# A Practical Unification of Multi-stage Programming and Macros

Nicolas Stucki
EPFL
Switzerland
nicolas.stucki@epfl.ch

Aggelos Biboudis
EPFL
Switzerland
aggelos.biboudis@epfl.ch

Martin Odersky
EPFL
Switzerland
martin.odersky@epfl.ch

## Abstract

Program generation is indispensable. We propose a novel unification of two existing metaprogramming techniques: multi-stage programming and hygienic generative macros. The former supports runtime code generation and execution in a type-safe manner while the latter offers compile-time code generation.

In this work we draw upon a long line of research on metaprogramming, starting with Lisp, MetaML and MetaOCaml. We provide direct support for quotes, splices and top-level splices, all regulated uniformly by a level-counting *Phase Consistency Principle*. Our design enables the construction and combination of code values for both expressions and types. Moreover, code generation can happen either at runtime *à la* MetaML or at compile time, in a macro fashion, *à la* MacroML.

We provide an implementation of our design in Scala and we present two case studies. The first implements the Hidden Markov Model, Shonan Challenge for HPC. The second implements the staged streaming library Strymonas.

**CCS Concepts** • **Software and its engineering → Language features**; *Macro languages*;

**Keywords** Macros, Multi-stage programming, Scala

## 1 Introduction

Generative programming [9] is widely used in scenarios such as code configuration of libraries, code optimizations [43] and DSL implementations [8, 41]. There are various kinds of program generation systems ranging from completely syntax-based and unhygienic, to fully typed [35]. Modern macro systems, like Racket's, can extend the syntax of the language [11]. On the flipside, other program generation systems may provide a fixed set of constructs offering staged evaluation [10, 16] like MetaML [38] and MetaOCaml [6, 20, 21, 23].

The latter techniques established a new programming paradigm, called *Multi-stage Programming* (MSP) offering a principled, well-scoped and type-safe approach to code generation [37]. Programmers make use of two constructs, *quote* and *splice*, to delay and compose representations of expressions. Conceptually, users are able to manually indicate which parts of their program are dynamic and which static. Even though this technique is inspired by advancements in partial evaluation [26] it proved useful to have it in a programming language with first-class support. Part of the power of this programming model, comes from a regulation mechanism that attributes *levels* to terms [36]; these systems are type-safe in a modular way (type checking the generator ensures the validity of the generated code). Nowadays, gaining inspiration from MetaML and MetaOCaml, many programming languages provide support for similar mechanisms such as F#, Haskell (Template Haskell [33] and later Typed Template Haskell [15]), Converge [42] and others. While MSP is primarily a metaprogramming technique for *runtime* code generation it has been shown that its semantics can specify compile-time metaprogramming as well.

MacroML [12] showed that the treatment of staged evaluation can form the basis for generative macros (i.e. macros that cannot inspect code) or more precisely, *function inlining*. Theoretically it has been proven that MacroML's interpretation is a denotational semantics where MetaML is the internal language of the model. Monnier et al.[25] first expressed inlining as staged computation but MacroML offered a user-level perspective by reusing the same mechanisms of quotes and splices; where splices can appear at the top-level (not nested in a quote). Modular Macros [44] prototyped a compile-time variant of MetaOCaml which also comprises part of our inspiration.

```
def power_s(x: Expr[Double], n: Int): Expr[Double] =
  if (n == 0) '(1.0)
  else if (n % 2 == 1) '(~x * ~power_s(x, n - 1))
  else '{ val y = ~x * ~x; ~power_s('(y), n / 2) }

inline def power(x: Double, inline n: Int): Double =
  ~power_s('(x), n)

val x = 2
// 1) staged, runtime generation
val power5 = ('{ (x: Double) => ~power_s('(x), 5)}).run
power5(x)
// 2) macro, compile-time generation
power(x, 5)
// Both generate: { val y = x * x; val y2 = y * y; x * y2 }
```

**Figure 1.** Power function, staged or inlined

While the same line of work inspired many metaprogramming libraries and language features, to our knowledge built-in support for both run-time MSP and generative macros has not been implemented previously in a unifying manner. We advocate that such a unification has a two-fold benefit: 1) users rely on a single abstraction to express code generation and 2) having a single subsystem in the compiler favors maintainability. Our view regarding top-level splices is on par with the benefits of MSP on domain-specific optimizations[7, 21, 22]: in modern programming languages, inlining (à la C++) with a *sufficiently smart* partial evaluator is not necessarily equivalent with domain-specific optimizations that can be done at compile-time.

In our work a staged library can be used, unaltered, either as a macro or a run-time code generator. We illustrate staging and macros via the folklore example of a simple power function, which has been used for demonstrating partial evaluation techniques. The power_s, staged function is defined recursively using the basic method of exponentiation by squaring. The inline function power becomes a macro by expanding power_s. In Figure 1 we see two different ways to use it: 1) staged; generation happens at runtime and 2) inlined generation happens at compile-time.

**Contributions** In this paper, inspired from MetaML and MacroML we present a practical implementation of *homogeneous generative metaprogramming* (HGMP) for Scala:

- We present a design with *quotes*, *splices*, and *top-level splices* to support both MSP and macros simultaneously.
- We extend the operation of splicing to handle *terms* and *types* uniformly.
- We present how our system operates under a MetaML-inspired check, *Phase Consistency Principle* (PCP), that regulates free variable accesses in quoted and spliced expressions and types *uniformly, for both MSP and macros*.

Scala is a multi-paradigm programming language for the JVM offering a metaprogramming API called *scala.reflect* [5]. scala.reflect supports type-aware, runtime and compile-time code generation providing an expressive and powerful system to the user (both generative and analytical). Besides the success of scala.reflect, the API exposed compiler internals and gave rise to portability problems between compiler versions [24]. We implemented our system for the Dotty [39] compiler for Scala and we believe that the design is portable in other languages as well.

**Organization** First, in Section 2, we introduce a motivating example to explain the high-level semantics of quotes and splices. In Section 3 we present PCP and the details of multi-staging and macros. In Section 4 we discuss how to implement *cross-stage persistence* (CSP) in this system. In Section 5 we show how to simplify the handling of type splices in quoted code. In Section 6 we discuss lifted lambdas and $\beta$-reduction optimizations. Section 7 describes the implementation in Dotty. Section 8 presents two case studies[1]: (i) we give a sample solution to the Hidden Markov Model challenge as specified in Shonan Challenge for Generative Programming [1] and (ii) we port Strymonas [22], a staged library for streams. In Section 9 we discuss the related work and conclude in Section 10.

## 2 Overview of Quotes and Splices

Our metaprogramming system is built on two well-known fundamental operations: quotation[2] and splicing. A quotation is expressed as '(...) or '{...} for expressions (both forms are equivalent) and as '[...] for types. Splicing is expressed with the ~ prefix operator.

