KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Alexander Slesarenko

Scalan: polytypic library for nested parallelism in Scala

Moscow
2011

Alexander Slesarenko
Scalan: polytypic library for nested parallelism in Scala.

The paper presents a specialized Scala library to express nested data parallelism in Scala language using its advanced features for polytypic and datatype-generic programming, DSL embedding and type-level programming. The proposed library implements polytypic (type-indexed), non-parametric representations of parallel arrays. The approach is illustrated by a series of increasingly sophisticated examples followed by a description of key implementation details.

Александр Слесаренко
Scalan: политиповая библиотека на языке Scala для вложенного параллелизма

В работе представлена библиотека на языке Scala для описания вложенного параллелизма. Используются возможности языка Scala для обобщенного программирования, построения встроенных проблемно-ориентированных языков (DSL), а также программирования на уровне типов. В предложенной библиотеке используется непараметрическое представление параллельных массивов. Подход иллюстрируется серией примеров и сопровождается описанием ключевых деталей реализации.

Contents

1 Introduction

Programming is hard and parallel programming is an order of magnitude harder. Every computer these days is, in fact, a parallel computer and we have to find a way to cope with this complexity. One particular approach to parallel programming is to "go functional way" since pure functional languages are by-default safe for parallel evaluation, whereas imperative languages are by-default unsafe.

A particularly promising and well-studied approach to employing large numbers of processors (100-s and 1000-s) is data parallelism in general and *nested data parallelism* in particular. Blelloch's seminal work on NESL [3] showed that nested data parallelism can be efficiently implemented by mapping it to flat parallelism using flattening transformation [1]. It is possible to combine flexible and high-level programming model (nested data parallelism) with fast, scalable and implementation specific execution model (flat data parallelism). Recent work [16] showed that this approach can be extended to support higher-order functional languages such as Haskell.

Programmers of parallel computer encounter nested parallelism whenever they write a routine that performs parallel operations, and in turn want to call that routine in parallel. This occurs naturally in many applications [2]. Most data-parallel programming environments [9, 22, 8], however, do not permit the expression of nested parallelism. This forces the programmer to exploit only one level of parallelism or to create its own implementation of nested parallelism. Both of these alternatives tend to produce code that is harder to maintain and less modular than code described at a higher-level with nested parallel constructs. Not permitting the expression of nested parallelism is analogous to not permitting nested loops in serial languages.

In this paper we describe *Scalan* - a polytypic [14] library for Scala, which embodies nested data parallelism in a modern, general-purpose language [24]. In contrast to [16] our approach is to rely solely on the existing language features while implementing nested parallelism in the Scala language. The library can be regarded as an embedded domain specific language (DSL) with the purely functional semantic.

This paper makes the following main contributions.

1. We present Scalan library for expressing nested parallelism in Scala code, for that we use series of examples.

2. We show internal non-parametric representation of parallel arrays and describe how that representation is constructed depending on type of array element thus specifying parallel arrays as *type-indexed types* [10].

3. Following [6] we present Scala encodings of parallel arrays as type-indexed types by employing advanced generic programming techniques available in Scala [7].

4. We present a reference implementation (sequential) of our approach and a series of examples that illustrate a range of nested parallelism scenarios.

5. We also show how nested parallelism can be used to encode lambda calculus by implementing a simple interpreter. To our knowledge we are the first to describe this application of nested parallelism.

The purely-functional interface of Scalan comes in the form of collective operations such as maps, filters, permutations etc., that emphasise an algorithm's high-level structure.

The reference implementation described in this work employs the standard Scala arrays and in fact is sequential (not parallel)[1] Thus, its performance is not so good to say the least. First of all, it aims to show the usability of Scala language for encoding nested parallel algorithms and underlying non-parametric representations. Second, we are not considering any parallel speedups or benchmarking as it is a subject of future development.

In this paper we assume familiarity with the Scala language [24].

The rest of the paper is organized as follows. We describe why the library is called polytypic. Then we present Scalan as a DSL which is polymorphically embedded in the Scala language. Than we give a number of examples showing how Scalan library (DSL) can be used to express nested data-parallel algorithms. Then we describe a non-parametric type-indexed representations of parallel arrays. Next we show how the advanced Scala language features can be used to implement these representations. And finally, in Section 7, we conclude by considering some limitations of our approach and possible lines of further research.

2 Why polytypic?

Polytypism [15] differs from both parametric polymorphism and ad-hoc polymorphism (overloading). A definition of polymorphic function such as

```
def head[A](a: List[A]): A
```

can be seen as a family of functions one for each instance of `A` as a monomorphic type. There need only be one definition of `head`; the typechecker ensures that values of type `A` are never used. A polymorphic function can be implemented *parametrically* - as a single function that works on boxed values.

