Fluid Quotes: Metaprogramming across Abstraction Boundaries with Dependent Types

Shadaj Laddad
University of California, Berkeley
USA
shadaj@berkeley.edu

Koushik Sen
University of California, Berkeley
USA
ksen@cs.berkeley.edu

Abstract

Object-oriented programming, functional programming, and metaprogramming each offer a unique axis of abstraction that enables modular code. Macros, a common technique for metaprogramming, capture ASTs as quotes to let users manipulate them in the host language. However, macros are often at odds with other programming techniques since they can only process code written at the call-site and cannot analyze code behind abstraction boundaries such as variables and methods. Furthermore, the quotes generated for macro expansion exist only at compile-time and cannot be passed around in user code. Multi-stage programming treats quotes as runtime values to address this problem, but introduces the cost of running the compiler when splicing quotes. This forces developers to choose between low runtime overhead and modularity. What if we could have the best of both worlds? We introduce fluid quotes, a new technique that uses dependent types to let users pass quotes through abstraction boundaries in runtime code while splicing them ahead-of-time. This technique enables new metaprogramming capabilities by eliminating the traditional requirement of co-locating parameter expressions with call-sites. Fluid quotes capture not only source code but also associated runtime context to ensure correctness. In addition, they can be composed into larger expressions without any macro code. We demonstrate the capabilities of fluid quotes through two specific applications: optimizing data processing pipelines and making language integrated queries more flexible.

CCS Concepts: • **Software and its engineering → Language features**.

Keywords: metaprogramming, dependent types, inlining, functional programming

GPCE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8174-1/20/11.
https://doi.org/10.1145/3425898.3426953

ACM Reference Format:
Shadaj Laddad and Koushik Sen. 2020. Fluid Quotes: Metaprogramming across Abstraction Boundaries with Dependent Types. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426953

1 Introduction

Macros are a powerful tool that enable developers to use metaprogramming without interacting with compiler internals. With macros, developers can define custom expansions that transform the ASTs used at the call-site into a new expression. Quotes in languages such as LISP and Scala make it easy to define macros by allowing users to capture code and splice it into larger expressions. Traditionally, quotes exist only at compile-time and cannot be passed around as values in user code without bundling the compiler for staged code execution [Taha and Sheard 1997]. This limits macro expansions since they can only analyze the code immediately available at the call-site.

Metaprogramming allows developers to perform optimizations similar to traditional inlining by splicing the quoted arguments into a larger expression that replaces the macro call. However, functional programming throws in a twist to this traditional formula. With higher-order functions, we need to inline not only the function body but also the calls to functions arguments. While modern languages such as Kotlin can inline such arguments [JetBrains 2018], they limit inlining to function arguments defined directly at the call-site.

The same limitation also appears when using metaprogramming to translate user code into other domains, such as synthesizing SQL queries from user code. Since macros can only analyze the ASTs directly available at call-site, they must treat function references as opaque values and cannot analyze their bodies. Furthermore, macros used to implement a chained API can only see other chained calls if they are lexically adjacent, which limits the ability to perform transformations across calls split by abstraction boundaries such as separately defined functions.

Multi-stage programming systems such as Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] allow mixing quotes into user code. This makes it easy to write

metaprogramming transformations on complex code graphs. However, to make this possible, LMS performs splicing at runtime by bundling the host language compiler. This introduces significant overhead and cannot be implemented in ahead-of-time compiled languages. Other systems such as Braid [Sampson et al. 2017] can perform expansions ahead-of-time but require users to separate macros from runtime code just like traditional systems [Flatt 2002].

We present **fluid quotes**, a new technique that overcomes these traditional metaprogramming limitations by storing quoted expressions in type members. Users can carry fluid quotes across abstraction boundaries and compose them through type-level data structures. We can splice a fluid quote at any place where its types can be resolved. With fluid quotes, developers no longer have to forgo the elegance of abstractions to gain the power of metaprogramming.

Fluid quotes can be implemented in any language that supports dependent types and the ability to extend the type-checker with custom expansions either through macros or custom compiler APIs. In this paper, we implement fluid quotes using Scala programming language and make it available as an open-source project. We evaluate fluid quotes by applying it to performance optimization, through a stream fusion implementation, and code transformation, through an extension of language-integrated queries.

1.1 Motivating Example

Consider a toy differentiable programming example as shown in Figure 1 where we want to calculate the derivative of a function. Let us assume that we have access to a `differentiate` macro that synthesizes the derivative of a given function. We decide to modularize some of the preprocessing logic into a separate function.

```
1  def differentiate(
2    fn: Double => Double
3  ): Double => Double = macro ...
4
5  def getPreprocessor = {
6    println("Init!")
7    (x: Double) => x * x
8  }
9
10 val preprocess = getPreprocessor // Init!
11 // does not compile
12 val derivative = differentiate(
13   (in: Double) => 2 * preprocess(in))
14 println(derivative(2)) // 8
```

Figure 1. Using macros for differentiable programming

Unfortunately, the modularized code above will not compile. The `differentiate` macro receives only the AST of line 15, which contains an opaque reference to the `preprocess` function. Without static access to the definition of this function, the macro cannot synthesize its derivative at compile-time. LMS can help by allowing us to synthesize derivatives at runtime, but introduces significant overhead since the synthesized code would have to be compiled at runtime. Another solution is to manually copy the definition of `preprocess` to replace the call on line 15, but this introduces duplication that breaks the modularity of our code. Yet another solution could be to return not only the original function in `getPreprocessor` but also its derivative. This would allow us to differentiate the function on line 15 by using the additional return value as the derivative of `preprocess(in)`, but would result in poor performance due to function call overhead. We would like the compiler to track the body of `preprocess` without changing the program structure.