If e is an expression, then '(e) or '{e} represent the opaque *typed abstract syntax tree* representing e. If T is a type, then '[T] represents the opaque *type structure* representing T. The precise definitions of typed abstract syntax tree or type structure do not matter for now, the expressions are used only to give some intuition that they represent code as a value. Conversely, ~e evaluates the expression e, which must yield a typed abstract syntax tree or type structure, and embeds the result as an expression (respectively, type) in the enclosing program. Informally, by quoting we delay the evaluation of an expression—or we *stage*, in MSP terms—and by splicing, we evaluate an expression before embedding the result in the surrounding quote.

Quotes and splices are duals of each other. For arbitrary expressions e: T and types T we have ~'(e) = e and ~'[T] = T; for arbitrary AST-typed expressions e2: Expr[T] and t: Type[T] we have '(~e) = e and '(~t) = t.

---

[1]The code of the case studies, along with unit tests and benchmarks are at https://github.com/nicolasstucki/dotty-staging-gpce-2018
[2]Or more accurately quasiquotation, which represents quotes with unquoted expressions getting evaluated first.

Quoted code values can have the following two types:

- `Expr[T]`: *typed abstract syntax trees* representing an expression of type `T`.
- `Type[T]`: *type structures* representing a type `T`.

Quoting can be seen as the function that takes expressions of type `T` to expressions of type `Expr[T]` and a type `T` to an expression of type `Type[T]`. Splicing takes expressions of type `Expr[T]` to expressions of type `T` and an expression of type `Type[T]` to a type `T`. For example, the code below presents `unrolled`, a recursive function which generates code that will explicitly perform the given operation for each element of a known list. The elements of the list are expressions themselves and the function maps expressions of integers to expressions of `Unit` (or statements). We use quotes to delay the representation of the return value and splice the result of the evaluation of `f(head)` and `unrolled(tail, f)`.

```
def unrolled(list: List[Expr[Int]], f: Expr[Int] =>
    Expr[Unit]): Expr[Unit] = list match {
  case head :: tail => '{ ~f(head); ~unrolled(tail, f) }
  case Nil => '()
}
unrolled(List('(1), '(2)), (i: Expr[Int]) => '(println(~i)))
// Generates: '{ println(1); println(2); () }
```

Similarly, it is also possible to splice types in the quoted code giving us the capability of creating expressions of types not known at compile time. In the example below `x` has type `Expr[T]` but we require `T` itself to be unknown.

```
def some[T](x: Expr[T], t:Type[T]): Expr[Some[T]] =
  '{ Some[~t](~x) }
def someType[T](t:Type[T]): Type[Some[T]] =
  '[Some[~t]]
```

In this section we showed how to unroll a loop for a known sequence of staged expressions. However, we have deliberately not yet discussed whether code generation happens at compile-time or run-time.

## 3 Unifying Multi-stage Programming and Macros

This section introduces our *Phase Consistency Principle* (PCP) and how we employ it to check that the staged code is consistent. Then, we will see how quotes and splices are used in multi-stage programming and macros alike.

To start, let us adapt the requirements of our `unrolled` example and instead of unrolling a loop for a known sequence of staged expressions we want to stage a loop for an unknown sequence. The following example shows what happens when we start nesting quotes, in splices, in quotes. `~f('(element))` is inside a quote, which means that the expression will generate some code that will be spliced in-place. Inside it we refer to `'(element)`, which is defined in the outer

quote. Additionally, we make this version generic on `T` with `Type[T]`, which is spliced in the type of `val element: ~t`.

```
def staged[T](arr: Expr[Array[T]], f: Expr[T] =>
    Expr[Unit])(implicit t: Type[T]): Expr[Unit] = '{
  var i: Int = 0
  while (i < (~arr).length) {
    val element: ~t = (~arr)(i)
    ~f('(element))
    i += 1
  }
}
```

**Intuition**   The stage in which the code is run is determined by the difference between the number of splice scopes and quote scopes in which the code is embedded.

- If there is a top-level splice—a splice not enclosed in quotes—the code is run at *compile-time* (i.e. as a macro).
- If the number of splices equals the number of quotes, the code is compiled and run as usual.
- If the number of quotes exceeds the number of splices, the code is staged. That is, it produces a *typed abstract syntax tree* or *type structure* at run-time. A quote excess of more than one corresponds to multi-staged programming.

### 3.1 Phase Consistency Principle

A fundamental *phase consistency principle* (PCP) regulates accesses to free variables in quoted and spliced code:

- *For any free variable reference x, the number of quoted scopes and the number of spliced scopes between the reference to x and the definition of x must be equal.*

Here, the self-reference to an object (`this`) counts as free variables. On the other hand, we assume that all imports are fully expanded and that `_root_` is not a free variable. So references to global definitions are allowed everywhere.

For example, in `staged`, `element` is consistent because there is one `~` and one `'` between the definition and its use. The same is true for `arr` and `t` even though there is a `'` first and then a `~`. The type `Int` of `var i: Int` is consistent as it is expanded to `_root_.scala.Int`, thus not considered a free variable. Primitive language operation such as `+=` in `i += 1` are also globally identifiable and hence not free variables. The variable `i` is consistent because it is only used locally in the `'`, i.e. it is not a free variable of any other quote or splice.

The phase consistency principle can be motivated as follows: first, suppose the result of a program `P` is some quoted code `'{ ... x ... }` that refers to a free variable `x` in `P`. This can be represented only by referring to the original variable `x`. Hence, the result of the program will need to persist the program state itself as one of its parts. This operation should not be considered positive in general as different stages might be run on different machines, as macros do. Hence this situation should be made illegal. Dually, suppose a top-level part of

a program is a spliced code ~{ ... x ... } that refers to a free variable x in P. This would mean that we refer during *construction* of P to a value that is available only during *execution* of P. This is of course impossible and therefore needs to be ruled out. Now, the small-step evaluation of a program will reduce quotes and splices in equal measures using cancellation rules which informally they state that: ~'(e) ⇒ e, '(~e) ⇒ e, ~'[T] ⇒ T and '[~T] ⇒ T. However, the evaluation will neither create or remove quotes (or splices) individually. So PCP ensures that the program elaboration will lead to neither of the two unwanted situations described above.

In what concerns the range of features it covers, principled metaprogramming is quite close to the MetaML family of languages. One difference is that MetaML does not have an equivalent of the PCP - quoted code in MetaML *can* access variables in its immediately enclosing environment, with some restrictions and caveats since such accesses involve serialization. However, this does not constitute a fundamental gain in expressiveness. Quotes and splices allow CSP by lifting which enable the implementation of such accesses within the confines of the PCP. This is further explained in section 4.1.

