Polyvariant Staging

Vojin Jovanovic Eugene Burmako Denys Shabalin Martin Odersky

École polytechnique fédérale de Lausanne—EPFL
{first}.{last}@epfl.ch

Abstract

In staged languages each term is executed at a single *compilation stage* that is determined based on: quotation (e.g., MetaOCaml), or types (e.g., LMS). To ensure cross-stage persistence (i.e., correctness) types of next/previous stage terms must annotated. Consequentially staged functions are *monovariant*: their arguments can not vary with respect to a compilation stage.

Type based staged languages require type annotations of all *next stage* terms, as well as implementing reification and code generation logic for all next stage types. Further, when we use staging at host language compile-time, all terms scheduled to execute at runtime require type annotations and all libraries used at runtime require logic for reification and code generation. Annotations are especially noticeable in languages with local type inference as types of method parameters must be provided by the programmer.

We introduce a type based staging extension for Scala where we transparently extend Scala with polymorphic binding-time information. With this information we enable parametrically polymorphic parameters to also be stage polymorphic. With this scheme defining stage polyvariant functions can be done through existing language features (i.e., bounded parametric polymorphism). Further, terms whose types are monomorphically annotated are executed in the earlier stage of compilation, in our case, at host language compile-time. Annotated types rely on the polymorphic binding time to represent merely *compile-time views* of original types. This obviates the necessity for reification and code generation logic. We compare our framework with LMS and show that in majority of programs we require less type annotations while we achieve same performance improvements.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Multi-Stage Programming, Partial Evaluation

1. Introduction

Multi-stage programming (or *staging*) is a meta-programming technique where compilation is separated in multiple *stages*. Execution of each stage outputs code that is executed in the next stage of compilation. The first stage of compilation happens at the *host language* compile time, the second stage happens at the host language runtime, the third stage happens at runtime of runtime generated code, etc. Different stages of compilation can be executed in the same language [**? ?**] or in different languages [**? ?**]. In this work we will focus on staging systems where all stages are in the same language and that, through static typing, assure that terms in the next stage are well typed.

Notable staging systems in statically typed languages are MetaOCaml [**? ?**] and LMS [**?**]. These systems were successfully applied as a *partial evaluatior* [**?**]: for removing abstraction overheads in high-level programs [**? ?**], for domain-specific languages [**? ? ?**], and for converting language interpreters into compilers [**? ?**]. Staging originates from research on two-level [**? ?**] and multi-level [**?**] calculi.

We show an example of how staging is used for partial evaluation of a function for computing the inner product of two vectors[1]:

```
def dot[T:Numeric](v1: Vector[T], v2: Vector[T]): T =
  (v1 zip v2).foldLeft(zero[T]) {
    case (prod, (cl, cr)) => prod + cl * cr
  }
```

In function `dot`, if vector sizes are constant, the inner product can be partially evaluated into a sum of products of vector components. To achieve partial evaluation, we must communicate to the staging system that operations on values of vector components should be executed in the next stage. The compilation stage in which a term is executed is determined by *code quotation* (in MetaOCaml) or by parametric types `Rep` (in LMS). In LMS marking that the vector size is statically known is achieved by annotating only vector elements with a `Rep` type[2]:

```
def dot[T:Numeric]
  (v1: Vector[Rep[T]], v2: Vector[Rep[T]]): Rep[T]
```

Here the `Rep` annotations on `Rep[T]` denote that elements of vectors will be known only in the next stage (in LMS, this is a stage after run-time compilation). After run-time compilation `zip`, `foldLeft`, and pattern matching inside the closure will not exist as they were evaluated in the previous stage of compilation (host language runtime). Note that in LMS unannotated code is always executed during host-language runtime and type-annotated code is executed after run-time compilation.

Staging at host language compile time. How can we use staging for programs whose values are statically known at the host language compile-time (the first stage)? Existing staging frameworks treat unannotated terms as runtime values of the host language and annotated terms as values in later stages of compilation. Even if we would take that the first stage is executed at the host language compile time, we would have to annotate all run-time values. Annotat-

[Copyright notice will appear here once 'preprint' option is removed.]

ing all values is cumbersome since host language run-time values comprise the majority of user programs (§**??**).