One compile-time option is to use inlining, but we cannot safely inline the `preprocess` variable since doing so would change the program behavior if the variable initialization has side-effects. For example, naively inlining `preprocess` would result in the logging line `println("Init!")` being executed every time the derivative is calculated. Whether it be classes, methods, or simply variables, the moment a function value is placed behind such an abstraction boundary we lose access to its definition. We could stretch our implementation and use internal compiler APIs to look up definition ASTs, but this quickly becomes extremely brittle.

What we need is a sound mechanism to propagate ASTs across all these boundaries at compile-time. With fluid quotes, we use types to carry expressions across abstractions. Applying fluid quotes to our example, we change the `getPreprocessor` function to return a fluid quote, whose type captures the AST of the function.

```
def getPreprocessor = {
  println("Initializing!")
  quote((in: Double) => in * in)
}
```

Then, we splice the function body with `spliceCall`. The `spliceCall` macro provided by the fluid quotes API inserts the body of the captured function at the call-site.

```
val preprocess = getPreprocessor
val derivative = differentiate((in: Double) =>
  2 * preprocess.spliceCall(in))
```

After `spliceCall` inlines the captured function, we have a regular block that can be analyzed by `differentiate`:

```
val derivative = differentiate((in: Double) =>
  2 * { val x = in; x * x }
)
println(derivative(2)) // 8
```

With fluid quotes, we are able to apply compile-time transformations without changing the structure of our code or introducing unnecessary duplication.

1.2 Contributions

In this paper, we make the following contributions:

- We introduce a new type-level encoding to pass quoted code through types and splice them at compile time. We describe how both expressions and function calls can be spliced through a natural API. (Section 2)
- We discuss how fluid quotes handle local definitions by generating closures through additional types and code transformations. We explore how this transformation preserves runtime semantics while minimizing the performance cost. (Section 3)
- We introduce a composition system that uses dependent types to combine fluid quotes at the type level without requiring additional macro code. We demonstrate how this allows writing short templates that can expand into performant code. (Section 4)
- We discuss how limitations of the type system affect fluid quotes. We explore extensions to fluid quotes as well as applications of closures and the composition API to overcome some of these restrictions. (Section 5)
- We evaluate applications of fluid quotes that enable performance gains and implement new metaprogramming capabilities. We also demonstrate how these applications extend beyond state-of-the-art capabilities in real-world languages. (Section 6)

2 Fluid Quotes

Fluid quotes, represented by the `FluidQuote` type, capture the AST of an expression to be spliced elsewhere in the code base. In this section, we consider simple examples that demonstrate the core architecture of fluid quotes. Ensuring correctness and ease-of-use involves complex challenges such as involving closures and type-level data structures, which we discuss in later sections.

Every `FluidQuote` tracks the AST of the captured expression. We need a mechanism that can propagate this data across abstraction boundaries at compile-time. Types, which already have propogation logic built into the compiler, are a perfect fit for this. We use a type member, `Expr`, to store the AST. Storing this data as an instance variable would not serve our purpose since they would only be resolvable at runtime. This is the core limitation of runtime staging systems, which we overcome with fluid quotes.

Any `FluidQuote` object can be spliced by the user wherever they want, with a few typing-related restrictions that we discuss in Section 5. The `splice` macro, available in `FluidQuote`, resolves the `Expr` type, parses the AST, and emits it. All together, the core definition of `FluidQuote` is quite concise.

```
trait FluidQuote[T] {
  type Expr
  def splice: T = macro ...
}
implicit def quote[T](expr: T): FluidQuote[T]
  = macro ...
```

`FluidQuote` uses two macros (`quote` and `splice`) for quoting and splicing. Our system requires that the host language execute macros during typechecking so that we can generate and resolve types. We use Scala macros [Burmako 2013, 2019] in our implementation to satisfy these requirements, but fluid quotes can be implemented in other languages by extending the typechecker.

To see these transformations in action, let's consider a simple guiding example:

```
val expr = quote(1 + 2)
expr.splice
```

In the following subsections, we will break this example down to see the internals of fluid quotes.

2.1 Quoting Expressions

The `quote` macro captures an expression of type `T` in an instance of `FluidQuote[T]`. In our guiding example, we store a `FluidQuote[Int]` that captures the desugared AST `1.+(2)`. The `quote` macro receives the AST of the passed expression and generates an instance of `FluidQuote` with the `Expr` type set to a string literals type representing the AST. As with traditional quoting, wrapping a block in `quote` changes its operational semantics since `quote` captures the block as an AST and does not evaluate it. After the `quote` expansion, the first line of our example looks as follows:

```
val expr = new FluidQuote[Int] {
  type Expr = "1.+(2)"
}
```

Note that `quote` is defined as an implicit conversion to automatically wrap any expression of type `T` into a `FluidQuote[T]` when the expected expression type is a fluid quote. This allows developers to replace function and by-name parameters [Odersky et al. 2008] with fluid quotes without any change to the end-user experience.

2.2 Splicing Expressions and Function Calls

The `splice` macro resolves the `Expr` dependent type of the `FluidQuote` instance and emits the captured AST. In our example, the `splice` macro expands to the original expression:

```
expr.splice // expands at compile time to
{ 1.+(2) }
```

Consider a more complex example where we capture a function with a fluid quote:

```
val doubleIt = quote((in: Int) => in * 2)
doubleIt.splice(5) // expands to
((in: Int) => in * 2)(5)
```

If we use `splice`, we introduce an unnecessary temporary function value. To avoid this, we offer special support for directly splicing function calls through `FluidFunctionN`

types that represent function values of different arities. In addition to inheriting `FluidQuote` features, fluid functions can directly splice a function call `spliceCall` macro. For example, we can use this in the earlier example:

```
val doubleIt = quote((in: Int) => in * 2)
doubleIt.spliceCall(5)
// expands at compile time to
val doubleIt = new FluidFunction1[Int, Int] {
  type Expr = "in.*(2)"
  type Params = "in"
)
{
  val in = 5 // expose arguments
  in.*(2) // splice body
}
```

To implement function call inlining, we introduce an additional type member `Params` that tracks the original parameter names. The `spliceCall` macro first exposes each argument as variables with the same name as the corresponding parameter. This takes advantage of definition shadowing, since even though there may be existing variables in scope with those names, the expanded macro block will only see the parameter variables. Now, all that is left is to splice the body of the function as if it were a regular `FluidQuote`.

