Semantics-Preserving Inlining for Metaprogramming

Nicolas Stucki
Aggelos Biboudis
nicolas.stucki@epfl.ch
biboudis@gmail.com
EPFL
Lausanne, Swizerland

Sébastien Doeraene
sebastien.doeraene@epfl.ch
Scala Center (EPFL)
Lausanne, Swizerland

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

Abstract

Inlining is used in many different ways in programming languages: some languages use it as a compiler-directive solely for optimization, some use it as a metaprogramming feature, and others lay their design in-between. This paper presents inlining through the lens of metaprogramming and we describe a powerful set of metaprogramming constructs that help programmers to unfold domain-specific decisions at compile-time. In a multi-paradigm language like Scala, the concern for generality of inlining poses several interesting questions and the challenge we tackle is to offer inlining without changing the model seen by the programmer. In this paper, we explore these questions by explaining the rationale behind the design of Scala-3's inlining capability and how it relates to its metaprogramming architecture.

Keywords: Inlining, OO, Metaprogramming, Macros

1 Introduction

Programming languages [1, 12, 14] usually offer inlining as a compiler directive for optimization purposes. In some of these, an inline directive is mandatory to trigger inlining, in others it is just a hint for the optimizer. The expectation from a users' perspective is simple: the semantic reasoning for a method call should remain unaffected by the presence of inlining. In other words, inlining is expected to be *semantics-preserving* and consequently this form of inlining can be done late in the compiler pipeline. Inlining is typically implemented in the backend of a compiler, where code representations are simpler to deal with. The motivation is simple and not far from the motivation behind the code below: we desire to avoid one method call, thus the method body itself replaces the call.

```
inline def square(x: Int): Int = x * x
square(y) // inlined as y*y to avoid a call to square at runtime
```

Some programming languages do not provide inlining at the language level and rely on automatic detection of inlining opportunities instead. Java and its dynamic features such as dynamic dispatch and hot execution paths, push the (much more complex) inlining further down the pipeline at the level of the JVM. Scala primarily relies on the JVM for performance, though it can inline while compiling to help a bit the JIT compiler.

However, there is another angle that we can view inlining from. Inlining can be seen as a form of metaprogramming, where inlining is the syntactic construct that turns a program into a program generator [8, 11]. Indeed, the snippet above describes a metaprogram, the metaprogram that is going to generate a method body at the call site. This may seem like a very simple, and obvious observation, but in this paper we show that metaprogramming through inlining can be seen as a structured, and *semantics-preserving* methodology for metaprogramming.

Typically, in the aforementioned programming languages, what is getting inlined is a piece of untyped code that is then typed at the call-site. Therefore the semantics are only defined at the call-site and there are no semantics to preserve.

```
// C++
#define square(X)  X * X // * does not have any semantics here
square(y) // inlined as y*y and then typed
```

Sometimes such inlining is done in a different language fragment (as in C++ templates) at other times it uses standard methods and calls, as in D inline functions and C++ *constrexpr* functions. In these cases, inlining is not necessarily semantics preserving, and it also usually does not provide type-safety guarantees at the definition site. In the former case, C++ allows code that may not generate valid code for some of the type parameters of the template, and is only checked for the specific type arguments. That is, there is no guarantee that an expanded inline call will always type-check. In the latter case, functions come with a complete spec on what their method bodies can include to be considered

SCALA '20, November 13, 2020, Virtual, USA

ACM ISBN 978-1-4503-8177-2/20/11…$15.00
https://doi.org/10.1145/3426426.3428486

as `constexpr`. The relevant keyword shows the intention of inlining and it does not mean that the function is necessarily going to be executed at compile-time.

Non-semantic inlining can be categorized as *syntactic-inlining*. To see the difference between the two, consider this example with overloaded methods:

```
def f(x: Any) = 1
def f(x: String) = 2
inline def g[T](x: T) = f(x)
g("abc")
```

When using semantics-preserving inlining, the `inline` could be dropped without changing the result. That means that the call `f(x)` would resolve to the first alternative and the result is 1. With syntactic-inlining, we inline the call to `g`, expanding it to `f("abc")`. This then resolves to the second alternative of `f`, yielding 2. So, in a sense, the syntactic-inlining replaces overloading-resolution with compile-time multi-method dispatch.

Syntactic-inlining is very powerful and has been used to great effect to produce heavily specialized code. But can also be difficult to reason about and it can lead to errors in expanded code that are hard to track down.

Other compile-time metaprogramming constructs have inlining as an implicit part of what they do. For instance, a macro in Lisp [4] or Scala 2 [2] moves the code of the macro to the call-site (this is a form of inlining) and then executes the macro's code at this point. Can we disentangle inlining from the other metaprogramming features? This is the approach followed in Scala 3 [18]. It offers a powerful set of metaprogramming constructs, including staging with quotes `'{..}` and splices `${..}` [3, 5, 13, 17]. Quotes delay the execution of code while splices compute code that will be inserted in a larger piece of code. Staging is turned from a runtime code-generation feature to a compile-time macro feature by combining it with inlining.