## 3.2 Supporting Multi-stage Programming

As discussed so far, the system allows code to be staged, i.e. be prepared to be executed at a later stage. To be able to consume the staged code, Expr[T] does not only provide the ~ prefix method, it also provides run that evaluates the code and returns a value of type T. Note that ~ and run both map from Expr[T] to T but only ~ is subject to the PCP, whereas run is just a normal method. We also provide a show method to display the code in String form.

```
def sumCodeFor(arr: Expr[Array[Int]]): Expr[Int] = '{
  var sum = 0
  ~staged(arr, x => '(sum += ~x))
  sum
}
val sumCode = '{ (arr: Array[Int]) => ~sumCodeFor('(arr)) }

println(sumCode.show)
// (arr: Array[Int]) => {
//   var sum: Int = 0
//   var i: Int = 0
//   while (i < arr.length) {
//     val element: Int = arr(i)
//     sum += element
//     i += 1
//   }
//   sum
// }

// evaluate the code of sumCode which return the function
val sum: Array[Int] => Int = sumCode.run
sum(Array(1, 2, 3)) // Returns 6
sum(Array(2, 3, 4, 5)) // Returns 14
```

***Limitations to Splicing*** Quotes and splices are duals as far as the PCP is concerned. But there is an additional restriction that needs to be imposed on splices to guarantee soundness: *code in splices must be free of scope extrusions*, which we guarantee by disallowing effects. The restriction prevents code like this:

```
var x: Expr[T] = _
'{ (y: T) => ~{ x = '(y); 1 } }
```

This code, if it was accepted, would *extrude* a reference to a quoted variable y from its scope. This means we subsequently access a variable outside the scope where it is defined, which is problematic. The code is clearly phase consistent, so we cannot use PCP to rule it out. Instead, we postulate a future effect system that can guarantee that splices are pure. In the absence of such a system we simply demand that spliced expressions are pure by convention, and allow for undefined compiler behavior if they are not.

A second limitation comes from the use of the method run in splices. Consider the following expression:

```
'{ (x: Int) => ~{ ('(x)).run; 1 } }
```

This is again phase correct but will lead us into trouble. Indeed, evaluating the run will reduce the expression ('(x)).run to x. But then the result

```
'{ (x: Int) => ~{ x; 1 } }
```

is no longer phase correct. To prevent this soundness hole it seems easiest to classify run as a side-effecting operation. It would thus be prevented from appearing in splices. In a base language with side-effects we'd have to do this anyway: Since run runs arbitrary code it can always produce a side effect if the code it runs produces one.

## 3.3 Supporting Macros

Seen by itself, quotes and splices-based metaprogramming looks more like a system for staging than one supporting macros. But combined with Dotty's inline[3] it can be used as a compile-time metaprogramming system as well. Effectively executing the staging at compile-time and generating the full program with no overhead at run-time.

***Inline*** In Dotty the inline keyword can be added to a val, def or a parameter to a inline def. A definition marked as inline will be inlined when the code is typed checked. Informally speaking, a val and a parameter marked as such, will be inlined only if they are a constant or an inlined constant of primitive value type (Boolean, Byte, Short, Int, Long, Float, Double, Char or String). Other values are disallowed to avoid

---

[3]Dotty's inline keyword guarantees inlining and inlines the code at type-checking time. [40]

moving any side effects and changing the semantics of the program.

Function definitions are always inlined in a semantic preserving way as they are in essence $\beta$-reductions. Parameters have *call by value* (CBV) semantics, hence they are evaluated before the invocation to the function and bound to local `val`s. If the parameters are marked as *call by name* (CBN) (which is realized by prefixing the type with `=>`) then the argument is directly inlined in each reference to the parameter. Inline parameters are inlined in the resulting code and guaranteed to be a constant value.

***Macro***   In combination with `inline`, macro elaboration can be understood as a combination of a staging library and a quoted program. An inline function, such as `Macros.sum` that contains a splice operation outside an enclosing quote, is called a *macro*. Macros are supposed to be expanded in a subsequent phase, i.e. in a quoted context. Therefore, they are also type checked as if they were in a quoted context. For instance, the definition of `sum` is type-checked as if it appeared inside quotes. This makes the call from `sum` to `sumCodeFor` phase-correct, even if we assume that both definitions are local.

```
object Macros {
  inline def sum(arr: Array[Int]): Int = ~sumCodeFor('(arr))
  def sumCodeFor(arr: Expr[Array[Int]]): Expr[Int] =
   ... // same definitions as before
}
```

On the other side we will have an `App` that will use the `sum` function.

```
object App {
  val array = Array(1, 2, 3)
  Macros.sum(array)
}
```

When this program is compiled it can be thought of as a quoted program that is being staged. Inlining the `sum` function would give the following phase correct `App`:

```
object App {
  val array = Array(1, 2, 3)
  ~Macros.sumCodeFor('(array))
}
```

Phase correctness is unchanged for `Macros` and `array`, inlining preserves PCP. But now we have a top-level splice in the `App`, which is not an issue as we assumed that `App` is a quoted program being staged. The next step is to evaluate `sumCodeFor('(array))` and place the result in the splice.

```
object App {
  val array = Array(1, 2, 3)
  ~('{ var sum = 0; ...; sum })
  // or { var sum = 0; ...; sum } by cancelation rule
}
```

The second role of `inline` in a macro is to make constants available in all stages. To illustrate this, consider the `sumN` function that makes use of a statically known size:

```
object Macros {
  inline def sumN(inline size: Int, arr: Array[Int]): Int =
    ~sumN_m(size, '(arr))
  def sumN_m(size: Int, arr: Expr[Array]): Expr[Int] =
    ... // implemented in section 4.1
}
```

The reference to `size` as an argument in `sumN_m(size, '(arr))` seems not phase-consistent, since `size` appears in a splice without an enclosing quote. Normally that would be a problem because it means that we need the *value* of `size` at compile-time, which is not available in general. But since `size` is an `inline` parameter, we know that at the macro expansion point `size` will be a known constant value. To reflect this, we will assume that all `inline` values are not free variables as they will be known after inlining:

- If `x` is an `inline` value (or an `inline` parameter of an `inline` function) it is not a free variable of the quote or splice.

Additionally we may also have macros with type parameters that are used inside a top-level splice. For example, the type parameter `T` used in the macro in the following version of `foreach` exemplifies this.

```
inline def foreach[T](arr: Array[T], f: T => Unit): Unit =
    ~staged[ T ](...)
```

When inlined the type `T` will become a known type, this implies that macro type parameters can have the same treatment as `inline` parameters.

- If `T` is a type parameter of an `inline` function, then `T` is not a free variable of the quote or splice.

***Avoiding an Interpreter***   Providing an interpreter for the full language is quite difficult, and it is even more difficult to make that interpreter run efficiently. To avoid needing a full interpreter, we can impose the following restrictions on the use of splices to simplify the evaluation of the code in top-level splices.

1. A top-level splice must appear in an `inline` function (turning that function into a macro).
2. Splices direcly inside splices are not allowed.
3. A macro is effectively final and it may not override other methods.
4. Macros are consumed in by other modules/libraries.

These restrictions allow us to stage and compile (at macro compilation time) the code that would be interpreted at macro expansion time, which entails that the macro will be expanded using compiled code. Which is faster and does not require the implementation of an AST interpreter for the full language.

## 4 Cross-stage Persistence

Cross-Stage Persistence refers to persisting some value or type, available in the current stage, for use in a future stage. We support persisting base types, ADT encodings (classes) and abstract types by copying using `Liftable`. Fully qualified names (terms or types) are always shared. Finally, polymorphic (e.g., def `lift[T](x: T) = '(x)`)) are not supported directly unless written as def `lift[T: Liftable](x: T) = x.toExpr`.