MacroML [**?**] expresses macros as two-stage computations that start executing from host language compile time. In MacroML, parameters of macros can be annotated as an early stage computation. These parameters can then used in escaped terms for compile-time computation. Terms scheduled for runtime execution, withing the escaped terms, again need to be quoted with brackets. This, in effect, imposes quotation for both escaping and brackets which requiring additional effort.

Code Duplication. Staging systems based on type annotations (e.g., LMS and type-directed partial evaluation [**?**]) inherently require code duplication as, a priory, no operations are defined on `Rep` annotated types. For example, in the LMS version of the `dot` function, all numeric types (i.e., `Rep[Int]`, `Rep[Double]`, etc.) must be re-implemented in order to typecheck the programs and achieve code generation for the next stage.

Sujeeth et al. [**?**] and Jovanovic et al. [**?**] propose generating code for the next stage computations based on a language specification. These approaches solve the problem, but they require writing additional specification for the libraries, require a large machinery for code generation, and support only restricted parts of Scala.

Annotating the Previous Stage. The main idea of this paper is that *annotated types* should denote computations that happen during the *previous stage* of compilation. The reason is that static terms appear less frequently than run-time terms in a large set of analyzed programs (§**??**). Therefore, annotating static terms introduces less overhead for the programmer.

We treat annotated types as *compile-time views* of existing data types. Compile-time view of a type denotes that all operations on that type are executed at host language compile time. We promote types to their compile-time views with the `@ct` annotation (e.g., `Int@ct`). Similarly, statically known terms can be promoted their compile time duals with the `ct` function on the term level. By having two views of the same type we obviate the need for introducing reification and code generation logic for existing types.

With compile-time views, to require that vectors `v1` and `v2` are static and to partially evaluate the function, a programmer needs to make a simple modification of the `dot` signature:

```
def dot[V: Numeric@ct]
  (v1: Vector[V]@ct, v2: Vector[V]@ct): V
```

Since, vector elements are polymorphic the result of the function can be a dynamic value, or a compile-time view that can be further used for compile-time computations. The binding time of the return type of `dot` will match the binding time of vector elements:

```
// [el1, el2, el3, el4] are dynamic decimals
dot(Vector(el1, el2), Vector(el3, el4))
  ↪ (el1 * el3 + el2 * el4): Double

// ct promotes static terms to compile-time views
dot(Vector(ct(2.0), ct(4.0)),
    Vector(ct(1.0), ct(10.0)))
  ↪ 42: Double@ct
```

In this paper we contribute to the state of the art:

- By introducing compile-time views (§**??**) as means to succinctly achieve type safe two-stage programming starting from host language compile time.

[1] All code examples are written in *Scala*. It is necessary to know the basics of Scala to comprehend this paper.

[2] In this work we use LMS as a representative of type-based staging systems.

```
package object scalact {
  final class ct extends StaticAnnotation

  @compileTimeOnly def ct[T](body: => T): T = ???
}
```

Figure 1. Interface of ScalaCT.

- By obviating the need for reification and code generation logic in type based staging systems.
- By demonstrating the usefulness of compile-time views in four case studies (§**??**): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [**? ? ?**].

We have implemented a staging extension for Scala ScalaCT[3]. ScalaCT has a minimal interface (§**??**) based on type annotations. We have evaluated performance gains and the validity of ScalaCT on all case studies (§**??**) and compared them to LMS. In all benchmarks (§**??**) our evaluator performs the same as LMS and gives significant performance gains compared to original programs.

2. Compile-Time Views in Scala

In this section we informally present ScalaCT, a staging extension for Scala based on compile-time views. ScalaCT is a compiler plugin that executes in a phase after the Scala type checker. The plugin takes as input typechecked Scala programs and uses type annotations [**?**] to track and verify information about the biding-time of terms. It supports only two stages of compilation: host language compile-time (types annotated with `@ct`) and host language run-time (unannotated code).