An ad-hoc polymorphic function, which can be idiomaticaly encoded in Scala like this (employing Typeclass of Concept pattern [7])

```
def +[A:Numeric](a:A, b:A): A // we use scala.math.Numeric[T]
```

is also a family of functions, one for each instance of the Numeric type resolved by the compiler and provided as an evidence for implicit parameter.

A polytypic function on the other hand is a function like this

[1] Source code is available at `http://github.com/scalan`.

```
def length[A,D[_]](d: D[A]): Int
```

where `D` ranges over type constructors [15]. We can apply `length` to instances of `List[A]` or `Tree[A]` for some kind of tree.

The polymorphism of a polytypic function is somewhere between parametric and ad-hoc polymorphism. A single definition of `length` suffices, but `length` has different instances in different contexts depending on type `D`. A polytypic function can be implemented parametricaly, but it need not be parametric and can also be implemented non-parametricaly without boxing values.

We are going to use this property in our library to represent instances of `PA[A]` type (see Fig. 2) non-paramerically using unboxed arrays along with an implementation of related polytypic methods and functions.

General approach to encoding of polytypic (or generic) functions in Scala is given in [25]. Inspired by these ideas, we implement polytypism in Scalan library but in a different, more specialized way.

3 Scalan as embedded DSL

It is well known that a domain specific language (DSL) can be embedded in an appropriate host language (see for example [12]). When embedding a DSL in a rich host language, the embedded DSL (EDSL) can reuse the syntax of the host language, its module system, typechecking(inference), existing libraries, its tool chain, and so on. This hinges on the power and the flexibility of the host language.

In *pure embedding* the domain types are directly implemented as host language types, and domain operations are modeled as host language functions on these types. This approach is similar to the development of a traditional library, which also exports types and functions to its clients, but DSL approach stresses the domain-specific concepts and operations during the design and implementation of the library.

The DSL approach is highly amenable to formal methods. The key point that one can reason directly *withing the domain semantics*, rather than withing the semantics of the programming language. Because the domain operations are defined in terms of the domain semantics, rather than the syntax of the DSL, this approach automatically yields compositional semantics with its well-known advantages, such as easier and modular reasoning about programs and improved composability.

However, while the semantics is in accordance with the laws (algebraic) of the domain, the approach based on pure embedding cannot utilize them for optimization purposes due to tight coupling of the host language and the embedded one. It also restricts the EDSL to a single opaque interpretation, which is not amenable to analysis or optimizations. Partial evaluation and multi-staged computation have been proposed [13], but domain-specific optimizations or other kinds of analyses are still not possible with pure embedding approach.

Recently, *polymorphic embedding* – a generalization of Hudak's approach – was proposed [11] to support multiple interpretations by complementing the functional abstraction mechanism with an object-oriented one. This approach introduces the main advantage of an external DSL, while maintaining the strengths of the embedded approach: compositionality and integration with the existing language. In this framework, optimizations and analyses are just special interpretations of the DSL program.

Taking advantage of the polymorphic embedding approach we have designed Scalan in a similar way. The ultimate goal is to expose Scalan as polymorphically embedded DSL in the Scala language. We are going to benefit from transformational nature of nested parallelism by providing specialized domain specific interpretations along with analisis and optimizations.

3.1 Getting started with Scalan

The first step to use Scalan is to make its definitions available in the program by inheriting them from **trait** `Scalan` defined in **package** `scalan.dsl`, as it is shown in Fig. 1. This makes all abstract definitions of Scalan DSL available inside `MySample` trait. According to the idea of polymorphic embedding, at this point, we don't impose any specific implementation details and use Scalan in an abstract way (type of methods `fromArray, map, toArray` given in Fig. 2 and 3).

Figure 1 Hello Scalan sample

```
package scalan.samples
import scalan.dsl._

trait HelloScalan extends Scalan {
  def hello(names: Array[String]) = {
    fromArray(names) map {name ⇒ "Hello, " + name + "!"} toArray
  }
}

import scalan.sequential._
object Sample extends HelloScalan with ScalanSequential {}
```

Later on, to construct a particular implementation (in this case with sequential semantics) and to run our code we need to import corresponding implementation package and create object by using *mixin composition* of our abstract `HelloScalan` trait with particular Scalan implementation (`ScalanSequential` in this case).