A macro is an inline function with a top-level splice. E.g.

```
inline def power(m: Double, inline n: Int): Double =
  ${ powerCode('{m}, '{n}) }
```

A call-site such as `power(x, 4)` is expanded by inlining its implementation and the value of argument `n` into it:

```
val m = x
${ powerCode('{m}, '{4}) }
```

The contents of this splice is then executed in the context of the call-site at compile-time.

When used in conjunction with other metaprogramming constructs, inlining has to be done early, typically during type checking, because that is when these other constructs apply. Furthermore, it makes sense that inlining by itself should be as "boring" as possible. It should be type-safe and semantics-preserving by default. At the same time, inlined definitions should be usable and composable in interesting ways. For instance, since normal methods can override methods in parent classes or implement abstract methods, it makes sense to allow the same flexibility for inlined methods, as far as is possible.

These deliberations lead us to the following principles:

1. *Semantics-preserving*: A call to an inline method should have exactly the same semantics as the same method without the inline modifier.
2. *Generality*: We want to be able to use and define inline methods as generally as possible, as long as (1) is satisfied.

In an object-oriented language like Scala, the concern for generality poses several interesting questions, which are answered in this paper:

- Can inline methods implement abstract methods?
- Can inline methods override concrete methods?
- Can inline methods be overridden themselves?
- Can inline methods be abstract themselves?

There is another question here that will influence the answers to these four questions.

- Can inline methods be called at runtime?

It will turn out that the answer to this question is "it depends". Some inline methods will need to be callable at runtime, in order to maintain semantics preservation. Others cannot be called at runtime because they use metaprogramming constructs that can only be executed at compile time.

In this paper, we explore these questions by presenting the rationale and design of Scala-3's inlining concept and how it relates to its metaprogramming architecture.

In section 2 we discuss how our design of inline functions is based on the principles above. In section 3 we extend the discussion to the design of inline methods. In section 4 we introduce a simple extension to inline functions that can affect the semantics at call-site (not the call itself). In section 5 we show some of the metaprogramming features that can be built on top of semantics-preserving inlining. We conclude by discussing the related work in section 8.

2 Inline Functions

We introduce the `inline` modifier to denote that a function is an inline function. A function with `inline` can be called as any other function would.

```
inline def logged[T](logger: Logger, x: T): Unit =
  logger.log(x)
```

Assuming that `Logger` has a proper definition of `log`, the code would type check. Inlining this code seems simple enough as shown below.

```
logged(new RefinedLogger, 3)
// expands to: (new RefinedLogger).log(3)
```

But what if the definitions of `log` were the following?

```
class Logger {
  def log(x: Any): Unit = println(x)
}

class RefinedLogger extends Logger {
  override def log(x: Any): Unit = println("Any: " + x)
  def log(x: Int): Unit = println("Int: " + x)
}
```

If we look at `logger.log(x)` we can see that the only option is to call `Logger.log(Any)`. By examining the inline site, one would argue that the method to be invoked should be `RefinedLogger.log(Int)`. However, this would imply a change in the semantics of the code after inlining where the overloading resolution results in different method selection depending on whether we use inlining or not. With or without inline, the code should perform the same operation, therefore we need to retain the original overload resolution to avoid breaking the first principle.

The elaboration of extension methods[1], implicit conversion, and implicit resolution must be preserved as these are part of the overload resolution. All these could change if they are performed with different type information, which could potentially end up calling different methods.

While overloading should not change, it is also possible to perform de-virtualization without breaking semantics. This is an optimization that precomputes the virtual dispatch resolution (override) that would otherwise happen at runtime. In the example, we would call `RefinedLogger.log(Any)` directly.

2.1 Inline Values

`val` definitions can also be marked as `inline`. An inline `val` inlines the contents of its right-hand side (RHS) as an `inline def` would. Unlike `inline defs`, when inlining an `inline val` we cannot inline any arbitrary RHS as it may recompute the values several times, which would break the evaluation order semantics.

```
inline val x = 4
x // replaced with 4

def z: Int = ...
inline val y = z // error: z is not a known value
y // cannot replace y with z as z may have side effects
```

Therefore we constrain the RHS to be pure and to be able to reduce to a value at compile-time.

This means that the RHS can only contain literal values or references that reduce to a literal constant such as another inline `val` or `def`.

[1]Extension methods allow one to add methods to a type after the type is defined–https://web.archive.org/web/20200421114625/https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html

2.2 Parameters of Inline Functions

To support *semantics-preserving* inlining in the presence of effects during the evaluation of arguments, the latter must be let-bound at the call-site. To illustrate this, consider the `square` inline function.

```
inline def square(x: Int): Int = x * x
```

This function could be called with an arbitrary parameter which could have side effects. As Scala provides by-value call semantics, the argument expression must be evaluated once and before the evaluation of the body of the function. The solution is to let-bind the evaluation of the expression passed as an argument to preserve the evaluation order. In the following example, the method call `n` contains an I/O operation:

```
def n: Int = scala.io.StdIn.readInt()

square(n) // expands to:
          //   val x1 = n
          //   x1 * x1
```

If we were to inline it as `n * n`, we would mistakenly read two numbers from the standard input. This shows why it is imperative to have let-bound arguments.