To the user, ScalaCT exposes a minimal interface (Figure **??**) with a single annotation `ct` and a single function `ct`.

Annotation `ct` is used on types (e.g., `Int@ct`) to promote them to their compile-time views. The annotation is integrated in the Scala's type system and, therefore, can be arbitrarily nested in different variants of types.

Since all operations on compile-time views are executed at compile time, non-generic method parameters and result types of compile-time views also become compile-time views. Generic parameters remain unchanged as their binding time is defined during corresponding type application. Table **??** shows how the `@ct` annotation can be placed on types and how it affects method signatures on annotated types.

In Table **??**, on `Int@ct` both parameters and result types of all methods are also compile-time views. On the other hand, `Vector[Int]@ct` has parameters of all methods transformed except the generic ones. In effect, this, makes higher order combinators of `Vector` operate on dynamic values, thus, function `f` passed to `map` accepts the dynamic value as input. Type `Vector[Int@ct]@ct` has all methods executed at compile-time. The return type of the function `map` on `Vector[Int@ct]@ct` can still be either dynamic or a compile-time view due to the type parameter `U`.

Annotation `ct` can be used to achieve simple inlining of statically known methods and functions. This is achieved by putting the annotation of the method/function definition:

```
@ct def dot[T: Numeric]
  (v1: Vector[T], v2: Vector[T]): T
```

Annotated methods will have an annotated method type

```
((v1: Vector[T], v2: Vector[T]) => T)@ct
```

[3] Source code: `https://github.com/scala-ct/scala-ct`.

Table 1. Compile-time views of types and their corresponding method signatures.

Annotated Type	Type's Method Signatures
`Int@ct`	`+(rhs: Int@ct): Int@ct`
`Vector[Int]@ct`	`map[U](f: (Int => U)@ct): Vector[U]@ct` `length: Int@ct`
`Vector[Int@ct]@ct`	`map[U](f: (Int@ct => U)@ct): Vector[U]@ct` `length: Int@ct`
`Map[Int@ct, Int]@ct`	`get(key: Int@ct): Option[Int]@ct`

which can not be written by the users. This is not the first time that inlining is achieved through partial evaluation [**?**].

Function `ct` is used at the term level for promoting literals, modules, and methods/functions into their compile-time views. Without `ct` we would not be able to instantiate compile-time views of types. Table **??** shows how different types of terms are promoted to their compile-time views. An exception for promoting terms to compile-time views is the `new` construct. Here we use the type annotation on the constructed type.

2.1 Tracking Binding-Time of Terms

Internally ScalaCT has additional type annotations for tracking the binding time of terms. Type of each term is annotated with either `dynamic`, `static`, or `ct`. `dynamic` denotes that the term can only be known at runtime, `static` that the term is known at compile-time but it will not be computed at compile time, and `ct` that the term will be computed at compile-time.

Tracking static terms was studied in the context of binding-time analysis in partial evaluation for typed [**?**] and untyped [**?**] languages. We use similar techniques, however, unlike in partial evaluation we do not evaluate static terms at compile time. They are tracked in the same way as `ct`, for verifying correctness of term promotions to `ct` and providing convenient implicit conversions. Static terms are evaluated only when they are explicitly marked by the programmer with `ct`.

In ScalaCT language literals, functions, direct class constructor calls with static arguments, and static method calls with static arguments are marked as static. Examples of static terms are

```
1.0, "1", (x: Int => x), new Cons(1, Nil), List(1,2,3)
```

2.2 Subtyping

We use subtyping of Scala to simplify tracking of binding times by introducing a subtyping relation between `dynamic`, `static`, and `ct`. We argue that a `static` type is a more specific `dynamic` as it is statically known and that `ct` is more specific than `static` as its operations are executed at compile time. Therefore we establish that

```
ct <: static <: dynamic
```

The use of subtyping simplifies tracking binding times of terms as in all cases where least upper bounds are calculated we can use the same mechanism for binding-times. An interesting example are the binding times of type parameters:

```
ct(List)(1, ct(2)): List[Int@static]@ct
ct(List)(ct(1), ct(2)): List[Int@ct]@ct
ct(List)((x: Int@dynamic), ct(2)): List[Int@dynamic]@ct
```

