instead as: $(\alpha{:}(\_{:}* \vdash^1 *)) \to (\beta : *) \to \alpha[\beta]$ avoiding type-level functions and creating a type-level function application.

Our core language of Mœbius includes functions ($\mathsf{fn}\ x \to e$), function application ($e_1\ e_2$), type abstractions ($\mathsf{Fn}\ \alpha^n \to e$) and type application ($e\ (\hat{\Psi}^n.T)$). We carry the level n of α as an annotation; this is needed for technical reasons when we define simultaneous substitution operations. Further, we include box and let box expressions to generate code and use code. In particular, $\mathsf{box}(\hat{\Gamma}^n.\ e)$ describes code e together with the list of (local) variables $\hat{\Gamma}^n$ which may be used in e. The level n states again that these bound variables are below level n and we ensure that n here is greater than 0 during typing, as an expression e at level 0 is an ordinary expression, not a code fragment. To use a program e_1 that generates a code template, we bind a contextual variable u to the result of evaluation of e_1 and use the result in a body e_2. This is accomplished via $\mathsf{let\ box}\ (\hat{\Gamma}^n.u) = e_1\ \mathsf{in}\ e_2$.

A substitution is formed by extending a given substitution with a contextual term ($\hat{\Psi}^n.e$), a contextual type ($\hat{\Psi}^n.T$), a term variable x, or a type variable α. The latter extensions are necessary to allow us to treat substitutions as simultaneous substitutions. In particular, when pushing a substitution σ inside a function $\mathsf{Fn}\ \alpha^n \to e$, we need to extend it with a mapping for α. However, we would need to eta-expand α based on its type to obtain a proper contextual object, as α by itself would not be a legitimate type. Therefore, we allow extensions of substitutions with an identity mapping, which is written as $\sigma; \alpha^n$. Similarly, if we push the substitution inside $\mathsf{box}(\hat{\Gamma}^n.\ e)$ then we may need to extend the simultaneous substitution with mappings for the variables listed in $\hat{\Gamma}$, but we lack the type information to expand a term variable x to a proper contextual term. This issue is not new and is handled similarly in Nanevski et al. [2008] in a slightly different setting.

In contrast to the two-zone formulation of the modal lambda-calculus in Nanevski et al. [2008], which has different substitution operations for ordinary bound variables (variables at level 0) and meta-variables (variables at level 1), our substitution encompasses both kinds of substitutions.

Remark 1: We define the level of a context: $\mathsf{level}(\Psi) = n$ iff for all declarations $x{:}(\_ \vdash^k \_)$ in Ψ have $k < n$. In other words, the level of context Ψ ensures that all variables in Ψ are at strictly smaller levels. We often make the level explicit as a superscript, Ψ^n, which describes a context Ψ where $\mathsf{level}(\Psi) = n$. Note that we abuse the functional notation here, since level does not have a unique result for a given context Ψ.

Remark 2: The erasure of type annotations from the context Ψ^n drops all the type and kind declarations from Ψ, but retains the level to obtain $\hat{\Psi}^n$. Abusing notation, we often simply write $\hat{\Psi}^n$ for the erasure of type annotations from the context Ψ^n.

Remark 3: We write $\mathsf{id}(\hat{\Phi})$ for the identity substitution that has domain Φ. It is defined as follows:

$$\begin{array}{lcl} \mathsf{id}(\cdot) & = & \cdot \\ \mathsf{id}(\hat{\Phi},\ x^n) & = & \mathsf{id}(\hat{\Phi}); x^n \\ \mathsf{id}(\hat{\Phi},\ \alpha^n) & = & \mathsf{id}(\hat{\Phi}); \alpha^n \end{array}$$

Before moving to the typing rules, we take a closer look first at our typing contexts and then at our substitutions, because they are both key parts of the design of our language and they hopefully give a good general intuition about the way the system works.

$$
\begin{array}{lcll}
\text{Merging contexts:} \quad \Psi \oplus \Phi = \Gamma & & & \\
\cdot \oplus \Phi & = & \Phi & \\
\Psi \oplus \cdot & = & \Psi & \\
\Psi, x{:}(\Gamma \vdash^n K) \oplus \Phi, y{:}(\Gamma' \vdash^k K') & = & (\Psi, x{:}(\Gamma \vdash^n K) \oplus \Phi), y{:}(\Gamma' \vdash^k K') & \text{if } k \leq n \\
\Psi, x{:}(\Gamma \vdash^n K) \oplus \Phi, y{:}(\Gamma' \vdash^k K') & = & (\Psi \oplus \Phi, y{:}(\Gamma' \vdash^k K')), x{:}(\Gamma \vdash^n K) & \text{otherwise} \\
\\
\text{Chopping lower context:} \quad \Psi|_n \; = \; \Phi & & & \\
(\cdot)|_n & = & \cdot & \\
(\Psi, x{:}(\Phi \vdash^k K))|_n & = & \Psi|_n & \text{if } k < n \\
(\Psi, x{:}(\Phi \vdash^k K))|_n & = & \Psi, x{:}(\Phi \vdash^k K) & \text{otherwise}
\end{array}
$$

Fig. 2. Merging and chopping of contexts

3.2 Context Operations

First, we note that we maintain order in a context and hence contexts must be sorted according to the level of assumptions $x{:}(\Psi \vdash^n T)$. One should conceptualize this ordered context as a stack of subcontexts, one for each level of variables. Let $\Psi(k)$ be the subcontext of Ψ^n with only assumptions of level k. Then, $\Psi^n = \Psi(n-1), \Psi(n-2), \ldots, \Psi(1), \Psi(0)$. Slightly abusing notation we write $\Psi(i), \Psi(i-1)$ for appending the context $\Psi(i)$ and the context $\Psi(i-1)$. Note that $\Psi(0)$ contains only declarations of type $(\cdot \vdash^0 T)$ and we recover our ordinary typing context which simply contains variable declarations $x{:}T$. The context Ψ^n is essentially an n-ary zone generalization of the two-zone context present in Davies and Pfenning [2001] work. We can recover their two zones when $n = 1$. We opt here for a flattened presentation of the n-ary zones of contexts in order to simplify operations on contexts. Keeping the context sorted not only provides conceptual guidance, but also helps us to handle cleanly the dependencies among variable declarations.

Keeping the context sorted comes at a cost: extending a context must preserve this invariant. However, restricting a context, which is necessary as we move up one stage, i.e. we move inside a box, is simpler. We therefore define and explain two main operations on context: restricting a context and merging two contexts. To chop off all variables below level n from an ordered context Γ, we write $\Gamma|_n$. To merge two ordered contexts Γ and Ψ we write $\Gamma \oplus \Psi$. With merging defined, insertion of a new assumption into a context is a special case. For compactness, we write K for a type or a kind in the definition of the context operations in Fig. 2.

The chopping operation for the lower context allows us to drop all variable assumptions below the given level from a context. If $k \leq n$, then $(\Psi^k)|_n = \cdot$.

Merging of two independent, sorted contexts is akin to the merge step of the merge-sort algorithm and therefore inherits many of its properties. In particular, the merge of two sorted independent contexts is again a sorted context. It is also stable, in the sense that the relative positions of any two assumptions in Ψ^n or in Φ^k is preserved in $\Psi^n \oplus \Phi^k$.

When we merge two sorted contexts Ψ and Φ, i.e. when for every declaration $x{:}(\_ \vdash^n \_)$ in Ψ, we have $\text{level}(\Phi) \leq n + 1$, we simply write Ψ, Φ (see [Jang et al. 2021] for the definition). While defining a separate append operation on sorted contexts is not necessary, it is often conceptually easier to read and use. It is also more concise.

3.3 Simultaneous Substitution Operation

We concentrate here on the simultaneous substitution operation, but the single substitution operation follows similar ideas.

In our grammar, simultaneous substitutions are defined without their domain. However, when applying a simultaneous substitution σ, we can always recover its domain $\hat{\Psi}$. The idea is that σ provides instantiations for all variables in $\hat{\Psi}$ where $\text{level}(\hat{\Psi}) = n$. All variables at levels greater or equal to n are treated as global variables and are untouched by the substitution operation. We note that since contexts are ordered, simultaneous substitutions are also ordered.

We describe here in detail applying a substitution to a type $[\sigma/\hat{\Phi}]T$. Note that the substitution operation is written in prefix. This is in contrast to the closure of a variable with a substitution which is written in postfix (see $x[\sigma]$ and $\alpha[\sigma]$ resp.). Due to space, we omit the definition for $[\sigma/\hat{\Phi}]e$ which applies the substitution σ to a term e, composing substitutions, and applying a substitution to a context. They can be found in [Jang et al. 2021]. In the definition of these substitution operations, we rely on chopping, merging, and appending simultaneous substitutions, written $(\sigma/\hat{\Phi})|_k$, $(\sigma/\hat{\Phi}) \oplus (\sigma'/\hat{\Psi})$ and $(\sigma/\hat{\Phi}), (\sigma'/\hat{\Psi})$. Those operations correspond to the equivalent context operations and are also defined in [Jang et al. 2021].

$$
\begin{array}{lll}
[\sigma/\hat{\Psi}](\alpha[\sigma']) & = \alpha[\sigma''] & \alpha \notin \hat{\Psi} \text{ and } [\sigma/\hat{\Psi}]\sigma' = \sigma'' \\
[\sigma/\hat{\Psi}](\alpha[\sigma']) & = T' & \text{lkp}(\sigma/\hat{\Psi})\ \alpha = (\hat{\Phi}^n.T) \text{ and } \text{level}(\hat{\Psi}) > n \\
 & & \text{and } [\sigma/\hat{\Psi}]\sigma' = \sigma'' \text{ and } [\sigma''/\hat{\Phi}]T = T' \\
[\sigma/\hat{\Psi}](\alpha[\sigma']) & = \beta[\sigma''] & \text{lkp}(\sigma/\hat{\Psi})\ \alpha = \beta^n \text{ and } \text{level}(\hat{\Psi}) > n \text{ and } [\sigma/\hat{\Psi}]\sigma' = \sigma'' \\
[\sigma/\hat{\Psi}](T \to S) & = T' \to S' & [\sigma/\hat{\Psi}]T = T' \text{ and } [\sigma/\hat{\Psi}]S = S' \\
[\sigma/\hat{\Psi}]((\alpha{:}(\Phi \vdash^n *)) \to T) & = ((\alpha{:}(\Phi \vdash^n *)) \to T') & \text{level}(\hat{\Psi}) > n \text{ and } [(\sigma/\hat{\Psi}) \oplus (\alpha/\alpha^n)]T = T' \\
[\sigma/\hat{\Psi}]((\alpha{:}(\Phi \vdash^n *)) \to T) & = ((\alpha{:}(\Phi \vdash^n *)) \to T') & \text{level}(\hat{\Psi}) \leq n \text{ and } [\sigma/\hat{\Psi}]T = T' \\
[\sigma/\hat{\Psi}](\lceil \Phi \vdash^n T \rceil) & = \lceil \Phi' \vdash^n T' \rceil & \text{level}(\hat{\Psi}) \geq n \text{ and } (\sigma/\hat{\Psi})|_n = \sigma'/\hat{\Psi}' \text{ and} \\
 & & [\sigma'/\hat{\Psi}']\Phi = \Phi' \text{ and } [(\sigma'/\hat{\Psi}), (\text{id}(\hat{\Phi})/\hat{\Phi})]T = T' \\
[\sigma/\hat{\Psi}](\lceil \Phi \vdash^n T \rceil) & = \lceil \Phi \vdash^n T \rceil & \text{level}(\hat{\Psi}) < n
\end{array}
$$

Fig. 3. Simultaneous Substitution Operation for Types: $[\sigma/\hat{\Psi}]T = S$

The substitution operation $[\sigma/\hat{\Psi}]T$ is then mostly straightforward and applied recursively to T. In the variable case $\alpha[\sigma']$, we distinguish between three cases:

If α is not in $\hat{\Psi}$, then α denotes a "global" variable; we leave α untouched and only apply σ to σ'.