Scala also provides by-name parameters. These parameters need to be evaluated each time they are referred to.

```
inline def twice(thunk: =>Unit): Int = {
  thunk
  thunk
}

twice { print("Hello!") } // prints: Hello!Hello!
// expands to:
//   def thunk = print("Hello!")
//   thunk
//   thunk
```

Instead of binding them to a `val` we bind them to a `def`. We do not replace each reference `thunk` by `print("Hello!")` to avoid code duplication.

Constant folding. After methods and `vals` are inlined we can perform constant folding optimizations on primitive types. This implies that constants are propagated and primitive operations are performed on them.

```
square(3) // expands to: val x1 = 3; x1 * x1
          // optimized to: 3 * 3
          // then optimized to: 9
```

Additionally, if constant folding evaluates the condition of an `if` to a known value, then we can partially evaluate the if and eliminate one of the branches. This allows a limited but simple way to generate simplified code. More complex and domain-specific optimizations demand the use of custom metalanguage code with macros.

Inline parameters. In some cases, we do not want to let-bind the arguments and instead we wish to just inline them directly where they are used. For this purpose, we allow parameters to be marked as `inline`, but only in `inline` functions. This makes it a metaprogramming feature as it provides semantics that are not expressible in normal functions. These parameters have, by construction, semantics that are similar to by-name and may generate duplicated code. The `inline` parameters allow further specialization of code by duplicating code and allowing each copy to be specialized in a different way. This specialization might come from further inlining or by one of the metaprogramming features.

For example, it is possible to remove closure allocations early on using inline parameters.

```
inline def tabulate3[T](inline f: Int => T): List[T] =
  List(f(0), f(1), f(2))

tabulate3(x => 2*x)
// expands to: List((x => 2*x)(0), (x => 2*x)(1), (x => 2*x)(2))
// which reduces to : List(0, 2, 4)
```

Without the inline parameter, we would have been forced to let-bind the instantiation of the closure which may have side effects. An optimizer might remove it only if there are no side effects where we can ensure that the whole expression is ignored.

2.3 Recursion

Inline functions can call other inline functions and in particular themselves. Calls to an inline function `f` within another inline function `g` are not immediately inlined within the body of `g`. Instead they are only inlined once `g` has itself be inlined in third, non-`inline` function `h`.

```
inline def f() = 3
inline def g() = f() // f not inlined here
def h() = g() // first inlines g then inlines f
```

Now consider the recursive inline function `power`.

```
inline def power(x: Double, n: Int): Double =
  if (n == 0) 1.0
  else if (n == 1) x
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)

power(expr, 10)
// expands to:
//   val x = expr.      // x^1
//   val x1 = x * x     // x^2
//   val x2 = x1 * x1   // x^4
//   val x3 = x2 * x    // x^5
//   x3 * x3            // x^10
```

Note the importance of parameter semantics: if `x` would not be let-bound the computation would be linear instead of the expected logarithmic time. In this example, we assume that `n` will be a constant and that it can be constant folded in the conditions of the `if`s. In turn, we assumed that after constant folding only one branch will be kept and eventually will stop the recursion. This will not always be the case.

With recursive inlining we introduce potentially non- terminating inline expansions. Consider the previous example, but with an unknown value of `n`.

```
power(expr, m)
// expands in a first step to:
//   val n = m
//   if (n == 0) 1.0
//   else if (n == 1) x
//   else if (n % 2 == 1) x * power(x, n - 1)
//   else power(x * x, n / 2)
```

It is apparent that we could take one more unfolding step to the next call of `power` and then recursively do the same again, so we would never end. The expansion will continue until a predefined maximum inline depth limit[2] is reached and fail compilation.

As we ensure that all calls to inline functions are inlined or a compilation failure occurs, we never need to call these methods at runtime. This implies that the inline function definitions can be removed from the generated code.

2.4 Inline If

An `inline if` provides a variant of `if` that must be constant-folded in its condition to eliminate one of the branches. If that cannot be done, an error is emitted and no further expansion within the `if` is attempted. Using `inline if` ensures that we always partially evaluate the `if` at compile-time. An `inline if` and an `if` have the exact same semantics at runtime.

This is also useful as an explicit convergence check when using recursive inline functions.

```
inline def power(x: Double, n: Int): Double =
  inline if (n == 0) 1.0
  else inline if (n == 1) x
  else inline if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)

power(expr, m)
// expands in a first step to:
//   val n = m
//   inline if (n == 0) 1.0
//   else inline if (n == 1) x
//   else inline if (n % 2 == 1) x * power(x, n - 1)
//   else power(x * x, n / 2)
```

As `n==0` does not have a known value at compile-time, the expansion fails and no further nested expansions are attempted. The same happens for the other nested `inline if`.

3 Inline Methods

Inline can also be used for methods in classes or traits. Inline methods will be able to access object fields and interact with virtual dispatch.

