Staging for Generic Programming in Space and Time

Georg Ofenbeck  
Department of Computer Science  
ETH Zurich  
Switzerland  
ofenbeck@inf.ethz.ch

Tiark Rompf  
Department of Computer Science  
Purdue University  
USA  
tiark@purdue.edu

Markus Püschel  
Department of Computer Science  
ETH Zurich  
Switzerland  
pueschel@inf.ethz.ch

Abstract

Metaprogramming is among the most promising candidates to solve the abstraction vs performance trade-off that plagues software engineering through specialization. Metaprogramming has been used to enable low-overhead generic programming for a long time, with C++ templates being one of the most prominent examples. But often a single, fixed pattern of specialization is not enough, and more flexibility is needed. Hence, this paper seeks to apply generic programming techniques to challenges in metaprogramming, in particular to abstract over the execution stage of individual program expressions. We thus extend the scope of generic programming into the dimension of time. The resulting notion of stage polymorphism enables novel abstractions in the design of program generators, which we develop and explore in this paper. We present one possible implementation, in Scala using the lightweight modular staging (LMS) framework, and apply it to two important case studies: convolution on images and the fast Fourier transform (FFT).

CCS Concepts · Software and its engineering → Polymorphism; Source code generation;

Keywords · staging, polymorphism, generic programming, high performance program generation, FFT

ACM Reference Format:
https://doi.org/10.1145/3136040.3136060

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPCE’17, October 23–24, 2017, Vancouver, Canada  
© 2017 Association for Computing Machinery.  
ACM ISBN 978-1-4503-5524-7/17/10...

1 Introduction

Generic programming [25] embodies the idea of parameterizing algorithms over aspects such as data representations or subcomputations. A generic sorting algorithm, e.g., might abstract over the underlying data type, access model, comparison operator, and whether to sort in ascending or descending order. Generic programming is widely used, and supported in all common programming languages through language facilities (e.g., type classes in Haskell) and libraries. Benefits include reduced code duplication and better software scalability or maintainability. In many cases, however, abstraction techniques unfortunately come at the price of performance due to runtime overheads or because the abstractions hide optimization opportunities from the compiler.

Staging for generic programming in space. A potential remedy for the abstraction / performance trade-off is specialization, which can be achieved in a general way via metaprogramming techniques such as macros, templates, or runtime code generation. In all such cases, the generic computation is divided into (at least) two stages: the runtime (or target) stage and the meta stage. Hence, we speak of a staging transformation [19]. The basic idea is to eliminate the abstraction overhead through computation in the meta stage, and execute only specialized code in the target stage.

Example: Fast Fourier transform. Specialization typically consists of precomputation of values and simplification of the code based on the parameters known at metatime. The effect is both improved runtime and smaller code size. A good case study is the recursive fast Fourier transform (FFT). If the input size $n$ is known, the recursion strategy (for which

### Table 1. Stage polymorphism, i.e., generic programming in time, enables fully automatic generation of fixed-size and general-size libraries with specialized base cases from a single high-level specification. Here, we compare our system (SpiralS) to state-of-the-art FFT implementations.

<table>
<thead>
<tr>
<th></th>
<th>JTransform</th>
<th>FFTW</th>
<th>Spiral</th>
<th>SpiralS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project LOC</td>
<td>80 K</td>
<td>50 K</td>
<td>1 M</td>
<td>~0.2 K core logic + ~1.2 K full</td>
</tr>
<tr>
<td>Library</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infrastructure</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>generation</td>
<td>fixed + library</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
there are degrees of freedom) can be fixed at metatime and thus all recursive calls can be inlined to yield one monolithic function. Further, the needed constants (called twiddle factors), which depend on \( n \), can be precomputed and included. Combined with abstractions to capture different data formats and different unrolling and scalar replacement strategies, one can build an FFT generator for fixed sizes. But what if \( n \) is not known? In this case, the code has to be recursive and needs facilities to control the freedom in recursion, twiddle factors may be computed once and reused or computed on the fly, and a suite of optimized base cases for small sizes is needed. The latter are completely unrolled to enable further simplifications and alleviate the overhead from recursing all the way [11]. In summary, the impact of knowing \( n \) or not on the resulting code is profound. The question we ask in this paper is what mechanisms are needed in such situations to support both cases in one generative framework to enable "generic programming in time."

**Generic programming in time.** Classical staging and metaprogramming facilities fix the execution stage for each piece of code, and hence, only support generic programming in space. While there are always ways of also making the stage (i.e., the time of computation) generic, e.g., in C++ by combining preprocessor statements with template metaprogramming, the goal of this paper is to provide a more principled solution to generic programming in time, without requiring external custom tool-chains, and using only standard generic programming mechanisms to provide true generic programming in time.

We present a solution, implemented in Scala using the lightweight modular staging (LMS) framework [32]. We chose LMS because it encodes staging through types, i.e., operations to be performed in the meta stage only differ in a marker type that signals the compiler to redirect the expression to the staging framework. This is achieved using standard techniques such as operator overloading. These marker types are regular Scala types; therefore all already existing mechanisms for generic programming on types can be exploited to abstract over whether a given expression is staged or not, and hence achieve stage polymorphism.

**Contributions.** In this paper we present a principled approach that adds stage polymorphism to the generic programming tool set. In particular:
- We give an overview of selected performance transformations that can be described through stage polymorphism, such as tiling, precomputation (Section 3.2) and inlining (Section 3.3)
- We compose genericity in time with genericity in space (Section 3.4) to enable scalarization of data structures (Section 3.5).
- We introduce a mechanism that enables the automated encapsulation and exposure of target-time constructs from within meta-time containers using isomorphisms encoded as type classes (Section 3.6) and we describe how this helps with loop tiling and in generating function definitions with flexible signatures (Section 3.7).
- We illustrate how target-stage checks can transform target information into meta information in a form of bounded static variation (Section 3.8) that is crucial for specializing divide-and-conquer algorithms (Section 3.9).
- We combine these mechanisms to specialize recursive function call graphs ahead of time, similar to procedure cloning [7], but under full control of the programmer (Section 3.10).
- We demonstrate all introduced techniques on a running example and two case studies (Section 4): convolution in image processing and the FFT.

