Multi-stage Programming with Generative and Analytical Macros

Nicolas Stucki
EPFL
Lausanne, Switzerland
nicolas.stucki@epfl.ch

Jonathan Immanuel Brachthäuser
EPFL
Lausanne, Switzerland
jonathan.brachthauser@epfl.ch

Martin Odersky
EPFL
Lausanne, Switzerland
martin.odersky@epfl.ch

Abstract

In metaprogramming, code generation and code analysis are complementary. Traditionally, principled metaprogramming extensions for programming languages, like MetaML and BER MetaOCaml, offer strong foundations for code generation but lack equivalent support for code analysis. Similarly, existing macro systems are biased towards the code generation aspect.

In this work, we present a calculus for macros featuring both code generation and code analysis. The calculus directly models separate compilation of macros, internalizing a commonly neglected aspect of macros. The system ensures that the generated code is well-typed and hygienic.

We implement our system in Scala 3, provide a formalization, and prove its soundness.

CCS Concepts: • **Software and its engineering → Macro languages**; **Semantics**; *Patterns.*

Keywords: multi-stage programming, metaprogramming, macro systems, formalization

ACM Reference Format:
Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage Programming with Generative and Analytical Macros. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486609.3487203

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487203

1 Introduction

Generative programming [5] is used in scenarios such as code configuration of libraries, code optimizations [28], and DSL implementations [4, 26]. There are various kinds of program generation systems ranging from syntax-based and unhygienic, to fully typed [14, 21]. Modern macro systems, like Racket's, can extend the syntax of the language [8] to create hierarchies of domain-specific languages [1]. In this paper, we are not concerned with Racket-like language extensibility, but rather macros that can generate/analize code of expressions at compile-time.

Principled approaches to metaprogramming, such as MetaML [25] and BER MetaOCaml [3, 11–13], offer strong foundations for expression code generation. These systems focus on runtime code generation and ensure static-safety (well-typed and hygienic) and cross-stage safety. They usually provide cross-stage persistence (CSP), the ability to refer to values from previous stages, at the cost of not supporting *cross-platform portability* [25]. Cross-platform portability, as defined by Taha and Sheard [25], describes the ability to move code, generated on one machine, to a (potentially different) machine to compile and execute it there. This ability is necessary for multi-stage macros in compiled languages with separate compilation. To move code from one machine to the next, cross-platform portability requires code serialization, either in the form of source code or as a serialized intermediate representation. Multi-stage systems that support portability usually need to make some compromises. Some miss analytical capabilities while others allow both generative and analytical macros by resorting to advanced type system machinery.

MacroML [9] extended MetaML to provide a multi-stage macro system showing that *"multi-stage programming languages are a good foundation for the semantics based design of macro systems"* [9]. By design, MacroML took a conservative approach not to blur the distinction between code and data, explicitly avoids dynamic scoping, and lacks analytical macros. Squid [16–18] provides a multi-stage system for generative and analytical macros. Unlike MacroML, Squid uses statically typed dynamic scoping, tracking free term variables in types, to provide type-safe analytical macros.

Requirements. We identify the following requirements that a design of a multi-stage macro system for compiled languages should meet.

- *Cross-platform portability.* It should be possible to use generated code on different machines.
- *Static-safety.* Generated code should be hygienic and well-typed.
- *Cross-stage safety.* Access to variables should only be allowed at stages where they are available.
- *Generative/Analytical.* Programmers should be able to generate as well as analyze and decompose code.

We present a formal calculus that captures the fundamental aspects of the Scala 3 multi-stage macro system. The formalization and implementation advance the state of the art, satisfying the requirements listed above.

Contributions. In particular, this paper makes the following contributions.

- *We introduce a calculus* ($\lambda^{\blacktriangle}$) *for well-typed and hygienic multi-stage metaprogramming that allows both generative and analytical macros* (Section 3). The calculus supports quotes and splices (Section 3.1), cross-platform portability (Section 3.2), and analytical macros via pattern matching (Section 3.3).
- *We prove soundness of* $\lambda^{\blacktriangle}$ in terms of the standard theorems for progress (Theorems 3.5 and 3.14) and preservation (Theorems 3.6 and 3.15).
- *We fully implement* $\lambda^{\blacktriangle}$ as a production-ready system in the Scala 3 programming language (Section 4).

2 Multi-Stage Macros in Scala 3

In this section, we offer an informal overview of the features that our calculus provides and discuss how they relate to the design requirements stated in the previous section. All examples in this section are expressed in Scala 3, using our implementation. In Section 3, we present $\lambda^{\blacktriangle}$ formally.

2.1 Generative Multi-Stage Programming

Our implementation in Scala 3 supports generative multi-stage programming. Here we give an overview of the various features that together constitute the meta-programming API.

Quotes and Splices. Multi-stage programming in Scala 3 uses quotes `'{..}` to delay/stage execution of code and splices `${..}` to evaluate and insert code into quotes. Quoted expressions are typed as `Expr[T]` with a covariant type parameter `T`. With these two concepts, it is easy to write statically safe code generators. The following examples shows naive implementation of the x^n mathematical operation.

```
import scala.quoted.*
def powCode(x: Expr[Int], n: Int)(using Quotes): Expr[Int] =
  if n == 0 then Expr(1) // lift 1 into '{ 1 }
  else '{ $x * ${ powCode(x, n-1) } }
```

As shown in the example, our implementation provides the `Expr` operation that lifts a value into a quoted expression.

Macros. We can use the same splicing abstraction to express macros. In our system, a macro consists of top-level splices (not nested in any quote). Conceptually, the contents of the splice are evaluated one stage earlier than the program. Or, in other words, the contents are evaluated while compiling the program. The generated code resulting from the macro replaces the splice in the program.

```
def power2(x: Int): Int =
  ${ powCode('x, 2) } // x * x * 1
```

Stage consistency. We define the *staging level* of some code as the number of quotes minus the number of splices surrounding said code. In general, it is never possible to access a local variable from a lower staging level as it does not yet exist.

```
def badPower(x: Int, n: Int): Int =
  ${ powCode('x, n) } // ERROR value of `n` not known yet
```

In the context of macros and *cross-platform portability*, that is, macros compiled on one machine but potentially executed on another, we cannot support cross-stage persistence of local variables.

```
def badPowCode(x: Expr[Int], n: Int)(using Quotes) =
  // ERROR `n` potentially not available in the next machine
  '{ power($x, n) }

def power(x: Int, n: Int): Int =
  if n == 0 then 1
  else power(x, n-1)
```

Therefore, in our system, local variables can only be accessed at precisely the same staging level. For global definitions, such as `powCode`, the rules are slightly different. It is possible to generate code that contains a reference to a *global* definition such as in `'{ power(2, 4) }`. This is a limited form of cross-stage persistence, where we refer to the already compiled code for `power`. Each compilation step will lower the staging level by 1 while keeping global definitions. In consequence, we can refer to compiled definitions in macros such as `powCode` in `${ powCode('x, 2) }`. We disallow splices within top-level splices.

Inlining. Since using the splices in the middle of a program is not as ergonomic as calling a function, we hide the staging mechanism from end-users of macros and have a uniform way of calling macros and normal functions. For this, we restrict the use of top-level splices to only appear in so-called *inline methods* [22]. This mechanism is not part of the formalization of the present paper but helps to see how macros will be used in practice.

```
// inline macro definition
inline def inlinePower(x: Int, n: Int): Int =
  ${ powCodeFor('x, 'n) }
  // `powCodeFor` defined in next section
```

```
// user code
def power2(x: Int): Int =
  inlinePower(x, 2) // x * x * 1
```

The evaluation of the macro will only happen when the code is inlined into `power2`. When inlined, the code is equivalent to the previous definition of `power2`. An important consequence is that none of the arguments or the return type of the macro will have to mention the `Expr` types, encapsulating all aspects of metaprogramming from the end users.

2.2 Analytical Multi-Stage Programming

By nature, macros consume and produce program fragments of type `Expr`. Analytical macros inspect the code they receive as arguments to perform some analysis or transformation.

Value analysis. To be able to generate optimized code using `powCode`, the macro implementation `powCodeFor` needs to first determine whether the argument passed as parameter `n` is a constant value. This can be achieved via *unlifting* using the `Expr` extractor from our library implementation that will only match if `n` is a quoted constant and extracts its value.

```
def powCodeFor(x: Expr[Int], n: Expr[Int])(using Quotes) =
  n match
    // it is a constant: unlift code n='{m} into number m
    case Expr(m) ⇒ powCode(x, m)
    // not known: call power at runtime
    case _ ⇒ '{ power($x, $n) }
```

Structural analysis. It is sometimes necessary to analyze the structure of the code or decompose the code into its parts. A classic example is an embedded DSL, where a macro knows a set of definitions that it can reinterpret while compiling the code using a macro (for instance, to perform optimizations). In the following example, we extend our previous implementation of `powCode` with the ability to look into `x` to provide further optimizations.

```
def powCode(x: Expr[Int], n: Int)(using Quotes) =
  x match
    case '{ power($y, ${Expr(m)}) } ⇒ // we have (y^m)^n
      powCode(y, n * m) // generate code for y^(n*m)
    case _ ⇒
      if n == 0 then '{ 1 }
      else '{ $x * ${ powCode(x, n-1) } }
```

The pattern `${..}` extracts code as an expression, and it is either bound as `$y` or matched against a nested pattern as in `${Expr(m)}`.