With these core features, developers can use fluid quotes to pass around expressions and splice them at compile-time. However, there are still additional challenges in maintaining correctness and simplifying the user experience. We discuss these in the following sections.

3 Capturing Environments with Closures

Fluid quotes often refer to local context through variables, classes, and methods that are not directly accessible at the splice sites. We handle such situations by generating closures that the spliced code uses to access the local symbols. Our closures are a combination of traditional closures, which track the relevant symbols, and syntactic closures [Bawden and Rees 1988], which ensure macro hygiene by preventing surrounding code from shadowing quoted references.

Consider the following example where the quoted code uses local references:

```
1  def add1(num: Int) = quote(num + 1)
2  val quotedAdd1To123 = add1(123)
3
4  {
5    val num = 456 // attempts to shadow num
6    quotedAdd1To123.splice // should be 124
7  }
```

Figure 2. An example where closures are needed.

We must offer the capabilities of traditional closures to track the local 123 value referenced by the fluid quote, as well as syntactic closures to ensure that the `num` definition on line 5 is not leaked into the spliced code. We support this by storing a closure object in the fluid quote which contains forwarders to the original local symbols. With our closure transformation system, the code above is transformed into:

```
1  def add1(num: Int) = new FluidQuote[Int] {
2    type Expr = "numFwd.+(1)"
3
4    type ClosureRefs = "numFwd"
5    class Closure {
6      val numFwd = num
7    }
8
9    val closure = new Closure
10 }
11
12 val quotedAdd1To123 = add1(123)
13 {
14   val num = 456
15   val closure = quotedAdd1To123.closure
16   closure.numFwd.+(1)
17 }
```

Figure 3. The final code after closure transformations have taken place.

Here, the local reference `num` is captured with `numFwd` in the `Closure` class and retrieved as `closure.numFwd` at the splice site. In the following subsections, we will examine how we generate closures and use them in the quoted code.

3.1 Closure Generation Process

Our closure generation process starts by identifying any reference that is not publicly accessible or defined within the quoted expression. These must be captured with a closure. This step is similar to how Spores [Miller et al. 2014] identifies free variables. However unlike Spores, which requires users to explicitly declare closure elements to minimize serialization cost, fluid quotes do so automatically. Since fluid quotes are not intended to be transmitted over the network, serialization cost is not a concern.

For each identified external references, we add a forwarder to the generated closure class. We split these references into several groups: `Method` for method calls, `Constructor` for object instantiation, and `Immutable` for immutable variables. We use a separate type `Mutable` for mutable variables since the value may change after the closure is generated. While `Mutable` is used for mutable variable reads, we use the `Mutator` type for assignments.

$$\frac{cType = \texttt{Constructor} \quad typeSig = \texttt{class \$\_(\$p0: \$T0, ...)}}{F(expr, typeSig, cType, name) = \texttt{"def \$name(\$p0: \$T0, ...) = new \$expr(\$p0, ...)"}}$$

$$\frac{cType = \texttt{Method} \quad typeSig = \texttt{def \$\_(\$p0: \$T0, ...): \_}}{F(expr, typeSig, cType, name) = \texttt{"def \$name(\$p0: \$T0, ...) = \$expr(\$p0, ...)"}}$$

$$\frac{cType = \texttt{Immutable}}{F(expr, typeSig, cType, name) = \texttt{"val \$name = \$expr"}}$$

$$\frac{cType = \texttt{Mutable}}{F(expr, typeSig, cType, name) = \texttt{"def \$name = \$expr"}}$$

$$\frac{cType = \texttt{Mutator} \quad typeSig = \texttt{var \_: \$T}}{F(expr, typeSig, cType, name) = \texttt{"def \$name(newValue: \$T) = \{ \$expr = newValue \}"}}$$

Figure 4. Closure forwarder generation rules.

Each of these types come with a rule to generate a forwarder in the closure class. The following artificial example exercises each of these rules:

```
val quotedLocal = {
  def localMethod(a: Int) = a + 1
  class LocalClass(b: Int)
  val localImmutable = 123
  var localMutable = 456

  quote {
    localMutable = 1
    math.max(
      localMethod(1) +
        new LocalClass(1).hashCode,
      localImmutable + localMutable
    )
  }
  // expands to
  new FluidQuote[Int] {
    ...
    class Closure {
      def localMethodFwd(a: Int) =
        localMethod(a)
      def LocalClassFwd(b: Int) =
        new LocalClass(b)
      val localImmutableFwd = localImmutable
      def localMutableFwd = localMutable
      def localMutableMutator(newValue: Int) =
        { localMutable = newValue }
    }
  }
}
```

Formally, we define a forwarder generator which generates each of the members of the closure class one-by-one. We define this as the function $F(expr, typeSig, cType, name)$ where $expr$ is code to reference the local symbol (e.g. `"localImmutable"`), $typeSig$ is the type signature of the referred symbol (e.g. `val localImmutable: Int`), $cType$ is the reference type (e.g. `Immutable`), and $name$ is the name to use for the forwarder (e.g. `"localImmutableFwd"`). We describe this generator through rules in Figure 4.

We are now ready to use closures in the quoted code.