Notable exception are control flow constructs for which the original Scala rules for least upper bounds do not hold. The binding-time of control flow constructs does not depend only on the return type of the branches but also on the conditionals. For example, if both branches of an `if` construct are `static` the result can still be `dynamic` if the condition is `dynamic`. Here subtyping also helps as the binding type of the return value is simply calculated as `lub(c, thn, elz)` where `lub(tps: Type*)` is a function for computing the least upper bounds of types, and `c`, `thn`, `elz` are respectively binding times of the condition, the then branch, and the else branch. The same principles can be applied for pattern matching.

2.3 Well-Formedness of Compile-Time Views

Earlier stages of computation can not depend on values from later stages. This property, defined as *cross-stage persistence* [**? ?**], imposes that all operations on compile-time views must known at compile time.

To satisfy cross-stage persistence ScalaCT verifies that binding time of composite types (e.g., polymorphic types, function types, record types, etc.) is always a subtype of the binding time of their components. In the following example, we show malformed types and examples of terms that are inconsistent:

```
xs: List[Int@ct]      => ct(Predef).println(xs.head)
fn: (Int@ct=>Int@ct) => ct(Predef).println(fn(ct(1)))
```

In the first example the program would, according to the semantics of `@ct`, print a head of the list at compile time. However, the head of the list is known only in the runtime stage. In the second example the program should print the result of `fn` at compile time but the body of the function will be known only at runtime. By causality such examples are not possible.

On functions/methods the `ct` annotation requires that function/method bodies are known at compile-time. Otherwise, inlining of such functions/methods would not be possible at compile-time. In Scala, method bodies are statically known in objects and classes with final methods, thus, the `ct` annotation is only applicable on such methods.

2.4 Minimizing the Number of Annotations

One of the design goals of ScalaCT is to minimize the number of superfluous staging annotations. We achieve this by implicitly adding annotations and term promotions that would with high probability be added by the user.

Adding `ct` to Functions. Due to cross-stage persistence if a single parameter of a function is annotated with `ct` the whole function must also be `ct`. Since forgetting this annotation would only result in an error we implicitly add the `ct` annotation when at least one parameter type or the result type is marked as `ct`.

Implicit Conversions. If method parameters require the compile-time view of a type the corresponding arguments in method application would always have to be promoted to `ct`. In some libraries this could require an inconveniently large number of annotations.

To minimize the number of required annotations we introduce *implicit conversions* from certain `static` terms to `ct` terms. Note that we use a custom mechanism for implicit conversions based on type annotations. This is necessary as Scala implicit conversions are oblivious to type annotations (§**??**).

The conversions support translation of language literals, direct class constructor calls with static arguments, and static method

Table 2. Promotion of terms to their compile-time views.

Promoted Term	Term's Type
`ct(Vector)(1, 2, 3)`	`: Vector[Int]@ct`
`ct(Vector)(ct(1), ct(2), ct(3))`	`: Vector[Int@ct]@ct`
`ct((x: Int@ct) => x)`	`: (Int@ct => Int@ct)@ct`
`ct((x: Int) => x)`	`: (Int => Int)@ct`
`new (::@ct)(1, Nil)`	`: (::[Int])@ct`
`new (::@ct)(ct(1), ct(Nil))`	`: (::[Int@ct])@ct`

calls with static arguments into their compile-time views. Since our compile-time evaluator does not use Asai's [**?** **?**] method to keep track of the value of each static term, we disallow implicit conversions of terms with static variables.

For example, for a factorial function

```
def fact(n: Int @ct): Int@ct =
  if (n == 0) 1 else fact(n - 1)
```

we will not require promotions of literals `0`, and `1`. Furthermore, the function can be invoked without promoting the argument into it's compile-time view:

```
fact(5)
  ↪ 120
```

Without implicit conversions the factorial functions would be more verbose

```
def fact(n: Int @ct): Int@ct =
  if (n == ct(0)) ct(1) else fact(n - ct(1))
```

as well as each function application (`fact(ct(5))`).