If α is in $\hat{\Psi}$, then we have either a corresponding instantiation $(\hat{\Phi}^n.T)/\alpha^n$ or simply β^n for α in $\sigma/\hat{\Psi}$ (see [Jang et al. 2021] for the definition of the lookup operation, $\text{lkp}((\sigma/\hat{\Psi}))\ \alpha$). In both cases, we apply the simultaneous substitution σ to σ' giving us some substitution σ''. In the former case (where $(\hat{\Phi}^n.T)/\alpha^n$), we now apply the substitution σ'' to T. Applying the substitution will terminate, as σ'' provides instantiations for variables at lower levels than n. In the latter case (where β^n/α^n), we simply pair β with σ'' creating a new closure.

For polymorphically quantified types, we push the substitution inside and extend the substitution with the identity, if its level n satisfies $n < \text{level}(\hat{\Psi})$. Although we quantify over (contextual) type variables that have type $(\Phi \vdash^n *)$, we do not have to apply the substitution to Φ itself, as Φ is a context containing only type variable declarations[1]. For boxed types such as $\lceil \Phi \vdash^n T \rceil$, we push σ inside the box, if $\text{level}(\hat{\Psi}) \geq n$. To do this, we first drop all the mappings for variables below n from the substitution σ, since those variables will be replaced by Φ. Then, we apply $\sigma'/\hat{\Psi}'$ to Φ (see

[1]Generalizing this to contain both type and term variable declarations is straightforward and follows the similar principle as in the boxed type case where we apply to Φ the substitution $(\sigma/\hat{\Psi})|_n$.

[Jang et al. 2021]for the definition). Further, we extend σ' with the identity mapping for variables in Φ before applying it to T. If $\text{level}(\hat{\Psi}) \leq n$, then we do not apply the substitution to Φ or T, since it concerns variables that will be replaced by Φ and those variables are locally bound.

Theorem 3.1 (Termination). *The substitution operation $[\sigma/\hat{\Phi}]T$, $[\sigma/\hat{\Phi}]e$, and $[\sigma/\hat{\Phi}]\sigma'$ terminates.*

Proof. By induction on the $\text{level}(\hat{\Phi})$ and structure of T and e and σ'. Either $\text{level}(\hat{\Phi})$ stays the same and T (resp. e or σ) is decreasing or $\text{level}(\hat{\Phi})$ is decreasing. □

3.4 Typing Rules

With the context and substitution operations in place, we can now define the typing rules for the multi-level contextual modal lambda-calculus in an elegant way. The levels provide us with enough structure to keep track of the scope of variables, terms and types.

To distinguish the notation for contextual types, written as $\lceil \Psi \vdash^n T \rceil$ from the typing judgment, we use $\Vdash$ for all the typing judgments.

We begin with defining well-formed contexts. A context $\Gamma, x{:}(\Phi \vdash^n T)$ is well-formed, if Γ is well-formed, Φ is well-formed with respect to $\Gamma|_n$, and T is well-kinded in the context $(\Gamma|_n, \Phi)$.

Well-formed contexts: $\boxed{\Vdash \Gamma}$

$$\frac{\Vdash \Gamma \quad \text{level}(\Phi) \leq n \quad \Vdash \Gamma|_n, \Phi \quad \Gamma|_n, \Phi \Vdash T}{\Vdash \Gamma, x{:}(\Phi \vdash^n T)} \qquad \frac{\Vdash \Gamma \quad \text{level}(\Phi) \leq n \quad \Vdash \Gamma|_n, \Phi}{\Vdash \Gamma, \alpha{:}(\Phi \vdash^n *)} \qquad \frac{}{\Vdash \cdot}$$

With the context operations in place, the kinding rules for types are straightforward.

Kinding rules for types: $\boxed{\Gamma \Vdash T}$

$$\frac{\Gamma(\alpha) = (\Phi \vdash^n *) \quad \Gamma \Vdash \sigma : \Phi}{\Gamma \Vdash \alpha[\sigma]}$$

$$\frac{\Gamma \Vdash S \quad \Gamma \Vdash T}{\Gamma \Vdash S \to T} \qquad \frac{\Vdash \Gamma|_n, \Phi \quad \Gamma \oplus \alpha{:}(\Phi \vdash^n *) \Vdash T}{\Gamma \Vdash (\alpha{:}(\Phi \vdash^n *)) \to T} \qquad \frac{\Vdash \Gamma|_n, \Phi \quad \Gamma|_n, \Phi \Vdash T}{\Gamma \Vdash \lceil \Phi \vdash^n T \rceil} \; n > 0$$

Type variables $\alpha[\sigma]$ where $\alpha : (\Phi \vdash^n *)$ in Γ are well-kinded, if the associated substitution σ maps variables from Φ to the present context Γ. Function types, $S \to T$ are well-kinded, if both S and T are well-kinded. When we check that $(\alpha{:}(\Phi \vdash^n *)) \to T$ is well-kinded in a context Γ, we ensure that Φ is well-formed with respect to $\Gamma|_n$, as the local context Φ will replace all the variables at levels below n in the original Γ; this is accomplished by $\Vdash \Gamma|_n, \Phi$. We then check that T is well-formed, in the context Γ extended with the assumption $\alpha{:}(\Phi \vdash^n *)$ where the assumption is inserted at the appropriate position in Γ. For the kinding of $\lceil \Phi \vdash^n T \rceil$, we proceed similarly. We again replace all the variables at levels below n in the original Γ with Φ using the previously defined context operations.

Using the context operations for restricting and extending contexts, the typing rules for expressions (Fig. 4) are mostly straightforward and follow the general principle that we have seen before. To show that a function $\text{fn } x \to e$ has type $S \to T$, we extend the context Γ with the declaration $x{:}(\cdot \vdash^0 S)$ and check that the body e has type T. As we remarked before, no typing context with assumptions below level 0 exists, and this declaration can be viewed as $x{:}T$. However, giving it level 0 allows for a uniform treatment. To show that a type abstraction, $\text{Fn } \alpha^n \to e$, has the appropriate type, we extend the context with type variable declaration for α.

Applications, $e_1\ e_2$ are defined as expected. For type applications, $e\ (\hat{\Phi}^n.T)$, we need to be a bit more careful: in addition to verifying that e has type $(\alpha{:}(\Phi \vdash^n *)) \to S$, we again verify that T is well-kinded in a new context, where we replace the assumptions at level below n in Γ with the

$$\frac{\Gamma(x) = (\Phi \vdash^n T) \quad \Gamma \Vdash \sigma : \Phi}{\Gamma \Vdash x[\sigma] : [\sigma/\hat{\Phi}]T} \qquad \frac{\Gamma, x{:}(\cdot \vdash^0 S) \Vdash e : T}{\Gamma \Vdash \mathsf{fn}\ x \to e : S \to T} \qquad \frac{\Gamma \Vdash e_1 : S \to T \quad \Gamma \Vdash e_2 : S}{\Gamma \Vdash e_1\ e_2 : T}$$

$$\frac{\Gamma \oplus \alpha{:}(\Phi \vdash^n *) \Vdash e : T}{\Gamma \Vdash \mathsf{Fn}\ \alpha^n \to e : (\alpha{:}(\Phi \vdash^n *)) \to T} \qquad \frac{\Gamma \Vdash e : (\alpha{:}(\Phi \vdash^n *)) \to S \quad \Gamma|_n, \Phi \Vdash T}{\Gamma \Vdash e\ (\hat{\Phi}^n.T) : [\hat{\Phi}^n.T/\alpha^n]S}$$

$$\frac{\Gamma|_n, \Phi \Vdash e : T}{\Gamma \Vdash \mathsf{box}(\hat{\Phi}^n.\, e) : \lceil \Phi \vdash^n T \rceil} \qquad \frac{\Gamma \Vdash e_1 : \lceil \Phi \vdash^n S \rceil \quad \Gamma \oplus u{:}(\Phi \vdash^n S) \Vdash e_2 : T}{\Gamma \Vdash \mathsf{let\ box}\ (\hat{\Phi}^n.u) = e_1\ \mathsf{in}\ e_2 : T}$$

Fig. 4. Typing rules for expressions $\boxed{\Gamma \Vdash e : T}$

$$\frac{}{\Gamma \Vdash \cdot : \cdot} \qquad \frac{\Gamma \Vdash \sigma : \Gamma' \quad (\sigma/\hat{\Gamma}')|_n = (\sigma'/\hat{\Gamma}'') \quad \Gamma|_n, [\sigma'/\hat{\Gamma}'']\Psi \Vdash e : [(\sigma'/\hat{\Gamma}''), (\mathsf{id}(\hat{\Psi})/\hat{\Psi})]T}{\Gamma \Vdash (\sigma, \hat{\Psi}^n.e)\ :\ (\Gamma', x{:}(\Psi \vdash^n T))}$$

$$\frac{\Gamma \Vdash \sigma : \Gamma' \quad (\sigma/\hat{\Gamma}')|_n = (\sigma''/\hat{\Gamma}'') \quad \Gamma(x) = ([\sigma''/\hat{\Gamma}'']\Psi \vdash^n [\sigma''/\hat{\Gamma}'']T)}{\Gamma \Vdash (\sigma; x)\ :\ (\Gamma', x{:}(\Psi \vdash^n T))}$$

Fig. 5. Typing rules for substitutions $\boxed{\Gamma \Vdash \sigma : \Phi}$

declarations from Φ. To accomplish this, we again rely on our context operations, first restricting Γ and then appending Φ. As the type of e will be polymorphic, namely $(\alpha{:}(\Phi \vdash^n *)) \to S$, we return as the type of $e\ (\hat{\Phi}^n.T)$ the type S where we have replaced α with $(\hat{\Phi}^n.T)$.