3.2 Quoted Code Transformation

We transform the quoted expression to use the closure as needed. Continuing our example, the fluid quote tracks the transformed AST in the `Expr` type member:

```
val quotedLocal = {
  ... // same as before
  quote { ... } // expands to
  new FluidQuote[Int] {
    type Expr = """{
      localMutableMutator(1)
      _root_.scala.math.max(
        localMethodFwd(1) +
          LocalClassFwd(1).hashCode,
        localImmutableFwd +
          localMutableFwd
      )
    }"""
    ...
  }
}
```

We avoid forwarders when possible since they come with a runtime cost. When we detect a reference to a global public value, we replace it with a path from the root package. In our example, we detect that `math.max` is public so replace it with `_root_.scala.math.max` and do not generate a forwarder. This is similar to Squid [Parreaux et al. 2017], which performs the transformation to maintain macro hygiene. To further reduce the closure size, when handling selection chains, we only create a closure for the shortest prefix that is not publicly accessible. For example, when processing `new LocalClass(1).hashCode` we detect that `hashCode`

is public so we only need to generate a forwarder for the `LocalClass` constructor instead of the entire expression.

Finally, we store the transformed tree as a string literal type, an instance of the closure, and a list of closure references into the fluid quote. We introduce two new type members: `Closure` to track the type of the closure object and `ClosureRefs` to track the closure reference list. In addition, we store an instance of the closure in the `closure` variable.

```
val quotedLocal = {
  ...
  quote { ... } // expands to
  new FluidQuote[Int] {
    ...
    class Closure { ... }
    type ClosureRefs =
      "localMethodFwd,LocalClassFwd,..."
    val closure = new Closure
  }
}
```

The closure reference list is necessary to re-transform the references at splice sites to refer to the associated `FluidQuote` instance. To finish our example, we can splice the quoted code and see how the expanded code uses the closure:

```
quotedLocal.splice
// expands to
{
  val closure = quotedLocal.closure
  closure.localMutableMutator(1)
  _root_.scala.math.max(
    closure.localMethodFwd(1) +
      closure.LocalClassFwd(1).hashCode,
    closure.localImmutableFwd +
      closure.localMutableFwd
  )
}
```

With closures, developers can splice fluid quotes containing any code wherever they want while ensuring correctness.

4 Composing Fluid Quotes

Fluid quotes can be combined by splicing them into a larger quoted expression. This composition system makes it possible to create code templates without writing macros. This is quite similar to MacroML [Ganz et al. 2001], which implements type-safe generative macros by tracking the types of expressions captured by quotes. However, unlike MacroML which executes macros as separate blocks of code, we use type-level operations to combine quoted expressions.

Consider a simple code template that adds two fluid quotes together. We can define this with the composition API:

```
def quotedAddValues(
  a: FluidQuote[Int],  b: FluidQuote[Int]
) = {
  quote(a.splice + b.splice)
}
```

With this template definition, we can call the function normally, but get back a fully resolved `FluidQuote` type that can be spliced.

```
def firstQuote = quote(1)
def secondQuote = quote(2)

val addedValues = quotedAddValues(
  firstQuote, secondQuote
)

println(addedValues.splice)
// expands at compile time to
println(1.+(2))
```

When compiling the body of the `quotedAddValues` function, the dependent types of `a` and `b` are unknown and therefore cannot be spliced using the techniques described so far. To handle this, we store references to the unknown types as a type-level linked list that can be resolved later and refer to the unknown fluid quotes in the serialized expression. In addition, we store the runtime fluid quote values of the sub-expressions so that their closures are accessible when splicing the composed expression.

```
def quotedAddValues(
  a: FluidQuote[Int],  b: FluidQuote[Int]
) = {
  quote(a.splice + b.splice)
  // expands to
  new FluidQuote[Int] {
    type Expr = "$expr0.+($expr1)"

    type ReferencedExprs = ...
    val referencedExprs = new Cons {
      type Head = a.type
      val head = a // expr0

      type Tail = ...
      val tail = new Cons {
        type Head = b.type
        val head = b // expr1

        type Tail = Nil
        val tail = Nil
      }
    }
  }
}
```

This approach to templating only supports generative macros which do not inspect the passed ASTs. This is limited

compared to general macros, but makes it significantly easier to write short templates. For example, if a developer wanted to write a template that runs a loop with a maximum number of iterations, they could do so easily with the composition feature.

```
def cappedLoop(condition: FluidQuote[Boolean],
               maxIterations: FluidQuote[Int])
              (thunk: FluidQuote[Unit]) = {
  quote {
    var itersLeft = maxIterations.splice
    while (itersLeft > 0 && condition.splice) {
      thunk.splice
      itersLeft -= 1
    }
  }
}
```

Calling this template and splicing the returned expression results in expanded code—all without writing any macro code. Furthermore, all the elements of the template are inlined directly into the final expression so we have no function call overhead.

```
cappedLoop(
  readLine("Try again?") == "y", 5
)(println("Trying again!")).splice
// expands at compile time to
var itersLeft = 5
while (
  itersLeft > 0 &&
    readLine("Try again?" == "y")
) {
  println("Trying again!")
  itersLeft -= 1
}
```

This templating style offers a lightweight solution to implementing library-specific optimizations while still ensuring flexibility of abstractions. Compared to regular inlining, using fluid quotes makes it possible to handle situations where template parameters may not be defined directly at the call site but passed in from elsewhere.

5 Splicing Restrictions

Fluid quotes come with a few restrictions since they depend on type propagation implemented by the host language. Specifically, they cannot be spliced when their dependent types are unknown and can face ambiguities when dealing with dynamically selected expressions. We offer partial remedies that help developers deal with such situations.

A fluid quote cannot be spliced in an environment where it is defined as a parameter, since its type members will be abstract types in that context. This mirrors a fundamental limitation of a general inlining system, since it is impossible to know the AST of a parameter when compiling a method body. When `splice` is called on a fluid quote whose expression type cannot be resolved, the macro raises an error.

```
def myExprSplicer(expr: FluidQuote[Int]): Int =
  expr.splice
  // ^ error: the AST of `expr` is unknown here
```

As a workaround for such situations, we include a runtime fallback that lets users execute the expression represented by a fluid quote without inlining it. This is implemented as an additional method of `FluidQuote` that returns the result of executing the expression.

```
quote(1 + 2)
// expands to
new FluidQuote[Int] {
  // ...
  def runtimeFallback: Int = 1.+(2)
}
```

In a more complex situation, the typechecker may face an ambiguity when branches return different `FluidQuote` types. In this case, the returned types will correspond to the least upper bound of the types across all the branches. This effectively erases the `Expr` type member since taking the least upper bound of two string literal types erases the type.