Patterns may contain two kinds of references, global reference such as `power` in `'{ power(...) }` or references to bindings defined in the pattern such as `x` in `case '{ (x: Int) ⇒ x }`.

Closed patterns. When extracting an expression from a quote, we need to make sure that we do not extrude any variables from the scope where they are defined.

```
'{ (x: Int) ⇒ x + 1 } match
  case '{ (y: Int) ⇒ $z } ⇒
    // should not match, otherwise: z = '{ x + 1 }
```

In this example, we see that the pattern should not match. Otherwise, any use of the expression `z` would contain an unbound reference to `x`. To avoid any such extrusion, we only match on a `${..}` if its expression is closed under the definitions within the pattern.

HOAS patterns. To allow extracting expression that may have extruded references we offer a *higher-order abstract syntax* (HOAS) [19] pattern `$f(y)` (or `$f(y1, ...)`). This pattern will η-expand the sub-expression with respect to y and bind it to f. The variables that might have been extruded will be replaced by the arguments of the lambda.

```
'{ ((x: Int) ⇒ x + 1).apply(2) } match
  case '{ ((y: Int) ⇒ $f(y)).apply($z: Int) } ⇒
    // f may contain references to x
    // f = (y: Expr[Int]) ⇒ '{ $y + 1 }
    f(z) // generates '{ 2 + 1 }
```

This approach was also used in earlier Squid [17] versions. The use of HOAS allows us to keep the involved types simple. The η-expanded sub-expression can be typed with a simple function type. This way we can avoid scope extrusion without resorting to involved type-level machinery of tracking free variables. HOAS patterns without parameters are considered closed patterns.

Summary. We showed the features of our system that support generative and analytical multi-stage programming. We showed how cross-platform portability relates to macros and how it influences the design of cross-stage safety. We showed how we guarantee cross-stage safety with stage consistency and HOAS patterns.

3 Multi-Stage Macros Calculus

In this section, we present the $\lambda^{\blacktriangle}$ *multi-stage macros calculus*. The presentation is organized into four parts. We first introduce the core calculus, which extends the simply-typed lambda calculus with support for quotes and splices. The two abstractions are at the core of cross-stage safety and static-safety. In a second step, we extend the calculus to capture the semantics of compilation of macros. An abstraction that supports cross-platform portability. Then we extend the core calculus with quote analysis by adding quoted pattern matching. In a final extension, we combine compilation and

macros with the pattern matching extension. We show the soundness of individual extensions and their combination.

3.1 Core Calculus: Quotes and Splices

This first core calculus captures the fundamental semantics of programs that operate on and produce code. To this end, it extends STLC with the ability to delay the computation by quoting the code and the ability to compose delayed computations by splicing.

Figure 1 defines the syntax and semantics of this calculus. We use the notation $[x \mapsto t_2]t_1$ to denote the standard capture-avoiding substitution of t_2 for x within t_1. As usual, we follow Barendregt [2] and require that all variable names are globally unique. We also only distinguish terms up to renaming.

3.1.1 Syntax. The calculus features the standard forms of simply-typed lambda calculus, that is, constants c, variables x, abstraction $\lambda x{:}T.t$, and application $t\ t$. In order to express the introductory examples, we also add support for fixpoint computation $\mathbf{fix}\ t$. The two most important additions to the term syntax are quotation $\lceil t \rceil$ (instead of `'{t}`) and splicing $\lfloor t \rfloor$ (instead of `${t}`). The syntax of types includes built-in types C, function types of the form $T{\to}T$, and the type of quoted terms $\lceil T \rceil$ (corresponds to `Expr[T]`). That is, for a term t of type T, the quoted term $\lceil t \rceil$ has type $\lceil T \rceil$.

Example 3.1. Within the core calculus, we can easily write a function that can generate complex code. The `def powCode` of Section 2 can be encoded in this calculus for a numerical type $\mathbf{N}$ as follows:

$$\begin{aligned}&\mathbf{fix}\ \lambda rec{:}\lceil \mathbf{N}\rceil{\to}\mathbf{N}{\to}\lceil \mathbf{N}\rceil.\\ &\quad \lambda x{:}\lceil \mathbf{N}\rceil.\lambda n{:}\mathbf{N}.\mathit{ifIsZero}\ n\ \lceil 1\rceil\ \lceil \mathit{mult}\ \lfloor x\rfloor\ \lfloor \mathit{rec}\ x\ (n{-}1)\rfloor\rceil\end{aligned}$$

3.1.2 Environments. As usual, environments Γ are lists of bindings $x :^i T$. However, they do not only track the type of each binding T, but also at which *staging level* i a variable has been introduced. The *staging level* i is a number in $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. We consider bindings at different levels as disjoint. That is, looking up the binding for x at level j in environment $\Gamma = \Gamma_1, x :^i T, \Gamma_2$ only succeeds, if $i = j$. For simplicity, we require well-formedness to establish that environments contain a single binding of each name x, ruling out $\Gamma, x :^i T, x :^j T$ by construction. By definition, the environment guarantees cross-stage safety. Notably, this implies that there is no cross-stage persistence of local variables.

Example 3.2. Though the calculus itself does not support CSP, it is possible to use values in later stages by lifting and splicing. In the following example, we lift a boolean ($\mathbf{B}$) constant into a quote containing the constant.

$$\lambda x{:}\mathbf{B}.\mathit{ifIsTrue}\ x\ \lceil \mathit{true}\rceil\ \lceil \mathit{false}\rceil$$

3.1.3 Typing. Typing judgments take the form $\Gamma \vdash^i t : T$ and assign a term t the type T. However, they are also parameterized by the staging level i. Conceptually, the level starts at 0, increases each time we encounter a quote (T-Quote), and decreases each time we encounter a splice (T-Splice). All other typing rules maintain the same level in their premises. Splices cannot be typed at level 0 to avoid negative staging levels. As can be seen in rule T-Abs, bindings are added to the environment with the level at which they are defined. Similarly, a variable can only be typed if it is referenced at the same level it was defined in (T-Var).

Example 3.3. Tracking of levels ensures *cross-stage safety* of variables. Both type derivations below fail.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x :^1 T \in \emptyset, x :^0 T}\textsc{Fail}}{\emptyset, x :^0 T \vdash^1 x : T}}{\emptyset, x :^0 T \vdash^0 \lceil x\rceil : \lceil T\rceil}}{\emptyset \vdash^0 \lambda x{:}T.\lceil x\rceil : T{\to}\lceil T\rceil} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x :^0 \lceil T\rceil \in \emptyset, x :^1 \lceil T\rceil}\textsc{Fail}}{\emptyset, x :^1 \lceil T\rceil \vdash^0 x : \lceil T\rceil}}{\emptyset, x :^1 \lceil T\rceil \vdash^1 \lfloor x\rfloor : T}}{\emptyset \vdash^1 \lambda x{:}\lceil T\rceil.\lfloor x\rfloor : \lceil T\rceil{\to}T}$$

3.1.4 Evaluation. We present the semantics of our calculus in terms of a small-step operational semantics (Figure 1). Like the typing judgments, the evaluation relation is also indexed by a staging level i. Also similar to typing, the index starts at 0, increases each time it goes in a quote (E-Quote), and decreases each time it goes into a splice (E-Splice). At level 0, the semantics follow the usual STLC semantics and we perform β-reduction (E-Beta) and fix-point computation (E-Fix-Red). The rules E-App-1, E-App-2, and E-Fix express the usual congruences. The congruences E-Quote and E-Splice modify the levels accordingly. Intuitively, the calculus not only performs β-reduction on level 0, but also seeks to reduce all splices at level 1 (E-Splice-Red). To achieve this, at levels greater than 0, we need to evaluate under lambdas (E-Abs) in case it contains a level 1 splice.

Example 3.4. The specifics of the operational semantics are illustrated by the following example, which makes use of both reductions rules E-Beta and E-Splice-Red. The resulting expression is a value according to our definition since it does not contain any level-1 splice.

$$\begin{aligned}&\lceil \lambda x{:}T.\lfloor (\lambda y{:}\lceil T\rceil.y)\ \lceil f\ x\rceil\rfloor\rceil\\ \longrightarrow\ &\lceil \lambda x{:}T.\lfloor \lceil f\ x\rceil\rfloor\rceil && \textsc{E-Beta}\\ \longrightarrow\ &\lceil \lambda x{:}T.f\ x\rceil && \textsc{E-Splice-Red}\end{aligned}$$

3.1.5 Values. While it may appear non-standard, the definition of values follows directly from the operational semantics. Intuitively, a term is a value, if it is a constant (V-Const), an abstraction (V-Abs-0), or a quote (V-Quote) that does not contain any level-1 splices.