The rules for box and letbox again appropriately restrict and extend Γ based on the level where $n > 0$. This side condition simply allows us to distinguish between code, i.e. terms that are boxed and represent syntax, and programs, i.e. terms that will be evaluated and run.

A substitution $(\sigma, \hat{\Psi}^n.e)$ provides a mapping from $(\Gamma', x{:}(\Psi \vdash^n T))$ to the context Γ, if σ maps variables from Γ' to Γ and if e is well-typed. As the type $(\Psi \vdash^n T)$ is well-formed with respect to Γ', we will need to transport both Ψ and T to Γ. Since Ψ depends only on $\Gamma'|_n$, we only need to apply the restricted substitution $(\sigma/\hat{\Gamma}')|_n = (\sigma''/\hat{\Gamma}'')$ to Ψ. Since we work with simultaneous substitution, we need to extend this restricted substitution with the identity mapping for Ψ, before we can apply it to the type T and effectively move the type to the new context $(\Gamma|_n, [\sigma''/\hat{\Gamma}'']\Psi)$ in which e will be well-typed. The cases where we extend a substitution with a variable (i.e. $\sigma; x^n$) is a special case of the previous cases. The extension of a substitution with a type $(\hat{\Psi}^n.T)$ or a type variable α^n follows the similar idea and are a special case of handling the term extension. We omit them here due to space, but their definition is given in [Jang et al. 2021].

3.5 Substitution Properties

We now establish substitution properties. Recall that a context $\Psi^k = \Psi(k{-}1), \Psi(k{-}2), \ldots, \Psi(1), \Psi(0)$ where $\Psi(j)$ is the subcontext of Ψ^k with only assumptions of level j. For easier readability, we will simply write $\Gamma_1, \alpha{:}(\Phi \vdash^n), \Gamma_0$ for a context where all assumptions in Γ_1 are at level n or above and all assumptions in Γ_0 are at level n or below.

We first state that the generation of identity substitutions yields well-typed substitutions.

Lemma 3.2 (Identity Substitution). $\Gamma, \Phi \Vdash \mathsf{id}(\hat{\Phi}) : \Phi$

Proof. By induction on Φ. □

Next, we generalize a property that already exists in Nanevski et al. [2008] and Pientka [2003]. In this previous work, it states that the substitution for locally bound variables and the substitution operation for meta-variables commute. Here we state more generally that substitution for a variable with level m and substitution for a variable with level n commute, as long as m is greater or equal to n.

Lemma 3.3 (Commuting Substitutions).

(1) If $m \geq n$, *then* $[(\hat{\Psi}^m.S_2)/\beta^m][(\hat{\Phi}^n.S_1)/\alpha^n]T = [(\hat{\Phi}^n.[(\hat{\Psi}^m.S_2)/\beta^m]S_1)/\alpha^n][(\hat{\Psi}^m.S_2)/\beta^m]T$.
(2) If $m \geq n$, *then* $[(\hat{\Psi}^m.e_2)/y^m][(\hat{\Phi}^n.e_1)/x^n]e_0 = [(\hat{\Phi}^n.[(\hat{\Psi}^m.e_2)/y^m]e_1)/x^n][(\hat{\Psi}^m.e_2)/y^m]e_0$.

Proof. By induction on T and e_0. □

Last, we state the substitution properties for type variables, term variables and simultaneous substitutions. Since the single substitution operation relies on simultaneous substitution, we prove them all mutually. However, for clarity, we state them separately.

Lemma 3.4 (Type Substitution Lemma). *Assuming* $\Vdash \Gamma_1, \alpha{:}(\Phi \vdash^n *), \Gamma_0$ *and* $\Gamma_1, \Phi \Vdash T$.

(1) Then $\Vdash \Gamma_1, [(\hat{\Phi}^n.T)/\alpha^n]\Gamma_0$.
(2) If $\Gamma_1, \alpha{:}(\Phi \vdash^n *), \Gamma_0 \Vdash S$ *then* $\Gamma_1, [(\hat{\Phi}^n.T)/\alpha^n]\Gamma_0 \Vdash [(\hat{\Phi}^n.T)/\alpha^n]S$.
(3) If $\Gamma_1, \alpha{:}(\Phi \vdash^n *), \Gamma_0 \Vdash e_2 : S$ *then* $\Gamma_1, [(\hat{\Phi}^n.T)/\alpha^n]\Gamma_0 \Vdash [(\hat{\Phi}^n.T)/\alpha^n]e_2 : [(\hat{\Phi}^n.T)/\alpha^n]S$.
(4) If $\Gamma_1, \alpha{:}(\Phi \vdash^n *), \Gamma_0 \Vdash \sigma : \Psi$ *then* $\Gamma_1, [(\hat{\Phi}^n.T)/\alpha^n]\Gamma_0 \Vdash [(\hat{\Phi}^n.T)/\alpha^n]\sigma : [(\hat{\Phi}^n.T)/\alpha^n]\Psi$.

Lemma 3.5 (Term Substitution Lemma). *Assuming* $\Vdash \Gamma_1, u{:}(\Phi \vdash^n T), \Gamma_0$ *and* $\Gamma_1, \Phi \Vdash e : T$.

(1) If $\Gamma_1, u{:}(\Phi \vdash^n T), \Gamma_0 \Vdash e_2 : S$ *then* $\Gamma_1, \Gamma_0 \Vdash [(\hat{\Phi}^n.e)/u]e_2 : S$.
(2) If $\Gamma_1, u{:}(\Phi \vdash^n T), \Gamma_0 \Vdash \sigma : \Psi$ *then* $\Gamma_1, \Gamma_0 \Vdash [(\hat{\Phi}^n.e)/u]\sigma : \Psi$.

Lemma 3.6 (Simultaneous Substitution Lemma). *Assuming* $\Gamma_1, \Phi \Vdash \sigma : \Gamma_0$.

(1) If $\Vdash \Gamma_1, \Gamma_0, \Psi$ *then* $\Vdash \Gamma_1, \Phi, ([\sigma/\hat{\Gamma}_0]\Psi)$.
(2) If $\Gamma_1, \Gamma_0 \Vdash S$ *then* $\Gamma_1, \Phi \Vdash [\sigma/\hat{\Gamma}_0]S$.
(3) If $\Gamma_1, \Gamma_0 \Vdash e : S$ *then* $\Gamma_1, \Phi \Vdash [\sigma/\hat{\Gamma}_0]e : [\sigma/\hat{\Gamma}_0]S$.
(4) If $\Gamma_1, \Gamma_0 \Vdash \sigma_2 : \Psi$ *then* $\Gamma_1, \Phi \Vdash [\sigma/\hat{\Gamma}_0]\sigma_2 : [\sigma/\hat{\Gamma}_0]\Psi$.

Proof. By induction on the first derivation. For more details, see [Jang et al. 2021]. □

3.6 Local Soundness and Completeness

With the substitution properties in place, we establish local soundness and completeness properties of our calculus. Local soundness guarantees that our typing rules are not too strong, i.e. they do not allow us to infer more than we should. Dually, local completeness guarantees that the rules are sufficiently strong.

The local soundness property also gives natural rise to reduction rules in our operational semantics and can be viewed as showing that types are preserved during reduction.

Function type. For local soundness, we refer to the term substitution lemma 3.5 to obtain $\mathcal{D}'$.

Local Soundness:

$$\frac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \Gamma, x{:}(\cdot \vdash^0 T) \Vdash e & \Gamma \Vdash e' : T\end{array}}{\Gamma \Vdash (\mathsf{fn}\ x \to e)\ e' : T \to S} \implies \begin{array}{c}\mathcal{D}' \\ \Gamma \Vdash [(\cdot.e')/x^0]e : S\end{array}$$

Local completeness derivation follows directly from the given typing rules.

Local Completeness:

$$\dfrac{\mathcal{D}}{\Gamma \Vdash e : T \to S} \implies \dfrac{\dfrac{\mathcal{D}}{\Gamma, x{:}(\cdot \vdash^0 T) \Vdash e : T \to S} \quad \dfrac{\overline{\Gamma, x{:}(\cdot \vdash^0 T) \Vdash \cdot : \cdot}}{\Gamma, x{:}(\cdot \vdash^0 T) \Vdash x[\cdot] : T}}{\dfrac{\Gamma, x{:}(\cdot \vdash^0 T) \Vdash e\ x[\cdot] : S}{\Gamma \Vdash \mathsf{fn}\ x \to e\ x[\cdot] : T \to S}}$$

Polymorphic type. Local soundness and completeness for the polymorphic type are similar.

Local Soundness:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \oplus \alpha{:}(\Phi \vdash^n *) \Vdash e : S}}{\Gamma \Vdash \mathsf{Fn}\ \alpha^n \to e : (\alpha{:}(\Phi \vdash^n *)) \to S} \quad \dfrac{\mathcal{E}}{\Gamma|_n, \Phi \Vdash T}}{\Gamma \Vdash (\mathsf{Fn}\ \alpha^n \to e)\ (\hat{\Phi}^n.T) : [\hat{\Phi}^n.T/\alpha^n]S} \implies \dfrac{\mathcal{D}'}{\Gamma \Vdash [(\hat{\Phi}^n.T)/\alpha^n]e : [\hat{\Phi}^n.T/\alpha^n]S}$$

We note that $\Gamma \oplus \alpha{:}(\Phi \vdash^n *)$ results in an ordered context of the form $\Gamma_1, \alpha{:}(\Phi \vdash^n *), \Gamma_0$ where all declarations in Γ_1 are at level n or above and all declarations in Γ_0 are below n. The derivation $\mathcal{D}'$ is then obtained using the type substitution lemma 3.4.

Local Completeness:

$$\dfrac{\mathcal{D}}{\Gamma \Vdash e : (\alpha{:}(\Phi \vdash^n *)) \to S} \implies \dfrac{\dfrac{\mathcal{D}}{\Gamma \oplus \alpha{:}(\Phi \vdash^n *) \Vdash e : (\alpha{:}(\Phi \vdash^n *)) \to S} \quad \dfrac{\overline{\Gamma|_n, \Phi \Vdash \mathsf{id}(\hat{\Phi}) : \Phi}}{\Gamma|_n, \Phi \Vdash \alpha[\mathsf{id}(\hat{\Phi})]}}{\dfrac{\Gamma \oplus \alpha{:}(\Phi \vdash^n *) \Vdash e\ (\hat{\Phi}^n.\alpha[\mathsf{id}(\hat{\Phi})]) : S}{\Gamma \Vdash \mathsf{Fn}\ \alpha^n \to e\ (\hat{\Phi}^n.\alpha[\mathsf{id}(\hat{\Phi})]) : (\alpha{:}(\Phi \vdash^n *)) \to S}}$$

We use the identity substitution lemma 3.2 to justify the derivation $\Gamma|_n, \Phi \Vdash \alpha[\mathsf{id}(\hat{\Phi})]$ and exploit the fact that $[(\hat{\Phi}^n.\alpha[\mathsf{id}(\hat{\Phi})])/\alpha^n]S = S$.