We note that all the above was implemented as a natural extension of existing generic programming patterns. In [28], we already used the idea of abstraction over the execution stage, but only as a means to control the style of the generated code (e.g., unrolled or scalarized) and support different data layouts for fixed-size FFTs. In this paper, we fully develop the idea in a more general context. In particular this allows abstraction in situations were the resulting code is fundamentally altered as in the case of fixed size vs general size FFTs mentioned before. This paper is the first to generate both from a single code base.

The FFT case study (Section 4.2) is particularly significant, as it is a well-studied application of program generation. Table 1 compares our implementation, called SpiralS, to state-of-the-art systems, which range from fully handmade kernels (JTransforms: [46]), over generated codelets, i.e., fixed-size kernels (FFTW: [11, 12]), to generated kernels plus library infrastructure (Spiral: [29, 30]). Our system SpiralS is the first to generate both fixed-size and general-size libraries with specialized base cases from a single high-level implementation, which takes under 200 lines of code for the core algorithm, and just about 1200 lines total—an order of magnitude reduction in complexity and code size compared to state-of-the-art systems, even if one takes into account that systems like Spiral provide large amounts of additional functionality along orthogonal dimensions, e.g., support for classes of linear transforms other than the FFT, or generation of FPGA implementations [9], which render a literal LOC comparison meaningless.

All code presented in this paper, including the case studies, is available at [27]. We emphasize that we consider in particular the FFT case study a key part of our contribution.

## 2 Background

We present a brief introduction to relevant generic- and metaprogramming techniques in Scala.

**Type parameterization and abstract type members.**
Scala supports type parameterization on classes and methods [1, 26]. The syntax to define both is:

```scala
class GenericClass[T](constructor_param: T) {
  val member: T = constructor_param
  def method[X](param: X): X = param
}
```
Here, T is the generic type parameter to the class and X is the type parameter to the method in this example. The syntax at the instantiation site is

```scala
val inst = new GenericClass3 // inferred [Int]
val inst_explicit = new GenericClass[Long](3)
```
Alternatively the same behavior can be achieved through the use of abstract type members:

```scala
abstract class GenericClass2 {
  type T   // abstract type member
  val member: T   // abstract value member
  def method[X](param: X): X = param }
```
which can be instantiated through refinement:

```scala
val inst2 = new GenericClass2 {
  type T = Int
  val member: T = 3 }
```

We use both versions in this paper as they offer different trade-offs in terms of code verbosity.

**Type classes and implicits.** First introduced in Haskell, type classes [45] enable a natively typed flavor of *ad-hoc* polymorphism. Take, for example a generic method that doubles a given number:

```scala
def generic_method[T: Numeric](p: T) = p + p
```
This method is generic in the type T, but imposes the restriction that it must be a numeric type. The benefit of type classes over inheritance is that we can retroactively add functionality to existing data types. For example, we can provide a type class implementation of Numeric for type `Pair[Int,Int]`. However, since `Pair` is a generic class, we could not make it inherit from a numeric base type.

Scala implements type classes using normal classes and objects, but with the help of *implicits*. A possible interface definition for `Numeric` could look like this:

```scala
trait Numeric[T] {  
  def interface_plus(l: T, r: T): T  
  class Ops(lhs: T) {    
    def +(rhs: T) = interface_plus(lhs, rhs)  
  }
}
```
Here we defined a class that specifies the required operations on the generic type T. In addition, we define a class `Ops` for syntactic convenience. As a second component, we provide an implementation of `Numeric` for every numeric type:

```scala
implicit object IntIsNumeric extends Numeric[Int] {    
  def interface_plus(l: Int, r: Int): Int = l + r }

implicit object LongIsNumeric extends Numeric[Long] {    
  def interface_plus(l: Long, r: Long): Long = l + r }
```
These objects serve as evidence that the types `Int` and `Long` are in fact numeric types. Note that we have used the `implicit` keyword, which means that the compiler will automatically insert a reference to these objects whenever an implicit parameter of type `Numeric[Int]` or `Numeric[Long]` is required in a method application. The third component is an implicit conversion method, which will wrap a numeric value in the corresponding `Ops` class:

```scala
implicit def toOps[T](x: T)(implicit num: Numeric[T]): num.Ops = new num.Ops(x)
```

With these facilities in place, we can explain how the original generic method is desugared by the Scala compiler:

```scala
def generic_method[T: Numeric](p: T) = p + p   // original

def generic_method_desugar[T](p: T) // desugared
  (implicit ev: Numeric[T]) = toOps(p)(ev).+(p)
```
Multiple implicits are used to enable the infix syntax of `p + p`. Note that with this design one can retroactively add behavior to a class, which, from a user syntax point of view, looks like the addition of new methods. We use this design pattern extensively within the work presented in this paper.

