# Annotating the Earlier Stage

## Succint Type-Driven Staging at Compile Time

Vojin Jovanovic
École Polytechnique Fédérale de Lausanne - EPFL
vojin.jovanovic@epfl.ch

Martin Odersky
École Polytechnique Fédérale de Lausanne - EPFL
martin.odersky@epfl.ch

## Abstract

***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 flavor of meta-programming 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 run-time of runtime generated code, etc. Different stages of compilation can be executed in the same language [20, 29] or in different languages [4, 11].

Notable, staging systems for statically typed languages are MetaOCaml [5, 29] and LMS [23]. These systems were successfully applied as a *partial evaluatior* [15]: for removing abstraction overheads in high-level programs [6, 23], for domain-specific languages [7, 16, 28], and for converting language interpreters into compilers [12, 25]. Staging originates from research on two-level [9, 20] and multi-level [10] 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[V:Numeric](v1: Vector[V], v2: Vector[V]): V =
  (v1 zip v2).foldLeft(zero[V]) {
    case (prod, (cl, cr)) => prod + cl * cr
  }
```



In function `dot`, if the vector sizes are invariant during program runtime 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 framework that the operations on values of vector components should be executed in the next stage (after run-time compilation). The compilation stage in which a term will be executed is determined by *code quotation* (in MetaOCaml) or by specific parametric types `Rep` (in LMS). In LMS we mark that the vector size is statically known by annotating only vector elements with a `Rep` type[2]:

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

Here the `Rep` annotations on `V` denote that elements of vectors will be known only in the next stage (after run-time compilation). At this stage the `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 the unquoted/unannotated code is always executed during host-language runtime and quoted/type-annotated code is executed after run-time compilation.

**First Problem.** How can we use staging for programs whose values are statically known at the host language compile-time (the first stage)? All staging frameworks treat unannotated terms as runtime values of the host language and annotated terms as values of in stages of compilation. Even if we would start staging one stage earlier (at the host language compile-time), we would have to annotate all run-time values. Annotating all values is cumbersome since host language run-time values comprise the majority of user programs (c.f., §5).

Programming languages Idris [3] and D [1] try to solve this problem by allowing the `static` annotation on function arguments. 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 ex-

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

[2] In this work we use LMS as it is the only staging framework in Scala.

pressed, 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 deterministically propagated through return types of functions so `static` in Idris and D is a partial evaluation construct.

**Second Problem.** Staging systems based on type annotations (e.g., LMS and type-directed partial evaluation [8]) inherently require duplication of code as, a priory, no operations are defined on `Rep` annotated types. For example, in the `dot` function all numerical types (e.g., `Rep[Int]`, `Rep[Double]`, etc.) must be re-implemented in order to typecheck the programs and achieve code generation for the next stage.

Suereth et al. [26] and Jovanovic et al. [17] 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.

The main idea of this paper is that *annotated types* should denote computations that happen during the *previous stage* of compilation instead of the next stage of compilation. The reason is two-fold: *i)* annotating code of previous stages succinctly express compile-time execution and *ii)* in staged programs the static terms appear less frequently than run-time terms, and in order to bear minimum overhead for the users, it is better to add annotation overhead to static terms.

We treat annotated types as a *compile-time view*s of existing data types and therefore no code duplication is necessary. Compile-time views make all operations on types executed in the host language compile time. Existing types can be promoted to their compile-time duals with the `@ct` annotation at the type level, and with the `ct` function on the term level. Consequently, due to the integration with the type system, the control over staging is fine-grained and polymorphic, thus, term level promotions obviate code duplication for static data structures.

With compile-time views, to require that vectors `v1` and `v2` are static and to partially evaluate the function, a programmer would need 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, static value, or a compile-time value 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

// static terms are internally tracked through types
dot(Vector(2.0, 4.0), Vector(1.0, 10.0))
  ↪ (2.0 * 1.0 + 4.0 * 10.0): Double@static

// ct promotes static terms to compile-time
dot(Vector(ct(2), ct(4)),
    Vector(ct(1), ct(10)))
  ↪ 42: Double@ct
```

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

- By introducing compile-time views (§2) as means to: *i)* succinctly achieve staging at host language compile-time and to *ii)* avoid code duplication in type based staging systems.
- By demonstrating the usefulness of compile-time views in four case studies (§3): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [14, 22, 30].
- By introducing the $F_{i<:}$ calculus (§**??**) that in a fine-grained way captures the user's intent about partial evaluation. The calculus is based on $F_{<:}$ with lazy records which makes it suitable for representing modern multi-paradigm languages with object oriented features. Finally, we formally define evaluation rules for $F_{i<:}$.
- By providing a *translation scheme* from data types in object oriented languages (polymorphic classes and methods) into their dual compile-time views in the $F_{i<:}$ calculus (§**??**).