Multi-level contextual type. Local soundness and completeness follows similar ideas. For local soundness, we note that $\Gamma \oplus u{:}(\Phi \vdash^n T)$ results in an ordered context Γ' where the declaration $u{:}(\Phi \vdash^n T)$ is inserted at the appropriate level. The derivation $\mathcal{D}'$ is then obtained by referring to the term substitution lemma 3.5 using $(\hat{\Phi}^n.e)$ for u, $\mathcal{E}$, and $\mathcal{D}$.

Local Soundness:

$$\dfrac{\dfrac{\dfrac{\mathcal{E}}{\Gamma|_n, \Phi \Vdash e : T}}{\Gamma \Vdash \mathsf{box}(\hat{\Phi}^n.\, e) : \lceil \Phi \vdash^n T \rceil} \quad \dfrac{\mathcal{D}}{\Gamma \oplus u{:}(\Phi \vdash^n T) \Vdash e_2 : S}}{\Gamma \Vdash \mathsf{let\,box}\ (\hat{\Phi}^n.u) = \mathsf{box}(\hat{\Phi}^n.\, e)\ \mathsf{in}\ e_2 : S} \implies \dfrac{\mathcal{D}'}{\Gamma \Vdash [(\hat{\Phi}^n.e)/u^n]e_2 : S}$$

For local completeness, we again use Lemma 3.2 for identity substitutions to justify the derivation $(\Gamma \oplus u{:}(\Phi \vdash^n T))|_n, \Phi \Vdash u[\mathsf{id}(\hat{\Phi})] : T$.

Local Completeness:

$$\dfrac{\mathcal{D}}{\Gamma \Vdash e : \lceil \Phi \vdash^n T \rceil} \implies \dfrac{\dfrac{\mathcal{D}}{\Gamma \Vdash e : \lceil \Phi \vdash^n T \rceil} \quad \dfrac{\dfrac{\overline{(\Gamma \oplus u{:}(\Phi \vdash^n T))|_n, \Phi \Vdash \mathsf{id}(\hat{\Phi}) : \Phi}}{(\Gamma \oplus u{:}(\Phi \vdash^n T))|_n, \Phi \Vdash u[\mathsf{id}(\hat{\Phi})] : T}}{\Gamma \oplus u{:}(\Phi \vdash^n T) \Vdash \mathsf{box}(\hat{\Phi}.\, u[\mathsf{id}(\hat{\Phi})]) : \lceil \Phi \vdash^n T \rceil}}{\Gamma \Vdash \mathsf{let\,box}\ (\hat{\Phi}^n.u) = e\ \mathsf{in}\ \mathsf{box}(\hat{\Phi}.\, u[\mathsf{id}(\hat{\Phi})]) : \lceil \Phi \vdash^n T \rceil}$$

4 EXTENSION TO PATTERN MATCHING

We now extend the multi-level programming foundation to support pattern matching on code, i.e. expressions of type $\lceil \Gamma \vdash^n T \rceil$. This generalizes the expressions $\mathsf{let\,box}\ (\hat{\Gamma}^n.u) = e_1\ \mathsf{in}\ e_2$ to support not only binding of code to a variable u, but also inspecting this piece of code by pattern matching. In general, one may view the expression $\mathsf{let\,box}\ (\hat{\Gamma}^n.u) = e_1\ \mathsf{in}\ e_2$, as a match on e_1 where

the pattern consists only of a pattern variable u and e_2 is the branch of the case-expressions. We view code patterns as higher-order abstract syntax trees; this abstraction allows us to choose any encoding internally in the implementation.

Adding pattern matching on code in a type-safe manner has been a challenge for two reasons:

1) Representation of Code and Refinement of Types. In general, we want to pattern match on $\lceil \Gamma \vdash^n \alpha \rceil$ (where we omit the identity substitution that is associated with the type variable α for better readability). As a consequence, a pattern may impose some constraints on α. For example, a code pattern $\mathsf{box}(\hat{\Gamma}.\, \mathsf{fn}\ x \to p)$ which has type $\lceil \Gamma \vdash^n \beta_1 \to \beta_2 \rceil$ generates the constraint that $\alpha := \beta_1 \to \beta_2$. A similar situation exists in dependently typed systems. Hence, it is not surprising that constraints arise.

2) Characterizing pattern variables at different stages. A key question is what type to assign to pattern variables in code and type patterns. When we capture a code pattern $\mathsf{box}(\hat{\Gamma}.\, \mathsf{fn}\ x \to p)$, it seems natural that we assign p the type β_2 in the context Γ extended with the declaration for x of type β_1. This follows, for example, the approach taken in the proof environment Beluga[Pientka 2008; Pientka and Cave 2015; Pientka and Dunfield 2010] where the pattern variable p depends on the bound variables from $\hat{\Gamma}$ extended with x. However, in our setting, we want to also allow code patterns of the form $\mathsf{box}(\hat{\Gamma}.\, \mathsf{let\ box}\ (\hat{\Phi}^k.u) = p\ \mathsf{in}\ q)$. As a consequence, we need to capture the type of q, which may depend not only on the variables from $\hat{\Gamma}$, but also on the variable u. Or we might want to allow code patterns that themselves contain code! For example, $\mathsf{box}(\hat{\Gamma}.\, \mathsf{box}(\hat{\Phi}.\, p))$. This is where our multi-level context approach pays off. Since we associate every variable with a level, we simply view q in the extended context with the variable u where u has been inserted at the appropriate position. Similarly, in the code pattern $\mathsf{box}(\hat{\Gamma}^k.\, \mathsf{box}(\hat{\Phi}^n.\, p))$, the levels associated with $\hat{\Gamma}$ and $\hat{\Phi}$ give us the structure to determine in which context p is making sense.

These two considerations lead us to generalize our term language and our contexts. In particular, we track two kinds of constraints: the first one $\alpha := (\Psi \vdash^n T)$ refines a type declaration $\alpha : (\Psi \vdash^*)$ and the second one # denotes an inconsistent set of constraints. The latter allows for elegant handling of impossible cases that may arise. We note that these constraints arise only during typing, and programmers will write only pure contexts, i.e. contexts without any constraints.

$$
\begin{array}{llll}
\text{Terms} & e, p, q & ::= & \ldots \mid \mathsf{case}_{\lceil \Phi \vdash^k T \rceil}\ e\ \mathsf{of}\ \overrightarrow{(\Psi_i.(\hat{\Phi}_i.p_i) : (\Phi_i \vdash^k T_i) \to e_i)} \\
\text{Context} & \Gamma, \Psi, \Phi & ::= & \ldots \mid \Gamma, \alpha := (\hat{\Psi}^n.T) : (\Psi \vdash^n *) \mid \Gamma, \#
\end{array}
$$

For simplicity and clarity, we define pattern matching and branches in a case-expression in such a way that all pattern variables are explicitly listed as part of the branch in the context Ψ_i. The context Φ_i contains all the locally bound variables that may be used in the pattern p_i. We concentrate here on patterns that are terms, but not case-expressions themselves to explain the main ideas. In practice, we would infer the type associated with those pattern variables. We note that for all variables $x{:}(\_ \vdash^n \_)$ in Ψ_i, we have that $n > \mathsf{level}(\Phi_i)$, since pattern variables abstract over holes/variables in a code pattern. Further, these pattern variables themselves may depend on the context Φ_i where all variables in Φ_i denote bound variables within the code pattern and hence are variables defined at levels higher than $n+1$. We also add a type annotation $(\Phi_i \vdash^k T_i)$ to each pattern which states that the pattern p_i has type T_i in the context Ψ_i, Φ_i.

Last but not least, we add a type annotation to the case-expression itself. It describes the type of the scrutinee e. This is used during run-time where we first find the compatible branch by matching the type of the scrutinee against one of type annotations in branches and subsequently match the scrutinee against the pattern in the selected branch. Before giving a more detailed explanation of the static and dynamic semantics for case-expressions and turning our previous kinding and

typing rules from Sec. 3.4 into shallow pattern rules (see Sec. 4.4), we consider the generation and handling of type constraints next.

4.1 Typing Modulo

To handle constraints during type checking, we extend our definition of well-formed contexts s.t. $\Gamma, \alpha := (\Psi \vdash^n T)$ is a well-formed context if Γ is well-formed, and the type T is well-formed in $\Gamma|_n, \Psi$. We also must ensure that there are no circularities in Γ. A context $\Gamma, \#$ which contains a contradiction marked by $\#$ is well-formed, if Γ is.

$$\text{Additional rules for context well-formedness : } \boxed{\Vdash \Gamma}$$

$$\frac{\Vdash \Gamma \quad \Vdash (\Gamma|_n, \Psi) \quad (\Gamma|_n, \Psi) \Vdash T}{\Vdash \Gamma, \alpha := (\hat{\Psi}^n.T) : (\Psi \vdash^n *)} \qquad \frac{\Vdash \Gamma}{\Vdash \Gamma, \#}$$

We also revisit the typing rules for substitutions. To ensure that $\sigma,\ (\hat{\Psi}^k.T)$ is a well-typed substitution for a context $\Phi, \alpha{:=}(\hat{\Psi}^k.S) : (\Psi \vdash^k *)$, we check in addition to the fact that σ and T are well-typed, that T and S are equal in the range of the substitution. We also note that, if the domain of a substitution contains a contradiction, then the range must also contain a contradiction.

$$\text{Additional rules for well-typed substitutions: } \boxed{\Gamma \Vdash \sigma : \Phi}$$

$$\frac{\Gamma \Vdash \sigma : \Phi \qquad \Gamma|_k, \Psi \Vdash T = [((\sigma/\hat{\Phi})|_k), (\mathrm{id}(\hat{\Psi})/\hat{\Psi})]S \qquad \Gamma|_k, \Psi \Vdash T}{\Gamma \Vdash \sigma,\ (\hat{\Psi}^k.T) : \Phi, \alpha{:=}(\hat{\Psi}^k.S) : (\Psi \vdash^k *)} \qquad \frac{\# \in \Gamma \quad \Gamma \Vdash \sigma : \Phi}{\Gamma \Vdash \sigma : \Phi, \#}$$

Last, we add a type conversion rule that lets us prove that two types are equal modulo the constraints.

$$\frac{\Psi \Vdash e : S \quad \Psi \Vdash S = T}{\Psi \Vdash e : T}$$

4.2 Type Equality Modulo Constraints

Since we accumulate equality constraints in our context Ψ during type checking, declarative equality is not purely structural – it can also exploit the equality constraints. Structural equality on types modulo constraints is defined using the judgment: $\Psi \Vdash T = S$.