3.1.6 Soundness. We show the soundness of the calculus by proving the standard progress and preservation theorems.

Theorem 3.5 (Progress for Terms). *If $\emptyset \vdash^i t : T$, then t is a value $\vdash^i t$ $\mathbf{vl}$ or there exists t' such $t \longrightarrow^i t'$.*

Theorem 3.6 (Preservation for Terms). *If $\Gamma \vdash^i t : T$ and $t \longrightarrow^i t'$, then $\Gamma \vdash^i t' : T$.*

Syntax

$$\textit{Term} \quad t \quad ::= \quad c \mid x \mid \lambda x{:}T.t \mid t\ t \mid \mathbf{fix}\ t \mid \lceil t \rceil \mid \lfloor t \rfloor$$
$$\textit{Type} \quad T \quad ::= \quad C \mid T \to T \mid \lceil T \rceil$$

Environment

$$\textit{Typing environment} \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, x :^i T$$
$$\textit{Level} \quad i \quad \in \quad \mathbb{N}_0$$

Typing Rules

$$\frac{}{\Gamma \vdash^i c : C}\ \text{(T-Const)} \qquad \frac{x :^i T \in \Gamma}{\Gamma \vdash^i x : T}\ \text{(T-Var)} \qquad \frac{\Gamma, x :^i T_1 \vdash^i t_2 : T_2}{\Gamma \vdash^i \lambda x{:}T_1.t_2 : T_1 \to T_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash^i t_1 : T_1 \to T_2 \qquad \Gamma \vdash^i t_2 : T_1}{\Gamma \vdash^i t_1\ t_2 : T_2}\ \text{(T-App)}$$

$$\frac{\Gamma \vdash^i t : T \to T}{\Gamma \vdash^i \mathbf{fix}\ t : T}\ \text{(T-Fix)} \qquad \frac{\Gamma \vdash^{i+1} t : T}{\Gamma \vdash^i \lceil t \rceil : \lceil T \rceil}\ \text{(T-Quote)} \qquad \frac{\Gamma \vdash^{i-1} t : \lceil T \rceil \qquad i \geq 1}{\Gamma \vdash^i \lfloor t \rfloor : T}\ \text{(T-Splice)}$$

Operational Semantics

$$\frac{t_1 \longrightarrow^i t_1'}{t_1\ t_2 \longrightarrow^i t_1'\ t_2}\ \text{(E-App-1)} \qquad \frac{t \longrightarrow^i t' \qquad i \geq 1}{\lambda x{:}T.t \longrightarrow^i \lambda x{:}T.t'}\ \text{(E-Abs)} \qquad \frac{\vdash^0 t_2\ \mathbf{vl}}{(\lambda x{:}T_1.t_1)\ t_2 \longrightarrow^0 [x \mapsto t_2]t_1}\ \text{(E-Beta)}$$

$$\frac{\vdash^i t_1\ \mathbf{vl} \qquad t_2 \longrightarrow^i t_2'}{t_1\ t_2 \longrightarrow^i t_1\ t_2'}\ \text{(E-App-2)} \qquad \frac{t \longrightarrow^i t'}{\mathbf{fix}\ t \longrightarrow^i \mathbf{fix}\ t'}\ \text{(E-Fix)} \qquad \mathbf{fix}\ \lambda x{:}T.t \longrightarrow^0 [x \mapsto \mathbf{fix}\ \lambda x{:}T.t]t\ \text{(E-Fix-Red)}$$

$$\frac{t \longrightarrow^{i+1} t'}{\lceil t \rceil \longrightarrow^i \lceil t' \rceil}\ \text{(E-Quote)} \qquad \frac{t \longrightarrow^{i-1} t' \qquad i \geq 1}{\lfloor t \rfloor \longrightarrow^i \lfloor t' \rfloor}\ \text{(E-Splice)} \qquad \frac{\vdash^1 t\ \mathbf{vl}}{\lfloor \lceil t \rceil \rfloor \longrightarrow^1 t}\ \text{(E-Splice-Red)}$$

Values

$$\frac{}{\vdash^i c\ \mathbf{vl}}\ \text{(V-Const)} \qquad \frac{}{\vdash^0 \lambda x{:}T.t\ \mathbf{vl}}\ \text{(V-Abs-0)} \qquad \frac{\vdash^{i+1} t\ \mathbf{vl}}{\vdash^i \lceil t \rceil\ \mathbf{vl}}\ \text{(V-Quote)} \qquad \frac{i \geq 1}{\vdash^i x\ \mathbf{vl}}\ \text{(V-Var)}$$

$$\frac{\vdash^i t\ \mathbf{vl} \qquad i \geq 1}{\vdash^i \mathbf{fix}\ t\ \mathbf{vl}}\ \text{(V-Fix)} \qquad \frac{\vdash^i t\ \mathbf{vl} \qquad i \geq 1}{\vdash^i \lambda x{:}T.t\ \mathbf{vl}}\ \text{(V-Abs)} \qquad \frac{\vdash^i t_1\ \mathbf{vl} \qquad \vdash^i t_2\ \mathbf{vl} \qquad i \geq 1}{\vdash^i t_1\ t_2\ \mathbf{vl}}\ \text{(V-App)} \qquad \frac{\vdash^{i-1} t\ \mathbf{vl} \qquad i \geq 2}{\vdash^i \lfloor t \rfloor\ \mathbf{vl}}\ \text{(V-Splice)}$$

Figure 1. Core Calculus

The full proofs can be found in Appendix A [24], here we only point out the structure and define key lemmas.

As usual, progress requires us to show that values take canonical forms. This standard lemma trivially extends to our non-standard definition of values.

Lemma 3.7 (Canonical Forms)**.**
If $\vdash^0 t\ \mathbf{vl}$ *and*

- $t : C$*, then* $t = c$ *for some* c*.*
- $t : T_1 \to T_2$*, then* $t = \lambda x{:}T_1.t_1$ *for some* x *and* t_1*.*
- $t : \lceil T \rceil$*, then* $t = \lceil t_1 \rceil$ *for some* t_1*.*

Proof (of Lemma 3.7). By case analysis on the value definition $\vdash^0 t\ \mathbf{vl}$. □

To prove progress, we also need to prove a more general variant allowing the typing context Γ to only contain bindings on a level greater than 0. To capture this property, we define $\Gamma^{\geq 1}$ as a restricted typing context:

Definition 3.8 (Restricted Typing Context)**.**

$$\Gamma^{\geq 1} \quad ::= \quad \emptyset \ \mid\ \Gamma^{\geq 1}, x :^i T \quad \text{for } i \geq 1$$

Using the restricted context, we define the generalized version of progress as:

Lemma 3.9 (Extended Progress for Terms)**.** *If* $\Gamma^{\geq 1} \vdash^i t : T$*, then* t *is a value* $\vdash^i t\ \mathbf{vl}$ *or there exists* t' *such* $t \longrightarrow^i t'$

Proof (of Theorem 3.5). The proof of progress trivially follows from Lemma 3.9, by choosing $\Gamma^{\geq 1} = \emptyset$. □

For the proof of preservation, we need to adjust the standard substitution lemma to our setting with levels.

Lemma 3.10 (Substitution)**.** $\forall i, j \in \mathbb{N}_0$*, if* $\Gamma \vdash^j t_1 : T_1$ *and* $\Gamma, x :^j T_1 \vdash^i t_2 : T_2$ *then* $\Gamma \vdash^i [x \mapsto t_1]t_2 : T_2$*.*

Proof of Preservation (Theorem 3.6). Induction over the typing derivation, using the substitution lemma (Lemma 3.10). □

3.2 Macros Extension: Cross-Platform Portability

The calculus from the previous section allowed us to write programs that generate code using quotes and splices.

Extended Syntax

$$\textit{Program} \quad p \quad ::= \quad \texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p \mid \texttt{eval}\ t$$

Libraries

$$\textit{Library typing} \quad \Sigma \quad ::= \quad \emptyset \mid \Sigma, x{:}T$$

$$\textit{Runtime library} \quad \Omega \quad ::= \quad \emptyset \mid \Omega, x{=}t$$

Extended Typing Rules

$$\frac{x{:}T \in \Sigma}{\Gamma \mid \Sigma \vdash^i x : T}\ \text{(T-Link)} \qquad \frac{\emptyset \mid \Sigma \vdash^0 t : T}{\Sigma \vdash \texttt{eval}\ t : T}\ \text{(T-Eval)} \qquad \frac{\emptyset \mid \Sigma \vdash^1 t : T_1 \qquad \Sigma, x{:}T_1 \vdash p : T_2}{\Sigma \vdash \texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p : T_2}\ \text{(T-Def)}$$

Extended Values

$$\frac{\vdash^0 t\ \ \mathbf{vl}}{\vdash \texttt{eval}\ t\ \ \mathbf{vl}}\ \text{(V-Eval)}$$

Extended Operational Semantics

$$\frac{x{=}t \in \Omega}{x \longrightarrow^0_\Omega t}\ \text{(E-Link)} \qquad \frac{t \longrightarrow^0_\Omega t'}{\texttt{eval}\ t \mid \Omega \longrightarrow \texttt{eval}\ t' \mid \Omega}\ \text{(E-Eval)} \qquad \frac{t \longrightarrow^1_\Omega t'}{\texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p \mid \Omega \longrightarrow \texttt{def}\ x = \lceil t' \rceil\ \texttt{in}\ p \mid \Omega}\ \text{(E-Macro)} \qquad \frac{\vdash^1 t\ \ \mathbf{vl}}{\texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p \mid \Omega \longrightarrow p \mid \Omega, x{=}t}\ \text{(E-Compile)}$$

Figure 2. Macros Extension

In realistic compiled languages, code is first compiled on some machine and then used on a (potentially) different machine. If a macro is executed when compiling, the code it generates will also be compiled and then used on another machine. This implies that we need cross-platform portability to compile the generated code. If a macro definition is itself compiled, we need to compile a program containing quoted code. This requires a form of serialization in practice, but for this to be sound it also requires cross-platform portability.