[2]This limit can be increased by the user if necessary

3.1 Members and Bridges

An inline method may refer in its body to the `this` reference of the current class or to any private member. Let us consider the following inline method defined in a class.

```
class InlineLogger {

  private var count = 0

  inline def log[T](op: () => T): Unit = {
    val result = op() // may contain call to log
    count += 1
    println(count + "> " + result)
  }
}
```

First, the method evaluates the operation, then it updates the private field `count`, and then prints it with the result. Note that the operation may contain nested calls to `log` which would use the current count.

```
def inlineLogger = new InlineLogger
inlineLogger.log(() => 5)
// naive expansion:
//    val ths = inlineLogger
//    val result = (() => 5)()
//    ths.count += 1
//    println(ths.count + "> " + result)
```

We need to make sure the prefix of the application (i.e., the receiver) is only evaluated once by let-binding it to `ths`. Then we use `ths` in place of `this` in the inlined code. Unfortunately, the inlined code contains a reference to the private field `count` which is not accessible from the call-site (under the JVM model). This does not break *semantics-preservation* but does greatly limit what could be used in the body of an inline method.

To lift this limitation, we instead generate bridges for all members that may not be accessible at the call-site. For the `count` we would create a getter and setter that make the bridge possible. This ensures that when the call is inlined all references are still accessible.

```
class InlineLogger {

  private var count = 0

  def inline$count: Int = // only in generated code
    count

  def inline$count_=(x: Int): Unit = // only in generated code
    count = x

  inline def log[T](op: () => T): Unit = {
    val result = op()
    this.inline$count_=(this.inline$count + 1)
    println(this.inline$count + "> " + result)
  }
}
```

3.2 Overloads

As inline methods must be *semantics-preserving*, the definition and resolution of overloads should not be affected. The overload resolution algorithm does not need any modification, hence it considers all inline and non-inline functions as equivalent. For example, the following variants perform the same overload resolution.

```
def log(msg: String): Unit = ...
def log(x: Any): Unit = ...
log("a")
```

```
inline def log(msg: String): Unit = ...
inline def log(x: Any): Unit = ...
log("a")
```

```
def log(msg: String): Unit = ...
inline def log(x: Any): Unit = ...
log("a")
```

```
inline def log(msg: String): Unit = ...
def log(x: Any): Unit = ...
log("a")
```

3.3 Abstract Methods and Overrides

Inline methods implementing interfaces. Consider the following example, where we have an inline definition implementing a non-inline abstract method.

```
trait Logger {
  def log[T](op: () => T): Unit
}

class InlineLogger extends Logger {
  inline def log[T](op: () => T): Unit = println(op())
}
```

If we have an instance of `InlineLogger` we can just inline the code. But now we also allow calls to `Logger.log` which will not be inlined.

```
def logged[T](logger: Logger, x: () => T): Unit =
  logger.log(x)

logged(new InlineLogger, 3)
```

This implies that we have a call to `Logger.log` at runtime which should be dispatched to `InlineLogger.log`. Therefore if the inline method implements an interface we cannot ensure it will be completely inlined and we must retain the code at runtime.

Inline methods overriding normal methods. Consider the following example, where we have an inline definition that overrides a non-inline method.

```
class Logger {
  def log[T](op: () => T): Unit = println(op())
}

class NoLogger extends Logger {
  inline def log[T](op: () => T): Unit = ()
}
```

If we have an instance of `NoLogger` we can just inline the code. But once again we also allow calls to `Logger.log`, which will not be inlined. Unlike with the implementation of the abstract method, it would be tempting to say that, as there exists an implementation of `log`, we could remove `NoLogger.log` from the generated code. However, that would not be semantics-preserving. In order to ensure that calling `Logger.log` on a `NoLogger` does indeed no logging, we must also keep the implementation of `NoLogger.log` at runtime. Then, virtual dispatch will be able to find the correct implementation at runtime.

Overriding inline methods. Consider the following example, where we have an inline method that is overridden by another method.

```
class Logger {
  inline def log[T](op: () => T): Unit = println(op())
}

class NoLogger extends Logger {
  /*inline*/ def log[T](op: () => T): Unit = ()
}
```

This time we turned things around and are trying to override an inline method with any method (inline or not). Using the same `logged` example we have a different way in which semantics-preservation fails.

```
def logged[T](logger: Logger, x: T): Unit =
  logger.log(x) // expanded to the contents Logger.log

logged(new NoLogger)(3)
```

As `log` is inlined from `Logger` before we know which logger we are using, we will always call `Logger.log`. Instead, we would have expected to call `NoLogger.log` which is a semantic breakage.

In general, no inline method can be safely overridden as it bypasses virtual dispatch resolution. Therefore all inline methods are *effectively final*.