Implicit conversions are safe in all cases except when the user should be warned about potential code explosion. For example, a user could accidentally call `fact` with a very large and cause code explosion without even knowing that staging is happening. If `ct` was required it would remind the users about the potential problems. In the desing of ScalaCT we decided to prefer less annotations over code-explosion prevention.

3. Case Studies

In this section we present selected use-cases for compile-time views that, at the same time, demonstrate step-by-step the mechanics behind ScalaCT. We start by inlining a simple function with staging (§**??**), then do the canonical staging example of the integer power function (§**??**), then we demonstrate how variable argument functions can be desugared into the core functionality (§**??**). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-class related abstraction an be removed (§**??**).

3.1 Inlining Expressed Through Staging

Function inlining can be expressed as staged computation [**?**]. Inlining is achieved when a statically known function body is applied with symbolic arguments. In ScalaCT we use the `ct` annotation on functions and methods to achieve inlining:

```
@ct def zero[T](implicit num: Numeric[T]) = num.zero

zero[Double]
  ↪ num.zero
```

3.2 Recursion

The canonical example in staging literature is partial evaluation of the power function where exponent is an integer:

```
def pow(base: Double, exp: Int): Double =
  if (exp == 0) 1 else base * pow(base, exp - 1)
```

When the exponent (`exp`) is statically known this function can be partially evaluated into `exp` multiplications of the `base` argument, significantly improving performance [**?**].

With compile-time views making `pow` partially evaluated requires adding only one annotation:

```
def pow(base: Double, exp: Int@ct): Double =
  if (exp == 0) 1 else base * pow(base, exp - 1)
```

To satisfy cross-stage persistence (§**??**) the `pow` must be `@ct`. This annotation is automatically added by ScalaCT as described in §**??**. In the example, the `ct` annotation on `exp` requires that the function must be called with a compile-time view of `Int`. ScalaCT ensures that the definiton of the `pow` function does not cause infinite recursion at compile-time by invoking the power function only when the value of the `ct` arguments is known.

The application of the function `pow` with a constant exponent produces:

```
pow(base, 4)
  ↪ base * base * base * base * 1
```

Constant 4 is promoted to `ct` by the implicit conversions (§**??**).

3.3 Variable Argument Functions

Variable argument functions appear in widely used languages like Java, C#, and Scala. Such arguments are typically passed in the function body inside of the data structure (e.g. `Seq[T]` in Scala). When applied with variable arguments the size of the data-structure is statically known and all operations on them can be partially evaluated. However, sometimes, the function is called with arguments of dynamic size. For example, function `min` that accepts multiple integers

```
def min(vs: Int*): Int = vs.tail.foldLeft(vs.head) {
  (min, el) => if (el < min) el else min
}
```

can be called either with statically known arguments (e.g., `min(1,2)`) or with dynamic arguments:

```
val values: Seq[Int] = ... // dynamic value
min(values: _*)
```

Ideally, we would be able to achieve partial evaluation if the arguments are of statically known size and avoid partial evaluation in case of dynamic arguments. To this end we translate the method `min` into a partially evaluated version and a dynamic version. The call to these methods is dispatched, at compile-time, by the `min` method which checks if arguments are statically known. Desugaring of `min` is shown in Figure **??**.

3.4 Removing Abstraction Overhead of Type-Classes

Type-classes are omnipresent in everyday programming as they allow abstraction over generic parameters (e.g., `Numeric` abstracts over numeric values). Unfortunately, type-classes introduce *dy-*

```
def min(vs: Int*): Int = macro
  if (isVarargs(vs)) q"min_CT(vs)"
  else q"min_D(vs)"

def min_CT(vs: Seq[Int]@ct): Int =
  vs.tail.foldLeft(vs.head) { (min, el) =>
    if (el < min) el else min
  }

def min_D(vs: Seq[Int]): Int =
  vs.tail.foldLeft(vs.head) {
    (min, el) => if (el < min) el else min
  }
```