We can run this sample in Scala console like this:

```
scala> import scalan.samples._
import scalan.samples._
scala> Sample.hello(Array("Alex", "Ilya"))
res1: Array[java.lang.String] = Array(Hello, Alex!, Hello, Ilya!)
```

3.2 Expressing parallelism with types

There is a lot of different ways to express parallelism. Depending on the programming model, one can use some implementation of actors, parallel collections, task parallel library or other specialized primitives.

In NESL the only way to say that something should happen in parallel was to use parallel array datatypes and *array comprehensions* [3], as the language construct to express parallel operations over collections of data. This is analogous to *list comprehensions* found in many programming languages.

In Scalan we follow this *type directed* approach.

1. We use datatypes to express parallelism.
2. We let the programmer to express parallel algorithms on the highest possible level, without actually introducing a parallelism in the language. (In fact Scalan can be viewed as a language with purely sequential semantic.)
3. We do it in a compositional way so that the parallel function once defined can be used in the definition of the others (in a nested way).

This can have a lot of benefits especially in polymorphic embedding settings. Take as an example type declaration from Fig. 6.

```
type Vector = PA[Float]
```

Here we declare `Vector` to be a *parallel array* type by using the type constructor `PA[A]` which is inherited as abstract type from **trait** `Scalan` and defined like this:

```
type PA[A] <: PArray[A] // parallel array of A
```

We will elaborate on `PA` type constructor later in this section. Using `Vector` type in the definition of `dotProduct` makes it a *parallel function*. This is the only way to express parallelism in Scalan and we will refer to Scala definitions that use parallel datatypes as *parallel functions*.

3.3 Type constructor of parallel arrays: PA[A]

In Scalan we support ideas of NESL and overall nested parallelism by:

1. introducing a type constructor `PA[A]` of parallel arrays (see.Fig 2) to express parallelism;
2. defining a set of polytypic [14] operations (generic array combinators) shown in Fig. 2 and 3. They can be applyed to any datatype `PA[A]` provided that the type `A` is constructed according to the rules in Fig. 4. This way we support non-parametric representations of datatypes;
3. integrating PA datatype with Scala's for-comprehensions by implementing abstract methods `map`, `flatMap` and `filter` of trait `PArray[A]` (Fig. 2).

Figure 2 Parallel array operations (OO style)

```
trait PArray[A] {
  def length: Int
  def toArray: Array[A]
  def map[B](f: A ⇒ B): PA[B]
  def flatMap[B:Elem](f:A⇒ PA[B]): PA[B]
  def filter(f: A⇒ Bool): PA[A]
  def apply(i: Int)
  def zip[B](b: PA[B]): PA[(A, B)]
  def zipWith[B,C](f: A⇒ B⇒ C)(that: PA[B]): PA[C]

  // retrieve elements from specified indexes
  def backPermute(idxs: PA[Int]): PA[A]

  // place elements to specified indexes
  def permute(idxs: PA[Int]): PA[A]
  def slice(start:Int, len:Int): PA[A]
  def ++(that: PA[A]): PA[A]
  ...

  // see companion source code [28]
}
```

Figure 3 Parallel array operations (functional style)

```
trait Arrays extends ArraysBase {
  def fromArray[A: Elem](x: Array[A]): PA[A]
  def length[A: Elem](a: PA[A]): Int
  def index [A: Elem](a: PA[A], i: Int): A
  def replicate[A](count: Int, v: A): PA[A]
  def tabulate[A](f:Int)(f:Int ⇒ A): PA[A]
  def map[A,B](f: A ⇒ B)(a: PA[A]): PA[B]
  def zip[A,B](a: PA[A], b: PA[B]): PA[(A, B)]
  def unzip[A: Elem, B: Elem](a: PA[(A, B)]): (PA[A], PA[B])
  def concat[A](a: PA[PA[A]]): PA[A]
  def unconcat[A,B](a: PA[PA[A]])(b:PA[B]): PA[PA[B]]
  def flatMap[A,B](f:A⇒ PA[B])(a: PA[A]): PA[B]
  def filter[A](f: A⇒ Bool)(a: PA[A]): PA[A]
  ...

  // see companion source code [28]
}
```

Since Scala is a multiparadigm language, we support two styles of array operations: in object-oriented (Fig. 2) and functional style (Fig. 3) so that a programmer can choose the style that is better for his/her intent. In most cases the functional style versions are simply wrappers around the OO style versions (see the definition of `map, flatMap, filter, pack` etc in the source code [28]). But some operations have only one style of implementation (see `++, unzip, unconcat`).