For example, in the code below, the static type of `branch` will be a `FluidQuote` but with all its type members erased since it is impossible to know at compile time which branch will be taken.

```
def branch(a: FluidQuote[Int],
           b: FluidQuote[Int]) =
  if (readLine("Use a?") == "yes") a else b
```

This can be partially remedied by replacing the branch with a custom branch that separately stores the types of all the fluid quotes and captures at runtime the branch taken. When this branched fluid quote is spliced, we splice all of the fluid quotes but check the branch condition at runtime to execute the corresponding expression. We can implement this idea with the closure and templating support discussed in the previous sections:

```
def quoteBranch[T](cond: Boolean)
                  (exprIf: FluidQuote[T],
                   exprElse: FluidQuote[T]) =
  quote(
    if (cond) exprIf.splice
    else exprElse.splice
  )

def branch(a: FluidQuote[Int],
           b: FluidQuote[Int]) =
  quoteBranch(readLine("Use a?") == "yes")(a, b)
```

By moving the branch into the quoted expression while still evaluating the condition outside to preserve the semantics of the program, users are able to dynamically pick between different fluid quotes.

6 Applications and Evaluation

With fluid quotes, library developers can easily implement performance optimizations and code transformations that require non-local program knowledge without limiting the abstractions in user code. We discuss two specific applications of fluid quotes: optimizing collection processing and improving the flexibility of language integrated queries.

6.1 Performant Collection Processing

In recent times, functional programming has grown into a popular technique to express transformations on data collections. By breaking down complex transformations into functional primitives [Burge 1975] such as `map` to transform individual elements, `filter` to select subsets of elements, and `scan` to accumulate aggregate results, users can write modular code. In addition, this approach is more scalable since each primitive can have an optimized implementation for the target platform such as specialized hardware or distributed systems. For example, functional transformations form the core of Apache Spark RDDs [Zaharia et al. 2012] to process data at scale.

However, using functions to transform data also comes with significant overhead. Since every transformation step is defined by a separate function, executing a pipeline on a single element involves a series of function calls that each add their own overhead. While advanced JIT compilers can eliminate some of this overhead [Prokopec et al. 2017], they struggle to apply optimizations such as vectorization since the data processing is not expressed as a traditional loop. The situation is even worse for ahead-of-time compilers since the transformation functions are distributed across the codebase and cannot be brought together for inlining.

Consider a simple vector transformation composed of many individual pieces:

```
val myVector = Vector(1, 2, ...)
myVector.view
  .map(_ + 1).map(_ + 2)
  .scanLeft(0)(_ + _).toVector
```

Note that we use `view` to prevent the creation of intermediate collections since we only care about the final result and without it performance would be even worse. If we manually fuse together the transformation steps, we are able to achieve an approximately **2.7x** speedup when running on OpenJDK 13 with a 2.9 GHz Intel i7-7820HQ and 16 GB RAM.

```
myVector.map {
  var acc = 0
  (in: Int) => {
    val step1 = in + 1
    val step2 = step1 + 2
    acc = acc + step2
    acc
  }
}
```

We present **pipelines**, a library based on fluid quotes that enables users to define transformations with functional APIs that compile down to a single optimized transformation without any function call overhead. To keep pipelines collection-agnostic, they only generate a function that emits the next output value when called with a new input value. As a result, pipelines are parameterized on both the input type as well as the output type. Pipelines only support stream operations such as `scan` instead of operations that produce a single output such as `fold`, in order to simultaneously support streaming data and concrete collections. With pipelines, users can get the same speedup as in the previous example without having to do the fusion manually.

Consider a program where we accumulate the sum of the squares of a collection of integers. Pipelines allow us to define independent steps and generate a fused transformation with no function call overhead.

```
def squareNums(nums: PipelineModel[Int, Int]) =
  nums.map(n => n * n)

def accumSum(nums: PipelineModel[Int, Int]) =
  nums.scanLeft(0)((acc, cur) => acc + cur)

val pipeline = accumSum(squareNums(
  PipelineModel.root[Int]))
val transformation = pipeline.instance
println((1 to 5).map(transformation))
// Seq(1, 5, 14, 30, 55)
```

Through fluid quotes, the `instance` macro emits a fused transformation that is nearly identical to hand-written code.

```
val transformation = {
  var acc0 = 0
  (in: Int) => {
    val step1 = in
    val step2 = step1 * step1
    acc0 = acc0 + step2
    acc0
  }
}
```

By combining all the transformation steps into a single function, pipelines generate an optimized transformation that eliminates the overhead of calling each step individually. We describe this process in the following subsections.

6.1.1 Pipeline Structure IR.

In order to perform stream fusion optimizations, we must first know the structure of the entire transformation graph. Pipeline structures use a simple intermediate representation that tracks transformation dependencies through parent references and user code through fluid quotes. We define this IR in Figure 5.

Pipelines store this transformation graph in a type member `Structure` similar to how fluid quotes store ASTs. Since

Pipeline Structure Types	
RootNode	Represents a root stream, which is the source of input values.
MapNode[P, op]	Maps each element $e \in P$ to $op(e)$.
ZipNode[$P1$, $P2$]	Merges each pair of elements $e1 \in P1$ and $e2 \in P2$ into a tuple $(e1, e2)$.
ScanLeftNode[P, $init$, op]	Transforms the stream P by accumulating to a state variable (initialized to $init$), updating the state to $op(prevState, e)$ when receiving $e \in P$, and outputting the new value of $state$.

Figure 5. Structure Types for Pipelines.

some fluid quotes may require a closure to handle local references, we also store a matching runtime object `structure` in the `PipelineModel` which contains the fluid quote values.