We have implemented and open-sourced[3] a staging extension for Scala ScalaCT. ScalaCT has a minimal interface (§2) based on type annotations. We have evaluated performance gains and the validity of the partial evaluator on all case studies (§3) and compared them to LMS. In all benchmarks our evaluator gives significant performance gains compared to original programs and performs equivalently to LMS. ScalaCT is implemented according to the formal translation scheme (§**??**) from object oriented features of Scala to the $F_{i<:}$ calculus.

## 2. Compile-Time Views in Scala

In the section we informally present ScalaCT, a staging extension for Scala based on the compile-time views. ScalaCT is a compiler plugin that executes in a phase after the Scala type checker. The plugin takes as input pre-typed Scala programs and uses type annotations [21] 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 2) with annotations `ct` and `inline`, and functions `ct` and `inline`.

**Annotation** `ct` is used at the type level (e.g., `Int@ct`) and denotes a compile-time view of a type. The annotation

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

```
package object scalact {

  final class ct extends StaticAnnotation
  final class inline extends StaticAnnotation

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

}
```

**Figure 1.** Interface of the ScalaCT.

is integrated in the Scala's type system and, therefore, can be arbitrarily nested in different variants of types. Table 1 shows how the `@ct` annotation can be placed on types and how it, due to the translation to the compile-time views (Figure **??**), changes method signatures on annotated types.

In Table 1, `Int@ct` is a non-polymorphic type and therefore according to the translation to the compile-time view (Figure **??**) parameters of all methods will also be compile-time views. On the other hand, `Vector[Int]@ct` will have 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 parts executed at compile-time. The return type of the function `map` can still be both dynamic and a compile-time view: due to the type parameter `U`.

**Annotation inline** can be used only at the term level on statically known methods and functions. It denotes that the method/function will be inlined during compilation time. In other words, `inline` is marking that the function application is a compile-time computation and that application should be removed by partial evaluation. This is not the first time that inlining is achieved through partial evaluation [18].

Internally `inline` can be expressed in terms of the `ct` annotation. A method

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

will have an internal method type

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

that can not be written by the users. We choose the name `inline` to be consistent with the existing Scala `inline` annotation.

**Functions `ct` and `inline`** are used at the term level for promoting Scala objects and methods/functions into their compile-time views. Without the the `ct` and `inline` we would not be able to instantiate compile-time views of types. Table 2 shows how different types of terms are promoted to their compile-time views.

Function `ct` can be applied to objects (e.g., `Vector`) to provide a compile-time view over their methods. When those objects have generic parameters, `ct` be used to promote the arguments, and thus, the result types of these functions. When applied, on functions `ct` promotes the compile-time view as well as its arguments and the return type.

Function `inline` can be applied on functions/methods to promote only the function/method to their compile time views without promoting the arguments. This function can be seen as a shallow version of `ct` that makes only the outer type a compile-time view.

### 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 [19] and untyped [13] languages. We use similar techniques (described in §**??**), however, unlike in partial evaluation we do not evaluate static terms at compile time. They are tracked for verifying correctness and providing convenient implict 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 Least Upper Bounds

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 verification of validty of function calls and helps computing the least upper bounds of terms. For example, validity of function calls:

```
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 least upper bound rules do not hold. The binding-time of control flow constructs does not depend only on return type of the body but also the conditional []. 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` is a function for computing least upper bounds, and `c`,

**Table 1.** Types and corresponding method signatures after translation to their compile-time views.

| 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`<br>`length: Int@ct` |
| `Vector[Int@ct]@ct` | `map[U](f: (Int@ct => U)@ct): Vector[U]@ct`<br>`length: Int@ct` |
| `Map[Int@ct, Int]@ct` | `get(key: Int@ct): Option[Int]@ct` |

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

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

`thn`, `elz` are respectively binding times of the condition, the then branch, and the else branch.

### 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* [29, 31], imposes that all operations on compile-time views must known at compile time.

To satisfy cross-stage persistence ScalaCT verifies that composite dynamic types (e.g., polymorphic-types, function types, record types, etc.) are not composed of compile-time views. The intuition is that all method parameters (including `this`) of compile time views must either be a compile-time view or them selves type variables. In the following example, we show malformed types and examples of terms that are inconsistent with causality

```
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 should print the head of the dynamically known list at compile time. In the second example the statement should print the result of `fn` at compile time but the body of the function is unknown.

The `inline` annotation promotes only function/method bodies to compile-time views. In effect, this requires only the method/function body to be known at compile time. Method bodies are statically known in objects and classes with final methods, thus, the `inline` annotation is only applicable on such methods.