Note that `PA[A]` has `PArray[A]` as the upper bound constraint which should be satisfied in descendent traits while providing concrete implementation according to polymorphic embedding design pattern. This makes it possible to think of parallel arrays as instances of the `PArray[A]` trait while writing code using Scalan as DSL.

3.4 What is *element type*?

In the definition of the `Vector` type we use the `Float` type as the type of element of parallel array (we will call it *element type* or *PA element type*). The types of array elements can be defined inductively according to the rules defined in Fig. 4. Given an *element type* `A` we can define the type of parallel array `PA[A]` with elements of the type `A`.

Figure 4 Element types

$$
\begin{array}{rll}
T := & Unit|Int|Boolean|Float|Char & (N_1) \\
| & (T1, T2) & (N_2) \\
| & (T1|T2) & (N_3) \\
| & (PA[T]) & (N_4) \\
| & (Tree[T]) & (N_5)
\end{array}
$$

Type constructor $(A|B)$ defined as $type\ |[A,B] = Either[A,B]$ (which has $Left$ and $Right$ constructors) and constructor $Tree$ as $case\ class\ Tree[A](value : A, children : PA[Tree[A]])$

For a given parallel array type `PA[A]` the concrete type of its internal representation depends on the element type `A`, specifically on the type structure (how the type is constructed). Representation types are built automatically by the library for each PA element type. This type-indexed behavior of PA types allows us to define generic representation rules ones and for all PA element types (see Fig. 15). We elaborate on this in the later sections. What is important here is that we can abstract this representations away leaving the details to the library.

3.5 Polytypic nature of parallel operations

The same way as a representation of a parallel array is automatically constructed based on its element type (see Fig. 15), the operations over this representation are automatically composed on the basis of the structure of the element type.

Figure 5 Polytypic operation `map`

```
// PA[Unit]
def map[R:Elem](f: Unit ⇒ R): PA[R] = {
  element[R].tabulate(len)(i ⇒ f())
}
// PA[A]
def map[R:Elem](f: A ⇒ R) = {
  element[R].tabulate(arr.length)(i ⇒ f(arr(i)))
}
// PA[(A,B)]
def map[R:Elem](f: ((A,B)) ⇒ R): PA[R] = {
  val len = length
  element[R].tabulate(len)(i ⇒ f(a(i),b(i)))
}
// PA[(A|B)]
def map[R:Elem](f: (A|B) ⇒ R): PA[R] = {
  val len = length
  element[R].tabulate(len)(i ⇒ f(index(i)))
}
// PA[PA[A]]
def map[R:Elem](f: PA[A] ⇒ R): PA[R] = {
  val len = length
  element[R].tabulate(len)(i ⇒
    {val (p,l) = segments(i); f(arr.slice(p,l))})
}
// PA[Tree[A]]
def map[R:Elem](f: Tree[A] ⇒ R): PA[R] = {
  val len = length
  element[R].tabulate(len)(i ⇒ {val t = this(i); f(t)})
}
```

This is achieved by implementing each abstract operation declared in base trait `PArray[A]` in all representation classes shown in Fig. 14.

Consider the function `map` declared in the trait `PArray[A]` (see Fig. 2). The implementation of this function for each representation class from Fig. 14 is shown in Fig. 5. We have to provide such implementation for each type constructor in the rules (see Fig. 4) of building an element type.

4 Scalan by examples

In this section we describe the Scalan library from the point of view of the programmer, illustrating our description with a series of examples.

Figure 6 Dot-product of two vectors

```
import scalan.dsl._

trait DslSamples extends Scalan {
  type Vector = PA[Float]

  // using for-comprehensions
  def dotProduct(v1: Vector, v2: Vector): Float =
    sum(for ((f1,f2) <- v1 zip v2) yield f1 * f2)

  // more idiomatic Scala
  def dotProduct2(v1: Vector, v2: Vector): Float =
    (v1, v2).zippedPA.map{ _ * _ }.sum
}

trait StdSamples {
  type Vector = Array[Float]

  def dotProduct(v1: Vector, v2: Vector): Float =
    (for ((f1,f2) <- v1 zip v2) yield f1 * f2).sum

  def dotProduct2(v1: Vector, v2: Vector): Float =
    (v1, v2).zipped.map{ _ * _ }.sum
}
```

4.1 Parallel dot-product

Consider the scalar multiplication of two vectors shown in Fig. 6. We show two versions, a parallel and a sequential one in comparisson, to emphasize the similarity of parallel and sequential code. In the parallel version we define the `Vector` type as a parallel array. Thus we employed the type directed approach to expressing parallelism. The parallel version of `dotProduct2`, being literally identical to the sequential version, has an internal representation and implementation that are completely different from the sequential ones.