To capture the semantics of compiling programs, we extend the previous calculus with library function definitions. These definitions will be compiled before they are used. The calculus also adds a restricted notion of CSP that is compatible with cross-platform portability. Figure 2 defines the syntax and semantics of the extended $\lambda^{\blacktriangle}$ calculus.

3.2.1 Syntax. The macro calculus extends the syntax of Figure 1 adding a syntactic form for programs p. Programs are lists of library bindings $\texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p$ ending in a single term $\texttt{eval}\ t$ that will be evaluated after all library bindings have been compiled. Here, $\texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p$ represents the definition of a library function that is made available as x in the remaining program. The implementation t, which is given as *code*, is compiled and then added to a simplified store, which we use to model the set of compiled functions loaded in the program. In this program, a macro is a splice within t. The syntactic form $\texttt{eval}\ t$ represents a program that will be evaluated without being compiled. In practice, $\texttt{eval}\ t$ would be an interpreted call to the `main` method.

3.2.2 Libraries. Figure 2 introduces store-like runtime libraries Ω that map library function names x to their implementation. It also adds a new typing environment Σ, which types libraries Ω and allows us to track references to compiled programs. Values will only ever be added to the library Ω but never updated. Importantly, the bindings in Σ (and Ω) are not annotated with staging levels and are thus staging-level agnostic.

3.2.3 Typing. The typing judgements in Figure 2 extend the ones presented in Figure 1. We modify the judgement form $\Gamma \vdash^i t : T$ to also track the library typing Σ as $\Gamma \mid \Sigma \vdash^i t : T$. All existing rules simply pass Σ unmodified to their premises. To type programs, we add a new typing judgment $\Sigma \vdash p : T$ where Σ tracks the library bindings. When typing a library definition $\texttt{def}\ x = \lceil t \rceil\ \texttt{in}\ p$ (T-Def), we type the term t at level 1. This way the term t can contain splices, which in turn allows us to model macros. The rest of the program is typed adding x to the library environment Σ. Typing $\texttt{eval}\ t$ (T-Eval) simply types t at level 0 (like in Section 3.1) but adds the Σ, which will not change in the remainder of the derivation. To be able to access library functions, we add rule T-Link, which looks up the signature of a free variable x in Σ. We assume that $dom(\Sigma)$ and $dom(\Gamma)$ are disjoint and hence there cannot be any ambiguity with (T-Var). Note that, unlike rule T-Var, variables in Σ are stage polymorphic, therefore library functions display a form of cross-stage persistence.

Example 3.11. Library functions in Σ can be used at any level after they are compiled. This is illustrated by the typing derivation below, where f is used at staging levels 0 and 1. Assuming that $f{:}\lceil C \rceil \to C \in \Sigma$, we can see that all premises are satisfiable with $T_1 = T_3 = \lceil C \rceil$ and $T_2 = C$.

$$\dfrac{\dfrac{f{:}T_1 \to C \in \Sigma}{\emptyset \mid \Sigma \vdash^0 f : T_1 \to C} \qquad \dfrac{\dfrac{\dfrac{f{:}T_3 \to T_2 \in \Sigma}{\emptyset \mid \Sigma \vdash^1 f{:}T_3 \to T_2} \qquad \dfrac{\emptyset \mid \Sigma \vdash^0 c : C \quad T_3 = \lceil C \rceil}{\emptyset \mid \Sigma \vdash^1 \lceil c \rceil : T_3}}{\emptyset \mid \Sigma \vdash^1 f\ \lceil c \rceil : T_2} \quad T_1 = \lceil T_2 \rceil}{\emptyset \mid \Sigma \vdash^0 \lceil f\ \lceil c \rceil \rceil : T_1}}{\emptyset \mid \Sigma \vdash^0 f\ \lceil f\ \lceil c \rceil \rceil : C}}{\Sigma \vdash \texttt{eval}\ f\ \lceil f\ \lceil c \rceil \rceil : C}$$

3.2.4 Evaluation. Like in the case of typing, the operational semantics in Figure 2 extends the one presented in Figure 1 by modifying the relation $t \longrightarrow^i t'$ to be indexed with a runtime library Ω as $t \longrightarrow^i_\Omega t'$. Also, like in typing, all existing rules simply pass on Ω to their premises. To specify

the evaluation of programs, we introduce a new relation $p \mid \Omega \longrightarrow p' \mid \Omega'$, where a program p with a library Ω evaluates to a program p' with a potentially extended library Ω'. For a library definition **def** $x = \lceil t \rceil$ **in** p, where t is a value (*i.e.*, does not contain macros), the library store is updated with a binding $x{=}t$ and the definition is removed (E-Compile). Importantly, in this process we are taking a t at staging level-1 and compile it, making it available as a runtime dependency in the rest of the program; the compiled library function t can be used on arbitrary levels, including level 0. If the library function t in a definition **def** $x = \lceil t \rceil$ **in** p still contains splices at staging level 1 (macros), we first need to evaluate them (E-Macro). Using the reduction of the core calculus from Section 3.1, the contents of the macros will be evaluated at that point. This will produce a quote value that is then canceled with the splice. To evaluate the final expression **eval** t, we simply reduce the term t using the runtime dependencies in Ω (E-Eval). At this point, Ω is fixed and will not change, and it will just propagate down to allow E-Link to use Ω. A reference x to a library function typed with T-Link at level 0 will lead to x being replaced by the compiled code from Ω (E-Macro). We say that we "link the reference with the compiled function". At any other level $i \geq 1$, we simply keep the reference to x, since it is considered a value.

3.2.5 Values. The definition of values in Figure 2 is kept unchanged. We merely also add a value definition for programs $\vdash p$ **vl**. The only program value is **eval** t where t is required to be a value (V-Eval).

Example 3.12. The following example illustrates evaluation in the calculus. We define a library function *powCode* (implemented as in Example 3.1) as a macro taking a quoted base x and an exponent n of numeric type (N); and a library function $power^2$ using the macro with 2 as the exponent.

$\Omega \quad \emptyset$

def *powCode* = $\lceil$**fix** $\lambda rec{:}\lceil \mathbf{N} \rceil \to \mathbf{N} \to \lceil \mathbf{N} \rceil . \ldots \rceil$ **in**
def $power^2$ = $\lceil \lambda x{:}\mathbf{N}. \lfloor$ *powCode* $\lceil x \rceil$ 2 $\rfloor \rceil$ **in eval** $power^2$ 3

First, we *compile* the macro definition *powCode* since it does not contain any splices at level 1, storing it in Ω.

$\Omega \quad \emptyset$, *powCode*=**fix** $\lambda rec{:}\lceil \mathbf{N} \rceil \to \mathbf{N} \to \lceil \mathbf{N} \rceil . \ldots$

def $power^2$ = $\lceil \lambda x{:}\mathbf{N}. \lfloor$ *powCode* $\lceil x \rceil$ 2 $\rfloor \rceil$ **in eval** $power^2$ 3

Next, we perform *macro expansion* and evaluate the code in the splice of $power^2$.

$\Omega \quad \emptyset$, *powCode*=**fix** $\lambda rec{:}\lceil \mathbf{N} \rceil \to \mathbf{N} \to \lceil \mathbf{N} \rceil . \ldots$

def $power^2$ = $\lceil \lambda x{:}\mathbf{N}. \lfloor \lceil x * x * 1 \rceil \rfloor \rceil$ **in eval** $power^2$ 3

Performing splice canceling results in:

$\Omega \quad \emptyset$, *powCode*=**fix** $\lambda rec{:}\lceil \mathbf{N} \rceil \to \mathbf{N} \to \lceil \mathbf{N} \rceil . \ldots$

def $power^2$ = $\lceil \lambda x{:}\mathbf{N}. x * x * 1 \rceil$ **in eval** $power^2$ 3

As before, we compile $power^2$, which now does not contain splices at level 0.