Other than the `instance` macro, which will analyze the transformation graph and emit the final fused transformation, none of the other methods in our implementation involve macros. This is possible because fluid quotes use type members so we do not need anything beyond the core language. This makes it easy for library authors to adopt fluid quotes to implement library-specific optimizations without introducing the significant engineering overhead of learning compiler-level APIs to implement macros.

6.1.2 Fused Transformation Generation. Once a pipeline has been created, we can resolve its full structure and generate an optimized transformation with all the individual steps fused together. This all happens in the `instance` macro.

The macro starts by recursively converting the structure type into objects which let us easily manipulate the computation graph. These objects store both the fluid quotes and parent references originally encoded through types. For example, if we have the following pipeline model:

```
val pipeline = PipelineModel.root[Int]
  .map(_ + 123).map(_ + 1)
```

The type member `pipeline.Structure` would be

```
MapNode[
  MapNode[RootNode, quote(_ + 123).type],
  quote(_ + 1).type
]
```

Now, generating a fused transformation simplifies to a traversal over the structure tree. The `instance` macro splices the fluid quotes for each node and generates code to pass data between nodes. The fused transformation for the example pipeline looks like the following:

```
(input: Int) => {
  val step1 = input + 123
  step1 + 1
}
```

When processing a `MapNode`, the macro splices a call to the node's fluid quote and add a new statement storing the result. For `ScanLeftNodes`, it adds a new state variable outside the function and update it inside the function by splicing a call to the accumulator function. Handling `ZipNode` is the simplest, since there are no fluid quotes involved and all the macro has to do is generate a new variable that stores a tuple of the two parent outputs.

6.1.3 Runtime Node Merging. To maintain correctness with stateful transformations, we must ensure that every stream node is calculated once in each execution of the pipeline. At compile time, we cannot tell if two nodes in the structure tree represent the same stream, since we have no way of identifying referential equality until the objects are instantiated. To handle this, we introduce a runtime check that de-duplicates computations.

Pipelines identify candidates for duplicated calculations by searching for nodes with identical contents and parents according to their types. Even when nodes have identical types, we must still perform a runtime check because two instances of the same pipeline may be created separately, which would result in identical types for pipelines that are not referentially equal. These nodes must be executed independently since they may have side effects.

Whenever the `instance` macro detects that an identical existing node has already been processed, it emits a branch around the output of the current node to check for referential equality and skip the computation. For example, consider the following code:

```
val shared = PipelineModel.root[Unit].map { _ =>
  println("hello!"); 1
}
val zipped = shared.zip(shared)
zipped.instance
```

When processing the second parent of the `ZipNode`, pipelines detects a potential duplication and generate a check to only calculates the `MapNode` if its runtime value is not referentially equal to the first parent:

```
(in: Unit) => {
  val step1 = { println("hello!"); 1 }
  val step2 = if (zipped.structure.p1 eq
                  zipped.structure.p2) step1
              else { println("hello!"); 1 }
  (step1, step2)
}
```

Since we only emit runtime checks if duplication is possible, we are able to generate a fused transformation that matches the expected behavior while minimizing the amount of extra code to perform de-duplication.

6.1.4 Benchmarks. We compare our pipelines implementation against Scala views, which implement a similar lazy evaluation scheme. Our benchmarks consist of transforming a vector of elements into either a scalar or a new vector. For views, we first convert the vector into a view and then perform the transformation. For pipelines, we build a transformation and pass each input element through it.

We benchmark our implementation of pipelines with various tasks expressed in a functional programming style. Our workloads that produce scalars do not involve constructing new collections, so the majority of computation time is spent in transforming elements. On the other hand, workloads producing new vectors closely match real-world uses of functional programming. We run all our workloads on a collection of 50 floating point elements $x_1 ... x_n$:

- **sumOfSquares** (scalar): calculates $\sum_{i=1}^{n} x_i^2$
- **sumSlidingProduct** (scalar): calculates $\sum_{i=1}^{n} \prod_{j=1}^{i} x_j$
- **tenMapAdds** (vector): transforms $x_i \rightarrow x_i + 1 + ... + 10$
- **addToSquare** (vector): transforms $x_i \rightarrow x_i + x_i^2$
- **multiplyStringifyAddParse** (vector): multiplies x_i by 1000 and casts to an integer, converts to a string, appends "10" to the string, and parses into an integer

We run benchmarks with JMH, which eliminates noise due to JIT compilation, on OpenJDK 13 with a 2.9 GHz Intel i7-7820HQ and 16 Gb RAM. The following plot summarizes the results of these benchmarks.

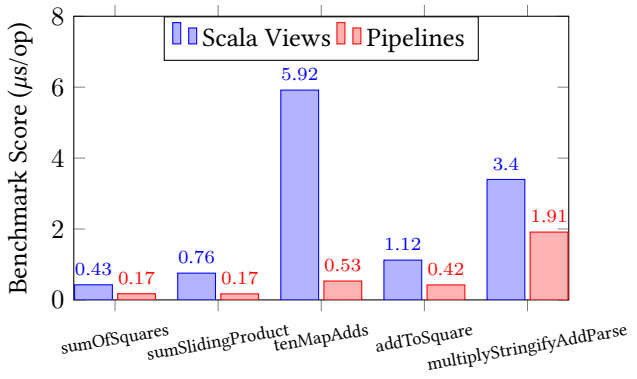

Figure 6. Results of pipeline benchmarks (lower is better).

With pipelines, users can define complex stream transformations with modular code while still generating efficient imperative code without the overhead of individual function calls. In all our benchmarks, pipelines are able to generate significant speedups by eliminating intermediate function calls, which results in friendlier bytecode that the JIT can more thoroughly optimize. Even in the "multiplyStringifyAddParse" workload where there are a small number of complex steps, the transformations enabled by fluid quotes almost double the performance. With pipelines involving many simple steps, such as the "tenMapAdds" workload, we see that pipelines can result in order-of-magnitude performance improvements.