Abstract inline methods. Consider the following example of an abstract inline method.

```
trait AbstInlineLogger {
  inline def log[T](op: () => T): Unit
}
```

It would be possible to implement this interface with a non-inline function as it would perfectly preserve the semantics. But this does not offer any expressivity advantage over a normal abstract method. Instead, we will restrict it to only be implementable by inline methods to guarantee that the calls can be inlined. Unlike plain abstract methods, the abstract inline method does not enforce the implementations of inline methods to be retained at runtime.

```
class InlineLogger extends AbstInlineLogger {
  inline def log[T](op: () => T): Unit = println(op())
}

class NoLogger extends AbstInlineLogger {
  inline def log[T](op: () => T): Unit = ()
}
```

It is clear that all implementations of `log` will be inlined if we know statically the receiver of the `log` call which defines the methods. But, can we ever call `AbstInlineLogger.log` directly and still have it inlined?

```
def logged[T](logger: AbstInlineLogger, x: () => T): Unit =
  logger.log(x) // error: cannot inline abstract method
```

Calling it directly will not work as it is impossible to inline. But, by inlining the previous code we can get this abstraction to work.

```
inline def logged[T](logger: AbstInlineLogger, x: () => T): Unit =
  logger.log(x)

logged(new InlineLogger, () => 5)
logged(new NoLogger, () => 6)
```

Now, when `logged` is inlined, the call `logger.log` gets de-virtualized at compile-time and then can be inlined. Crucially, with abstract inline methods, we provide a way to guarantee that all calls to such methods are de-virtualized and inlined at compile-time.

Inline methods overriding with inline parameters. Consider the following example where a method is overridden with an inline method and its parameter is marked inline.

```
class Logger {
  def log[T](x: T): Unit = println(x)
}

class NoLogger extends Logger {
  inline def log[T](inline x: T): Unit = ()
}

val noLogger: Logger = new NoLogger
noLogger.log(f()) // expands to: ()

val logger: Logger = noLogger
logger.log(f())
```

Here, inline has a deeper effect and provides the possibility to override the call semantics. Whenever we call `Logger.log`, the arguments will be evaluated with the standard by-value semantics. In this case, this implies the evaluation of `f()` which might have side effects. But, when calling `NoLogger.log` the evaluation of the argument is just dropped. As a consequence, the call semantics changed and this pattern should not be allowed.

Abstract inline methods and inline parameters. Consider the following example of an abstract inline method with an inline parameter. We implement it with a method that has the same signature.

```
trait AbstInlineLogger {
  inline def log[T](inline x: T): Unit
}

class InlineLogger extends AbstInlineLogger {
  inline def log[T](inline x: T): Unit = println(x)
}

class NoLogger extends AbstInlineLogger {
  inline def log[T](inline x: T): Unit = ()
}

inline def logged[T](logger: AbstInlineLogger, inline x: T): Unit =
  logger.log(x)

val inlineLogger = new InlineLogger
logged(inlineLogger, f()) // expands to: println(f())

val noLogger = new NoLogger
logged(noLogger, f()) // expands to: ()
```

With this pattern, the semantics of the inline parameter `x` are preserved across all abstractions, until the de-virtualization of `AbstInlineLogger.log` into `InlineLogger.log` (which preserves the call to `f()`) and `NoLogger.log` (which eliminates it). Therefore, we can allow abstract methods to have inline parameters, as long as all its implementations use corresponding inline parameters as well. This pattern shows another useful reason to have abstract inline methods: regular abstract methods cannot have inline parameters, as we saw earlier, while abstract inline methods can.

Now, consider an alternative implementation of `NoLogger` that does evaluate the argument but does not print it.

```
class NoLogger extends AbstInlineLogger {
  inline def log[T](inline x: T): Unit = {
    val y = x
    ()
  }
}
val noLogger = new NoLogger
logged(noLogger, f())
// expands to:
//   val y = f()
//   ()
```

It also works, but we just emulated by-value parameters. Instead, we could also just mark the parameter as a normal by-value parameter and let it take care of the binding. This does not contradict the `AbstInlineLogger.log` interface.

```
class NoLogger extends AbstInlineLogger {
  inline def log[T](x: T): Unit = ()
}
val noLogger = new NoLogger
noLogger(f())
// expands to:
//   val x = f()
//   ()
```

Inline methods summary. All inline methods are `final`. Abstract inline methods can only be implemented by inline methods. If an inline method overrides/implements a normal method then it must be retained (i.e. cannot be erased). Retained methods cannot have inline parameters.

4 Transparent Inlining

A simple but powerful metaprogramming extension to inlining is the ability to *refine* the type of an expression after the call is inlined. The inline call is typed and inlined retaining its elaboration as with normal inline functions. But instead of typing the inlined expression as the return time of the inline function, we take the precise type of the inlined expression. This unlocks the ability to change the semantics at the call-site around the inlined call, without changing the semantics of the code that was inlined.