**Lightweight modular staging.** LMS [32] is a multi-stage programming approach implemented in Scala. LMS allows a programmer to specify parts of the program to be delayed to a later stage of execution. Compared to related approaches based on syntactic distinctions like quasiquotes [24], LMS uses only types to distinguish between present-stage and future-stage operations. For example, the operation

```scala
val (a,b): (Int,Int) = (3,4)
val c: Int = a + b
```
will execute the operation, while the staged equivalent

```scala
val (a,b): (Rep[Int],Rep[Int]) = (3,4)
val c: Rep[Int] = a + b
```
uses the higher-kindred type `Rep[_]` as a marker type to redirect the compiler to use an alternative plus implementation

```scala
def infix_+(lhs: Rep[Int], rhs: Rep[Int]): Rep[Int]
```
Instead of executing the arithmetic operation directly, the staged variant will create a symbolic representation of the plus operation and return a symbolic identifier as the result. The details of this symbolic representation, its management and final unparsing are not relevant for this paper. We focus on describing the polymorphism between regular code and its staged counterpart, which relies solely on the ability to describe polymorphism over T and `Rep[T]`. The approach utilized therefore is runtime meta-programming: Scala code is compiled normally, and when the compiled code is run, which we call meta-time, it generates specialized code. This generated code is then compiled and executed offline, which we call target-time. Code generation at runtime provides capabilities that are not immediately achieved at compile time. Examples include specializing code based on data that becomes only available at meta-time (e.g., data read from a file on a user’s machine), or generating code for a different language such as C for performance.

### 3 Stage Polymorphism

In this section we demonstrate how to extend generic programming to incorporate the dimension of time using LMS-style type-driven staging. We first present a running example and properties we would like to abstract. The following subsections will then alternate between introducing a concept
used to achieve these abstractions and its application on the running example.

### 3.1 Running Example and Abstraction Goals

Our running example program does not perform a practically relevant computation but is designed to combine patterns that occur in many numeric algorithms of practical relevance, namely divide-and-conquer algorithms over arrays. The program first scales each array element with an external given value and then multiplies it by a call to a trigonometric function whose arguments depend on the position and array size. Afterwards, each value is added to its neighbour (in essence a low-pass filter). Finally, it splits the array into two halves and recurses on them until a base case is reached.

```scala
def recurse(a: Array[Double], s: Double): Array[Double] = 
  if (a.length < 2) a 
  else {
    val scaled = scale(a,s)
    val sum = sum(scaled)
    val (l, r) = split(sum)
    recurse(l, s) ++ recurse(r, s) 
  }

def split(a: Array[Double]) = a.splitAt(a.length / 2)

def scale(a: Array[Double], s: Double) = 
  a.zipWithIndex map { (ele, idx) =>
  ele * Math.sin((idx + a.length) % 10) * s }

def sum(a: Array[Double]) = 
  (0 until a.length - 1).foldLeft(Array.empty[Double]) {
  (acc, i) => acc += (a(i) + a(i + 1)) 
  }
```

**Abstraction goals.** For this simple algorithm we want to derive a staged, generic implementation that abstracts over multiple aspects of the code related to when values are computed and how the code is specialized. In particular:

- **The input scaling factor** $s$ is provided as a single value. If the value is known at meta-time we want to specialize for it. Alternatively, if only known at target-time, perform a runtime check on the value and depending on the outcome potentially invoke a specialized version.
- **The input array** $a$ should either be passed as in the sketched code above or alternatively be provided as a list of scalar variables and return the result in the same fashion. The latter version could, e.g., be required for targeting hardware descriptions and potentially require the input and output lists to scale to arbitrary arities.
- **The array size** reduces with each divide and conquer step converging towards the base case. The design goal is to perform a runtime check on the array size and if it is smaller then a threshold, invoke a version of the code that is specialized to the problem size. This size-specialized version should recurse through other size-specialized versions towards the base case.
- **The array data layout.** Specializing on the array size as mentioned above should also scalarize the array, if it is not already already passed as a list of scalars.
- **The loops.** For the scaling loop, we want to perform unrolling whenever the array size is known statically. This is the case if the input and output pair are lists of scalars or if we are within one of the size-specialized variants within the recursion. For the sum loop we want to employ tiling.
- **The trigonometric function.** Whenever applicable, we want to precompute the value of this function.
- **The recursion.** We want to inline all functions, with the exception of the recursive function itself, which we only want to inline if the size is statically known and inlining therefore terminates.

In the remainder of the section we will alternate between the concepts needed to achieve these goals and their application to our example.

### 3.2 Concept: Abstracting over Precomputation

**Meta vs runtime computation.** As introduced in Section 2, LMS uses types to distinguish between meta and target stage computation. To perform a sine function at the target stage the code would take the form

```scala
def sin_target(x: Rep[Double]): Rep[Double] = ...
```

To instead perform the computation at the meta stage, and only at the end move the result to the target stage, we simply leave out the staging annotation type for the input.

```scala
def sin_meta(x: Double): Double = Math.sin(x)
```

LMS provides an implicit conversion method `Const`, which can be used to move a primitive value from the meta to the target stage. In particular, this conversion enables us to call `sin_target` with either a meta or a target value. But calling the meta version with a target value will yield a type error:

```
val (s,c): (Rep[Double],Double) = (...,2.0)
```

```scala
sin_target(s)
sin_target(c) // ok, implicit conversion
```

```scala
sin_meta(s) // error: expected Double, not Rep[Double]
```

This simple example already demonstrates how the type checking prevents us from mixing meta and target phases in invalid combinations.