### 4.1 Lifting Expressions

Consider the implementation of `sumN_m` used in the previous macro:

```
def sumN_m(size: Int, arr: Expr[Array[Int]]): Expr[Int] = '{
  assert((~arr).length == ~size.toExpr)
  var sum = 0
  ~unrolled(List.tabulate(size)(_.toExpr),
      x => '(sum += (~arr)(~x)))
  sum
}
```

The assertion, `assert((~arr).length == ~size.toExpr)`, looks suspicious. The parameter `size` is declared as an `Int`, yet it is converted to an `Expr[Int]` with `toExpr`. Shouldn't `size` be quoted? In fact, this would not work since replacing `size` by `'(size)` in the clause would not be phase correct.

What happens instead is an extension method `toExpr` is added. The expression `size.toExpr` is then expanded to the equivalent of:

```
implicity[Liftable[Int]].toExpr(size)
```

The extension method says that values of types implementing the `Liftable` type class can be lifted (serialized) to `Expr` values using `toExpr` when `scala.quoted._` is imported. We provide instance definitions of `Liftable` for several types including `Boolean`, `String`, and all primitive number types. For example, `Int` values can be converted to `Expr[Int]` values by wrapping the value in a `Literal` tree node. This makes use of the underlying tree representation in the compiler for efficiency. But the `Liftable` instances are nevertheless *not magic* in the sense that they could all be defined in a user program without knowing anything about the representation of `Expr` trees. For instance, here is a possible instance of `Liftable[Boolean]`:

```
implicit def BooleanIsLiftable: Liftable[Boolean] = new {
  def toExpr(bool: Boolean): Expr[Boolean] =
    if (bool) '(true)
    else '(false)
}
```

Once we can lift bits, we can work our way up. For instance, here is a possible implementation of `Liftable[Int]` that does not use the underlying tree machinery:

```
implicit def IntIsLiftable: Liftable[Int] = new {
  def toExpr(n: Int): Expr[Int] = n match {
    case Int.MinValue     => '(Int.MinValue)
    case _ if n < 0       => '(-(~toExpr(n)))
    case 0                => '(0)
    case _ if n % 2 == 0  => '(~toExpr(n / 2) * 2)
    case _                => '(~toExpr(n / 2) * 2 + 1)
  }
}
```

Since `Liftable` is a type class, its instances can be conditional. For example, a `List` is liftable if its element is:

```
implicit def ListIsLiftable[T: Liftable: Type]:
    Liftable[List[T]] = new {
  def toExpr(xs: List[T]): Expr[List[T]] = xs match {
    case x :: xs1 => '(~x.toExpr :: ~toExpr(xs1))
    case Nil => '(Nil: List[T])
  }
}
```

In the end, `Liftable` resembles very much a serialization framework. Like the latter, it can be derived systematically for all collections, case classes and enums.

### 4.2 Implicitly Lifted Types

The metaprogramming system has to be able to take a type `T` and convert it to a type structure of type `Type[T]` that can be spliced. This means that all free variables of the type `T` refer to types and values defined in the current stage.

For a reference to a global class, this is easy, just issue the fully qualified name of the class. Members of reifiable types are handled by just reifying the containing type together with the member name. But what to do about references to type parameters or local type definitions that are not defined in the current stage? Here, we cannot construct the `Type[T]` tree directly, so we need to get it from a possibly recursive implicit search. For instance, to provide `implicitly[Type[List[T]]]`, the lifted type `Type[List[T]]` required by `ListIsLiftable` where `T` is not defined in the current stage. We construct the type constructor of `List` applied to the splice of the result of searching for an implicit `Type[T]`, which is equivalent to `'[List[~implicitly[Type[T]]]]`.

## 5 Healing Phase of Types

To avoid clutter, the compiler tries to heal a phase-incorrect reference to a type to a spliced lifted type, by rewriting `T` to `~implicitly[Type[T]]`. For instance, the user-level definition of `staged` would be rewritten, replacing the reference to `T` with `~implicitly[Type[T]]`. The `implicitly` query succeeds because there is an implicit value of type `Type[T]` available (namely the evidence parameter corresponding to the context bound `Type`[4]), and the reference to that value is phase-correct.

---

[4]The notation `T: Type` is called a context bound and it is a shorthand for the `(implicit t: Type[T])` parameter in the original signature.

If that was not the case, the phase inconsistency for `T` would be reported as an error.

```scala
def staged[ T: Type ](arr: Expr[Array[T]], f: Expr[T] =>
    Expr[Unit]): Expr[Unit] = '{
  var i = 0
  while (i < (~arr).length) {
    val element: T = (~arr)(i)
    ~f('(element))
    i += 1
  }
}
```

## 6  Staged Lambdas

When staging programs in a functional language there are two unavoidable abstractions: staged lambda `Expr[T => U]` and staging lambda `Expr[T] => Expr[U]`. The former is a function that will exist in the next stage whereas the second one is a function that exists in the current stage.

Below we show an instance where these two do not match. The `'(f)` has type `Expr[Int => Unit]` and `staged` expects an `Expr[Int] => Expr[Unit]`. In general we it is practical to have a mechanism to go from `Expr[T => U]` to `Expr[T] => Expr[U]` and *vice versa* (as desribed in [38]).

```scala
inline def foreach(arr: Array[Int], f: Int => Unit): Unit =
    ~staged('(arr), '(f) )
def staged(arr: Expr[Array[Int]],
    f: Expr[Int] => Expr[Unit] ): Expr[Unit] = ...
```

We provide a conversion from `Expr[T => U]` to `Expr[U] => Expr[T]` with the decorator `AsFunction`. This decorator gives `Expr` the `apply` operation of an applicative functor, where `Expr`s over function types can be applied to `Expr` arguments. The definition of `AsFunction(f).apply(x)` is assumed to be functionally the same as `'((~f)(~x))`, however it optimizes this call by returning the result of beta-reducing `f(x)` if `f` is a known lambda expression[5].

The `AsFunction` decorator distributes applications of `Expr` over function arrows:

```scala
AsFunction(_).apply: Expr[T => U] => (Expr[T] => Expr[U])
```

We can use the conversion in our previous `foreach` example as follows

```scala
~foreach('(arr), x => ('(f))(x))
```

Its dual, let's call it `reflect`, can be defined in user space as follows:

```scala
def reflect[T: Type, U: Type](f: Expr[T] => Expr[U]):
    Expr[T => U] = '{ (x: T) => ~f('(x)) }
```

---

[5]Without the $\beta$-reduction requirement it is possible to implement in user code.

## 7  Implementation

The described metaprogramming system is implemented in the Dotty compiler [39] directly, however it can be ported to other ecosystems as well. The necessary ingredients to port the design in other ecosystems are the following:

- A typed and lexically-scoped language.
- Syntax support for quotes and splices.
- Support for the serialization of *typed* code.
- Support for separate compilation or the use of an existing interpreter (for macros).