$\Omega \quad \emptyset$, *powCode*=..., $power^2 = \lambda x{:}\mathbf{N}. x*x*1$

eval $power^2$ 3

Finally, we evaluate the *main* program:

$\Omega \quad \emptyset$, *powCode*=..., $power^2$=...

eval 9

3.2.6 Soundness. To state the progress and preservation theorems for the $\lambda^{\blacktriangle}$ calculus, we introduce an auxiliary relation $\Sigma \Vdash \Omega$, which means that Ω is well-typed under environment Σ.

Definition 3.13. (Library Typing) $\Sigma \Vdash \Omega$ if and only if

$$dom(\Sigma) = dom(\Omega) \ \wedge$$
$$\emptyset \mid \Sigma \vdash^i \Omega(x) : \Sigma(x) \text{ for all } x \in dom(\Omega) \text{ and all } i \in \mathbb{N}_0$$

Using this definition, we state the soundness of the calculus in terms of progress and preservation for programs. Again, the full proofs can be found in Appendix B [24].

Theorem 3.14 (Progress for Programs). *If $\Sigma \vdash p : T$, then p is a value $\vdash p$ **vl** or, for any Ω such that $\Sigma \Vdash \Omega$, there exists p' and Ω' such $p \mid \Omega \longrightarrow p' \mid \Omega'$.*

Theorem 3.15 (Preservation for Programs). *If $\Sigma \vdash p : T$, $\Sigma \Vdash \Omega$ and $p \mid \Omega \longrightarrow p' \mid \Omega'$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \vdash p' : T$ and $\Sigma' \Vdash \Omega'$.*

Since we added rule T-Link, we need to revisit the lemmas from the previous section. The Canonical Forms (Lemma 3.7) still holds because there was no change in value definitions.

We also need to restate the various theorems and lemmas for terms to account for libraries and library typing.

Theorem 3.16 (Progress for Terms). *If $\emptyset \mid \Sigma \vdash^i t : T$ and $\Sigma \Vdash \Omega$, then t is a value $\vdash^i t$ **vl** or there exists t' such $t \longrightarrow^i_\Omega t'$.*

Lemma 3.17 (Extended Progress for Terms). *If $\Gamma^{\geq 1} \mid \Sigma \vdash^i t : T$ and $\Sigma \Vdash \Omega$, then t is a value $\vdash^i t$ **vl** or there exists t' such $t \longrightarrow^i_\Omega t'$.*

Theorem 3.18 (Preservation for Terms). *If $\Gamma \mid \Sigma \vdash^i t : T$, $t \longrightarrow^i_\Omega t'$ and $\Sigma \Vdash \Omega$, then $\Gamma \mid \Sigma \vdash^i t' : T$.*

Lemma 3.19 (Substitution). *$\forall i, j \in \mathbb{N}_0$, if $\Gamma \mid \Sigma \vdash^j t_1 : T_1$ and $\Gamma, x :^j T_1 \mid \Sigma \vdash^i t_2 : T_2$ then $\Gamma \mid \Sigma \vdash^i [x \mapsto t_1]t_2 : T_2$.*

Most of the proofs carry through unchanged. To show preservation we additionally need the following lemma.

Lemma 3.20 (Σ-Weakening). *If $\Sigma \vdash p : T$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' \vdash p : T$*

3.3 Analytical Extension: Quote Matching

We now extend the calculus of Section 3.1 with support for analytical macros. To this end, we add a pattern matching construct that can deconstruct a piece of code into its components. Subexpressions of the code can be selectively extracted using a *bind pattern*. Importantly, patterns can only match on a subset of the language, specifically STLC+Fix. Figure 3 contains the syntax and semantics of this calculus.

Extended Syntax

$$\textit{Terms} \quad t \quad ::= \quad \ldots \mid t \ \texttt{match} \ \lceil t \rceil \ \texttt{then} \ t \ \texttt{else} \ t \mid \llbracket x \rrbracket_T^{\overline{x_k:T_K}}$$

Extended Environment

$$\textit{Pattern bindings} \quad \Phi \quad ::= \quad \emptyset \mid \Phi, x \mapsto x$$

Extended Typing Rules

$$\frac{\Gamma \vdash^i t_s : \lceil T_p \rceil \qquad \emptyset \vdash^{i+1} t_p : T_p \dashv \Gamma_t \qquad \Gamma; \Gamma_t \vdash^i t_t : T \qquad \Gamma \vdash^i t_e : T}{\Gamma \vdash^i t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e : T}$$
(T-Match)

$$\Gamma_p \vdash^i c : C \dashv \emptyset$$
(T-Pat-Const)

$$\frac{x :^i T \in \Gamma_p}{\Gamma_p \vdash^i x : T \dashv \emptyset}$$
(T-Pat-Var)

$$\frac{\Gamma_p, x :^i T_1 \vdash^i t : T_2 \dashv \Gamma_t}{\Gamma_p \vdash^i \lambda x{:}T_1.t : T_1 \to T_2 \dashv \Gamma_t}$$
(T-Pat-Abs)

$$\frac{\Gamma_p \vdash^i t_1 : T_1 \to T_2 \dashv \Gamma_{t_1} \qquad \Gamma_p \vdash^i t_2 : T_1 \dashv \Gamma_{t_2}}{\Gamma_p \vdash^i t_1 \ t_2 : T_2 \dashv \Gamma_{t_1}; \Gamma_{t_2}}$$
(T-Pat-App)

$$\frac{\Gamma_p \vdash^i t : T \to T \dashv \Gamma_t}{\Gamma_p \vdash^i \texttt{fix} \ t : T \dashv \Gamma_t}$$
(T-Pat-Fix)

$$\frac{\overline{x_k :^i T_k \in \Gamma_p}}{\Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{x_k:T_K}} : T \dashv \emptyset, x :^{i-1} \overline{\lceil T_k \rceil \to} \lceil T \rceil}$$
(T-Pat-Bind)

Extended Operational Semantics

$$\frac{\vdash^1 t_s \ \textbf{vl} \qquad t_s \boxdot t_p \ / \ t_t \Longrightarrow^{\emptyset} t_t'}{\lceil t_s \rceil \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \longrightarrow^0 t_t'}$$
(E-Match-Succ)

$$\frac{t_s \longrightarrow^0 t_s'}{t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \longrightarrow^0 t_s' \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e}$$
(E-Match-Scrut)

$$\frac{\vdash^1 t_s \ \textbf{vl} \qquad t_s \boxdot t_p \ / \ t_t \not\Longrightarrow^{\emptyset} t_t'}{\lceil t_s \rceil \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \longrightarrow^0 t_e}$$
(E-Match-Fail)

$$\frac{\vdash^0 t_s \ \textbf{vl} \qquad t_t \longrightarrow^i t_t' \qquad i \geq 1}{t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \longrightarrow^i t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t' \ \texttt{else} \ t_e}$$
(E-Match-Then)

$$\frac{\vdash^0 t_s \ \textbf{vl} \qquad \vdash^0 t_t \ \textbf{vl} \qquad t_e \longrightarrow^i t_e' \qquad i \geq 1}{t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \longrightarrow^i t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e'}$$
(E-Match-Else)

$$c \boxdot c \ / \ t_t \Longrightarrow^{\Phi} t_t$$
(E-Pat-Const)

$$\frac{t_{s1} \boxdot t_{p1} \ / \ t_{t1} \Longrightarrow^{\Phi} t_{t2} \qquad t_{s2} \boxdot t_{p2} \ / \ t_{t2} \Longrightarrow^{\Phi} t_{t3}}{t_{s1} \ t_{s2} \boxdot t_{p1} \ t_{p2} \ / \ t_{t1} \Longrightarrow^{\Phi} t_{t3}}$$
(E-Pat-App)

$$\frac{t_s \boxdot t_p \ / \ t_t \Longrightarrow^{\Phi} t_t'}{\texttt{fix} \ t_s \boxdot \texttt{fix} \ t_p \ / \ t_t \Longrightarrow^{\Phi} t_t'}$$
(E-Pat-Fix)

$$\Phi(x_p) \boxdot x_p \ / \ t_t \Longrightarrow^{\Phi} t_t$$
(E-Pat-Var)

$$\frac{t_s \boxdot t_p \ / \ t_t \Longrightarrow^{\Phi, x_p \mapsto x_s} t_t'}{\lambda x_s{:}T.t_s \boxdot \lambda x_p{:}T.t_p \ / \ t_t \Longrightarrow^{\Phi} t_t'}$$
(E-Pat-Abs)

$$\frac{FV(t_s) \cap range(\Phi) \subseteq \{\overline{\Phi(x_k)}\} \qquad t_s' = \overline{\lambda x_k' : \lceil T_k \rceil}.\ [\overline{\Phi(x_k) \mapsto \lfloor x_k' \rfloor}] \lceil t_s \rceil}{t_s \boxdot \llbracket x \rrbracket_T^{\overline{x_k:T_K}} \ / \ t_t \Longrightarrow^{\Phi} [x \mapsto t_s'] t_t}$$
(E-Pat-Bind)