Figure 2. Function `min` is desugared into a `min` macro that based on the binding time of the arguments dispatches to the partially evaluated version (`min_CT`) for statically known varargs or to the original min function for dynamic arguments `min_D`.

```
object Numeric {
  implicit def dnum: Numeric[Double]@ct =
    ct(DoubleNumeric)
  def zero[T](implicit num: Numeric[T]@ct): T =
    num.zero
}

trait Numeric[T] {
  def plus(x: T, y: T): T
  def times(x: T, y: T): T
  def zero: T
}

object DoubleNumeric extends Numeric[Double] {
  def plus(x: Double, y: Double): Double = x + y
  def times(x: Double, y: Double): Double = x * y
  def zero: Double = 0.0
}
```

Figure 3. Removing abstraction overheads of type classes.

namic dispatch on every call [**?**] and, thus, impose a performance penalty. Type-classes are in most of the cases statically known. Here we show how with ScalaCT we can remove all abstraction overheads of type classes.

In Scala, type classes are implemented with objects and implicit parameters [**?**]. In Figure **??**, we define a `trait Numeric` serves as an interface for all numeric types. Then we define a concrete implementation of `Numeric` for type `Double` (`DoubleNumeric`). The `DoubleNumeric` is than passed as an implicit argument `dnum` to all methods that use it (e.g., `zero`).

When `zero` is applied first the implicit argument (`dnum`) gets inlined due to the `ct` annotation of the return type, then the function `zero` gets inlined. Since `dnum` returns a compile-time view of `DoubleNumerc` the method `zero` on `dnum` is evaluated at compile time. The constant `0.0` is promoted to `ct` since `DoubleNumeric` is a compile time view. Finally the `ct(0.0)` result is coerced to a dynamic value by the signature of `Numeric.zero`. The compile-time execution is shown in the following snippet

```
Numeric.zero[Double]
  ↪ Numeric.zero[Double](DoubleNumeric)
  ↪ ct(DoubleNumeric).zero
  ↪ (ct(0.0): Double)
  ↪ 0.0
```

3.5 Inner Product of Vectors

Here we demonstrate how the introductory example (§**??**) is partially evaluated through staging. We start with the desugared `dot` function (i.e., all implicit operations are shown):

```
def dot[V](v1: Vector[V]@ct, v2: Vector[V]@ct)
  (implicit num: Numeric[V]@ct): V =
  (v1 zip v2).foldLeft(zero[V](num)) {
    case (prod, (cl, cr)) => prod + cl * cr
  }
```

Function `dot` is generic in the type of vector elements. This will reflect upon the staging annotations as well (`ct` and `static`). When we apply the `dot` function with static arguments we will get the vector with static elements back:

```
dot[Double@static](
  ct(Vector)(2.0, 4.0), ct(Vector)(1.0, 10.0))(
  Numeric.dnum)
↪
  (ct(Vector)(2.0, 4.0) zip ct(Vector)(1.0, 10.0))
    .foldLeft(ct(0.0)) {
      case (prod, (cl, cr)) => prod + cl * cr
    }
↪ (2.0 * 1.0 + 4.0 * 10.0): Double@static
```

When `dot` is evaluated with the `ct` elements the last step will further execute to a single compile-time value that can further be used in compile-time computations:

```
dot[Double@ct](
  ct(Vector)(ct(2.0), ct(4.0)),
  ct(Vector)(ct(1.0), ct(10.0)))(Numeric.dnum)
↪ ct(2.0) * ct(1.0) + ct(4.0) * ct(10.0)
↪ 42.0: Double@ct
```

4. Discussion

To distinguish terms executed at compile-time from terms executed at runtime with type annotations we have the following possibilities:

1. Annotate types of all terms that should be executed at runtime. Here all types analyzed LMS and realized that this is not an option.
2. Annotate types of terms that should be executed at runtime but introduce scopes (e.g., method bodies) for which this rule applies. In this way we would avoid annotating types of all runtime terms. This approach is taken by MacroML where macro functions are executed at compile time and quoted terms are executed at runtime. First approach is, also, a special case of this approach where there is a single scope for the whole language.
3. Annotate types of terms that are executed at compile time. This approach is used with ScalaCT and annotated types are called compile-time views.