Many languages offer optimizations trying to solve similar problems to pipelines. We discuss how pipelines offer a more complete optimization scheme in the following subsections.

6.1.5 Comparison to Haskell Stream Fusion. Haskell, where stream based transformations like `map` and `filter` form the primary interface for collection processing, includes stream fusion optimizations to avoid function call overhead [Coutts et al. 2007]. These optimizations identify successive stream transformations and merge them into a single operation.

While this technique works well to optimize short pieces of collection transformations, it does not scale to larger applications where transformation segments may be split across multiple functions. Because Haskell's stream optimizer works by matching on ASTs, it cannot identify relationships between transformations spread in separate functions and so each transformation chunk is compiled into a separate stream that cannot be fused further.

By utilizing type members to store the transformation graph, fluid quotes enable us to statically resolve the entire stream structure. Instead of fusing within a local context, pipelines instead gathers the entire stream structure and optimizes with a birds-eye view of the transformation graph, which significantly increases the flexibility of the system.

6.1.6 Comparison to Rust Iterators. Rust iterators [Mozilla 2020] allows transforming collections using functional programming primitives such as `map` and `filter` while generating "zero-cost" code that is as efficient as hand-written loops. Rust achieves this with its trait system, which enables libraries to associate behavior with runtime data while avoiding the cost of dynamic dispatch.

Rust iterators build up a tree of traits at compile time since each transformation inlines the parent iterator. The final trait contains a flat transformation without function calls to previous steps. However, because these transformations work by wrapping a new trait implementation at the call site of each transformation, Rust iterators cannot handle graphs where an iterator is used along multiple paths. For example, consider the following code:

```
let v1 = vec![1, 2, 3];
let base_iter = v1.iter().map(|i| i + 1);

let iter_1 = base_iter.map(|i| i + 1);
let iter_2 = base_iter.map(|i| i * 2);
let final_iter = iter_1.zip(iter_2)
```

```
  .map(|(x, y)| x + y);

dbg!(final_iter.collect::<Vec<i32>>());
```

Since iterators work by inlining all the transformations up to the latest step, the function `|i| i + 1` will be double inlined since it shows up in both branches of the `zip`. This introduces unnecessary computation and would lead to unexpected behavior if the function had side effects. In order to support such graphs, we need an overall view to identify which nodes can refer to the same runtime pipeline. With fluid quotes, we use types to resolve a complete graph that can be used to detect such shared paths.

6.2 Flexible Language-Integrated Queries

Fluid quotes also make code transformation techniques more powerful. We evaluate this capability by extending Quill [Brasil 2020], a library for language integrated queries [Torgersen 2006] that generates SQL queries from a quoted domain-specific language [Najd et al. 2016].

With Quill, users interact with database tables in a manner similar to regular collections, but the operations and functions passed into them are tracked at compile time to generate an equivalent SQL query. For example, we can define a type that describes the columns of a table in our database and query rows from that table. The result is type-safe, familiar code that is able to express complex queries.

```
case class Circle(color: String, radius: Float)
ctx.run(ctx.query[Circle]
  .filter(_.color == "red")
  .map(_.radius).avg)
// generated query:
// "SELECT AVG(c.radius) FROM Circle c
// WHERE c.color = 'red'"
```

However, Quill faces a significant limitation when it comes to user abstractions. Because Quill, like other existing strategies for metaprogramming, cannot reach into code defined behind an abstraction boundary, it often has to fall back to generating queries at runtime. This happens when Quill cannot resolve the full set of operations that the user defined for the query. If we modify the example to abstract out the predicate, Quill emits a compile-time warning saying that the query must be generated at runtime.

```
def averageCircleRadius(
  predicate: ctx.Quoted[Circle => Boolean]
) = ctx.query[Circle].filter(predicate(_))
      .map(_.radius).avg

ctx.run(averageCircleRadius(
  ctx.quote((c: Circle) => c.color == “red”)
))
// warning: "Dynamic query"
```

We improve this situation by using fluid quotes to track the functions passed into query operations and use type members to track the overall structure of the query at compile time. Then, when the user needs to perform the query, we can generate code that Quill can directly handle at compile time instead of needing to wait for runtime generation that adds overhead to the system.

We define our extension as the `FluidQuery` class, which offers similar APIs as Quill but stores the query functions as fluid quotes instead of immediately analyzing them at the call-site. This allows users to define abstractions in their code while taking advantage of the compile-time query generation capabilities. We can rewrite the example above using `FluidQuery` and see that the query is generated at compile-time instead of runtime:

```
def averageCircleRadius(
  predicate: FluidFunction1[Circle, Boolean]
) = FluidQuery.query[Circle].filter(predicate)
      .map(_.radius).avg

ctx.run(averageCircleRadius(
  (c: Circle) => c.color == "red"
).get(ctx))
// generated query:
// "SELECT AVG(c.radius) FROM Circle c
// WHERE c.color = 'red'"
```

Similar to pipelines, the `FluidQuery` class stores fluid functions in a type-level data structure that captures all the operations in a hierarchical manner. For example, the final query structure type in the above example is:

```
AvgNode[
  MapNode[
    FilterNode[
      RootNode,
      quote((c: Circle) =>
        c.color == "red").type
    ],
    quote(_.radius).type
  ]
]
```

Once this type is constructed, we can unwrap the structure into a regular Quill query when the `get` method is called. This unwrapping is similar to pipelines but without any of the de-duplication logic since there is no `zip` transformation. In our example, the call to `get` is expanded into:

```
ctx.run(ctx.query[Circle]
  .filter((c: Circle) => c.color == "red")
  .map(_.radius).avg)
```

One notable limitation of our implementation is the lack of support for `flatMap`, a common operator used when combining multiple tables in a single query. This limitation exists because `flatMap` would need to take a function returning a `FluidQuery` instance, but due to the lack of dependent function types [Rapoport and Lhoták 2019] in Scala 2, the type

members of the inner `FluidQuery` would be erased and we would not be able to resolve the structure of that component at compile-time. We hope to address this limitation in future work through the dependent type capabilities of Scala 3.