**Abstraction over precomputation.** The example above duplicates code between the two implementations of the sine. But since staging annotations are regular Scala types, we can exploit all mechanisms for generic programming on types to abstract over the choice of `Rep[Double]` and `Double`. We can formulate the stage-polyorphic sine function as:

```scala
def sin.polym[R[_]: IRep]@_1: IRep[T: R[Double]]: R[Double]
```

where we use a type class `IRep` defined as

```scala
trait IRep[R[_]] { def sin(lhs: R[Double]): R[Double] } 
```

and two evidence objects defined as

```scala
type NR[T] = T
implicit object isRep extends IRep[Rep] { // target
  def sin(x: Rep[Double]): Rep[Double] = ...
}
implicit object isNoRep extends IRep[NR] { // meta
  def sin(x: NR[Double]): NR[Double] = Math.sin(x)
}
```
Staging for Generic Programming in Space and Time

Note that we use exactly the same type class design pattern as introduced in Section 2, with only one minor tweak: Instead of regular parametric types we use higher-kind types, since we are abstracting over a type constructor that is applied to base types. We exploit the fact that we can define a higher-kind identity type NR[T] = T to describe the meta phase, therefore inlining meta functions by default, yielding at the target phase

```
val x = s + s; x + x
```

If we want to create a function definition at the target level, we can use an operator `fundef`:

```
val target = fundef(metaf) // metaf from above
```

The signature of `fundef` is

```
```

i.e., the return value is again a meta-level function which, when called, will generate a call to the generated function definition. Hence, functions are not `Rep` values themselves, and we have use the Scala type system to ensure that all generated functions are first order and all call targets are known. Yet, target functions remain first-class values at the meta level.

This way of defining `fundef` with an identity signature enables automatic stage polymorphism: we can decide whether to invoke `fundef` based on any dynamic condition

```
val maybeInline = if (shouldInline) metaf else fundef(metaf)
```

A key use case in our running example will be the specialization of recursion patterns, as discussed later in Section 3.10.

### 3.4 Concept: Combining Axes of Polymorphism

Stage polymorphism uses the same mechanisms as those to achieve regular data abstraction. Hence, combining stage polymorphism with existing generic programming design patterns is straightforward. We implement an abstract data container

```
abstract class AC[R[_]: IRep,T] {
    // AC is short
    // for AbstractContainer
    def apply(i: R[Int]): T
    def update(i: R[Int], y: T)
    def length(): R[Int]
}
```

with two instantiations

```
class StagedArray(val data: Rep[Array[Double]]) extends AC[Rep,Rep[Double]] {
    def apply(i: Int) = data(i)
    def update(i: Int, y: Double)
    def length(): Int
}
```

Calling `loop_poly` with a meta value for the range `l`, the fold will actually be executed at meta time, therefore resulting in fully unrolled target code. On the other hand, if the function is called with a target value, a staged loop will be created. Of course we could also factor out this functionality into a stage-polymorphic version of `foldLeft`.

**Inlining**. Function invocations at the meta level are invisible to the target level, e.g.,

```
val s: Rep[Int] = ...
val metaf: Rep[Int] => Rep[Int] = (in: Rep[Int]) => in + in
```

will be executed during the meta phase, therefore `fundef` is called with a target value, a staged loop will be created. Of course we could also factor out this functionality into a stage-polymorphic version of `foldLeft`.

Calling `loop_poly` with a meta value for the range `l`, the fold will actually be executed at meta time, therefore resulting in fully unrolled target code. On the other hand, if the function is called with a target value, a staged loop will be created. Of course we could also factor out this functionality into a stage-polymorphic version of `foldLeft`.

```
val t = sin(((i + a.length()) % 10).toDouble())
```

Given that we have exactly the same type class design pattern as introduced in Section 2, with only one minor tweak: instead of regular parametric types we use higher-kind types, since we are abstracting over a type constructor that is applied to base types. We exploit the fact that we can define a higher-kind identity type `NR[T] = T` to describe the meta level, therefore inlining meta functions by default, yielding at the target level

```
val x = s + s; x + x
```

If we want to create a function definition at the target level, we can use an operator `fundef`:

```
val target = fundef(metaf) // metaf from above
```

The signature of `fundef` is

```
```

i.e., the return value is again a meta-level function which, when called, will generate a call to the generated function definition. Hence, functions are not `Rep` values themselves, and we have use the Scala type system to ensure that all generated functions are first order and all call targets are known. Yet, target functions remain first-class values at the meta level.

This way of defining `fundef` with an identity signature enables automatic stage polymorphism: we can decide whether to invoke `fundef` based on any dynamic condition

```
val maybeInline = if (shouldInline) metaf else fundef(metaf)
```

A key use case in our running example will be the specialization of recursion patterns, as discussed later in Section 3.10.
3.5 Application

We reformulate our implementation in terms of the abstract container, e.g., the function header `loop_poly` in Section 3.3 takes the form

```scala
def loop_poly[R[_]](a: AC, s: Rep[Double])(0 until a.length) ...
```

Note that we got rid of one parameter and instead use the length function of the abstract container. Given a meta array of scalars this will execute the loop at meta time and pass meta-time values to the sine function, thus enabling unrolling and precomputation if possible.

**Scalarization.** Using this data abstraction we can formulate scalarization as a conversion operation within the abstract container as

```scala
abstract class AC[R[_]: IRep,T] {
  def scalarize(size: Int): MetaArrayofScalars }
```

with the implementation in the `StagedArray` subclass

```scala
def scalarize(size: Int): MetaArrayofScalars = {
  val scalars = new MetaArrayofScalars{
    new Array[Rep[Double]](size)
  }
  for (i <- 0 until size) scalars(i) = data(Const(i))
  scalars
}
```

and a simple identity function in the other subclass. Note that the typing enforces the intuition that scalarization is bound to an array size known at meta time.