Most structural equality rules are as expected, and we omit them for brevity. To, for example, compare two function types for equality, we simply compare their components. For polymorphic types, we similarly compare their components and make sure to extend the context with the declaration for α when comparing the result types. If Ψ contains a contradiction, then we simply succeed. The only interesting case is : $\alpha[\sigma] = T$ where $\alpha{:=}(\hat{\Phi}^n.S) : (\Phi \vdash^n *) \in \Psi$. In this case, we continue to compare T with $[\sigma/\hat{\Phi}]S$. The complete set of structural equality rules can be found in [Jang et al. 2021].

4.3 Typing Rule for Case-Expressions

We now discuss the typing rule for case-expressions which is the centrepiece of supporting typed code analysis.

$$\frac{\Psi \Vdash e : \lceil \Phi \vdash^k T \rceil \qquad \Psi \Vdash \lceil \Phi \vdash^k T \rceil \qquad \text{For all } i \begin{cases} \Vdash (\Psi_i, \Phi_i) & \Psi_i; (\Phi_i)^k \Vdash T_i & \Psi_i; (\Phi_i)^k \Vdash p_i : T_i \\ \Vdash (\Psi_i \oplus \Psi|_k) & (\Psi_i \oplus \Psi|_k) \Vdash (\Phi \vdash^k T) = (\Phi_i \vdash^k T_i) \searrow \Gamma_i & \Gamma_i, \Psi|^k \Vdash e_i : S \end{cases}}{\Psi \Vdash \mathsf{case}_{\lceil \Phi \vdash^k T \rceil}\ e\ \mathsf{of}\ \overrightarrow{(\Psi_i.(\hat{\Phi}_i.p_i) : (\Phi_i \vdash^k T_i) \longrightarrow e_i)} : S}$$

We first check that the guard e has type $\lceil \Phi \vdash^k T \rceil$ and that the given type annotation is a well-kinded type. For each branch, we then check the following conditions:

(1) The type annotation in the pattern is well-kinded. This entails verifying that the context Ψ_i, Φ_i is well-formed and that the type T_i is well-kinded. Recall that Ψ_i, Φ_i is only defined, when all declarations in Ψ_i are at higher levels than k, i.e. the level of Φ_i.
(2) The pattern p_i has type T_i in the context Ψ_i, Φ_i. In fact, we ensure something stronger, namely that Ψ_i contains the pattern variables and the code pattern $\mathsf{box}(\hat{\Phi}_i.\, p_i)$ has type $\lceil \Phi_i \vdash^k T_i \rceil$. This is accomplished by the judgment $\Psi_i; \Phi_i \Vdash p_i : T_i$ (see Sec. 4.4).
(3) We match the type of the pattern, $(\Phi_i \vdash^k T_i)$ against the type of the scrutinee $(\Phi \vdash^k T)$ using the judgment

$$(\Psi_i \oplus \Psi|_k) \Vdash (\Phi \vdash^k T) = (\Phi_i \vdash^k T_i) \searrow \Gamma_i$$

This generates a new context Γ_i which constrains some type variables in $\Psi|_k$. Recall that all variables in Ψ which are below k will be replaced by Φ_i, and hence only the declarations in $\Psi|_k$ matter. Further, $\Psi_i \oplus \Psi|_k$ only contains variable declarations at levels above k and therefore also Γ_i contains only variables above k.
Last, we ensure that the resulting context Γ_i is well-formed, by starting with the joint context $(\Psi_i \oplus \Psi|_k)$. The order will in fact ensure that type variables in $\Psi|_k$ (the type of the scrutinee) can be constrained by variables in Ψ_i (the type of the pattern). We describe matching and the generation of constraints in Sec. 4.5.
(4) Finally, we check that the body e_i of the branch has type S in the constrained context Γ_i extended with the $\Psi|^k$ where we remove all assumptions that are at level k or higher. Note that all removed assumptions are in fact present in Γ_i and have possibly been refined in the previous step.

We now describe pattern typing and constraint generation.

4.4 Pattern Typing

Pattern kinding and typing rules are derived from kinding and typing rules for types and terms in Sec. 3.4. They are a special case of those rules. In all the pattern typing judgments, we separate the pattern variables Ψ from the bound variables Γ, writing $\Psi; \Gamma^n$ instead of Ψ, Γ where $\mathsf{level}(\Gamma) = n$ and for every declaration $x{:}(\_ \vdash^j \_)$ in Ψ, we have $j > n$. Although this separation is not strictly necessary for ensuring that a pattern is well-typed, it is necessary when we define unification on code and types, since only pattern variables in Ψ can be instantiated, while bound variables in Γ remain fixed. Further, to ensure that unification falls into the decidable higher-order pattern fragment (see Miller [1991]), pattern variables must be associated with a variable substitution. For simplicity, we choose here the identity substitution, which we in fact omit for better readability.

We first consider type patterns (see Fig. 6). For type variables, we distinguish two cases. When the type variable α is in Ψ, it must have type $(\Gamma \vdash^n *)$ and describes a pattern type variable. A bound variable occurrence, $\alpha[\sigma]$, is well-kinded, if α is declared in Γ to have kind $(\Phi \vdash^k *)$ and σ is a substitution mapping variables from Φ to $\Psi; \Gamma$. Pattern kinding for function types $\beta \to \alpha$ is straightforward: each pattern variable must be declared in Ψ with kind $(\Gamma \vdash^n *)$. For polymorphic type patterns, $(\alpha{:}(\Phi \vdash^k *)) \to \beta$, we note that β is the pattern variable that makes sense in the extended context $(\Gamma \oplus \alpha{:}(\Phi \vdash^k *))$. Note that in general, k could be greater than $\mathsf{level}(\Gamma) = n$. Hence, the type variable α can increase the overall level of the extended context, if $k > n$, and the extended context has level $\max(n, k)$. Therefore, the pattern type variable β must be declared in Ψ with kind $((\Gamma \oplus \alpha{:}(\Phi \vdash^k *)) \vdash^{\max(n,k)} *)$. Last, we consider the contextual type pattern, $\lceil \Phi \vdash^k \beta \rceil$, where β is the pattern type variable. It is meaningful in the context $(\Gamma|_k, \Phi)$. The level of this resulting context

$$\frac{\alpha : (\Phi \vdash^k *) \in \Gamma \quad \Psi; \Gamma \Vdash \sigma : \Phi}{\Psi; \Gamma^n \Vdash \alpha[\sigma]} \qquad \frac{\alpha : (\Gamma \vdash^n *) \in \Psi}{\Psi; \Gamma^n \Vdash \alpha}$$

$$\frac{k > 0 \quad \Vdash (\Gamma|_k, \Phi) \quad \beta : ((\Gamma|_k, \Phi) \vdash^{\max(n,k)} *) \in \Psi}{\Psi; \Gamma^n \Vdash \lceil \Phi \vdash^k \beta \rceil}$$

$$\frac{\beta : (\Gamma \vdash^n *) \in \Psi \quad \alpha : (\Gamma \vdash^n *) \in \Psi}{\Psi; \Gamma^n \Vdash \beta \to \alpha} \qquad \frac{\Vdash \Psi, \Gamma|_k, \Phi \quad \beta : ((\Gamma \oplus \alpha{:}(\Phi \vdash^k *)) \vdash^{\max(n,k)} *) \in \Psi}{\Psi; \Gamma^n \Vdash (\alpha{:}(\Phi \vdash^k *)) \to \beta}$$

Fig. 6. Type Pattern rules: $\boxed{\Psi; \Gamma^n \Vdash T}$

$$\frac{x : (\Phi \vdash^k T) \in \Gamma \quad \Psi; \Gamma^n \Vdash \sigma : \Phi}{\Psi; \Gamma^n \Vdash x[\sigma] : [\sigma/\hat{\Phi}]T} \qquad \frac{x : (\Gamma \vdash^n T) \in \Psi}{\Psi; \Gamma^n \Vdash x : T} \qquad \frac{p{:}(\Gamma, x{:}(\cdot \vdash^0 S) \vdash^n T) \in \Psi}{\Psi; \Gamma^n \Vdash \mathsf{fn}\ x \to p : S \to T}$$

$$\frac{p{:}(\Gamma \vdash^n S \to T) \in \Psi \quad q{:}(\Gamma \vdash^n S) \in \Psi}{\Psi; \Gamma^n \Vdash p\ q : T} \qquad \frac{p{:}(\Gamma \oplus \alpha{:}(\Phi \vdash^k *) \vdash^{\max(n,k)} T) \in \Psi}{\Psi; \Gamma^n \Vdash \mathsf{Fn}\ \alpha^k \to p : (\alpha{:}(\Phi \vdash^k *)) \to T}$$

$$\frac{p : (\Gamma \vdash^n (\alpha{:}(\Phi \vdash^k *)) \to S) \in \Psi \quad \beta{:}(\Gamma|_k, \Phi \vdash^{\max(n,k)} *) \in \Psi}{\Psi; \Gamma^n \Vdash p\ (\hat{\Phi}^k.\beta) : [\hat{\Phi}^k.\beta[\mathrm{id}(\hat{\Phi})]/\alpha^k]S}$$

$$\frac{k > 0 \quad p{:}(\Gamma|_k, \Phi \vdash^{\max(n,k)} T) \in \Psi}{\Psi; \Gamma^n \Vdash \mathsf{box}(\hat{\Phi}^k.\, p) : \lceil \Phi \vdash^k T \rceil} \qquad \frac{p{:}(\Gamma \vdash^n \lceil \Phi \vdash^k S \rceil) \in \Psi \quad q{:}(\Gamma \oplus u{:}(\Phi \vdash^k S) \vdash^{\max(n,k)} T) \in \Psi}{\Psi; \Gamma^n \Vdash \mathsf{let\,box}\ (\hat{\Phi}^k.u) = p\ \mathsf{in}\ q : T}$$

Fig. 7. Code Pattern Typing $\boxed{\Psi; \Gamma^n \Vdash p : T}$

is $\max(n, k)$ depending on whether n is greater or less than k. Hence, it must be declared in Ψ with kind $((\Gamma|_k, \Phi) \vdash^{\max(n,k)} *)$.