We use the `transparent` keyword to enable this feature.

```
transparent inline def choose(b: Boolean): Any =
  if (b) 3 else "three"

val obj1: Int = choose(true)
val obj2: String = choose(false)
```

This may be used to influence type inference, overload resolution, and implicit resolution at the call-site. But as with all inlines, it may not change the elaboration of the inlined code. To illustrate this, consider the following code where we have a definition of a method that is overwritten, overloaded and returns a more precise type.

```
class A {
  def f(a: A): A = ...
}
class B extends A {
  override def f(a: A): B = ...
  def f(x: B): String = ...
}

transparent inline def g(inline a1: A, inline a2: A): A =
  a1.f(a2)
val b: B = ???
val y = g(b, b) // expands to: val y: B = b.f(b)

def h(a: A): Unit = println("A")
def h(b: B): Unit = println("B")
h(y) // prints "B" because g is transparent (otherwise would be "A")
```

From section 3, we know that for the call semantics to be preserved we need to make sure that the inlined call to `f` should be to `B.f(A)` at runtime. Before inlining the code in `g`, the call to `f` returned an `A` as we were calling `A.f(A)`. After inlining the code in `g`, this same call gets de-virualized and we know that we actually call `B.f(A)` and it returns a `B`. Hence the inlined expression is of type `B` and `y` is inferred to be a `B` as well. Then the rest of the code outside the call is subject to this more precise type.

The call to `h(y)` will be statically resolved to a call to the `h(B)` overload as `y` was typed as `B`. On the other hand, if `g` had been a normal function or a non-transparent inline function, the type of `y` would have been `A`. In this case the overload resolution would have chosen `h(A)`. This shows how transparent inline function can affect the semantics around their call-site.

To be able to propagate the types as we do we need to inline while typing. Any non-transparent inlining can be performed after typing.

Each call to a transparent inline will respect the semantic preservation. In contrast to non-transparent inline function, replacing it with the same function without transparent may change the semantics of the overall program. It is possible to emulate the transparent semantics adding casts that align with the results of implementation.

```
/*transparent*/ inline def choose(b: Boolean): Any =
  if (b) 3 else "three"
val obj1: Int = choose(true).asInstanceOf[Int]
val obj2: String = choose(false).asInstanceOf[String]
```

For this to be sound we would need to prove that these casts have exactly the type of the result based on the types or statically known values of the arguments.

5 Metaprogramming

With metaprogramming, we introduce metalanguage features that allow code manipulation. In general, these features only manipulate code in place which maintains the metaprogramming abstractions simple. This is possible because `inline` takes care of placing the metaprogram where it needs to be. In most cases, these metaprogramming features do not have runtime semantics until expanded at the call-site. But all of them rely on the knowledge that the code around them or in their parameters preserved their semantics when inlining. These features may be combined with transparent inlining. In this section, we have a non-exhaustive list of metaprogramming features that are supported by inlining.

5.1 Inline Error

An `error` method provides a way to emit custom error messages. The error will be emitted if a call to `error` is inlined and not eliminated as a dead branch.

```
import scala.compiletime.error
inline def div(n: Int, m: Int): Int =
  inline if (m == 0) error("Cannot divide by 0") else n / m
```

`error` is not subject to the semantics-preservation principle, since it is illegal in code that is retained at run-time. The same observation applies to most metaprogramming features described in this section.

5.2 Inline Pattern Matching

This variant of `match` provides a way to match on the static type of some expression. It ensures that only one branch is kept. In the following example, the scrutinee, `x`, is an inline parameter that we can pattern match on at compile time.

```
transparent inline def half(inline x: Any): Any =
  inline x match {
    case x: Int => x / 2
    case x: Double => x / 2.0d
  }

half(1.0d) // expands to: 1.0d / 2.0d
half(2) // expands to: 2 / 2
val n: Any = 3
half(n) // error: n is not statically known to be an Int or a Double
```

The `inline match` will use the static type of the scrutinee and keep the branch that matches said type. For this to work, the patterns must be non-overlapping. Unlike the `inline if`, this reduction is not necessarily equivalent to its runtime counterpart when we have more type information.

5.3 Inline Summoning

If we need to summon implicit evidence provided by the call-site within a method we generally need to pass it as an argument of that method. But we may want to conditionally generate different code based on the existence of such implicit. This is not possible if it is part of the arguments as it would require it before expanding the code.

For this purpose, we introduce a set of delayed summoning (such as `summonInline` and `summonFrom`) that can be used within the body of an inline but will only be resolved at call-site.

```
import scala.compiletime.summonFrom
inline def setFor[T]: Set[T] =
  summonFrom {
    case ord: Ordering[T] => new TreeSet[T](ord)
    case _                => new HashSet[T]
  }
```

In a sense, `summonFrom` is a `transparent inline` as the expanded expression will have the type of the body of the chosen branch.