3.6 Concept: Isomorphism between Meta and Target

We have shown in Section 3.3 how LMS supports staging of functions of the form `Rep[A] => Rep[R]`. The `fundef` operator is also implemented for multi-parameter functions such as `(Rep[A],Rep[B]) => Rep[R]`, which is transformed to `Rep[(A,B) => R]` (up to 24 parameters). While working with this construct we hit a nuisance. Assume a meta container such as:

```scala
case class Complex(re: Rep[Double], im: Rep[Double])
```

This container type enables us to reason about complex numbers on the meta stage, without carrying the overhead to the target stage, as only the contained doubles will be visible at this point. Unfortunately, whenever we use this construct in the context of staging functions, additional boilerplate is required as function staging is only defined over `Rep` types. Therefore boilerplate code for composing and decomposing each meta container used is required. To remedy the situation we use another type class to describe types that are isomorphic to a list of `Rep` values:

```scala
trait ExposeRep[T] {
  def fresh: Vector[Rep[_]]
  val vec2t: Vector[Rep[_]] => T   // compose
  val t2vec: T => Vector[Rep[_]]   // decompose
}
```

This isomorphism captures the process of composing and decomposing meta containers. In addition, it carries an interface to create new symbols for all expressions it describes in Figure 1. Combining the expose type class with regular data abstraction. For encapsulating and exposing meta objects the type class redirects to the corresponding implementation via the abstract base class.

In the process. This is used, e.g., at the creation of lambdas to create symbolic inputs as entry point to the function. Given this construct we can now define a more general `fundef` operator with signature

```scala
```

which only requires that the argument and return types of the given function are isomorphic to a list of target types. Usage of this construct takes the form

```scala
val complex_val: Complex = ...
def complex_f(c: Complex): Complex = ...
// assume implicit ExposeRep[Complex] in context
val staged_complex_f = fundef(complex_f)
val result: Complex = staged_complex_f(complex_val)
```

For the price of providing an expose type class, the user is able to seamlessly work with staged functions over meta containers. Note that for non-encapsulated target types the expose type class is provided implicitly. The described technique is a crucial enabler for the ideas presented in the rest of this section.

3.7 Application

Dynamic meta values that are lifted to the target stage become static values within it. E.g., the value of `i` in

```scala
(0 until 10).foldLeft(0)((acc, i) => acc + Const(i))
```

does not require a boilerplate code for composition and decomposing each meta container used is required. To remedy the situation we use another type class to describe types that are isomorphic to a list of `Rep` values:

```scala
trait ExposeRep[T] {
  def fresh: Vector[Rep[_]]
  val vec2t: Vector[Rep[_]] => T   // compose
  val t2vec: T => Vector[Rep[_]]   // decompose
}
```

This isomorphism captures the process of composing and decomposing meta containers. In addition, it carries an interface to create new symbols for all expressions it describes in

The dynamic reusability of parameters lists at meta time. For our running example this is an essential requirement. We have been able to abstract over arrays vs scalars in Section 3.4, yet the function

```scala
Array[Rep[Double]] => Array[Rep[Double]]
```

would not have been expressible so far.
We combine the abstraction from Section 3.4 and the expose mechanism from Section 3.6 as sketched in Figure 1. We define an expose type class for the abstract container and resolve the composition and decomposition within by referring to member functions of the AC class. Note that we are required to provide the expose mechanism with a sample. This is due to the fact that, e.g., the length of a list of target time scalars is only known during meta time. This approach allows for easy composition of meta containers. For example, to describe the expose mechanism of a vector of complex numbers one can use the previously defined exposeRep[Complex] type class for each sub-element.

Using this technique also allows us to describe staged functions in general as only $A=R$ and encode multiple parameters and/or return types in the expose type class of $A$ or $R$. This eliminates the need to encode all function arities in the staging framework. For arities that are not supported by the target language as, e.g., multiple return values or above 22 parameters in the case of Scala, a corresponding transformation has to be done before unparsing. In the concrete case of Scala code we simply transform into a right-leaning nested tuple, which allows us to preserve all type information.

**Tiling.** Loop tiling is a common program transformation that is done to expose optimization opportunities for performance and change the access pattern of a loop. The following code expresses tiling in combination with stage polymorphism.

```scala
def tiling[R[_]: IRep] (a: Rep[Array[Int]], tiling: R[Int]): Rep[Int] = {
  val tiled: Rep[Int] = a.length / toRep(tiling)
  (0 until tiled).foldLeft(0) { (acc, idx) =>
    (0 until tiling).foldLeft(acc) { (acc2, idx_poly) =>
      val idx = idx * tiling + idx_poly
      acc2 + a(idx) + poly_sin(idx_poly)
    }
  }
}
```

The first loop executes until tiled, a staged value due to its dependency on the size of the array. This implies that the loop also can only be performed at target time. The tiling factor is stage polymorphic in this example. Given a meta-time tiling factor, the inner loop will be fully unrolled at target time. Furthermore we call a stage-polymorphic sine function within the loop. Given that the inner loop index is known at meta time, the sine can actually be precomputed ahead of time. This yields opportunities for algebraic simplifications. If we supply a target time tiling factor to the function, the inner loop will be visible at target stage and the sine function computed at target time. Providing a meta-time tiling factor unrolls the loop, precomputes values and potentially saves operations, ideal in a compute-bound situation. On the other hand, providing a target time tiling factor will not unroll the loop, saving code size and compute sine values on the fly, ideal for a memory-bound situation. We apply the same technique in a straightforward fashion to the running example.

**3.8 Concept: Information flow from target to meta**

All examples we have seen so far only allow information to flow from the meta stage to the target stage and not vice-versa. Indeed, it is not immediately apparent how such a "time travel" could be achieved: a $\text{Rep}[T]$ value cannot, in general, become a $T$ value without executing the generated code. However, we can achieve something similar in an indirect way, based on an observation in certain high performance libraries and JIT compilers. Many divide-and-conquer high performance libraries that work on a dynamic problem size perform a runtime check on the current problem size during each recursion step. If the checked value is smaller than a threshold they invoke a size-specialized variant (note that usually this is not the algorithmic base case of the recursion). In a similar fashion, JIT compilers observe runtime values provided to a function and if a value tends to be constant, they might create a specialized version of that function under guards. A similar patterns is known as binding-time improvement in partial evaluation under the name *bounded static variation*, or simply "The Trick" [10, 18].