4.2 Sparse matrix-vector multiplication

Next, consider the definition of `sparseVectorMul` in Fig. 7. We represent a sparse vector as a parallel array of pairs where the integer value represents the index of an element in the vector and the float value represents the value of the element (*compressed row format*). Having this representation, we can define the dot-product of sparse and dense vectors as a function over parallel arrays and thus expressing our intent for parallel evaluation.

Moreover, we can use the parallel function `sparseVectorMul` to define another

Figure 7 Sparse matrix-vector multiplication

```
import scalan.dsl._
trait  DslSamples extends Scalan {
  type VectorElem = (Int,Float)      // element index and value
  type SparseVector = PA[VectorElem] // store only non-zero elements
  type Vector = PA[Float]
  type Matrix = PA[SparseVector]     // array of rows

  def sparseVectorMul(sv: SparseVector, v: Vector) =
    sum(for ((i,value) <- sv) yield v(i) * value)

  def matrixVectorMul(matr: Matrix, vec: Vector) =
    for (row <- matr) yield sparseVectorMul(row, vec)
}
```

Figure 8 QuickSort

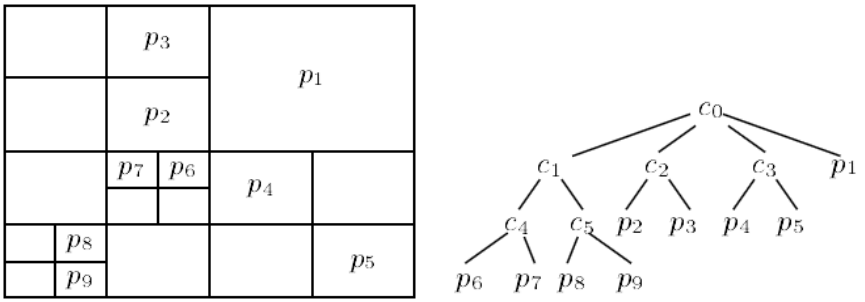
```
import scalan.dsl._

trait  DslSamples extends Scalan {
  def qsort(xs: PA[Int]): PA[Int] = {
    val len = xs.length
    if (len <= 1) xs
    else {
      val m = xs(len / 2)
      val smaller = for (x <- xs if x < m) yield x // for-comprehesions
      val greater = xs filter(x ⇒ x > m)       // or methods directly
      val equal = xs filter(_ == m)            // or even shorter
      val sg = fromArray(Array(smaller, greater))
      val sorted = for (sub <- sg) yield qsort(sub)
      sorted(0) ++ equal ++ sorted(1)
    }
  }
}
```

parallel function `matrixVectorMul` shown in Fig. 7 realizing the principle of composability inherent to nested data parallelism. We will see in later sections that the underling representation of the `Matrix` type enables both outer and inner parallelism to be used through the *flattening* techniques [1].

4.3 Quicksort

We can also use a parallel function inside its own definition i.e. recursively. Fig. 8 shows how the quick-sort recursive algorithm can be implemented in Scalan.

Figure 9 Points of mass and Barnes-Hut tree



Again we exploit the inner (sorting of subarrays) and the outer parallelism (recursive calls) by using parallel datatypes alone without introducing any parallel constructions in the language.

4.4 Working with hierarchical data

Consider a popular example of building Barnes-Hut tree that appears in many publications on nested parallelism and that we show here as well. Given a set of points on a plane, we have to build the centroid tree shown in Fig. 9. To represent this tree, we use the `Tree[A]` element type constructor from Fig. 4. Following [17] we provide a limited support for user-defined *recursive types* as parallel array element type. The type constructor `Tree[A]` has the following definition in the library.

```
case class Tree[A](
    value : A, // for each node
    children: PA[Tree[A]])
```

Using `Tree[A]`, we can define recursive parallel functions over recursive irregular data structures. We show in Fig. 10 how to encode Barnes-Hut tree building algoritm in Scalan, the example taken from [17]. The functions `splitArea`, `inArea`, `calcCentroid` have obvious implementations and we omit them for brevity. It is important to realize that while we are talking about irregular data structures they in fact have very regular and *flattened* representations that are given in Fig. 17 and will be discussed in Section 5.

4.5 Abstract syntax trees

Elaborating on the usage of the `Tree[A]` type constructor, we can represent even more irregular data structures exploiting the posibilities that come from the usage of the type constructor `(A|B)` (see Fig. 4). So far, we considered only parallel arrays with elements of the same type even in the case of the `buildTree` sample.