We describe code pattern typing in Fig. 7. A code pattern p is the sub-term in $\mathsf{box}(\hat{\Gamma}^n.\, p)$ which has type $\lceil \Gamma \vdash^n T \rceil$. Our code pattern typing rules follow the typing rules for terms given earlier, and we assume that $\lceil \Gamma \vdash^n T \rceil$ is a well-formed pattern type. We again separate the pattern variables (declared in Ψ) from the bound variables (declared in Γ) (see Fig. 7). A pattern variable x of type T is implicitly associated with the identity substitution and hence must be declared in Ψ with the type $(\Gamma \vdash^n T)$. A bound variable occurrence, written as a closure $x[\sigma]$, is well-typed, if x is declared to have type $(\Phi \vdash^k T)$ in Γ and σ is a substitution pattern that maps variables from Φ to $\Psi; \Gamma$. In code patterns for functions, $\mathsf{fn}\ x \to p$, of type $S \to T$, the pattern variable p is meaningful in the extended context $\Gamma, x{:}(\cdot \vdash^0 S)$. It hence must be declared in Ψ with type $(\Gamma, x{:}(\cdot \vdash^0 S) \vdash^n)T$. Code patterns for applications, $p\ q$ are straightforward given that we adapt a declarative formulation. Polymorphic code patterns, $\mathsf{Fn}\ \alpha^k \to p$, and type application patterns, $p\ (\hat{\Phi}^k.\beta)$, follow the same principles as in polymorphic type patterns discussed earlier; similarly, boxed code patterns, $\mathsf{box}(\hat{\Phi}^k.\, p)$, follow the same principles as for contextual type patterns. In each of these cases, the pattern variable is declared in the extended context at level $\max(n, k)$. For $\mathsf{let\,box}\ (\hat{\Phi}^k.u) = p\ \mathsf{in}\ q$, we follow the typing rules given earlier: p must be declared with type $(\Gamma \vdash^n \lceil \Phi \vdash^k S \rceil)$ and q with type $(\Gamma \oplus u{:}(\Phi \vdash^k S) \vdash^{\max(n,k)} T)$ in Ψ.

Last, we describe typing rules for substitution patterns (Fig. 8) which follow the corresponding typing rules for simultaneous substitutions.

$$\frac{}{\Psi; \Gamma^n \Vdash \cdot : \cdot} \qquad \frac{\Psi; \Gamma^n \Vdash \sigma : \Phi' \quad p{:}(\Gamma|_k, [\sigma/\hat{\Phi}']\Phi \vdash^{\max(n,k)} [\sigma/\hat{\Phi}', \mathrm{id}(\hat{\Phi})/\hat{\Phi}]T) \in \Psi}{\Psi; \Gamma^n \Vdash \sigma, (\hat{\Phi}^k.p) : (\Phi', x{:}(\Phi \vdash^k T))}$$

$$\frac{\Psi; \Gamma^n \Vdash \sigma : \Phi' \quad \alpha{:}(\Gamma|_k, \Phi \vdash^{\max(n,k)} *) \in \Psi}{\Psi; \Gamma^n \Vdash \sigma, (\hat{\Phi}^k.\alpha) : (\Phi', \alpha{:}(\Phi \vdash^k *))}$$

Fig. 8. Substitution Pattern Typing $\boxed{\Psi; \Gamma^n \Vdash \sigma : \Phi}$

$$\frac{\alpha{:}(\Phi \vdash^k *) \in \Gamma}{\Gamma; \Phi \Vdash \alpha = \alpha \searrow \Gamma} \qquad \frac{\Gamma = \Gamma_1, \alpha{:}(\Phi \vdash^k *), \Gamma_0 \quad \Gamma; \Phi \Vdash \alpha \in T}{\Gamma; \Phi \Vdash \alpha = T \searrow \Gamma, \#}$$

$$\frac{\Gamma = \Gamma_1, \alpha{:}(\Phi \vdash^k *), \Gamma_0 \quad \Gamma_1, \Phi \Vdash T \quad \Gamma; \Phi \Vdash \alpha \notin T \quad \Gamma' = \Gamma_1, \alpha{:=}(\hat{\Phi}.T) : (\Phi \vdash^k *), \Gamma_0}{\Gamma; \Phi \Vdash \alpha = T \searrow \Gamma'}$$

$$\frac{\Gamma = \Gamma_1, \alpha{:=}(\hat{\Phi}.T') : (\Phi \vdash^k *), \Gamma_0 \quad \Gamma; \Phi \Vdash \alpha \notin T \quad \Gamma; \Phi \Vdash T' = T \searrow \Gamma'}{\Gamma; \Phi \Vdash \alpha = T \searrow \Gamma'}$$

$$\frac{\alpha{:}(\Psi \vdash^k *) \in \Phi \quad \Gamma; \Phi \Vdash \sigma = \sigma' : \Psi \searrow \Gamma'}{\Gamma; \Phi \Vdash \alpha[\sigma] = \alpha[\sigma'] \searrow \Gamma'} \qquad \frac{\Gamma; \Phi|_n \Vdash \Psi = \Psi' \searrow \Gamma' \quad \Gamma'; (\Phi|_n, \Psi) \Vdash T = S \searrow \Gamma''}{\Gamma; \Phi \Vdash \lceil \Psi \vdash^n T \rceil = \lceil \Psi' \vdash^n S \rceil \searrow \Gamma''}$$

$$\frac{\Gamma; \Phi \Vdash T_1 = S_1 \searrow \Gamma_1 \quad \Gamma_1; \Phi \Vdash T_2 = S_2 \searrow \Gamma_0}{\Gamma; \Phi \Vdash T_1 \to T_2 = S_1 \to S_2 \searrow \Gamma_0} \qquad \frac{\Gamma; \Phi|_n \Vdash \Psi = \Psi' \quad \Gamma; \Phi \oplus (\alpha{:}(\Psi \vdash^n *)) \Vdash T = S \searrow \Gamma'}{\Gamma; \Phi \Vdash (\alpha{:}(\Psi \vdash^n *)) \to T = (\alpha{:}(\Psi' \vdash^n *)) \to S \searrow \Gamma'}$$

Fig. 9. Type Unification: $\boxed{\Gamma; \Phi \Vdash T = S \searrow \Gamma'}$

Whenever a pattern is well-typed using the pattern kinding and typing rules, the pattern is also well-typed when viewed as an expression.

Lemma 4.1 (Pattern Reflection). *Let* $\Vdash \Psi, \Gamma^n$.

(1) If $\Psi; \Gamma^n \Vdash T$ *then* $\Psi \Vdash \lceil \Gamma \vdash^n T \rceil$

(2) If $\Psi; \Gamma^n \Vdash p : T$ *then* $\Psi \Vdash \mathrm{box}(\hat{\Gamma}^n.\, p) : \lceil \Gamma \vdash^n T \rceil$

(3) If $\Psi; \Gamma^n \Vdash \sigma : \Phi$ *then* $\Psi, \Gamma^n \Vdash \sigma : \Phi$

Proof. For (3), by induction on the first derivation. For others, by case analysis. □

4.5 Unification on Types: Constraint Generation

We define the refinement of types using unification via the judgment

$$\Psi \Vdash (\Phi \vdash^k T) = (\Phi_i \vdash^k T_i) \searrow \Gamma$$

Without loss of generality, we assume that Ψ is a context that contains variables at levels higher than k. In particular, it contains the pattern variables.

$$\frac{\Gamma; \cdot \Vdash \Phi = \Phi_i \searrow \Gamma_1 \quad \Gamma_1; \Phi \Vdash T = T_i \searrow \Gamma_2}{\Gamma \Vdash (\Phi \vdash^k T) = (\Phi_i \vdash^k T_i) \searrow \Gamma_2}$$

To unify a contextual type $(\Phi \vdash^k T)$ against $(\Phi_i \vdash^k T_i)$, we first unify Φ against Φ_i and subsequently unify T against T_i given the (bound) variable context Φ. To separate between the bound variables Φ that are fixed and bound and pattern variables Γ that can be refined and instantiated,

$$\frac{}{\Gamma;\Gamma_0 \Vdash \cdot = \cdot \searrow \Gamma} \qquad \frac{\Gamma;\Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma' \quad \Gamma';\Gamma_0,\Psi|_n \Vdash \Psi' = \Phi' \searrow \Gamma''}{\Gamma;\Gamma_0 \Vdash (\Psi,\alpha{:}(\Psi' \vdash^n *)) = (\Phi,\beta{:}(\Phi' \vdash^n *)) \searrow \Gamma''}$$

$$\frac{\Gamma;\Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma' \quad \Gamma';\Gamma_0,\Psi|_n \Vdash \Psi' = \Phi' \searrow \Gamma'' \quad \Gamma'';\Gamma_0,\Psi|_n,\Psi' \Vdash T' = S' \searrow \Gamma'''}{\Gamma;\Gamma_0 \Vdash (\Psi,x{:}(\Psi' \vdash^n T')) = (\Phi,y{:}(\Phi' \vdash^n S')) \searrow \Gamma'''}$$

[All other cases yield $\Gamma, \#$]

Fig. 10. Context Unification: $\boxed{\Gamma;\Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma'}$

we again use the symbol ; and write $\Gamma;\Phi$. Context and type unification are then defined in Fig. 9 and Fig. 10 using the judgments $\Gamma;\Phi^k \Vdash T = S \searrow \Gamma'$ and $\Gamma;\Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma'$.

We only unify well-kinded types T and S; in particular, if T and S are well-typed in Γ,Φ^k, then they remain well-typed in Γ',Φ^k where Γ' is a refinement of Γ, i.e. some type variable declarations that occur in T and in S are constrained s.t. $\Gamma',\Phi \Vdash T = S$.

We write $\Gamma' \geq \Gamma$ to describes context refinement where a declaration $\alpha{:}(\Phi \vdash^n *)$ in Γ has been updated to $\alpha{:=}(\hat{\Phi}^n.T) : (\Phi \vdash^n *)$ and all variable declarations present in Γ are also present in Γ'.

Unification is defined recursively based on the type T and S. In particular, if $T = T_1 \to T_2$ and $S = S_1 \to S_2$, we first match T_1 against S_1 which returns a refined context Γ_1 and we subsequently match T_2 against S_2 yielding a further refinement Γ_2. We similarly proceed to match $\lceil \Psi \vdash^n T \rceil$ against $\lceil \Psi' \vdash^n S \rceil$ given the pattern variable context Γ and the bound variable context Φ. Since $\lceil \Psi \vdash^n T \rceil$ is well-kinded type pattern in $\Gamma;\Phi$, we know that T is a well-kinded type pattern in $\Gamma;\Phi|_n,\Psi$. We therefore first match Ψ against Ψ' in $\Gamma;\Phi|_n$. This returns an updated context Γ'. Next, we match T against S in the context $\Gamma';\Phi|_n,\Psi$.