**3.9 Application**

Inspired by this observation we can pre-initialize our running example algorithm with a function of the form:

```scala
def ini_specialize[R[_]: IRep, S[_]: IRep] (a: AC[R], s: S[Double]) = {
  if (s == 1.0) recurse[R,NoRep](a,1.0)
  else recurse(a,s)
}
```

If $s$ happened to be a meta-time value in the first place, the conditional will be resolved at meta time and no runtime overhead for checking will occur. If it is a target value, the conditional will be lifted into the target stage as well and two versions of the divide-and-conquer algorithm will be generated.

**3.10 Concept: Specializing Recursion Schemes**

We combine the previously introduced concepts to give the recurse function from the running example its final form:

```scala
def implicit val (expDyn, exppret): 
  (ExposeRep[Dyn[R]], ExposeRep[AC[R]])=(expDyn(stat),...)
def fun(dyn: Dyn[R]): AC[R] = {
  val mix = Mix[R](stat, dyn)
}
```


```java
mix.scaling = .... } // body
val function_signature = stat.genSig()
if (stat.inline) fun else fundef(fun,sig) }
```

Instead of passing multiple parameters, we rely on a single input meta container and single meta output container as introduced in Section 3.6. Their corresponding expose type classes are defined in the first line of the function. We formulate the actual computation within the body of an inner function fun. This function is optionally staged in the last line using the construct from Section 3.3. The snippet above we utilize the meta containers Stat[R], Dyn[R] and Mix[R]. They are in an inheritance relationship with a common super class

```java
class Base[R[_]: IRep](a: AC[R], scaling: ST[Double])
```

We use the three subclasses to give a view on either only the meta-time components, only the target-time components or their combination. This separation proved useful to avoid errors that can occur when mixing meta- and target-time components incorrectly. Within the body of the function we combine the Stat[R] and Dyn[R] aspect to create Mix[R], which can be safely used within. To allow for a dynamically sized arity within the recursion, we use the target-time static aspects encoded in Stat[R] to create the corresponding expose type class for Dyn[R] from it. As we are in a recursive function, we need to provide LMS with the means of detecting that we are in such a context. This is required as it will otherwise try to unfold the recursion during meta time. The separation into meta- and target-time components assists us in this task, as in the pattern above, the target-time layout is purely defined by meta-time values defined within Stat[R]. This is utilized in the second to last line, where we create a signature of the function based on the meta-time values used to create it. If we encountered the signature already we withhold the creation of a new function at target time and instead pass the exiting one. This full construct allows the recurse function to call itself, changing its function signature on the fly, thereby specializing itself. E.g., to scalarization within the recursion would take the following form within the body of the function

```java
val sc = a.scalarize(size) // scalarize
val (nstat, ndyn) = mix.cpy(a = sc, inline=true).split()
val rf = recurse(nstat) // create or lookup the function
val result_scalars = rf(ndyn) //call the function
```

where we create new Stat[R] and Dyn[R] objects. These are used to request a corresponding function from within recurse. In the case that the function signature defined by Stat[R] has been seen already, an existing generated target function will be returned.

---

3.11 Application

**Size specialization under a threshold.** In our example we wish to specialize the recursion on the size, once it is smaller than a given threshold. Ideally we want this to take the form of (pseudocode)

```java
if (size.isRep && size < threshold) size match {
  case n => recurse(size: NR[Int]=n, inline=true)
  case (n-1) => recurse(size: NR[Int]=n-1, inline=true)
  ...
}
```

We want to perform the check only on target time values, and, if the check succeeds, call a scalarized size-specialized version. Implementing this on the running example takes the form

```java
def sizecheck[R[_] : IRep](stat: Stat[R]):
  Dyn[R] => AC[R] = {
    def fun(dyn: Dyn[R]): AC[R] = {
      val mix = Mix[R](stat, dyn) // only check if target value
      if (ev.isRep && a.length() < size_threshold) binsearch(mix, toRep(a.length()), 0, 64)
      else ... // call regular recurse
    }
    val function_signature = stat.genSig()
    if (stat.inline) fun else fundef(fun,sig) }
```

Binsearch is a size check done in a binary-search-style fashion to minimize the cost of the comparisons. Note that binsearch and its recursive calls will be inlined within sizecheck at target time.

3.12 The Final Generic Implementation

Figure 3 gives a high-level overview over the connection of all components introduced within this section that form the final generic implementation of our algorithm fulfilling all requirements given in 3.1. The input can be given as an arbitrary sized list of scalars or as an array. The enabling ideas are given in Sections 3.7 and 3.4. In addition, a scaling factor is supplied that can be optionally a target-time constant (Section 3.2). Adopting the function header at target time on any given parameter list variation is done with the technique in Section 3.7. We perform a runtime check if the scaling factor is not constant already (Section 3.8. We enter the recursion loop in checksize, which, given a dynamically
2) A program generator that produces optimized code, two approaches are common:
1) An optimized library that supports a generic convolution taken with coefficients specified by the filter) of its eight neighbours and itself. Our generic implementation could serve as both a user-facing library or as a program generator.

Code specification. Our generic implementation abstracts over various optimizations that specialize the code at both code generation time and run time. At code generation time the user can specify the following shape defining properties of the algorithm:

- Block size. It is common practice to perform operations over the image in a blocked fashion as this yields better cache utilization.
- Unrolling factor. Each block is further tiled where the sub-loop is fully unrolled. This avoids loading the same data multiple times from the input as, e.g., for a full $3 \times 3$ filter each pixel of the original image is touched nine times.
- Symmetry of the filter matrix. The user may specify common symmetries (symmetric, antisymmetric) within the filter matrix, which reduce the operations count.
- Constant values of filter elements. Some or all of the filter values can be known at code generation time, which enables specialization and possibly simplification (if the values include 0, 1, -1, or duplicates).
- Decomposability of the filter. If the convolution is separable it can be split into two one-dimensional filters, possibly increasing locality. If the filter is not known at meta time, these are passed to the function.