Figure 10 Barnes-Hut tree building function

```
import scalan.dsl._

trait DslSamples extends Scalan {
  type Point = (Float, Float)        // x and y coordinates
  type Vector = (Float, Float)       // 2d vector
  type Force = Vector
  type Centroid = (Float, Point)     // mass and position
  type Particle = (Centroid, Vector) // position and velocity
  type Area = (Point, Point)         // by lower left and upper right
  type CentroidTree = Tree[Centroid] // tree of centroids

  def buildTree(area: Area, particles: PA[Particle]): CentroidTree = {
    if (particles.length == 1) {
      val ((m, loc), vel) = particles(0)
      Tree((m, loc), emptyArrayOf[CentroidTree])
    }
    else {
      val subtrees = for (
          a <- splitArea(area);
          val ps = for (p <- particles if inArea(a, p)) yield p
          if ps.length > 0
        )
        yield (buildTree(a, ps))

      val (m,l) = calcCentroid(subtrees)
      Tree((m, l), subtrees)
    }
  }
}
```

Figure 11 Abstract syntax tree as PA element

```
  type Expr = Tree[ExprNode]
  type ExprNode =
    ( String |       // Variables x, y, ... or env x → expr
    ( Int    |       // Int constant 1, 2, 3, ...
    ( String |       // constructor C(args)
    ( String |       // lambda abstraction λx → Expr
    ( Unit   |       // application (e1 e2)
      String )))))   // env item x → expr

  type Env = PA[Expr]
```

A more sophisticated example is given in Fig. 11, where we define `Expr` type to represent an abstract syntaxt tree (AST) for terms of the lambda calculus. Being able to encode a type of expressions using the constructors from Fig. 4, we automatically get a flattened and regular non-parametric representation (see next section 5) for the abstract syntax tree which we use here as an example of a very irregular data structure.

We might be interested in such a representation of expressions for a number of reasons.

1. To parallelize the execution of an operation (such as evaluation) over an instance of AST (inner parallelism).

2. To enable a parallel operation to be performed over a large collection of expressions (outer parallelism).

Consider the case when we have hundreds or thousands of expressions that we have to evaluate or transform. An example of such a transformation is multi-result supercompilaton [18] that can produce a lot of intermediate expressions that have to be analysed on the subsequent steps. Another example is genetic programming [26], where a population of thousands of computer programs(represented by ASTs) is evolved and on each step we need to estimate each program, often by evaluating it and comparing the results. It is the flattened representation of arrays that promise to enable a massive data parallelism in this cases.

Again we see the case of nested parallelism where we can combine the outer parallelism (over a collection of expressions) and the inner parallelism (operating over an expression) in a single flat parallel operation thus increasing the degree of parallelism.

Note that we cannot use recursive type definitions (direct of indirect) as they are not allowed for type synonims in Scala. Instead we use special encodings that we can abstract over by using constructors and extractors (see Fig. 12). We show an implementation for `Var`, `Lit` and `Con` and omit for others as they can be implemented in a similar manner (see an example in the source code [28]). The absence of recursive definitions can be considered a limitation of our approach but fortunately we can extend it to overcome such difficulties. We are going to elaborate on this in Section 7.

Now with the type `Expr`, constructors and extractors in hand, we can use flattened representation of lambda terms to implement a simple interpreter function `eval` with call-by-value semantic which enables us to automatically employ the inner (of expression) and the outer (of collection) parallelism (see Fig. 13).

5 Array representations

For each element-type `A` defined by means of constructors from Fig. 4 we want to specify how an instance of `PA[A]` is represented. That is we want to define a concrete representation type of parallel array depending on the element type

Figure 12 Constructors and extractors of AST

```
object Var {
  def apply(name: String): Expr = Tree(Left(name))
  def unapply(e: Expr) =
      e.value fold(v ⇒
          if (v != closureVarName) Some(v) else None, _ ⇒ None)
}
object Lit {
  def apply(n: Int): Expr = Tree(Right(Left(n)))
  def unapply(e: Expr) =
      e.value fold(_ ⇒ None, _.fold(n ⇒ Some(n), _ ⇒ None))
}
object Clo {
  def apply(e: Expr, env: Env): Expr =
      Tree(Left(closureVarName), singleton(e) ++ env)
  def unapply(e: Expr) = e.value fold(v ⇒
      if (v == closureVarName) {
        val expr = e.children(0)
        val env = e.children.slice(1, e.children.length-1)
        Some((expr, env))
      }
      else None, _ ⇒ None)
}
object Con {
  def apply(name: String, args: PA[Expr]): Expr =
      Tree(Right(Right(Left(name))), args)
  def unapply(e: Expr) = e.value fold(_ ⇒ None,
    _.fold(_ ⇒ None,
    _.fold(c ⇒ Some((c, e.children)), _ ⇒ None)))
}
object Lam {
  def apply(v: String, body: Expr): Expr =
      Tree(Right(Right(Right(Left(v)))), singleton(body))
  def unapply(e: Expr) = e.value fold(_ ⇒ None,
    _.fold(_ ⇒ None,
    _.fold(_ ⇒ None,
    _.fold(l ⇒ Some((l, e.children(0))), _ ⇒ None))))
}
object App {
  ... // see companion source code [28]
}
object EnvItem {
  ...
}
```