Extended Values

$$\frac{\vdash^i t_s \ \textbf{vl} \qquad \vdash^i t_t \ \textbf{vl} \qquad \vdash^i t_e \ \textbf{vl} \qquad i \geq 1}{\vdash^i t_s \ \texttt{match} \ \lceil t_p \rceil \ \texttt{then} \ t_t \ \texttt{else} \ t_e \ \textbf{vl}}$$
(V-Match)

Figure 3. Quote Pattern Match Extension

3.3.1 Syntax. The syntax of our extension is defined in Figure 3. It extends the calculus presented in Figure 1 with two new syntactic constructs: a pattern matching operation t_s `match` $\lceil t_p \rceil$ `then` t_t `else` t_e, and a bind pattern $\llbracket x \rrbracket_T^{\overline{x_k:T_K}}$. The former matches a scrutinee t_s against a pattern t_p. If the match succeeds the *then* branch t_t is evaluated, otherwise the *else* branch t_e is evaluated. A pattern t_p may contain any of the following language constructs: constants, references, lambdas, application, and fix operator. In addition, t_p may contain a bind pattern $\llbracket x \rrbracket_T^{\overline{x_k:T_K}}$, which will extract a sub-expression of type T from the quote and bind it to x. The extracted sub-expression is locally closed under $\overline{x_k:T_K}$, that is, it can only contain any of the x_k as free variables. We say the sub-expression is *locally closed*, since in addition to free

variables $\overline{x_k{:}T_K}$ bound in the pattern, it can contain free variables defined outside of the pattern. Bind is commonly used without any $\overline{x_k{:}T_K}$ as $[\![x]\!]_T$ to match a closed subexpression.

3.3.2 Typing. Typing a pattern match (T-Match) requires that the scrutinee of type $\lceil T_p \rceil$ will be matched against a pattern of type T_p. Patterns themselves are typed under a different typing judgment $\Gamma_p \vdash^i t : T \dashv \Gamma_t$. Here Γ_p represents an input and contains the bindings defined within the pattern. In contrast, Γ_t represents an output and contains binding introduced by the pattern, which is then made available in the *then* branch. The pattern is typed at level $i+1$ as if it was in a quote (as reflected by the syntax).

The rules for pattern typing mostly coincide with their typing counterparts. They only differ in their treatment of environments. First, the Γ_p environment tracks any binding added by a lambda pattern (T-Pat-Abs). It is used to lookup references in (T-Pat-Var) and to determine the types of free variables in a bind pattern (T-Pat-Bind). Second, the Γ_t environment collects x bindings added by $[\![x]\!]_T^{\overline{x_k:T_K}}$, which will be made available in the *then* branch.

The bind pattern $[\![x]\!]_T^{\overline{x_k:T_K}}$ matches against an arbitrary expression locally closed under $\overline{x_k{:}T_K}$. We represent this closed term as a curried function taking arguments of the corresponding types $\overline{T_k}$ (T-Pat-Bind). Hence, in the output environment, we bind x to a value of type $\overline{\lceil T_k \rceil \to \lceil T \rceil}$. As a special case of rule T-Pat-Bind, we match on a locally closed sub-expression where $\overline{x_k{:}T_K}$ is empty, and therefore the type of x is simply $\lceil T \rceil$. Note that the i in this typing judgment is only present to inform at which level x must be added in Γ_t.

3.3.3 Evaluation. Once more, the operational semantics in Figure 3 extends the operational semantics in Figure 1. First of all, to handle quoted pattern matching, we extend the reduction relation $t \longrightarrow^i t'$ with additional rules. At staging level 0, we first evaluate the scrutinee until it is a value (E-Match-Scrut). At all other levels, we also continue to reduce the *then* (E-Match-Then) and *else* branches (E-Match-Else).

To model the semantics of nested patterns, we introduce an additional reduction relation $t_s \boxdot t_p \;/\; t_t \Longrightarrow^{\Phi} t_t'$. It states that the sub-term t_s matches the sub-pattern t_p with a substitution map Φ, which provides a mapping from a variable defined in the scrutinee to one defined in the pattern. In addition to matching, the relation also transforms the *then* part of the match t_t into a t_t' where all bindings defined in the pattern are substituted.

To reduce a match operation, if the pattern matched, we will evaluate it into t_t' (E-Match-Succ) where t_t' does not contain any of the bindings defined in the pattern. We say that a pattern (or sub-pattern) did not match if $t_s \boxdot t_t \;/\; t_e \Longrightarrow^{\Phi} t_t'$, that is we cannot derive a match. Therefore, if the pattern did not match we evaluate to the *else* branch t_e (E-Match-Fail).

Exactly as in pattern typing, sub-pattern matching $t_s \boxdot t_p \;/\; t_t \Longrightarrow^{\Phi} t_t'$ can match the syntactic form of STLC+Fix, as well as bind sub-terms. Rules that do not introduce bindings in the *then* branch will not modify the t_t/t_t', they will only propagate the results from sub-evaluation. Rules E-Pat-Const, E-Pat-Fix, and E-Pat-App are straightforward. Interestingly, when matching a lambda, the substitution Φ will track the relationship between the binding in the pattern and in the scrutinee (E-Pat-Abs). When matching a reference to a binding defined in the pattern, we use Φ to know the name of the equivalent binding in the scrutinee (E-Pat-Var). We only match if those references are equivalent under the Φ mapping.

Finally, we have the bind pattern (E-Pat-Bind), which may match any sub-term as long as it has the correct type *and* the correct free variables. First, consider reduction, of the simplified $[\![x]\!]_T$ pattern using E-Pat-Bind.

$$\frac{FV(t_s) \cap range(\Phi) \subseteq \emptyset \qquad t_s' = \lceil t_s \rceil}{t_s \boxdot [\![x]\!]_T \;/\; t_t \Longrightarrow^{\Phi} [x \mapsto t_s']t_t}$$

In this case, the premise requires us to show that the intersection of the free variables of t_s and the range of Φ is empty. In other words, it means that t_s does not contain a reference to a binding that was defined in the scrutinee. Since it is only locally closed, it may still have references to binding defined *outside* of the pattern match as those will be valid when inserted in the *then* branch (*cf.*, rule T-Match). If the match succeeds, rule E-Pat-Bind reduces to $[x \mapsto \lceil t_s \rceil]t_t$. Now, consider the case where $\overline{x_k{:}T_K}$ is not empty. This implies that we will match a t_s that might contain references to bindings defined in the pattern. In this case, we cannot simply substitute t_s for x, we first need to bind all free variables. In particular, for each free variable $\Phi(x_k)$ defined in the pattern we η-expand by creating a lambda that receives a staged argument of type $x_k'{:}\lceil T_k \rceil$. In the body of that lambda we substitute $\Phi(x_k)$ with the $\lfloor x_k' \rfloor$. This process results in a curried lambda of the type $\overline{\lceil T_k \rceil \to \lceil T \rceil}$.

Example 3.21. In the following example, we show how to perform β-reduction at level 1 using quote matching. The code below is an encoding the HOAS pattern example of Section 2 for a numerical type **N**.

$$\lambda x{:}\lceil \mathbf{N} \rceil.x \;\; \texttt{match} \;\; \lceil (\lambda y{:}\mathbf{N}.[\![f]\!]_{\mathbf{N}}^{y:\mathbf{N}}) \; [\![z]\!]_{\mathbf{N}} \rceil \;\; \texttt{then} \; f \; z \; \texttt{else} \; x$$

3.3.4 Values. The value definition in Figure 3 extends the value definition in Figure 1. At level 0, the pattern match operations are evaluated away. At any other level, we need to ensure transitively that sub-terms do not have any splices at level 1. As the pattern is not evaluated by itself it is considered a value of its own.

Example 3.22. The ability to match individually quoted constants allows us to unlift quoted value into a value known in the current stage. In the example, we unlift a boolean (**B**)

constant returning the value applied to *succ* or *fail* if it is not a constant.

$$\lambda x{:}\mathbf{B}.\lambda succ{:}\mathbf{B}{\to}T.\lambda fail{:}T.\\ \quad x\ \texttt{match}\ \lceil true\rceil\ \texttt{then}\ succ\ true\ \texttt{else}\\ \quad\quad x\ \texttt{match}\ \lceil false\rceil\ \texttt{then}\ succ\ false\ \texttt{else}\ fail$$

3.3.5 Substitution. To handle the new syntactic form of pattern matching substitution is extended to homomorphically apply substitution to its sub-terms. As patterns can only refer to bindings defined in the pattern itself, as ensured by Γ_p, substitution does not need to go into the pattern.

3.3.6 Soundness. The statements of the progress and preservation theorems carry over unchanged from Section 3.1. Full proofs can be found in Appendix C [24]. Extending the proof of progress with a case for pattern matching is trivial; depending on the pattern reduction, we simply invoke E-Match-Succ or E-Match-Fail. The proof of preservation requires a few additional definitions and lemmas. First of all, we extend the substitution lemma to parallel substitutions:

Lemma 3.23 (Multi-Substitution). $\forall i, j \in \mathbb{N}_0$, *if* $\overline{\Gamma \vdash^j t_k : T_k}$ *and* $\Gamma, \overline{x :^j T_k} \vdash^i t : T$ *then* $\Gamma \vdash^i [\overline{x_k \mapsto t_k}]t : T$

Next, we state well-formedness of the substitution Φ with respect to environments Γ_p and Γ_δ.