### 7.1  Syntax changes

A splice `~e` on an expression of type `Expr[T]` is a normal prefix operator such as `def unary_~`. To make it work as a type operator on `Type[T]` as well, we need a syntax change that introduces prefix operators as types. With this addition, we can implement the type splice with `type unary_~`. Analogously to the situation with expressions, a prefix type operator such as `~e` is a shorthand for the type `e.unary_~`.

```scala
sealed abstract class Expr[T] {
  def  unary_~: T
}
sealed abstract class Type[T] {
  type unary_~ = T
}
```

Quotes are supported by introducing new tokens `'(`, `'{`, and `'[` and adding quoted variants `'(...)`, `'{...}` and `'[...]` to the valid expressions.

### 7.2  Implementation in Dotty

Quotes and splices are primitive forms in the generated typed abstract syntax trees. They are eliminated in an expansion phase after type checking and before starting the transformation of the trees to bytecode. This phase checks that the PCP holds, pickles contents of the quotes and expands top-level splices inserted by macros. All of these can be performed at the same time.

***PCP check***   To check phase consistency we traverse the tree top-down remembering the context stage. Each local definition in scope is recorded with its level and each reference to a definition is checked against the current stage.

```scala
// stage 0
'{ // stage 1
  val x = ... // stage 1 with (x -> 1)
 ~{ // stage 0 (x -> 1)
    val y = ... // stage 0 with (x -> 1, y -> 0)
    x // error: defined at level 1 but used in stage 0
  }
  // stage 1 (x -> 1)
  x // x is ok
}
```

**Pickling quotes**  If the outermost scope is a quote, we need to pickle [19] the contents of the quote to have it available at run-time. We implement this by pickling the tree as TASTY [27] binary, which is stored in a compacted string.

TASTY is the compact typed abstract syntax tree serialization format of Dotty. It usually pickles the full code after type checking and keeps it along the generated classfiles. This is used for separate and incremental compilation, documentation generation, language server for IDE, code decompilation and now quotes.

It is not possible to pickle the tree inside the quote directly as the contents of embedded splices are at stage 0 and may contain free variables. To avoid this we introduce *holes* in the trees that will be pickled in their place, each splice in the quote will have a hole that replaces it. Holes are encoded as a list of functions `fillHole`, each function contains the code that will be used to fill the $i_{th}$ hole. Each hole will have an argument list, listing variables defined in the quote and referenced inside the splice. These arguments (e.g., `'(x)` in the code below) will be quoted to retain phase consistency.

```
'{
  val x: Int = ...
 ~{ ... '{ ... x ... } ... }
}
// Is transformed to
'{
  val x: Int = ...
  ~fillHole(0).apply('(x))
}
```

The contents of the splices will be used to construct each element of the hole. Each element is a lambda that receives the quoted arguments and will return the evaluation of the code in the splice. The lambda will receive as parameters the quotes that were passed as arguments in the previous transformation. The quoted parameters need to be spliced in the body of the splice to keep phase consistency[6].

```
~{ ... '{ ... x ... } ... }
// Is transformed to
(x: Expr[Int]) => { ... '{ ... ~x ... } ... }
```

Once we applied the first transformation to the quoted code we can pickle it and keep the contents of the splices in a separate structure. We use `stagedQuote` to put together the parts of the quotes in some data structure. As an example consider the following quote:

```
val arr: Expr[Array[Int]] = ...
'{
  var sum = 0
  ~staged(arr, x => '(sum += ~x))
  sum
}
```

---

[6]Note that x must be inside some quote to be phase consistent in the first place.

Which will be transformed to the following code:

```
val arr: Expr[Array[Int]] = ...
stagedQuote(
  tasty = """[[ // PICKLED TASTY BINARY
    var sum = 0
    ~fillHole(0).apply('(sum))
    sum
    ]]""",
  fillHole = List(
    (sum: Expr[Int]) => staged(arr, x => '((~sum) += ~x))
  )
)
```

After the presented transformation, the contents of `fillHole` will use the same transformation recursively to pickle the inner quote: `'((~sum) += ~x)`.

**Compiling Macros**  To avoid the need for a complex interpreter when evaluating the code in top-level splices we use part of the pickling mechanism. For example in `sum` we do not wish to have to interpret `staged(...)` when inlining.

```
object Macros {
  inline def sum(arr: Array[Int]): Int = {
    var sum = 0
    ~staged('(arr), x => '(sum += ~x))
    sum
  }
}
```

The body of the macro is treated as quoted code and the tree is split into its parts.

Parameters of the macro are treated as defined outside of the quote and need to be added in the hole parameters. Parameters that were marked as `inline` are passed directly as values and lifted if used in a quote. We will get a version of the body that will have a hole in place of the original contents of the splices. The new version of the body of `sum` simply replaces the old one.

```
inline def sum(arr: Array[Int]): Int = {
  var sum = 0
  ~sum_hole(0).apply('(arr), '(sum))
  sum
}
```

Like with the pickled quotes we also get the contents of the splices in the form of a list of lambdas `sum_hole`. Which will be placed in a static method and compiled along with `sum`.

```
def sum_hole = List(
  (arr: Expr[Array[T]], sum: Expr[Int]) =>
    staged(arr, x => '((~sum) += ~x))
)
```

After this transformation, all top-level splices contain a tree with a call to a parameterless static method, a statically

known index and a list of quoted (or `inline`) arguments. The interpreter that handles the macro splice expansion only needs to be able to handle these trees.

**Unpickling quotes**   To unpickle quotes we unpickle most of the tree as usual in TASTY. But, if we encounter a hole it is filled using the corresponding `fillHole` for it. The index of the hole determines which `fillHole` must be used and the arguments of the hole are passed to the `fillHole(idx)`.

For inlined macros it is slightly different, as the tree will already be inlined with holes. Then we just need to load via reflection the corresponding `fillHole` and expand it normally.

**Running a quote**   When executing `Expr.run`, an instance of the compiler consumes the `Expr`. This is an instance of the normal Dotty compiler that is provided by a `quoted.Toolbox`. It provides caching and thread safety over the accesses to the compiler. Multiple instances can be created if needed. In the `Toolbox`, the compiler will load the tree from its TASTY and place the contents of the tree in a method of a new class. This class is compiled to bytecode and executed.

## 8   Case Studies

We present two case studies. Firstly, we give a sample solution to the Hidden Markov Model challenge as specified in the Shonan Challenge for Generative Programming [1]. This case study shows that our system captures the basic needs for abstraction and reusability of staged code. Secondly, we port Strymonas [22], a staged library for streams, showing that a more complex library can optimize pipelines either in a runtime or compile-time fashion, unaltered.

### 8.1   Case Study 1: Linear Algebra DSL

This case study presents a way to define a generic and composable *Linear Algebra DSL* that can be used on staged and non-staged code alike. We implemented the framework presented in [21] that provided optimizable matrix multiplication as part of the Shonan HMM challenge.

To simplify the presentation, in this section we will only show how to perform a vector dot product. We will present an implementation for vector dot product that can stage or unroll the operations, use statically known vectors or dynamically accessible ones, and work on any kind of elements. The same abstraction would be extended and composed for a matrix multiplication.