To unify two polymorphic types, $(\alpha{:}(\Psi \vdash^n *)) \to T$ against $(\alpha{:}(\Psi' \vdash^n *)) \to S$, we first match Ψ against Ψ' and subsequently match T against S. The first context match is not strictly necessary in our setting, as Ψ and Ψ' are type variable contexts and hence do not themselves contain any pattern variables themselves. Nevertheless, we keep the general form, as it highlights how one would extend it to a fully dependently typed system where types could also depend on term variables.

For pattern variables, where we match α against a type T, there are three cases. Note that we again omit writing the identity substitution that is associated with the pattern type variable α.

1) if T contains α (occur's check), then we return a context where we add a contradiction. The constraints are unsatisfiable, and hence, we can conclude anything.

2) if α is not yet constrained, we add $(\hat{\Phi}^k.T)$ as a constraint, making sure that T is well-typed in Γ_1,Φ. This will ensure that the resulting constrained context is well-formed.

3) if α was associated with a constraint $\alpha := (\hat{\Phi}^k.S)$, then we continue to match S against T.

We omit here the symmetric variable cases where $T = \alpha$ for compactness and lack of space, but they follow the same principle. Further, we mostly concentrate here on the cases where unification succeeds; the omitted cases, where T is not a pattern type variable and T and S do not share the top-level type constructor, lead to a contradiction. This is reflected in returning the context $\Gamma, \#$.

In practice, we use the given unification algorithm for bi-directional matching where both sides T and T_i have distinct unification variables. During type-checking, we match T_i (type of the pattern) against T (the type of the scrutinee), and we instantiate meta-variables in T. During runtime, the type T is concrete, and we match T against T_i to select the appropriate branch in the case-expression.

As a consequence, our unification algorithm is biased. For example, if β occurs before α in Γ, and we encounter a unification problem $\alpha = \beta$, we will instantiate α. If $\alpha \to \alpha$ is the type of the

pattern and β is the type of the scrutinee, the algorithm will fail. This is in line with our intuition that the pattern refines the type of the scrutinee, but not vice versa.

Unification on contexts proceeds recursively using the judgment $\Gamma; \Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma'$. Here Γ denote the pattern variables. The context Γ_0, Ψ and Γ_0, Φ are well-formed, i.e., all declarations in Γ are at higher levels than declarations in Γ_0, and all declarations in Γ_0 are at higher levels than declarations in Ψ and Φ. We maintain these criteria during unification, which helps us to cleanly manage variable dependencies.

Lemma 4.2. *If* $\Gamma' \geq \Gamma$ *and* $\Gamma \Vdash T = T'$ *then* $\Gamma' \Vdash T = T'$.

We next state and prove some properties of unification.

Lemma 4.3 (Well-defined Unification). *Assume* $\Vdash \Gamma, \Gamma_0, \Psi$.
(1) If $\Vdash \Gamma, \Gamma_0, \Phi$ *then there is a* Γ' *s.t.* $\Gamma; \Gamma_0 \Vdash \Psi = \Phi \searrow \Gamma'$.
(2) If $\Gamma, \Psi \Vdash T$ $\Gamma, \Psi \Vdash S$ *then there is a* Γ' *s.t.* $\Gamma; \Psi \Vdash T = S \searrow \Gamma'$

Lemma 4.4 (Soundness of Unification).
(1) If $\Gamma; \Gamma_0 \Vdash \Phi = \Psi \searrow \Gamma'$ *then* $\Gamma', \Gamma_0 \Vdash \Phi = \Psi$ *and* $\Gamma' \geq \Gamma$.
(2) If $\Vdash \Gamma, \Phi$ *and* $\Gamma; \Phi \Vdash T = S \searrow \Gamma'$ *then* $\Vdash \Gamma'$ *and* $\Gamma', \Phi \Vdash T = S$ *and* $\Gamma' \geq \Gamma$.

Lemma 4.5 (Unification is stable under substitution).
If $\Gamma, \Psi \Vdash T$ *and* $\Gamma, \Psi \Vdash S$ *and* $\Gamma; \Phi \Vdash T = S \searrow \Gamma'$ *and* $\Gamma_1 \Vdash \sigma : \Gamma$ *then there is* Γ_1' *such that* $\Gamma_1; [\sigma/\hat{\Gamma}]\Phi \Vdash [(\sigma/\hat{\Gamma}), (\mathrm{id}(\hat{\Phi})/\hat{\Phi})]T = [(\sigma/\hat{\Gamma}), (\mathrm{id}(\hat{\Phi})/\hat{\Phi})]S \searrow \Gamma_1'$ *and* $\Gamma_1' \Vdash \sigma : \Gamma'$

Proof. By induction on the structure of $\Gamma; \Phi \Vdash T = S \searrow \Gamma'$. Note that if Γ_1' contains $\#$, $\Gamma_1' \Vdash \sigma : \Gamma'$ becomes trivial with $\Gamma_1 \Vdash \sigma : \Gamma$. □

Lemma 4.6 (Unification is Compatible with Structural Equality).
For pure context Ψ, *if* $\Psi \Vdash T$ *and* $\Psi \Vdash S$ *and* $\mathcal{D} : \Phi \Vdash T = S$ *and* $\mathcal{E} : \cdot; \Phi \Vdash T = S \searrow \Gamma'$ *then* $\Gamma' = \cdot$

Proof. By induction on the structure of $\mathcal{D}$ and case analysis on the structure of $\mathcal{E}$. □

4.6 Operational Semantics for Case-Expressions

We define here the operational semantics of case-expressions and show that types are preserved. Together with the reduction rules for our core multi-level lambda-calculus that emerge from Sec. 3.6 this provides an operational semantics for Mœbius.

$$
\begin{array}{lcl}
(\mathsf{fn}\ x \to e)\ v & \Longrightarrow & [(\ .v)/x^0]e \\
(\mathsf{Fn}\ \alpha^n \to e)\ (\hat{\Phi}^n.T) & \Longrightarrow & [(\hat{\Phi}^n.T)/\alpha^n]e \\
\mathsf{let\ box}\ \hat{\Phi}^n.u = \mathsf{box}(\hat{\Phi}^n.\ e)\ \mathsf{in}\ e' & \Longrightarrow & [(\hat{\Phi}^n.e)/u^n]e'
\end{array}
$$

$$
\frac{\cdot \Vdash \sigma_i : \Psi_i \qquad \cdot \Vdash \lceil \sigma_i/\hat{\Psi}_i \rceil \lceil \Phi_i \vdash^k T_i \rceil = \lceil \Phi \vdash^k T \rceil \qquad \cdot \Vdash [\sigma_i/\hat{\Psi}_i]\mathsf{box}(\hat{\Phi}_i.\ q_i) = \mathsf{box}(\hat{\Phi}^k.\ e)}{\mathsf{case}_{\lceil \Phi \vdash^k T \rceil}\ (\mathsf{box}(\hat{\Phi}^k.\ e))\ \mathsf{of}\ \overrightarrow{(\Psi_i.(\hat{\Phi}_i.q_i) : (\Phi_i \vdash^k T_i) \to e_i')} \Longrightarrow [\sigma_i/\hat{\Psi}_i]e_i'}
$$

Fig. 11. Operational Semantics: $\boxed{e \Longrightarrow e'}$

Last, we prove type preservation. In the proof below, we only concentrate on case-expressions; the remaining cases follow the arguments for local soundness given in Sec. 3.6.

Theorem 4.7 (Type preservation). *If* $\Vdash e : T$ *and* $e \Longrightarrow e'$ *then* $\Vdash e' : T$.

Proof. By case analysis on the second derivation. We show below only the case where e is a case expression.

Case. $\mathsf{case}_{\lceil \Phi \vdash^k T \rceil}\ (\mathsf{box}(\hat{\Phi}^k.\, e))\ \mathsf{of}\ \overrightarrow{(\Psi_i.(\hat{\Phi}_i.q_i) : (\Phi_i \vdash^k T_i) \to e_i')} \Longrightarrow [\sigma_i/\hat{\Psi}_i]e_i'$

$\cdot \Vdash \mathsf{case}_{\lceil \Phi \vdash^k T \rceil}\ (\mathsf{box}(\hat{\Phi}^k.\, e))\ \mathsf{of}\ \overrightarrow{(\Psi_i.(\hat{\Phi}_i.q_i) : (\Phi_i \vdash^k T_i) \to e_i')} : S$ by assumption

$\Psi_i \Vdash \lceil \Phi_i \vdash^k T_i \rceil = \lceil \Phi \vdash^k T \rceil \searrow \Psi_i'$ by inversion

$\Psi_i' \Vdash e_i : S$ by inversion

there exists $\Vdash \sigma_i : \Psi_i$ by reduction rule for case

$\cdot \Vdash [\sigma_i/\hat{\Psi}_i]\lceil \Phi_i \vdash^k T_i \rceil = \lceil \Phi \vdash^k T \rceil \searrow \Gamma$

$\Gamma \Vdash \sigma_i : \Psi_i'$ by unification stability under substitution lemma 4.5

$\cdot \Vdash [\sigma_i/\hat{\Psi}_i]\lceil \Phi_i \vdash^k T_i \rceil = \lceil \Phi \vdash^k T \rceil$ by reduction rule for case

$\Gamma = \cdot$ by unification compatibility with structural equality 4.6

$\Psi_i' \geq \Psi_i$ and therefore $\hat{\Psi}_i' = \hat{\Psi}_i$ by soundness of unification lemma 4.4 and inversion on $\Vdash \sigma_i : \Psi_i'$

$\cdot \Vdash [\sigma_i/\hat{\Psi}_i']e_i : S$ by simultaneous substitution lemma 3.6 and the fact that $\cdot \Vdash S$

$\cdot \Vdash [\sigma_i/\hat{\Psi}_i]e_i : S$ since $\hat{\Psi}_i' = \hat{\Psi}_i$

□

5 RELATED WORK

Early staged computation systems. Davies and Pfenning [2001] first observed that the necessity modality of the S4 modal logic is ideally suited to distinguish between closed code which has type $\Box\tau$ and programs, which have type τ in the multi-staged programming setting. To characterize open code fragments, Davies [1996] proposed $\lambda^{\bigcirc}$ which corresponds to linear temporal logic. Open code is characterized by the type $\bigcirc\tau$. While this logical foundation provides an explanation for generating and splicing in open code, we cannot distinguish open from closed code and the type system of $\lambda^{\bigcirc}$ does not provide guarantees for its safe evaluation. The reason is exactly in the openness: it is not sound to evaluate an open expression before all of its free variables are bound. This openness also makes it difficult to support pattern matching on (open) code.