Figure 13 The Interpreter function

```
def eval(expr: Expr, env: Env): Expr = expr match {
  case l@Lit(n) ⇒ l
  case v@Var(x) ⇒ if (isPrim(x)) v else env(x)
  case Con(name, args) ⇒ Con(name, args map { eval(_, env) })
  case App(f, g) ⇒ {
    val vs = fromArray(Array(f, g)) map { eval(_, env) }
    val (f1,g1) = (vs(0), vs(1))
    f1 match {
      case Clo(Lam(x, e1), env1) ⇒
        val env2 = singleton(EnvItem(x, g1)) ++ env1
        eval(e1, env2)
      case Var("plus") ⇒ g1 match {
        case Con(name, args) if name == "P" && args.length == 2 ⇒
          (args(0), args(1)) match {
              case (Lit(a1),Lit(a2)) ⇒ Lit(a1 + a2)
          }
        case _ ⇒ error("constructor P expected: operation plus")
      }
      case _ ⇒ error("primitive function expected " + f)
    }
  }
  case Lam(_,_) ⇒ Clo(expr, env)
}

val x = Var("x")
val f = Lam("x", App(Var("plus"), Con("P", x, 1)))
val app = App(f, 100)
scala> eval(app, emptyArrayOf[Expr])
res0: Samples.Expr = Tree(Right(Left(101)),SeqTreeArray(None))

val apps = replicate(3, app)
val emptyEnv = emptyArrayOf[Expr]
apps map { eval(_, emptyEnv) }
scala> res4: Samples.PA[Samples.Expr] =
  SeqTreeArray(Some(
    SeqPairArray(
      SeqSumArray((true,true,true), (),
        SeqSumArray((false,false,false),(101,101,101),
          SeqSumArray((),(),
                      SeqSumArray((),(),SeqSumArray((),SeqUnitArray(0),()))))
        )),
      SeqNestedArray(SeqTreeArray(None),SeqPairArray((0,0,0),(0,0,0)))
    )))
```

Figure 14 Abstract representation classes

```
type Item[A] = (A, PA[Tree[A]])

abstract class UnitArray(val len: Int) extends PArray[Unit]

abstract class StdArray[A](val arr: Array[A]) extends PArray[A]

abstract class PairArray[A,B](val a: PA[A], val b: PA[B])
      extends PArray[(A,B)]

abstract class SumArray[A,B](val flags: PA[Boolean], val a:PA[A], val b:PA[B])
      extends PArray[(A|B)]

abstract class NestedArray[A](val arr: PA[A], val segments: PA[(Int,Int)])
      extends PArray[PA[A]] {

abstract class TreeArray[A](items: Option[PA[Item[A]]])
      extends PArray[Tree[A]]
```

Figure 15 Representation rules

$$
\begin{array}{lll}
Rep[\![PA[Unit]]\!] & \Rightarrow UnitArray & (R_0)\\
Rep[\![PA[T]]\!] & \Rightarrow Array[T] \text{ if } T = Int|Boolean|Float|Char & (R_1)\\
Rep[\![PA[(A,B)]]\!] & \Rightarrow PairArray(Rep[\![PA[A]]\!], Rep[\![PA[B]]\!]) & (R_2)\\
Rep[\![PA[(A|B)]]\!] & \Rightarrow SumArray(Rep[\![PA[Boolean]]\!], Rep[\![PA[A]]\!], Rep[\![PA[B]]\!]) & (R_3)\\
Rep[\![PA[PA[A]]]\!] & \Rightarrow NestedArray(Rep[\![PA[A]]\!], Rep[\![PA[(Int,Int)]]\!]) & (R_4)\\
Rep[\![PA[Tree[A]]]\!] & \Rightarrow TreeArray(Some(Rep[\![PA[Item[A]]]\!])) \text{ if not empty} & (R_5)\\
 & \Rightarrow TreeArray(None) \text{ otherwise} &
\end{array}
$$

i.e. as a type indexed type. We use Scala's abstract classes shown in Fig. 14 and representation rules from Fig. 15. The function *Rep* is a mapping from the abstract type `PA[A]` to a concrete representation type which is built recursively on the structure of the type `A`. This is similar to *flattened representations* [4] or *representation transformation* [5], but encoded in Scala.