#### 8.1.1   Ring Arithmetic

First we have to see how it is possible to abstract over operations that are staged and ones that are not staged. For this we will simply define an interpreter interface for our operations, in this case it will be the mathematical ring including subtraction. Apart from the operation, the interface will also provide the `zero` and `one` values for those operations.

```
trait Ring[T] {
  val zero: T
  val one: T
  val add: (x: T, y: T) => T
  val sub: (x: T, y: T) => T
  val mul: (x: T, y: T) => T
}

class RingInt extends Ring[Int] {
  val zero = 0
  val one = 1
  val add = (x, y) => x + y
  val sub = (x, y) => x - y
  val mul = (x, y) => x * y
}
```

As shown for a `Ring[Int]` all operations are just interpreted. If we implement a `Ring[Expr[Int]]` all operations will be staged. In fact `RingIntExpr` is a small staged interpreter, it will be a compiler for operations on `Int`.

```
class RingIntExpr extends Ring[Expr[Int]] {
  val zero = '(0)
  val one = '(1)
  val add = (x, y) => '(~x + ~y)
  val sub = (x, y) => '(~x - ~y)
  val mul = (x, y) => '(~x * ~y)
}
```

To implement rings on structured types such as a complex number we implement it generically based on a ring on its elements. This ring is used to perform all operations on the inner elements.

```
case class Complex[T](re: T, im: T)
class RingComplex[U](u: Ring[U]) extends Ring[Complex[U]] {
  val zero = Complex(u.zero, u.zero)
  val one  = Complex(u.one, u.zero)
  val add  = (x, y) => Complex(u.add(x.re, y.re),
                              u.add(x.im, y.im))
  val sub  = ...
  val mul  = ...
}
```

This implementation of `RingComplex` is polymorphic on the type of elements it contains. Hence it can be instantiated as `Complex[Int]` or `Complex[Expr[Int]]` by instantiating the rings with the complex ring with `RingInt` and `RingIntExpr` respectively. Using this composability, we can implement all possible combination of rings by only implementing the ring for each type twice (unstaged and staged).

#### 8.1.2   Vector Operations

Across this paper we have seen several implementations of a staged `foreach` operation that had a `while` loop or was unrolled. We will use a vector abstraction that abstracts both the element type and the index type. The `reduce` operation will be provided by the `VecOps[Idx, T]` interface.

```
case class Vec[Idx, T](size: Idx, get: Idx => T)

trait VecOps[Idx, T] {
  val reduce: ((T, T) => T, T, Vec[Idx, T]) => T
}
```

Now we can implement a version of the operation that executes the operations (`VecOps[Int, T]`) and one that stages the operations (`VecOps[Expr[Int], Expr[T]]`).

```
class StaticVecOps[T] extends VecOps[Int, T] {
  val reduce: ((T, T) => T, T, Vec[Int, T]) => T =
    (plus, zero, vec) => {
      var sum = zero
      for (i <- 0 until vec.size)
        sum = plus(sum, vec.get(i))
      sum
    }
}
class StagedVecOps[T: Type] extends VecOps[Expr[Int],
    Expr[T]] {
  val reduce: ((Expr[T], Expr[T]) => Expr[T], Expr[T],
    Vec[Expr[Int], Expr[T]]) => Expr[T] =
    (plus, zero, vec) => '{
      var sum = ~zero
      for (i <- 0 until ~vec.size)
        sum = ~plus('(sum), vec.get('(i)))
      sum
    }
}
```

### 8.1.3   Linear Algebra DSL

Now we can implement our linear algebra DSL that will provide the dot product on vectors. We both abstract on the vector operation and the element ring operations. It will first create a vector multiplying the elements using the ring and then it will be reduced using the operations of the ring.

```
class Blas1[Idx, T](r: Ring[T], ops: VecOps[Idx, T]) {
  def dot(v1: Vec[Idx, T], v2: Vec[Idx, T]): T = {
    val v3 = Vec(size, i => r.mul(v1.get(i), v2.get(i)))
    ops.reduce(r.add, r.zero, v3)
  }
}
```

This is all we need, now we can instantiate `Blas1` with different instances of `Ring` and `VecOps`.

```
// Computes the dot product on vectors of Int
val dotInt = new Blas1(new RingInt, new StaticVecOps).dot
// will compute the value 4
dotInt(
  Vec(5, i => i), // [0,1,2,3,4]
  Vec(5, i => i % 2)) // [0,1,0,1,0]

val RingComplexInt = new RingComplex(new RingInt)
// Computes the dot product on vectors of Complex[Int]
val dotComplexInt =
  new Blas1(RingComplexInt , new StaticVecOps).dot
// will compute the value Complex(-5, 13)
```

```
dotComplexInt(
  Vec(5, i => Complex(i,i)), // [0,1+i,2+2i,3+3i,4+4i]
  Vec(5, i => Complex(i % 2, i % 3))) // [0,1+i,2i,1,i]

// Staged loop of dot product on vectors of Expr[Int]
val dotStagedIntExpr =
  new Blas1(new RingIntExpr, new ExprVecOps).dot
// will generate '{ var sum = 0; for (i <- 0 until
//    arr1.size) sum = sum + arr1(i) * arr2(i); sum }
dotStagedIntExpr(
  Vec(5, i => '((~arr1)(~i.toExpr))),
  Vec(5, i => '((~arr2)(~i.toExpr)))

// Unrolles the computation of dot product on vectors of
//    Expr[Int]
val dotStaticIntExpr =
  new Blas1(new RingIntExpr, new StaticVecOps).dot
// will generate the code '{ 0*0 + 1*1 + 2*0 +3*1 +4*0 }
dotStaticIntExpr(
  Vec(5, i => i.toExpr), // ['(0),'(1),'(2),'(3),'(4)]
  Vec(5, i => (i % 2).toExpr)) // ['(0),'(1),'(0),'(1),'(0)]
```

### 8.1.4   Modular Optimizations

We will now see how to unroll the dot product of a stages vector with a known vector. The simple solution is to lift the second vector elements use `dotExprIntExpr` like we did in the previous example. A shortcoming of this approach is that it will not be able to partially evaluate lifted values.