To compare approach of ScalaCT with the first approach we analyzed 817 functions from the OptiML [**?**] DSL based on LMS. With the ScalaCT scheme OptiML would require more than 2x less annotations to implement.

Compared to the second approach our solution is simpler to comprehend and communicate. In the second approach there are two things that users need to understand when reasoning about staged programs: *i)* where does the compile time scope start, and *ii)* which terms are annotated. With ScalaCT the comprehension is simple: terms whose types are annotated with `ct` are executed at compile time.

It is also interesting to the second and third approaches. Here the number of annotations depends on the program. If the programs are mostly partially evaluated the second approach is better. These category of programs could also be regarded as code generators as most of the code is executed at compile time and produces large

outputs. When programs are comprised of mostly runtime values the approach of ScalaCT requires less annotations.

5. Evaluation

In this section we evaluate the amount of code that is obviated with ScalaCT compared to existing type directed staging systems (§**??**). Then we evaluate performance of ScalaCT compared to LMS and hand optimized code (§**??**)

5.1 Reduction in Code Duplication

Evaluating reduction of duplicated code (for reification and code generation) in type based staging systems is difficult as the factor varies from program to program. To avoid benchmark dependent results we instead calculate the lower bound on the duplication factor.

Given that we have a method on a type `T` whose body contains `n` lines of code (without the method definition). To introduce the same method on an annotated type `Rep[T]` we need another method for reification which has at least 1 line of code. Then we need code generation logic, which, if we use the same language should not have less lines than the original method plus at least one line for matching the reified method. For method of n lines we get a lower bound on the code duplication factor of:

$$2n + 3/n + 1$$

For single line methods ($n = 0$) the factor is 3 and for large methods ($n \to \infty$) it converges to 2.

5.2 Performance of Generated Code

In this section we compare performance of ScalaCT with LMS and original code. All benchmarks are executed on an Intel Core i7 processor (4960HQ) working frequency of 2.6 GHZ with 16GB of DDR3 with a working frequency of 1600 MHz. For all benchmarks we use Scala 2.11.5 and the HotSpot(TM) 64-Bit Server (24.51-b03) virtual machine. In all benchmarks the virtual machine is warmed up, no garbage collection happens, and all reported numbers are a mean of 5 measurements.

In Table **??** we show execution time normalized to original code for: *i)* `pow(42.0, 10)`, *ii)* `min(a, b, c, d, e)`, *iii)* inner product of two statically known vectors of size 50, and *iv)* the butterfly network of size 4 for fast Fourier transform `fft` (equivalent to code presented by Rompf and Odersky [**?**]). For all benchmarks the performance results are equivalent to LMS.

Table 3. Speedup of LMS and ScalaCT compared to the naive implementation of the algorithms.

Benchmark	LMS	ScalaCT
`pow`	221.75	221.70
`min`	1.82	1.79
`dot`	246.08	246.08
`fft`	12.14	12.88

6. Limitations

Type Annotations. Using type annotations for annotating the compilation stage is not ideal. Type annotations are not fully integrated into the Scala language. Major drawbacks are that overloading resolution and implicit search are oblivious about annotations. For example, if two methods have the same signatures but different staging annotations the compiler will report an error. Implicit search will fail in two ways: *i)* if two implicits with the same type are in scope but annotations differ the compiler will report and ambiguous implicit error and *ii)* if a method requires an implicit parameter with the `ct` annotation the compiler might provide an implicit argument without the annotation.

Type Annotation Position. Annotations in Scala can be used in many different positions and ScalaCT supports only some of them. Annotation `ct` can not be used in following positions: *i)* on classes, traits, and modules, *ii) in the list of inherited classes and traits*, *iii)* on the right hand side of the type variable definitions, and *iv)* on all terms outside the method definitions (constructors, constructor arguments, etc.).

Access Modifiers. Scala supports access modifiers of members. If methods that use ScalaCT internally access `private` members of the class ScalaCT will fail as all staged methods are inlined at the call site. Similar limitations exist with inlining functions in Scala. This problem could be circumvented inside Scala, however, the JVM will not allow this in the bytecode verification phase.