5.4 Inlining and Macros

Here is how we define a macro that generates code to compute the power of a number.

```
inline def power(x: Double, inline n: Int): Double =
  ${ powerCode('{x}, '{n}) }
```

The program is split into the macro definition `power` and code generators/analyzers `powerCode` and `powerUnrolled`.

```
def powerCode(x: Expr[Double], n: Expr[Long])(...): Expr[Int] =
  n.unlift match {
    case Some(m) => powerUnrolled(x, m) // statically known n
    case None => '{ Math.pow(${x}, ${n}) }
  }

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

The macro metalanguage provides two core constructs for code manipulation.

- `'{...}` quotes representing code fragments of type `Expr[T]` where `T` is the type of the code within

- `${...}` splices that insert code fragments into larger code fragments

The code directly in quotes delays the execution of the code, while the code within the splices computes a code fragment now. For example, `'{Math.pow(${x}, ${n})}` represents a snippet where we will insert the code of `x` and `n`. The un-lifting operation `Expr.unlift` allows us to extract the value of `n` if it is known at the call-site.

If we use a splice outside of a quote, as in `power`, we call it a macro. Such a splice will evaluate its contents at compile time. To make this evaluation efficient we require the code within the top-level splice to be a simple static call to a precompiled function. This way we only interpret a single reflective function call which then executes any user-defined compiled code. In theory it would be possible to let the users call this directly using the metalanguage.

```
${ powerCode('{x}, '{2}) } // would expand to: x * x
```

But if the users had to use the metalanguage directly, the usability of such a feature would have a high complexity cost. Instead, by using inline we can hide the metalanguage behind a normal method call that does not mention the metalanguage. We also avoid expanding the macros before the code is inlined.

```
inline def power(x: Double, inline n: Int): Double =
  ${ powerCode('{x}, '{n}) }
```

In this model, the macro expansion logic simply needs to evaluate and replace a piece of code. Now the users of this macro only need to know how to call a method.

```
power(x, 2) // expanded by inline to: ${ powerCode('{x}, '{2}) }
            // then by macro to: x * x
```

For macro overrides, we additionally expand the macro inside of the inline function[3] as it must exist at runtime. In this case, the macro should also be able to generate a generic fallback version of the code that does not have the call-site information.

Given that the implementation of the macro is done directly in the language rather than the metalanguage, the macro can execute arbitrary code at compile-time. This provides extra flexibility and expressivity that is not available when using the metalanguage constructs directly.

6 Implementation

Next, we describe the implementation of inlining, as described in this work, as merged in the Scala 3 compiler.

Inlining is performed while typing and inlines fully elaborated typed ASTs. The reason for this design choice is to support the implementation of transparent inlining. One of the very first steps we need to make is to obtain the typed ASTs. This can be done either via the definitions that we are currently typing or from a published TASTy (serialized AST in a binary format) [9] artifact. TASTy contains the fully elaborated typed ASTs of a complete class. From this artifact, it is relatively simple to extract the original AST of the method. Quoted code fragments are also encoded[4] in TASTy.

Once we have the AST, the next step is to performing β-reduction. Most of the complexity comes from making sure that during inlining we make all types as precise as possible without changing the resolution of overloads. When we perform the inlining, we make sure that all references in the code will be accessible at any inline site by generating public accessors if needed.

It is worth noting that overload resolution did not change. Extra checks where added to make sure that the override constraints hold. These constraints are summarized at the end of section 3. Furthermore, all inline method definitions are erased from the code except if marked as *retained*. The RHS of retained methods is evaluated as if inlined to execute any metaprogramming features.

7 Applicability

Using inlining as a compiler directive is already widely used and we advocate that the extra restrictions on method overriding are portable to any OO programming language.

The use of inlining as a base for metaprogramming could be used in other compiled languages in general. The quotes and splices were inspired by MetaOCaml for runtime code generation. They were transformed into compile-time code generation by simply allowing top-level splices inline methods. Other metaprogramming features like the error and pattern matching would also be useful in many languages.

8 Related Work

F# supports inlining of generic functions [14]. However, since generic numeric code–code that uses primitive operators–is treated differently for each numeric type, the mechanism of inlining demands specialized support for type inference. As a result, inline generic functions can have statically resolved type parameters whereas non-inline functions cannot. Scala, in combination with our work, does not infer a different type for the following method:

```
inline def f[T: Numeric](x: T, y: T): T = x + x * y
```

The compiler resolves overloaded methods uniformly and orthogonally to the inlining mechanism (note that `Numeric` is a view bound). F# infers statically resolved type parameters in the inferred type of the corresponding definition of f in F#. F# does not support the equivalent of inline if, inline match, inline overriding or the equivalent to transparent.

[3] And expand at all the call-sites.

[4] Details of the encoding can be found in [13]

In C++, inlining is a compiler hint that an optimizer may or may not follow. However, inlining is not binding compiler implementors to use inline substitution for any function that is not marked inline and vice versa. Similarly to our system, C++17 supports both function and variable inlining. It is worth noting that since C++ supports external linkage, linking behavior needs to be changed to support inlining.

`constexpr` was one of the additions in C++11 and proves crucial in simplifying template metaprogramming. A constant expression defines that an expression can be evaluated at compile-time and is implied to be inline. A constant function can return a constant value and may or may not be evaluated at compile time. In C++x20, `consteval` denotes *immediate functions* (not semantically equivalent to normal functions), which are guaranteed to be inlined and evaluated at compile-time. C++ supports `if constexpr` statements, similar to our `inline if`. In Scala, constant expressions are specified by a very limited set of rules, hence evaluation occurs only inside inline ifs and pattern matches. Right-hand sides of inline values and arguments for inline parameters must be constant expressions in the sense defined by the SLS § 6.24, including platform-specific extensions such as constant folding of pure numeric computations. `constexpr` comes with a very complex set of rules that defines what a `constexpr` function is; essentially a completely specified sub-language. In our work, we decide to abstain from strong compile-time evaluation guarantees to support semantics-preserving inlining. Since all the aforementioned variants of const expressions in C++ offer a very powerful set of compile-time evaluation in C++ also implies inlining, we can compare that aspect too. Firstly, constant expressions are strictly term-level features (as opposed to template metaprograms). In our work, as shown by transparent inlining we can refine the type of an expression after the call is inlined.

D [1] supports the usual compiler directive called inline. Like C++ it is an advice to the compiler. Similarly to C++, D is also equipped with a powerful template metaprogramming capability. While C++ uses a functional style for templates, in D a template looks like imperative code, so syntactically D is very close to what a user would write at the term level. Our `inline if`, similarly to D supports conditional compilation based on arguments.

D's *Compile Time Function Execution (CTFE)* is also part of the compile-time metaprogramming but on the interpretation side instead of merely inlining. D functions that are portable and free from side-effects can be executed at compile-time. While inlining is a declaration-level directive in our work, in D it is triggered by various "static" contexts such as a `static-if` or a dimension argument in a `static array`. One of the limitations in D is that the function source code must be available while in our work it can also be loaded from compiled code. In the compiler, CTFE comes after the AST manipulation phase (naming, type assignment, etc) has been completed and performs essentially interpretation much like C++. However, as in C++, D cannot introduce new types (or more precise) in the context.

MetaOCaml and MetaML [7, 15–17] offer a distinction between the metalanguage and an object language via staging annotations–brackets, escape and run. The aforementioned syntactic mondalities are introduced to denote where the evaluation needs to be deferred and we already cross the boundary of semantics-preserving code. At this point, we can navigate and guide freely the process of generating code from a quoted domain-specific language as shown in past work [6, 10, 13]. The macro system that comes in Scala 3, described in 5.4 essentially gains inspiration from these technologies and completes what we present in this work. Racket is considered to have one of the most advanced macro systems and racket macros can be viewed as compiler extensions that can expand syntax into existing forms. The work we present in this paper differs greatly from this direction.

Swift supports cross-module inlining and specialization with two attributes: `inlinable` and `usableFromInline`. The first can be applied to functions and methods, among others, exposing that declaration's implementation as part of the module's public interface. The second introduces a notion of an Application binary interface (ABI)-public declaration. Swift's attributes offer an inlining mechanism similar to C++. The most important distinction between Swift and our work is that inlinable declarations can only reference ABI-public declarations while we also support access to private methods via bridges. Our work provides automated detection and generation of said bridges at the cost of the potential of leaking private implementation details out of the public ABI.

9 Conclusion and Future Work

In this work we introduce an inline language primitive to support metaprogramming features. We showed the importance of preserving the semantics while inlining, including the implications of having methods and virtual dispatch. We listed a few metaprogramming features that use inlining and showed how the metalanguage takes advantage of inlining to remain simple.

Currently, we provide several out-of-the-box metaprogramming solutions to be used just by inlining while others require full-blown macros. For example, `inline if`, `inline match`, `summonInline` and `error` are all supported in some form by macros but we provide simpler primitives for those operations. As future work, we should identify a core set of metaprogramming features that are often required and provide implementations for them in the standard library.

Changing the implementation of transparent inline methods may break source compatibility, while a normal inline method may break binary compatibility. We need to explore how those can be mitigated and if it is possible to automatically detect all these cases. Swift's approach to the ABI might

be considered for this purpose, even though it has an extra syntactic overhead.

It would be good to formally prove the soundness of this inlining system. It would be interesting to prove the unsoundness of the system if the overriding restrictions are removed.

Acknowledgments

We thank the anonymous reviewers of the program committee for their constructive comments. We gratefully acknowledge funding by the Swiss National Science Foundation under grants 200021_166154 (Effects as Implicit Capabilities) and 407540_167213 (Programming Language Abstractions for Big Data).

References

[1] Andrei Alexandrescu. 2010. *The D Programming Language: The D Programming Lan_p1.* Addison-Wesley Professional.

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

[3] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604.

[4] Timothy P Hart. 1963. MACRO definitions for LISP. (1963).

[5] 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.

[6] 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* (Paris, France) *(POPL '17)*. ACM, 285–299.

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

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

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

[10] 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* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 127–136.

[11] 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.

[12] Bjarne Stroustrup. 2000. *The C++ programming language.* Pearson Education India.

[13] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. ACM, New York, NY, USA, 14–27.

[14] Don Syme. 2012. The F# 3.0 Language Specification. https://web.archive.org/web/20170325225238/http://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf.

[15] 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.

[16] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *In Proc. of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. ACM, New York, NY, USA, 26–37.

[17] Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *In Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. ACM, New York, NY, USA, 203–217.

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