In addition to the above choices that enable optimizations at code generation time, the implementation includes a runtime check for the following properties.

- Check if filter values are zero. For filter values specified at runtime, we can check if they are to zero and, if so, invoke a specialized version of the algorithm to reduce operations.
- Check if the filter has symmetry. For filter values that are not known at meta time, the symmetry will be automatically checked and exploited.
- Check if the filter is separable. If the filter is not known at meta time, the user can choose to check separability to invoke a specialized version.

Following Section 3.12 the generic filter implementation takes the form shown in Fig. 4. It differs from Fig. 3 in that it only optionally uses a recursion in the case of utilized runtime checks. The main computation is done without recursion, composed of tiling and the convolution core. Tiling is influenced by the meta time choice of tiling and unrolling factor and is not shown. We discuss the convolution core and the runtime checks in the following.
Exploiting symmetries in the convolution core. Symmetries in the filter matrix enable a reduction of the operations count. To achieve this, we add all input values that get scaled by the same value from the matrix; then we apply the now unique scalings to each corresponding such sum.

Given that we know the symmetry patterns statically within a function we can describe the reduction as follows:

```scala
// valuesym: Map[Int,Vector[Int]] is given as meta value
val symsum = valuesym.map { p =>
  val (scaleval,inputpos) = p
  inputpos.map(_ => getPixels()).reduce(_ => sum(_))
}
// summed all input values that use the same scale
val scaled = symsum.map(p => p._1 * p._2)
// and finally reduce across scales
val sumtotal = scaled.foldLeft(Pixel(0))(_ + _)
```

Note that this implementation automatically covers filter-matrix elements known at meta time as they will seamlessly combine with target-time values. For the special case of zero values, we rely on smart constructors within LMS to optimize the arithmetic operations during code construction.

Runtime specialization. It is worth noting that all previous optimizations are in principle also possible in vanilla LMS as they are done at code generation time and do not extend over lambdas. Runtime specialization, with ahead-of-time creation of the specialized cases, on the other hand, is only possible with the technique described in Section 3.10. We show a code snippet that performs the runtime check on each matrix element and specializes for zero values. Within an initialization function `convolution_ini` we perform specialization conditionally on a flag.

```scala
if (specialize && specialize_count < nr_entries)
  checkandreplace(specialize_count, mix, 0)
else
  convolution_core(stat)(dyn) // actual computation
```

The specialization takes the form:

```scala
def checkreplace(pos: Int, mix: Mix, check: Int) = {
  val inc_count = pmix.copy(spezialize_count += 1)
  if (entry(pos) == check) {
    // position becomes static
    val new_mix = pmix.copy(spezialize_count += 1)
    case 0 => inc_count.cpy(a = 0)/set matrix(0,0) 0
    case 1 => inc_count.cpy(b = 0)/set matrix(0,1) 0
  }
  else {
    // position stays dynamic
    val (old_stat, old_dyn) = inc_count.split()
    convolution_ini(old_stat)(old_dyn) }
}
```

During unparsing this will yield functions branching in a tree fashion during each element specialization as depicted in Fig. 5. Each leaf of the tree is a specialized version of the code. Since there are 9 filter elements, there are \(2^9\) code variants, i.e., the overall code size becomes rather large. Doing so, we effectively trade code size with the time it would take to invoke a code generator (including a JIT) at runtime to produce a specialized variant.

4.2 Fast Fourier Transform (FFT)

The FFT is a particular challenging algorithm to optimize as a number of complex transformations are needed. Prior generative work include FFTW [11, 12] and Spiral [29, 30]. FFTW generates the needed base cases (called codelets) for small sizes inside a hand-written general size library. Spiral can generate either code that is specialized to the input or a general-size library [44]. However, both cases use different generation pipelines because the differences in the generated code are much more profound than in the convolution example as explained already in the introduction.

We show that using our abstraction this genericity in time is also achievable for the FFT, resulting in a single unified generation pipeline (Table 1) with the code available at [27].

**Background: Recursive FFT.** A minimal recursive FFT implementation takes the form (pseudocode)

```scala
def fft(n: Int, in: Array[Double], out: Array[Double]) =
  if (n == 2) fft2(in,out) else {
    val k = choose_factor(n)
    for (i <- 0 until k) fft(k,in,out)
    for (i <- 0 until n) out(i) = twiddle(i) * out(i)
    for (i <- 0 until n/k) fft_strided(k,out,stride(n))
}
```

\(^2\)We use the same skeleton as used in Section 3.10, e.g., Mix is the combined meta and target info
Here, \( n \) and \( \text{out} \) specify the input and output stride at which the data is read and written, respectively. If the input size \( n \) is known at meta time, many specializations become possible including fixing the recursion strategy (choice of \( k \) at each step), precomputing the twiddle factors, unrolling and inlining the occurring small FFTs, which in turn enables further simplifications. The result is a sequence of nested loops. If \( n \) is not known, fast code is fundamentally different. The recursion stays generic and thus needs search at runtime, twiddles are precomputed once at runtime, or one the fly during computation, and to benefit from fast basic blocks, an entire suite of unrolled codelets (all small sizes and for both variants \( \text{fft} \) and \( \text{fft} \_\text{strided} \)) needs to be included. Vectorization and parallelization further complicates the requirements.

An example divide-and-conquer breakdown of an FFT is given in Fig. 7, either determined at meta time (if \( n = 1024 \) is known) or dynamically obtained by recursing within a general-size library. As said above, in both, the recursion is not followed until \( n = 2 \) but instead a specialized unrolled FFT of a larger size (called codelet) is called once the size is below a threshold (here: 64). This codelet is also computed recursively, but with the recursion unrolled and inlined.