Code Explosion. Annotation `ct` inlines all functions that are annotated and unrolls all loops on compile-time executed data structures. This can, for a staged function, lead to unnaceptable code explosion for some inputs while the code can behave regularly for other inputs. For example, calling `pow(0.5, 10000000)` on the function from §**??** will make unacceptably large code while `pow(0.5, 10)` will work as expected.

7. Related Work

MetaOCaml [**? ?**] is a staging extension for OCaml. It uses quotation to determine the stage in which the term is executed. Types of quoted terms are annotated to assure cross-stage persistence. Staging in MetaOCaml starts at host language runtime and can not express compile-time computations. Further, operations on annotated types do not get automatically promoted to the adequate stage of computation as with compile-time views. Finally, there are no implicit conversions so all stage promotions of terms must be explicit.

MacroML [**?**] is a language that translates macros into MetaML staging executed at compile time to provide a "clean" solution for macros. In MacroML, within the `let mac` construct function parameters can be annotated as an early stage computation. These parameters can then be used in escaped terms, i.e., terms scheduled to execute at compile time. Unlike ScalaCT, MacroML uses escapes and early parameters to mark terms scheduled for to execute at compile time. Within escapes terms scheduled for runtime again need to be marked with brackets. This kind of dual annotations are not required as compile-time views are automatically promoted to runtime terms.

In LMS [**?**] terms that are annotated with `Rep` types will be executed at the stage after runtime compilation. Therefore, LMS can not directly be used for compile time computation. Furthermore, LMS requires additional reification logic and code generation for all `Rep` types.

Programming language Idris [**?**] introduces the `static` annotation on function parameters to achieve partial evaluation. Annotation `static` denotes that the term is statically known and that all operations on that term should be executed at compile-time. However, since `static` is placed on terms rather then types, it can mark only *whole terms* as static. This restricts the number of programs that can be expressed, e.g., we could not express that vectors in the signature of *dot* are static only in size. Finally, information about `static` terms can not be propagated through return types of functions so `static` in Idris is a partial evaluation construct, i.e., it hints that partial evaluation should be applied if function arguments are static.

Hybrid Partial Evaluation (HPE) [**?**] is a technique for partial evaluation that does not perform binding time analysis (similarly to online partial evaluators) but relies on the user provided annotation

`CT`[1]. HPE implementations exist for both Java and Scala [**?**]. Although, `CT` is used for partial evaluation, it does not affect typing of user programs. Furthermore, behavior of `CT` in context of generics is not described. ScalaCT can be seen as statically typed version of hybrid partial evaluation with support for parametric polymorphism. Due to the support for parametric polymorphism ScalaCT can express compile-time data structures with dynamic data.

Forge [**?**], by Sujeeth et al., uses a DSL to declare a specification of the libraries. Forge then generates both unannotated and annotated code based on the specification. Their language also supports generating staged code (comprised of terms different from multiple stages). Forge specification and code generation supports only a subset of Scala guided towards the Delite [**? ?**] framework.

The Yin-Yang framework, by Jovanovic et al. [**?**], solves the problem of code duplication by generating reification and code generation logic based on Scala code of existing types. With their approach there is no code duplication for the supported language features. However, not all of the Scala language is supported and all generated terms are generated for the next stage, thus, making a stage distinction is impossible.

8. Conclusion

ScalaCT is only the first step towards a staging system for Scala that is fully integrated in the language. For complete integration it would be necessery to allow usage of `ct` on trait and class definitions as well as abstract types.

In ScalaCT annotation of a term determins the stage where it is computed. When ScalaCT is used only for partial evaluation this can lead to unnecessary code duplication as users need to provide two versions of the same code (one staged and one unstaged). For a practical system it is necessary to extend ScalaCT with annotations which denote that terms should be partially evaluated if they are statically known. We have achieved this for variable argument funcitons in §**??** with an ad-hoc mechanism, however, we seek for a principled solution.

[1] Name `ct` in ScalaCT is inspired by hybrid partial evaluation.