Definition 3.24 (Well-formedness of Φ). $\Gamma_p \mid \Gamma_\delta \Vdash \Phi$

We say Φ is well-formed with respect to Γ_p and Γ_δ, written $\Gamma_p \mid \Gamma_\delta \Vdash \Phi$, if and only if Φ is a bijection between $dom(\Gamma_p)$ and $dom(\Gamma_\delta)$, such that $dom(\Gamma_p) \cap dom(\Gamma_\delta) = \emptyset$ and $\forall x_p :^1 T \in \Gamma_p.\ \Phi(x_p) :^1 T \in \Gamma_\delta$.

Using Definition 3.24, we can state type preservation of the pattern reduction. That is, reducing a well typed match $t_s \boxdot p \ / \ t \Longrightarrow^\Phi t'$ results in a well-typed term t'.

Theorem 3.25 (Preservation of Pattern Reduction).
If (1), (2), (3), (4), (5) *then* $\Gamma \vdash^0 t' : T$.

$$\Gamma; \Gamma_\delta \vdash^1 t_s : T_1 \tag{1}$$
$$\Gamma_p \vdash^1 t_p : T_1 \dashv \Gamma_t \tag{2}$$
$$\Gamma; \Gamma_t \vdash^0 t : T \tag{3}$$
$$t_s \boxdot t_p \ / \ t \Longrightarrow^\Phi t' \tag{4}$$
$$\Gamma_p \mid \Gamma_\delta \Vdash \Phi \tag{5}$$

Here, premises (1) to (3) correspond to premises of rule T-Match. Finally, the match case in the proof of preservation follows as a corollary from Theorem 3.25:

Corollary 3.26 (Preservation for Match).
If $\Gamma \vdash^0 \lceil t_s\rceil\ \texttt{match}\ \lceil t_p\rceil\ \texttt{then}\ t_t\ \texttt{else}\ t_e : T$, $\vdash^1 t_s$ **vl** *and* $t_s \boxdot t_p \ / \ t_t \Longrightarrow^\emptyset t'_t$, *then* $\Gamma \vdash^0 t'_t : T$.

3.4 Full Calculus: Cross-Platform Quote Matching

We are now ready to connect all pieces in a final calculus that combines quotes and splices (Section 3.1), cross-platform global definitions (Section 3.2), and analytical macros (Section 3.3). From the combination of the different extensions emerges one additional feature: the ability to pattern match against global definitions. This becomes visible in Figure 4, which adds additional typing and reduction rules for the composed calculus.

3.4.1 Syntax. The syntax is simply the composition of the syntax of the base calculus (Figure 1), of the macro extension (Figure 2), and the pattern matching extension (Figure 3).

3.4.2 Typing. Figure 4 extends the typing judgement with one additional rule T-Pat-Link. This rule allows references to library functions to be typed in patterns. As usual with library functions, the reference can be at any level. Like in Section 3.2, we assume that all rules are modified to pass the library environment Σ through all rules of term and pattern typing.

3.4.3 Evaluation. The operational semantics in Figure 4 extends the combined operational semantics of calculi in Figure 2 and Figure 3. When matching a library function reference (E-Pat-Link) we match if the scrutinee is a reference to the same library function. This pattern does not change or use Φ in any way as it only needs to handle local variables. All other rules are modified to add the same Ω to all terms evaluation as previously done for terms in Section 3.2, patterns are not changed. Thus, reduction rules are of the forms $t \longrightarrow^i_\Omega t'$ for term reduction, $p \mid \Omega \longrightarrow p' \mid \Omega'$ for program reduction, and $t_s \boxdot t_p \ / \ t_t \Longrightarrow^\Phi t'_t$ for pattern reduction. The definition of values and substitution immediately arises from the combination of the different extensions.

Example 3.27. In the following example, we show how the combined calculus can be used to describe an optimization of our "DSL for mathematical operations". We match a numeric expression and check whether it is a call to our global *power* DSL function defined in a library. We then unlift the quoted number using an `unlift` function similar to the one in Example 3.22. The code below is an encoding of the last version of `def powCode` of Section 2 for a numerical type **N**.

$$\texttt{def}\ powCode = \lceil\\ \quad \texttt{fix}\ \lambda rec{:}\lceil\mathbf{N}\rceil{\to}\mathbf{N}{\to}\lceil\mathbf{N}\rceil.\\ \quad\quad \lambda e{:}\lceil\mathbf{N}\rceil.\lambda n{:}\mathbf{N}.\\ \quad\quad\quad e\ \texttt{match}\ \lceil power\ [\![x]\!]_\mathbf{N}\ [\![m_1]\!]_\mathbf{N}\rceil\\ \quad\quad\quad\quad \texttt{then}\ unlift\ m_1\ (\lambda m{:}\mathbf{N}.rec\ x\ (n+m))\ e\\ \quad\quad\quad\quad \texttt{else}\ \ldots\ \text{// generate code}\\ \rceil\ \texttt{in def}\ power^4 = \lceil\\ \quad \lambda x{:}\mathbf{N}.\lfloor powCode\ \lceil power\ x\ 2\rceil\ 2\rfloor\\ \rceil\ \texttt{in}\ p$$

3.4.4 Soundness. The proofs of progress and preservation carry over mostly unchanged. Full proofs can be found in Appendix D [24].

Extended Typing Rules

$$\frac{x{:}T \in \Sigma}{\Gamma_p \mid \Sigma \vdash^i x : T \dashv \emptyset}$$
(T-Pat-Link)

Extended Operational Semantics

$$x \boxdot x \;/\; t \Longrightarrow^{\Phi} t$$
(E-Pat-Link)

Figure 4. Global Definition Matching Extension

4 Implementation

We scaled our calculus to a practical, industrial-strength implementation of a multi-stage macro system in the programming language Scala 3 [23].

4.1 Relation to the Calculus

The static checks as performed by the type checker closely follow the rules of the calculus as described in this paper. The choice of syntax for quotes `'{t}` and for splices `${t}` follows the standard syntax rules of Scala's string interpolators `s"hello $word"` or `s"hello ${word}"` where `world` is spliced in the string. To lift or unlift values, we use the `Expr(x)` syntax which corresponds to `Expr.apply` in expression position and to `Expr.unapply` in pattern position. Global library function definition such as

def $x = \lceil f\ 3 \rceil$ **in**
def $y = \lceil \lambda a{:}\mathrm{N}.a \rceil$ **in**
def $z = \lceil$ **fix** $\lambda rec{:}\mathrm{N}.rec \rceil$ **in** ...

can be expressed in Scala as

```
def x: Int = f(3)
def y: Int ⇒ Int = (a: Int) ⇒ a
def z: Int = z
```

To support quoted pattern matching, we extend the pattern syntax to allow the pattern `'{pat}`. For example, the expression

s **match** $\lceil f\ [\![x]\!]_{\mathrm{N}} \rceil$ **then** t **else** e

can be expressed in Scala as

```
s match { case '{ f($x: Int) } ⇒ t; case _ ⇒ e }
```

where `f` is a global library reference and `$x` represents a locally closed bind pattern. Free variables in bind patterns, as in $[\![x]\!]^{y:\mathrm{N}}_{\mathrm{N}}$, can be specified as `$x(y):Int`. The following term in $\lambda^{\blacktriangle}$ that binds the body of a lambda to x and might have y as a free variable

s **match** $\lceil \lambda y{:}\mathrm{N}.[\![x]\!]^{y:\mathrm{N}}_{\mathrm{N}} \rceil$ **then** t **else** e

can be expressed in Scala as

```
s match
  case '{ (y: Int) ⇒ $x(y):Int } ⇒ t
  case _ ⇒ e
```

Finally, binding an applied function in a pattern

s **match** $\lceil [\![f]\!]_{\mathrm{N}\to\mathrm{N}}\ 3 \rceil$ **then** t **else** e

can be expressed in Scala as

```
s match { case '{ ($f: Int⇒ Int)(3) } ⇒ t; case _ ⇒ e }
```

4.2 Additional Features

In addition, the implementation also supports type polymorphism, existential types in patterns, and runtime multi-stage programming.

Type polymorphism. Type polymorphism is expressed by using the type-class `T:Type`.

```
def one[T: Type](e: Expr[T])(using Quotes): Expr[List[T]] =
  '{ List($e) }
```

The typeclass `Type` is necessary to know the non-erased type of `T` at runtime.

Existential types. Existential types in patterns allow the extraction and use of types that are not statically known.

```
def fuseMap(e: Expr[List[Int]])(using Quotes) =
  e match
    case '{ ($ls: List[u]).map[v]($f).map[Int]($g) } ⇒
      '{ $ls.map(x ⇒ $g($f(x))) }
    case _ ⇒ e
```

In the example, the pattern and uses of the extracted types (`u` and `v`) can be statically checked. In general, we use lower case letters for existential type bindings.