**Code specification.** Our generic implementation abstracts over various optimizations that specialize the code at meta time or runtime. Most important is the abstraction related to the input size due to the deep impact on the other optimizations and the resulting code (see above). The resulting code has about 200 LOC, and about 1200 LOC when including all class definitions.

- **Input size** which is known at meta or target time, causing deep consequences for the optimizations below.
- **Codelet size** specifies the threshold below which FFT sizes should be unrolled.

Figure 8 shows a full target-time call graph for a DFT accepting general size input, and utilizing fully specialized base cases for sizes up to 16. We highlighted the separation into infrastructure code (grey boxes), where the input size is unknown, and the calls into size specialized codelets (white boxes). Inspecting the graph, one can see that each base case exists multiple times, with varying parameter sets. This is the Cartesian product of the parameter sets and the input sizes we want to have base cases for. The call graph is exactly equal to the base case generation of the original Spiral system, but without invoking a second generator. This example is a strong motivation for a generative approach, as the number of required functions scales with the product of possible statically known parameters (e.g., number of static sizes for the base cases times the possible static input strides).

**Runtime comparison.** Our generic implementation outputs Java code. Fig. 8 compares its performance to JTransforms [46], an optimized Java library for the FFT on an Intel Core i7-4770K and the JVM Java HotSpot 64-Bit Server VM.
Related Work

Generic programming. One of the earliest mentions of the term “generic programming” as design methodology is by Musser and Stepanov [25]. A nice overview on the adoption of the concepts can be found in [13]. A similar review that is more recent can be found here [3]. Popular instantiations of data-type generic programming are “Data types à la carte” [42] and “Scrap your Boilerplate” (SYB) [22, 23].

Metaprogramming. One of the early well-known libraries that utilizes metaprogramming is Boost [38]. It utilizes template meta programming in C++, a technique that can be very challenging to utilize. Concepts [15] try to fix many of these challenges imposed including more compiler checks. More principled support for metaprogramming support is found across many other languages, such as Template Meta-Programming for Haskell [37], MetaOCaml [20], MetaML [43] and Macros for Scala [5], to name a few. Most of these languages or systems (with the exception of C++ templates) provide a version of syntactic annotations, brackets, escape, and (sometimes) run, which together provide a syntactic quasi-quotation facility similar to that found in Lisp but often extended with some static scoping and typing guarantees.

Staging based on types. Another line of metaprogramming approaches is based on embedded DSLs, leveraging the type system of the host language to distinguish meta-level from target-level terms. Lightweight Modular Staging (LMS) [32] is one instance of this approach. Immediately related work includes that of [6] and [17]. LMS draws inspiration from earlier work such as TaskGraph [2], a C++ framework for program generation and optimization.

Combining generic programming and metaprogramming. Staging and metaprogramming have been used in many ways to reduce the overhead of generic programming abstractions. The first explicit treatment, an implementation of SYB based on MetaOCaml, was presented by Yallop [47]. Earlier metaprogramming techniques that were inspired by generic programming approaches include polytypic staging [39] and isomorphic specialization [40], as well as work on combining deep and shallow embeddings of DSLs [41]. All of these were inspirational for our work.

Pre-computation and function specialization. Procedure cloning as an optimization step within a compiler was proposed by [8]. With the rise of just in time (JIT) compilers over the last decade, runtime specialization through JIT’s has become mainstream. Recent research [34] proposes value specialization in the context of a JavaScript JIT. In its Section 5, this work gives a nice overview over various code specialization flavors including static variants.

Stage polymorphism. The idea to explicitly abstract over staging decisions in a controlled and fine-grained way was first introduced in our previous work [28] which focuses on the mapping of a DSL based generator into Scala. This paper overlaps in that we also sketch the idea of abstracting over precomputation shown in Section 3.2 and combining it with standard generic programming shown in Section 3.4. We extend both techniques by the concepts shown within Sections 3.6 to 3.8 and restated them such that they compose with the extensions. The previous work was only capable of producing fixed size code, similar to e.g., FFTW. The new concepts not only allow us to produce general size libraries similar to e.g., [44], but also enable us to provide a fixed and a general size FFT generator through stage polymorphism.

Partial evaluation. Partial evaluation [18] is a program specialization technique that automatically splits programs into static/meta and dynamic/target computations. Notable systems include DyC [14] for C, JSpec/Tempo [35], the JSC Java Supercompiler [21], and Civet [36] for Java. Lancet [33] is a partial evaluator for Java bytecode built on top of LMS. Bounded static variation (“The Trick”) is discussed in the book by Jones, Gomard, and Sestoft [18], and has been related to Eta-expansion by Danvy et al. [10].

Partial evaluation and stage polymorphism. In one sense, a partial evaluator treats source expressions as polymorphic in their binding time. Notably the work by [16] explores polyvariancy in the context of partial evaluation. But experience suggests that it is not easy to generate exactly the desired specialization with fully automatic approaches, or to debug the outcome if something goes wrong. We view our approach to stage polymorphism as a promising middle ground between automatic partial evaluation and fully manual staging, which retains the benefit of code reuse, but makes the specialization fully programmable.

Conclusion

This paper presents one possible design of generic programming that abstracts over temporal aspects of code generation. The approach allows the composition of statically specialized and unspecialized code generation, even across function boundaries, within a single abstract generator. The presented techniques therefore enable a drastic reduction in code size for program generators. One application domain is the generation of high performance code as we demonstrated with the first generator that produces both fixed and general-size FFTs in a single pipeline.

Acknowledgments

This research was supported under NSF awards 1553471 and 1564207, and DOE award DE-SC0018050.
References