The desire to combine the advantages of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ has inspired a long line of research on type systems for staged computation, most notable being MetaML [Taha and Sheard 2000] and the work of Taha and Nielsen [2003] on environment classifiers which allows manipulation of open code and supports type inference. Environment classifiers give names for typing environments to specify when exactly its eval function can correctly insert code, but unlike contextual types it does not list the variables that may occur in an environment (or context) concretely. Instead, we reason about environments and their extensions abstractly. Inherent in the nature of that work is that we do not reason about the concrete variables that occur in a given piece of code. It seems hence less expressive than using contextual types, and, in particular, it seems difficult to extend to support pattern matching on code. As Taha [2000] observed, adding intensional analysis to MetaML would make many optimizations unsound. Environment classifiers also seem too coarse to characterize the dependencies that exist within the context of assumptions. This however seems critical when adding System F style polymorphism, as we do in this work, or extending such a system to dependent types.

Reflecting contexts in types. Subsequent work by Kim et al. [2006] characterizes open code using typing environments (using extensible records). Since their goal is to extend ML with Lisp's ability to write both hygienic (via capture-avoiding substitution) and unhygienic (via capturing substitution) templates, variables are treated symbolically, unlike with contextual types. As a result, code templates are not guaranteed to be lexically well-scoped.

Kiselyov et al. [2016] describes a staged calculus <NJ> that safely permits open code movements across different binding environments. The key to ensuring hygiene and type safety when manipulating open code is reflecting free variables of a code fragment in its type, which evokes

contextual modal type theory. However, unlike CMTT, that work uses environment classifiers with the argument that a type does not need to know exactly in which order free variables are bound. We would disagree. Especially when moving to a richer type system where we track dependencies among assumptions, we do need to know the order. Contextual types provide a more exact and general approach to model and reason about variable dependencies.

Rhiger [2012] shows a typed Kripke-style calculus for staged computation $\lambda^{[]}$, supporting evaluation under binders in future stages, manipulation of open code, and mutable state. It uses its own notion of contextual modal types, which models linear time (the typing judgment tracks all future stages), rather than the branching time of our contextual modal types. As in the work on MetaML, the context is still represented abstractly, and we hence lack the ability to express dependencies among assumptions.

Inspecting code. Closely related to using contextual types to model typed code fragments, Chen and Xi [2003, 2005] characterize typeful code representations pairing the list of value types that may appear in a given code together with the type of the code itself. This is reminiscent of what contextual types accomplish, but fundamentally uses a de Bruijn representation to model bound variables occurring in code. This is also a popular approach when modeling object languages in proof assistants and supporting mechanizing meta-theory (see for example [Benton et al. 2012]). In Chen and Xi [2005], the authors sketch how to add pattern matching on code, but it remains unclear how to exploit the full potential of pattern matching. In particular, the refinement of types, which happens when pattern matching on polymorphic code, is omitted.

One of the few works that support code analysis via pattern matching in typed multi-staged programming is Viera and Pardo [2006]. Similar to contextual types, the type of code fragments is annotated with a typing environment. However, unlike Mœbius, the type of the code fragment itself is not tracked, and hence we cannot fully reason about the type of open code. Instead, type-checking of evaluated code fragments is deferred to runtime. The complexity that arises from pattern matching on typed code fragments is hence sidestepped.

Parreaux et al. [2018] describe Squid, a Scala macro library which supports code generation and code inspection using a rewrite primitive. This allows programmers to define code transformations without extra code for the traversal through the program itself. This is very convenient in practice. Yet, from a theoretical point of view, it remains unclear how these rewrite rules are applied (i.e. in what context is a rule applied) and what conditions the rewrite rules must satisfy (i.e. should these rules be non-overlapping, deterministic in how they are applied, etc.).

Beside multi-staged programming: Typed Self-Interpreter. Many popular languages have a self-interpreter, that is, an interpreter for the language written in itself. The idea is to represent programs within the language and then recover a program from its representation. Most recently, Brown and Palsberg [2016] show that System F_ω is capable of representing a self-interpreter, a program that recovers a program from its representation and is implemented in the language itself. To circumvent the limits of the type language, they encode types and type-level functions extensionally as an instantiation function. Our work, which sits between System F_ω and System F, faced a similar problem and choses an intensional encoding instead. While we do not literally support type-level computation as in System F_ω, we do support contextual kinds and view contextual type variables as a closure. This simplifies the equational theory compared to System F_ω. Further, our notion of levels gives us the appropriate structure to describe type and code patterns in case-expressions enabling intensional analysis of code.

Modal Dependent Type Theory. Recently, there has been also interest in developing a dependently typed modal type theory that includes the box-modality. In particular, Gratzer et al. [2019] describe such a type theory which, in principle, allows us to describe the generation of dependently typed

and polymorphic code using **box** and **unbox**. Their modal dependent type theory exploits a Fitch-style system where locks are added to the context of assumptions. Those locks manage access to variables in a similar fashion as context stacks in Kripke-style modal type systems. Intuitively existing assumptions are locked when we enter a new stage and go under a box expression. But in that work, *all* prior assumptions are unlocked. This essentially removes the ability to reason about different stages, as all prior stages are treated the same; the system also has no ability to run code. Hence, this work lacks the ability to reason about multi-staged code and run it. It also does not support reasoning about open code nor pattern matching on code, which raise independent issues.

Pientka et al. [2019] present a Martin Löf type theory that uses the box modality. This work allows the generation and analysis of code using pattern matching. However code is represented in the logical framework LF and only a subset of the computation language can be directly embedded into LF. As such, the system is not a homogeneous programming language and not rich enough to generally model code that contains nested boxed and let box expressions. It also does not allow us to model multi-staged metaprogramming, as the system is fundamentally restricted to two stages.

Kawata and Igarashi [2019] develop a dependently typed multi-stage calculus based on the logical framework LF that supports execution of code and cross-stage persistence, but does not handle pattern matching or polymorphism. Their approach based on environment classifiers provides less fine-grained control compared to the use of contextual types and level annotations.

6 CONCLUSION

This paper describes a multi-staged metaprogramming foundation that, for the first time, supports the generation of typed polymorphic code and its analysis via pattern matching. This is accomplished by generalizing contextual types to a multi-level contextual type system. This allows us to disentangle the notions of levels and stages. While a stage refers to when at run-time code is generated vs executed, levels are annotations on code and the variables that appear inside the code. These level annotations on variables allow us to cleanly distinguish between holes whose value is supplied at runtime (global variables) and holes (local variables) that will be instantiated when the code is spliced into another code fragment.

The multi-level contextual types are the key to define pattern matching on code and type fragments and unlocking the full potential of typed meta-programming. More specifically, the multiple levels are crucial to support generating and inspecting code which itself generates code.

An important benefit of multi-level contextual types is that they allow us to work with high-level type and code pattern abstractions where we neither expose nor commit to a concrete representation of variable bindings and contexts. We see our work as a step towards building a general type-theoretic foundation for multi-staged metaprogramming that, on the one hand, enforces strong type guarantees and, on the other hand, makes it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing the reliability of and trust in the code we are producing and running.

In the immediate future, we aim to add context abstraction to Mœbius following the approach taken in Beluga. Abstracting over contexts is an orthogonal issue, which, we believe, is compatible with the foundation of Mœbius. In the longer term, we aim to generalize the foundations to dependent types. This would provide the first dependently-type meta-programming language.

ACKNOWLEDGMENTS

This work was funded by the Natural Sciences and Engineering Research Council of Canada (grant number 206263), Fonds de recherche du Québec - Nature et Technologies (grant number 253521), and a graduate fellowship from Fonds de recherche du Québec - Nature et Technologies (grant number 304215) awarded to the first author.

REFERENCES

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

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

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)*. 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, 5–17. https://doi.org/10.1145/2837614.2837623

Chiyan Chen and Hongwei Xi. 2003. Meta-Programming through Typeful Code Representation. In *8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*. 275–286. https://doi.org/10.1145/944746.944730

Chiyan Chen and Hongwei Xi. 2005. Meta-programming through Typeful Code Representation. *Journal of Functional Programming* 15, 5 (2005), 797–835. https://doi.org/10.1017/S0956796805005617

Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *11th 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. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a Modal Dependent Type Theory. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, ICFP (2019), 107:1–107:29. https://doi.org/10.1145/3341711

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2021. Moebius: Metaprogramming using Contextual Types – The stage where System F can pattern match on itself (Long Version). arXiv:2111.08099 [cs.PL]

Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *17th Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 11893)*. Springer, 53–72. https://doi.org/10.1007/978-3-030-34175-6_4

Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. 2006. A Polymorphic Modal Type System for Lisp-like Multi-Staged Languages. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. ACM Press, 257–268. https://doi.org/10.1145/1111037.1111060

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)*. 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

Dale Miller. 1991. Unification of Simply Typed Lambda-Terms as Logic Programming. In *8th International Logic Programming Conference*. MIT Press, 255–269.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3 (June 2008), 1–49. https://doi.org/10.1145/1352582.1352591

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying Analytic and Statically-Typed Quasiquotes. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (2018), 13:1–13:33. https://doi.org/10.1145/3158101

Brigitte Pientka. 2003. *Tabled higher-order logic programming*. Ph. D. Dissertation. Department of Computer Science, Carnegie Mellon University. CMU-CS-03-185.

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, 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. https://doi.org/10.1109/LICS.2019.8785683

Brigitte Pientka and Andrew Cave. 2015. Inductive Beluga:Programming Proofs (System Description). In *25th International Conference on Automated Deduction (CADE-25) (Lecture Notes in Computer Science (LNCS 9195))*. Springer, 272–281. https://doi.org/10.1007/978-3-319-21401-6_18

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

Brigitte Pientka and Jana Dunfield. 2010. Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description). In *5th International Joint Conference on Automated Reasoning (IJCAR'10) (Lecture Notes in Artificial Intelligence (LNAI 6173))*. Springer, 15–21. https://doi.org/10.1007/978-3-642-14203-1_2

Morten Rhiger. 2012. Staged Computation with Staged Lexical Scope. In *21st European Symposium on Programming Languages and Systems (ESOP'12) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 559–578. https://doi.org/10.1007/978-3-642-28869-2_28

Tim Sheard and Simon Peyton Jones. 2022. Template Meta-Programming for Haskell. In *ACM SIGPLAN Workshop on Haskell (Haskell'02)*. ACM, 1–16. https://doi.org/10.1145/581690.581691

Walid Taha. 2000. A Sound Reduction Semantics for Untyped CBN Multi-stage Computation. Or, the Theory of MetaML is Non-trivial (Extended Abstract). In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*. ACM, 34–43. https://doi.org/10.1145/328690.328697

Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM, 26–37. https://doi.org/10.1145/604131.604134

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)*. Springer, 157–173. https://doi.org/10.1007/978-3-642-41582-1_10

Marcos Viera and Alberto Pardo. 2006. A Multi-Stage Language with Intensional Analysis. In *5th International Conference on Generative Programming and Component Engineering (GPCE'06)*. ACM, 11–20. https://doi.org/10.1145/1173706.1173709