7 Related Work

7.1 Lightweight Modular Staging

Fluid quotes and Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] address many similar problems, such as optimizing hierarchical structures to eliminate abstraction costs and translation of user code into other domains. However, LMS can only operate at runtime since it tracks ASTs through runtime values, while fluid quotes can be analyzed ahead-of-time since the ASTs are tracked through types.

While fluid quotes offer a powerful tool for many situations, using the type system also introduces a few limitations that LMS does not face. By performing code generation at runtime, LMS is able to handle recursive structures, which fluid quotes cannot due to their use of the type system. This limitation shows up in one application of LMS: generating efficient code from parser combinator definitions [Béguet and Jonnalagedda 2014; Jonnalagedda et al. 2014].

An implementation of this system with fluid quotes would require capturing the entire model at the type-level, but this is not possible since recursive dependencies would lead to infinitely deep types. LMS can handle such situations better since it can perform runtime memoization of structures and avoid the infinite recursion challenge.

7.2 Unified Multi-Stage Programming and Macros

One of the main features of Scala 3 is a new quotation-based macro and multi-stage programming system [Stucki et al. 2018] that offers an interface similar to that of fluid quotes. However, like traditional systems, these macros can only expand ahead-of-time when all the quoted code is expressed at the call-site. Although the system offers a LMS-style alternative for situations where quoted code must be passed through runtime code, this requires the entire Scala compiler to be packaged along with the application and introduces overhead that may not be acceptable for some applications.

For example, the rewrite of the stream fusion library Strymonas [Kiselyov et al. 2017] cannot fuse streams ahead-of-time if they are defined outside macro implementations. This forces developers to split stream building into separate modules that complicates the code base significantly. As we saw in Section 6.1, fluid quotes enables us to implement a stream fusion system to process the entire transformation graph as a whole without the limiting where users define the transformation structure. This makes the stream fusion system in this paper a much more viable alternative to runtime equivalents such as views, since we preserve the object-oriented user experience and allow users to pass around streams in runtime code.

7.3 Static Staging with Braid

Braid [Sampson et al. 2017] is a system that enables ahead-of-time code generation through static stages. By offering a common language to write both runtime and code generation logic, Braid makes it easy to generate code for other domains and pass runtime values from one language into another. However, it requires developers to separate the metaprogramming code from runtime code in order to perform expansion at compile time. While this is acceptable for generating code snippets that cannot be compiled ahead-of-time such as WebGL, this does not work when developers need ahead-of-time expansions for regular runtime code.

Our stream fusion application demonstrates how fluid quotes can be used to generate code for other domains ahead-of-time while handling user abstractions. By using types, fluid quotes are able to eliminate the traditional dependency on explicitly separating metaprogramming code by using the typechecker to track ASTs. By executing all the expansions during the type checking phase, fluid quotes let users blend together runtime code and quotes while ensuring that they can be resolved ahead-of-time to be spliced elsewhere.

8 Conclusion

Metaprogramming is a powerful technique that lets libraries and developers define code transformations for performance optimization and translation into other domains. Macros make it easy for developers to quickly get started with metaprogramming by allowing them to use the same language for compile-time and runtime. However, macro code is executed at compile-time without access to runtime values, which limits their scope to call-site transformations. With fluid quotes, we have bridged the ahead-of-time expansion and runtime code universes through types. Fluid quotes track quoted ASTs in runtime code and allow users to splice them at compile-time wherever they want. Fluid quotes let macros reach beyond the call-site to access code passed around through abstraction boundaries such as objects, methods, and variables. In this paper, we discussed how we can achieve all these capabilities while maintaining correctness even when environment-specific data is involved. In addition, we explored how fluid quotes open up composition strategies where code templates can be defined just like higher-order functions. Finally, we demonstrated how fluid quotes enable collection processing optimizations and compile-time query generation even in the face of abstraction boundaries that limit traditional macros.

Acknowledgments

We would like to thank Jonathan Bachrach for his insightful feedback on early versions of this paper and applications of fluid quotes.

References

Alan Bawden and Jonathan Rees. 1988. Syntactic Closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88)*. Association for Computing Machinery, New York, NY, USA, 86–95. https://doi.org/10.1145/62678.62687

Eric Béguet and Manohar Jonnalagedda. 2014. Accelerating Parser Combinators with Macros. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. Association for Computing Machinery, New York, NY, USA, 7–17. https://doi.org/10.1145/2637647.2637653

Flavio Brasil. 2020. *Quill*. https://getquill.io/

W. H. Burge. 1975. Stream Processing Functions. *IBM Journal of Research and Development* 19, 1 (Jan 1975), 12–25. https://doi.org/10.1147/rd.191.0012

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

Eugene Burmako. 2019. *Macros: Blackbox vs Whitebox*. http://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

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

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

JetBrains. 2018. *Inline Functions and Reified Type Parameters - Kotlin Programming Language*. https://kotlinlang.org/docs/reference/inline-functions.html

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *SIGPLAN Not.* 52, 1 (Jan. 2017), 285–299. https://doi.org/10.1145/3093333.3009880

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–333.

Mozilla. 2020. Processing a Series of Items with Iterators - The Rust Programming Language. https://doc.rust-lang.org/book/ch13-02-iterators.html

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-Specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2847538.2847541

Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala*. Artima Inc.

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

Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002

Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 145 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360571

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

Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. *Proc. ACM Program. Lang.* 1, OOPSLA, Article Article 71 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133895

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 (GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3278122.3278139

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

Mads Torgersen. 2006. Language Integrated Query: Unified Querying across Data Sources and Programming Languages. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 736–737. https://doi.org/10.1145/1176617.1176700

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 2.