Instead, we will abstract the fact that we have a value of type `T` or `Expr[T]`. To achieve this we will define partially known values `PV[T]`.

```
sealed trait PV[T] {
  def expr(implicit l: Liftable[T]): Expr[T]
}
case class Sta[T](x: T) extends PV[T] { ... }
case class Dyn[T](x: Expr[T]) extends PV[T] { ... }
```

With this abstraction it is possible to define a `Ring[PV[U]]` that operates both on `Ring[U]` and `Ring[Expr[U]]`. In it is possible to perform constant folding optimizations statically known elements. In general, this ring can be composed with the rings for any given type.

```
class RingPV[U: Liftable](u: Ring[U], eu: Ring[Expr[U]])
    extends Ring[PV[U]] {
  val zero: PV[U] = Sta(u.zero)
  val one: PV[U] = Sta(u.one)
  val add = (x: PV[U], y: PV[U]) => (x, y) match {
    case (Sta(u.zero), x) => x // Constant fold zero
    case (x, Sta(u.zero)) => x // Constant fold zero
    case (Sta(x), Sta(y)) =>
      Sta(u.add(x, y)) // Evaluate at staging time
    case (x, y) =>
      Dyn(eu.add(x.expr, y.expr)) // Stage operation
  }
  val sub = ...
  val mul = ...
}
```

Using this ring we can optimize away all zeros from the dot product on vectors of `PV[Int]` and `Complex[PV[Int]]`. We do not use `PV[Complex[Int]]` as it would stage the complex before all optimizations can take place.

```
val RingPVInt =
  new RingPV[Int](new RingInt, new RingIntExpr)
// Staged loop of dot product on vectors of Int or Expr[Int]
val dotIntOptExpr =
  new Blas1(RingPVInt, new StaticVecOps).dot
// dotIntOptExpr will generate the code for
// '{ arr(1) + arr(3) }
dotIntOptExpr(
  Vec(5, i => Dyn('((~arr)(~i.toExpr)))),
  Vec[Int, PV[Int]](5, i => Sta((i % 2))) // [0,1,0,1,0]
).expr
```

### 8.2 Case Study 2: Stream Fusion, to Completeness

List processing has been a key abstraction in functional programming languages [3]; an abstraction that is tightly coupled with the notion of lazy evaluation [14]. A list processing library is typically equipped with a set of operators to create lists, transform and consume them into scalar or other kinds of data structures. `Data.List` in Haskell, a lazy programming language, relies on writing the list processing functions using appropriate data structures, providing a set of rewrite rules to identify patterns in the code and then relying on the optimizing phase of GHC [29] to apply them [13]. The expected result is to compile a pipeline into a low-level, tight-loop, with zero abstraction costs such as no intermediate data structures and heap-allocated objects. For Scala and similar eager programming languages, stream libraries are simulating laziness on their own, either by relying on `unfolds` (pull-based streams) or again `folds` (push-based streams) [2].

Strymonas, based on `unfolds` [22] implements a staged stream library that fuses pipelines generating tight-loops. Strymonas comes in two flavors, one in Scala/LMS and one in BER MetaOCaml. In this section we discuss a third port of this library in Scala demonstrating that now Scala is equipped with the necessary abstractions to support Strymonas. There are two kinds of combinators in this design: a) regular and b) `*Raw` versions. The former have the familiar signatures we know and the latter are used to pattern match on the stream shape (`Producer`) of a downstream combinator manipulating its shape accordingly. The latter can be seen as *code combinators* that operate on a "suitable intermediate representation" [7]. Additionally, they use CPS internally to enable let-insertion in stateful combinators. Since Strymonas is not relying on control effects our system can fully support it. Stream pipelines in Strymonas can be either staged or used as a macro, as shown in Section 1.

***A note on the performance of the generated code.*** The benchmarks in Figure 2 demonstrate that the use of macros elides the costs of runtime code-generation as expected. The
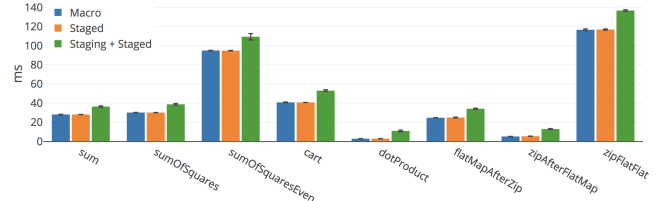


**Figure 2.** Strymonas microbenchmarks in msec / iteration. "Macro" and "Staged" is the execution time of the generated code. "Staging + Staged" is the time taken to stage the code at runtime and execute it. The first execution of "Staging" takes an additional 2.5 seconds to load the compiler.

macro and staged generated code were benchmarked by warming-up the code (to force JIT compilation). We also show the additional cost of staging and then running the resulting function. The overhead is the combination of compiling the code to bytecode, loading and JIT-compiling the code. Additionally on a cold JVM the first execution of `run` takes around 2.5 seconds to load the compiler. However, we omit it from the figure since it is amortized during warmup. Comparatively, macros do not incur such a performance penalty because the compiler is already loaded.

For our benchmarks we used the Java Microbenchmark Harness (JMH) [34] tool: a benchmarking tool for JVM-based languages that is part of the OpenJDK. The system we use runs an x64 OSX High Sierra 10.13.6 operating system on bare metal. It is equipped with a 4 GHz Intel Core i7 CPU (i7-6700K) having 4 physical and 8 logical cores. The total memory of the system is 16 GB of type 1867 MHz DDR3.

## 9 Related Work

Our system is heavily inspired by the long line of work by MetaML[38], MetaOCaml [6] and BER MetaOCaml[20]. We rely on the latter for most of our design decisions. We offer the capability of pretty printing generated code, but our system, contrary to BER MetaOCaml, compiles to native code first. In our case, native code (JVM bytecode) was simpler to implement since we rely on TASTY, the serialization format for typed syntax trees of Scala programs [27]. BER MetaOCaml offers the capability to programmers to process code values in their own way. We plan to make our system extensible in the same way but by relying on TASTY.

Modular Macros [44] offered a compile-time variant of BER MetaOCaml by introducing a new keyword to enable macro expansion. In their work they demonstrate that an existing staged library needs intrusive changes to sprinkle the code with the aforementioned keywords. In our case we just need one definition with a top-level splice and we reuse a staged library unchanged. Modular Macros is a separate project to BER MetaOCaml so the two techniques were not composed.

MacroML [12] pioneered compile-time version of MetaML showing at a theoretical level that the semantics of MetaML subsume macros; MacroML essentially translates macro programs to MetaML programs. Our work presents a confluence of macros and multi-stage programming in the same language (considering the imperative features of Scala, something left out from MacroML's development). Even though this merge was not demonstrated in the original work by Ganz et al. we believe that their work provides useful insights for the future foundations of our system.

Template Haskell [33] is a very expressive metaprogramming system that offers support for code generation not only of expressions but also definitions, instances and more. Template Haskell used the type class `lift` to perform CSP, we used the same technique for our `Liftable` construct. Code generation in Template Haskell is essentially untyped; the generated code is not guaranteed to be well-typed. Typed Template Haskell, on the other hand, also inspired by MetaML and MetaOCaml offers a more restrictive view in order to pursue a disciplined system for code generation. Typed Template Haskell is still considered to be unsound under side effects [18], providing the same static guarantees as MetaOCaml. To avoid these shortcomings we permit no side effects in splice operations as well. We regard side effects as an important aspect of programming code generators. The decision to disallow effects in splices was taken because it was a simple approach to avoid the unsoundness hole of scope-extrusion. At the moment, code generators and delimited control (e.g., like restricting the code generator's effects to the scope of generated binders [17]) was out of the scope of this paper but remains a goal of our future work.