Runtime multi-staging. The library function *run* compiles and executes quoted code at runtime. This allows programmers to perform runtime multi-stage programming.

```
import scala.quoted.staging.*
given Compiler = ...
val power2 = run('{ (x: Int) ⇒ ${powCode('x, 2)} })
```

Scope extrusion detection. As Scala is not a pure functional language, we need to make sure that the expressions are not extruded from their scope through side channels such as mutable state, exceptions, or even into a *run*. For example, we can abuse mutable state to extrude a quoted local variable follows:

```
var z: Expr[Int] = null
'{ val x = 7; ${ z = '{x}; ... } }
// z now contains '{x} and x will never be in scope anymore
```

To avoid this, we add runtime checks that track the stack of nested splices we are in when we create a quote. To know if an expression is extruded from its scope, we compare the stack of the creation site with the one at the call site and report a runtime error, if the two are not compatible. In our implementation, all quotation functionality is provided by means of an implicit/contextual instance of the type `Quotes`, which also serves as a representation of a persistent stack of scopes. Each quote will use the instance that is currently in scope and each splice will introduce a new instance.

5 Discussion and Related Work

LISP. LISP has a very simple way to treat *programs as data* based on the uniform representation of programs as lists. Quotation turns fragments of (unevaluated) code into data: `'42` is a number, `'a` is a symbol, `'(+ 3 4)` is a list of the quoted constituents. Quasiquotation—with a backquote—lets us escape inside a program fragment of e.g., a whole list with a comma operator that can *unquote* and evaluate a part of the quasiquoted expression e.g., `` `(1 2 ,(+ 3 4)) ``.

Racket. Racket has a sophisticated macro system. On one hand, contrary to Scala, Racket is dynamically-typed and on the other, Typed Racket will type-check all expressions at the run-time phase of the given module [27]. Despite these fundamental differences, it is worth noting that Racket supports pattern matching with quasiquotes (quasipatterns). Interestingly, Racket takes one step further and much like the quasiquote expression form, unquote and unquote-splicing escape back to normal patterns which is something that we do not support.

Multi-Stage-Programming. Multi-Stage Programming (MSP) transfers the concepts of quotes, quasiquotes, unquotes and staged evaluation [7, 10] in a statically-scoped, modularly-type safe environment [21]. MSP, popularised by MetaML [25] and MetaOCaml [3, 12, 13], made generative programming easier [5], effectively narrowing the gap of writing complex solutions of code manipulation such as: code optimizations [28] and DSL implementations [4, 26]. Fred McBride [15] highlights the need to bridge the gap of expressing *computer-aided manipulation of symbols*. Arguing that it is important to lower the cognitive barrier of reading and writing algebraic manipulators such as algebraic simplificators and integrators, he develops the first pattern matching facility for LISP; a form that provides a *natural description* to increase the user's *problem solving potential*. MacroML [9] used the quotation system of MetaML to define macros. The two fundamental quasiquotation operators in Scala 3 were inspired by MetaML/MacroML and BER MetaOCaml.

Template Haskell. Haskell was introduces to metaprogramming using quasiquotes with Template Haskell [20]. Neither MetaOCaml, MetaML, or Template Haskell support pattern matching with quasiquotes.

Squid. Squid [16, 18], a metaprogramming library for Scala, advances the state of the art of staging systems and puts quasiquotes at the center of user-defined optimizations. The user can pattern match over existing code and implement retroactive optimizations modularly. To the best of our knowledge, in an earlier version of Squid, Parreaux et al. [16] were the first to represent locally closed terms as HOAS functions. However, they later revisited the approach and instead track free variables in the type. While expressive, this approach requires advanced typing features such as path-dependent types, singleton types, and intersection types.

Modal logic. Our calculus is closely related to λ° [6] and $\lambda^{\Box}$ [7]. These calculi capture the temporal/modal logic essence of multistage programming. They only support code generation but not code analysis.

6 Future Work

While the calculus models separate compilation, as well as generative and analytical macros, it does not yet fully cover all features of our implementation in Scala 3. In particular, two important features are type polymorphism and existential pattern types. It would be interesting to extend the calculus in future work to account for these features. However, adding them will most likely result in additional complexity since it will require some kind of type unification.

The calculus assumes the lack of side channels, for which we use dynamic scope extrusion detection. It is worth formalizing this mechanism and precisely describe how it can be implemented at runtime or expressed as a static check.

Another way to extend the system would be to investigate the possibility to also allow matching on programs that use quotes, splices, and matches themselves. Adding pattern matching on quotes and splices seems to involve a more general notion of staged eta-expansion. In order to add support for matching on the match itself is less clear and would require some form of meta-pattern. While both extensions are interesting, we expect the metatheory to be significantly more involved, and thus neither of the two features is implemented in Scala 3.

7 Conclusion

We introduced a calculus ($\lambda^{\blacktriangle}$) for well-typed and hygienic multi-stage metaprogramming that allows both generative and analytical macros. We proved soundness of $\lambda^{\blacktriangle}$ and implemented it in Scala 3.

Acknowledgments

We gratefully acknowledge funding by the Swiss National Science Foundation under grant 407540_167213 (Programming Language Abstractions for Big Data).

References

[1] Leif Andersen, Stephen Chang, and Matthias Felleisen. 2017. Super 8 Languages for Making Movies (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 30 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110274

[2] Henk P. Barendregt. 1992. *Lambda Calculi with Types.* 117–309 pages.

[3] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proc. of the 2nd International Conference on Generative Programming and Component Engineering* (Erfurt, Germany) *(GPCE '03)*. Springer-Verlag, Berlin, Heidelberg, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

[4] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. *DSL Implementation in MetaOCaml, Template Haskell, and C++*. Springer Berlin Heidelberg, Berlin, Heidelberg, 51–72. https://doi.org/10.1007/978-3-540-25935-0_4

[5] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative Programming. In *Object-Oriented Technology ECOOP 2002 Workshop Reader*, Juan Hernández and Ana Moreira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 15–29. https://doi.org/10.1007/3-540-36208-8_2

[6] Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1, Article 1 (March 2017), 45 pages. https://doi.org/10.1145/3011069

[7] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. https://doi.org/10.1145/382780.382785

[8] Matthew Flatt. 2002. Composable and Compilable Macros: You Want It When? *SIGPLAN Not.* 37, 9 (Sept. 2002), 72–83. https://doi.org/10.1145/583852.581486

[9] Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. *SIGPLAN Not.* 36, 10 (Oct. 2001), 74–85. https://doi.org/10.1145/507669.507646

[10] Ulrik Jørring and William L. Scherlis. 1986. *Compilers and Staging Transformations.* Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/512644.512652

[11] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[12] Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages* 5, 1 (2018), 1–101. https://doi.org/10.1561/2500000038

[13] Oleg Kiselyov and Chung-chieh Shan. 2010. The MetaOCaml files - Status report and research proposal. In *ACM SIGPLAN Workshop on ML*.

[14] Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019), 39 pages. https://doi.org/10.1145/3354584

[15] Fred McBride. 1970. *Computer Aided Manipulation of Symbols.* Ph.D. Dissertation. Queen's University of Belfast.

[16] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted Staged Rewriting: A Practical Approach to Library-Defined Optimizations. *SIGPLAN Not.* 52, 12 (Oct. 2017), 131–145. https://doi.org/10.1145/3170492.3136043

[17] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: Type-Safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 56–66. https://doi.org/10.1145/3136000.3136005

[18] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017. Unifying Analytic and Statically-Typed Quasiquotes. *Proc. ACM Program. Lang.* 2, POPL, Article 13 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158101

[19] F. Pfenning and C. Elliott. 1988. Higher-Order Abstract Syntax. *SIGPLAN Not.* 23, 7 (June 1988), 199–208. https://doi.org/10.1145/960116.54010

[20] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[21] Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. 2017. Structured Program Generation Techniques. In *Grand Timely Topics in Software Engineering*, Jácome Cunha, João P. Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev (Eds.). Springer International Publishing, Cham, 154–178. https://doi.org/10.1007/978-3-319-60074-1_7

[22] Nicolas Stucki, Aggelos Biboudis, Sébastien Doeraene, and Martin Odersky. 2020. Semantics-Preserving Inlining for Metaprogramming. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala* (Virtual, USA) *(SCALA 2020)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3426426.3428486

[23] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-Stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3278122.3278139

[24] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. *Proof of Multi-Stage Programming with Generative and Analytical Macros.* Technical Report. EPFL.

[25] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. *SIGPLAN Not.* 32, 12 (Dec. 1997), 203–217. https://doi.org/10.1145/258994.259019

[26] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. *SIGPLAN Not.* 46, 6 (June 2011), 132–141. https://doi.org/10.1145/1993316.1993514

[27] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. 2021. *The Typed Racket Guide - Caveats and Limitations.*

[28] T Veldhuizen and E Gannon. 1998. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. of the 1998 SIAM Workshop: Object Oriented Methods for Interoperable Scientific and Engineering Computing*. 286–295.