ScalaCT: Type-Directed Staging at Compile-Time

Abstract

Staging systems decide the compilation stage in which a term is executed based on: quotation (e.g., MetaOCaml), or types (e.g., LMS). Type based staging systems, require type annotations of all future stage terms, as well as implementing reification and code generation logic for all future 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 require logic for reification and code generation. Number of annotations is especially noticeable in languages with local type inference, such as Scala, as method parameter types must be provided by the programmer.

We introduce a type based staging system for Scala where terms whose types are annotated are executed in the earlier stage of compilation, in our case, at host language compile-time. Annotated types represent merely compile-time views of original types and therefore no reification and code generation logic is necessary. 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 [20, 29] or in different languages [4, 11]. 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 for statically typed languages are MetaOCaml [5, 29] and LMS [23]. These systems were successfully applied as a *partial evaluator* [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[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 the 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 runtime 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 start staging 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 (§5).

TODO MetaML

Annotating the Next Stage. 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 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. [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.

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 (§5). 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 annotate types as 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
```

[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.

Since, vector elements are polymorphic the result of the function can be a dynamic 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
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 views
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 succinctly achieve type safe two-stage programming starting from host language compile time.
- 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 (§3): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [14, 22, 30].

We have implemented a staging extension for Scala ScalaCT[3]. ScalaCT has a minimal interface (§2) based on type annotations. We have evaluated performance gains and the validity of ScalaCT on all case studies (§3) and compared them to LMS. In all benchmarks (§4) 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 the 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 [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 a single annotation `ct` and a single function `ct`.

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

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

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

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

Figure 1. Interface of the ScalaCT.

Since all operations on compile-time views are executed at compile time, non-generic method parameters and result types are also compile-time views. Table 1 shows how the `@ct` annotation can be placed on types and how it affects method signatures on annotated types.

In Table 1, on `Int@ct` parameters and result types of all methods are 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` can still be both dynamic and 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:

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

This is not the first time that inlining is achieved through partial evaluation [18].

Annotated methods will have an annotated method type

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

which can not be written by the users.

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 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 methods have generic parameters, `ct` can be used to promote the arguments, and thus, the result types of these methods.

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

```scala
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

```scala
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:

```scala
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(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.

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

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

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`

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 (::@ct)(1, Nil)`	`: (::[Int])@ct`
`new (::@ct)(ct(1), ct(Nil))`	`: (::[Int@ct])@ct`
`ct((x: Int@ct) => x)`	`: (Int@ct => Int@ct)@ct`
`ct((x: Int) => x)`	`: (Int => Int)@ct`

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 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 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 some libraries this could require an inconveniently large number of annotations.

To minimize the number of required annotations 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): 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
```

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:

```
@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)
```

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 `@ct`. However, to reduce the number of required annotations we implicitly add the `ct` annotation when at least one parameter type 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
}
```

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.

```
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.

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 `ct` annotation of the return type, then the function `zero` gets inlined. Since `dnum` returns a 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. 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 (§1) 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. Evaluation

4.1 Reduction of Code Duplication

4.2 Performance Comparison

5. Discussion

@ct vs @rep count them

6. Limitations

- Interaction with type variables.
- Type variables.
- Type annotations and overloading and implicit search.

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

Benchmark	Hand Optimized	LMS	Scala Inline
`pow`			
`min`			
`dot`			
`fft`			

- Can not inherit from a compile time view.

7. Related Work

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

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.