F# supports code quotations that offer a quoting mechanism that is not opaque to the user effectively supporting analysis of F# expression trees at runtime. Programmers can quote expressions and they are offered the choice of getting back either a typed or an untyped expression tree. F# does not support multi-stage programming and currently lacks a code quotation compiler natively[7]. Furthermore, lifting is not supported. Finally, F# does not support splicing of types into quotations.

Scala offers experimental macros (called blackbox in Scala parlance) [4, 5]. The provided macros are quite different from our approach. Those macros expose directly an abstraction of the compiler's ASTs and the current compilation context. Scala Macros require specialized knowledge of the compiler internals. Quasiquotes, additionally, are implemented on top of macros using string interpolators [32] which simplify code generation. However, the user is still exposed to the same complex machinery, inherited from them. Scala also offers macros that can modify existing types in the system (whitebox and annotation macros). They have proven dangerously powerful; they can arbitrarily affect typing in unconventional ways giving rise to problems that can deteriorate IDE support, compiler evolution and code understanding.

Lightweight Modular Staging (LMS) offers support for Multi-stage Programming in Scala[31]. LMS departs from the use of explicit staging annotations by adopting a *type-based embedding*. On the contrary, a design choice of our system is to offer explicit annotations along the lines of MetaML. We believe that programming with quotes and splices reflects the textual nature of this kind of metaprogramming and gives the necessary visual feedback to the user, who needs to reason about code-fragments. LMS is a powerful system that preserves the execution order of staged computations and also offers an extensible Graph-based IR. On the flipside, two shortcomings of LMS, namely high compile times and the fact that it is based on a fork of the compiler were recently discussed as points of improvement [30].

Squid [28] 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. A shortcoming in Squid, implemented as a macro library, is that free variables must be marked explicitly. Furthermore, contexts are represented as contravariant structural types[8] which complicates the error messages.

## 10  Conclusion & Future Work

Metaprogramming has a reputation for being difficult and confusing. However with explicit `Expr`/`Type` types, quotes and splices it can become downright pleasant. A simple strategy first defines the underlying quoted or unquoted values using `Expr` and `Type` and then inserts quotes and splices to make the types line up. Phase consistency is at the same time a great guideline where to insert a quote or a splice and a vital sanity check that the result makes sense.

As future work we plan to study the formal properties of our system. Furthermore, we plan to complement it with a version of inline that not only provides $\beta$-reductions at the expression-level but also at the type-level.

---

[7]Splice types into Quotations–https://web.archive.org/web/20180712194211/ https://github.com/fsharp/fslang-suggestions/issues/584

---

[8]`type Code[+Typ, -Ctx]`

# References

[1] Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2013. Shonan Challenge for Generative Programming: Short Position Paper. In *Proc. of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM '13)*. ACM, New York, NY, USA, 147–154. https://doi.org/10.1145/2426890.2426917

[2] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2014. Clash of the Lambdas. In *Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '14)*. arXiv:cs.PL/1406.6631

[3] W. H. Burge. 1975. Stream Processing Functions. *IBM J. Res. Dev.* 19, 1 (Jan. 1975), 12–25. https://doi.org/10.1147/rd.191.0012

[4] Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proc. of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2489837.2489840

[5] Eugene Burmako. 2017. *Unification of Compile-Time and Runtime Metaprogramming in Scala.* Ph.D. Dissertation. Lausanne.

[6] 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 (GPCE '03)*. Springer-Verlag, Berlin, Heidelberg, 57–76. http://dl.acm.org/citation.cfm?id=954186.954190

[7] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. 2006. In Search of a Program Generator to Implement Generic Transformations for High-performance Computing. *Sci. Comput. Program.* 62, 1 (Sept. 2006), 25–46. https://doi.org/10.1016/j.scico.2005.10.013

[8] 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

[9] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative programming. In *European Conference on Object-Oriented Programming*. Springer, 15–29.

[10] 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

[11] Matthew Flatt. 2002. Composable and Compilable Macros:: You Want It when?. In *Proc. of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/581478.581486

[12] Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros As Multi-Stage Computations: Type-safe, Generative, Binding Macros in MacroML. In *Proc. of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 74–85. https://doi.org/10.1145/507635.507646

[13] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, 223–232. https://doi.org/10.1145/165180.165214

[14] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[15] Simon Peyton Jones. 2016. Template Haskell, 14 years on. .https://www.cl.cam.ac.uk/events/metaprog2016/Template-Haskell-Aug16.pptx.

[16] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*. ACM, New York, NY, USA, 86–96. https://doi.org/10.1145/512644.512652

[17] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the Stage: Staging with Delimited Control. In *Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '09)*. ACM, 111–120. https://doi.org/10.1145/1480945.1480962

[18] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for Impure Yet Hygienic Code Generation. *Sci. Comput. Program.* 112, P2 (Nov. 2015), 120–144. https://doi.org/10.1016/j.scico.2015.08.007

[19] Andrew J. Kennedy. 2004. FUNCTIONAL PEARL Pickler Combinators. *J. Funct. Program.* 14, 6 (Nov. 2004), 727–739. https://doi.org/10.1017/S0956796804005209

[20] 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.

[21] 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

[22] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 285–299. https://doi.org/10.1145/3009837.3009880

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

[24] Fengyun Liu and Eugene Burmako. 2017. Two approaches to portable macros. (2017).

[25] Stefan Monnier and Zhong Shao. 2003. Inlining As Staged Computation. *J. Funct. Program.* 13, 3 (May 2003), 647–676. https://doi.org/10.1017/S0956796802004616

[26] Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level Functional Languages.* Cambridge University Press, New York, NY, USA.

[27] Martin Odersky, Eugene Burmako, and Dmytro Petrashko. 2016. A TASTY Alternative. (2016).

[28] 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

[29] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

[30] Tiark Rompf. 2016. Reflections on LMS: Exploring Front-end Alternatives. In *Proc. of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2998392.2998399

[31] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

[32] Denys Shabalin. 2014. *Hygiene for scala.* Technical Report.

[33] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. In *Proc. of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[34] Aleksey Shipilev, Sergey Kuksenko, Anders Astrand, Staffan Friberg, and Henrik Loef. 2007. OpenJDK Code Tools: JMH. http://openjdk.java.net/projects/code-tools/jmh/.

[35] 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.

[36] Walid Taha. 1999. *Multi–Stage Programming: Its Theory and Applications.* Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

[37] Walid Taha. 2004. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, Christian Lengauer,

Don Batory, Charles Consel, and Martin Odersky (Eds.). Number 3016. Springer Berlin Heidelberg, 30–50.

[38] Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '97)*. ACM, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

[39] The Dotty Team. 2018. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20180630221002/http://dotty.epfl.ch/.

[40] The Dotty Team. 2018. Dotty Inline. https://web.archive.org/web/20171230163822/http://dotty.epfl.ch:80/docs/reference/inline.html. [Accessed: 2018-07-09].

[41] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1993498.1993514

[42] Laurence Tratt. 2008. Domain Specific Language Implementation via Compile-time Meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 31 (Oct. 2008), 40 pages. https://doi.org/10.1145/1391956.1391958

[43] 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.

[44] Jeremy Yallop and Leo White. 2015. Modular macros.