### 2.4 Implicit Conversions

If method parameters require compile-time views of a type the corresponding arguments in method application would always have to be promoted to `ct`. In practice this is not convenient as it requires an inconveniently large number of annotations. Staging is commonly used for optimization of libraries, and as such, it should not affect user code - users should not be aware of the internal operation of the library.

To address this issue we introduce implicit conversions from `static` terms to `ct` terms. The conversions support translation of language literals, direct class constructor calls with static arguments, and static method calls with static arguments into their compile-time views. Since our compile-time evaluator does not use Asai's [2, 27] method to keep track of the value of each static term, we dissalow implicit conversions of terms with static variables.

For example, for a factorial function

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

we will not require annotations on literals `0`, and `1`. Furthermore, the function can be invoked without promoting the literal `5` into it's compile-time view:

```
fact(5)
  ↪ 120
```

## 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 and the interesting applications. We start by inlining a simple function with staging (§3.1), then do the canonical staging example of the power function (§3.2), then we demonstrate how variable argument functions can be desugared into the core functionality (§3.3). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-class related abstraction an be removed (§3.5). For formal partial evaluation rules refer c.f. §**??**.

### 3.1 Inlining Expressed Through Staging

Function inlining can be expressed as staged computation [18]. Inlining is achieved when a statically known function body is applied with symbolic arguments. In ScalaCT we use the `inline` annotation on functions and methods to achieve inlining:

```
@inline 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)
```

When the exponent (`exp`) is statically known this function can be partially evaluated into `exp` multiplications of the `base` argument, significantly improving performance [5].

With compile-time views making`pow` partially evaluated requires adding two annotations:

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

To satisfy cross-stage persistence (§2.3) the `pow` must be `@inline`. However, to make reduce the number of required annotations we implicitly add the `inline` annotation when at least one parameter or the result type is marked as `ct`. 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 will produce:

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

Constant 4 is promoted to `ct` by the implicit conversions (§2.4).

### 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
}
```

```
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`.

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 2.

### 3.4 Removing Abstraction Overhead of Type-Classes

Type-classes are omnipresent in everyday programming as they provide allow abstraction over generic parameters (e.g., `Numeric` abstracts over numeric values). Unfortunately, type-classes introduce *dynamic dispatch* on every call [24] and are, 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 [22]. In Figure 3, 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 `inline` annotation, then the function zero gets inlined. Since `dnum` returns the compile-time view of `DoubleNumerc` the method `zero` is evaluated at compile time. The constant `0.0` is promoted to `ct` since `DoubleNumeric` is a compile time view (formally defined in §**??**). Finally the `ct(0.0)` result is coerced to a dynamic

```
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.

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 (§1) is partially evaluated through staging. We start with the desugared `dot` function (i.e., all implicit operations are shown):

```
@inline 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. Evaluation

### 4.1 Reduction of Code Duplication

### 4.2 Performance Comparison

**Table 3.** Performance comparison with LMS and hand optimized code.

| Benchmark | Hand Optimized | LMS | Scala Inline |
|---|---|---|---|
| pow | | | |
| min | | | |
| dot | | | |
| fft | | | |

## 5. Discussion

@ct vs @rep

## 6. Limitations

- Interaction with type variables.
- Type variables.
- Type annotations and overloading and implicit search.
- Can not inherit from a compile time view.

## 7. Related Work

## 8. Conclusion

## References


[1] The d programming language. `http://dlang.org/`.

[2] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–51, 2002.

[3] Edwin C Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming (ICFP)*, 2010.

[4] K. J Brown, A. K Sujeeth, H. J Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[5] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering*, 2003.

[6] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *Generative Programming and Component Engineering (GPCE)*, 2005.

[7] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. *Domain-Specific Program Generation*, 2004.

[8] Olivier Danvy. *Type-directed partial evaluation*. Springer, 1999.

[9] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Symposium on Logic in Computer Science (LICS)*, 1996.

[10] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

[11] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[12] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[13] Carsten K Gomard and Neil D Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(01):21–69, 1991.

[14] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transansactions on Programming Langanguages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[15] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[16] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.

[17] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, Koch C., and M Odersky. Yin-Yang: Concealing the deep embedding of DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2014.

[18] Stefan Monnier and Zhong Shao. Inlining as staged computation. *Journal of Functional Programming*, 13(03):647–676, 2003.

[19] F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed &lgr;-calculus. In *Symposium on Principles of Programming Languages (POPL)*, 1988.

[20] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*, volume 34. Cambridge University Press, 2005.

[21] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

[22] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

[23] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

[24] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanović, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Symposium on Principles of Programming Languages (POPL)*, 2013.

[25] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun, and Martin Odersky. Project Lancet: Surgical precision JIT compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[26] Arvind K Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2013.

[27] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2-3):101–142, 2001.

[28] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation (DSPG)*. 2004.

[29] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.

[30] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1989.

[31] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2010.