In the simplest case of `PA[Unit]` type, we just need to store the length of array (rule R_0). Rule R_1 says that any parallel array with elements of Scala's *value classes* is represented as an ordinary array of unboxed values (an important property if we want to achieve a better cache locality of array operations). When the element type is a pair, then the array of pairs is represented as a pair of arrays (rule R_2).

This has the following benefits.

1. Some operations like those shown in Fig. 16 have a very efficient implemen-

Figure 16 Efficient operations by non-parametric representation

```
def zip[A,B](a: PA[A], b: PA[B]): PA[(A, B)] = PairArray(a, b)

def unzip[A,B](a: PA[(A, B)]): (PA[A], PA[B]) = {
  val pa = a.asInstanceOf[PairArray[A, B]]
  (pa.a, pa.b)
}

def concat[A](a: PA[PA[A]]): PA[A] = a match {
  case nested: NestedArray[A] ⇒ nested.arr
}

def unconcat[A,B](a: PA[PA[A]])(b:PA[B])
                 (implicit ea: Elem[B], epa: Elem[PA[B]]): PA[PA[B]] =
  a match {
    case nested: NestedArray[A] ⇒ SeqNestedArray(b, nested.segments)
  }
```

tation which facilitates domain specific code optimizations.

2. It enables values to be stored unboxed in contrast to the $Array[(A, B)]$ type, where the array contains references to Tuple objects.

To represent the type `PA[(A|B)]`, we use a triple of arrays (rule R_3), where the `flags` array is used to distinguish between `Left` and `Right` case of the `Either` type. The invariant of this representation is that `length(flags) = length(a) + length(b)`.

Even more interesting is the rule R_4 where all the flattening happens. We use representation for `PA[PA[A]]` proposed in [2] and keep all elements of the nested parallel array in a single flat parallel array of values coupled with a parallel array of segment descriptors (which by rules R_2 and R_1 is represented as a pair of ordinary arrays).

The last rule R_5 defines the representation of a parallel array when its element type is a recursive data type. This representation is similar to one described in [17]. To make it more clear, we take the example tree from [17] (see Fig. 9) and show its encoding using our representation (see Fig. 17). Note that each level of the tree is represented as a flat parallel array containing all the elements from that level along with the segment descriptors that can be used to access those elements.

6 Scala encodings

Given the representation rules from Fig. 15, we want to implement these rules in the library. More specifically, our goal is to show how array representation

Figure 17 Barnes-Hut tree representation

```
val m = 1.0f
val vel = (0f, 0f)
val ps = fromArray(Array(
  ((m,(12f,12f)), vel),
  ((m,(6f,10f)), vel),
  ((m,(6f,14f)), vel),
  ((m,(10f,6f)), vel),
  ((m,(14f,2f)), vel),
  ((m,(7f,7f)), vel),
  ((m,(5f,7f)), vel),
  ((m,(3f,3f)), vel),
  ((m,(3f,1f)), vel)
))
val area = ((0f,0f),(16f,16f))
val tree = buildTree(area, ps)
scala> tree: Samples.CentroidTree =
Tree((9.0,(8.625,8.125)),
  TreeArray(Some(
    PairArray(
    PairArray((4.0, 2.0, 1.0, 2.0),
    PairArray((4.5, 12.0, 12.0, 6.0),
              (4.5, 4.0, 12.0, 12.0))),
    NestedArray(
      TreeArray(Some(
        PairArray(
        PairArray((2.0, 2.0, 1.0, 1.0, 1.0, 1.0 ),
        PairArray((3.0, 6.0, 14.0, 10.0, 6.0, 6.0 ),
                  (2.0, 7.0, 2.0, 6.0, 10.0, 14.0))),
        NestedArray(
          TreeArray(Some(
            PairArray(
            PairArray((1.0, 1.0, 1.0, 1.0),
            PairArray((3.0, 3.0, 7.0, 5.0),
                      (1.0, 3.0, 7.0, 7.0))),
            NestedArray(
              TreeArray(None),
              PairArray((0, 0, 0, 0), (0, 0, 0, 0))
            ) ))),
            PairArray((0, 2, 4, 4, 4, 4), (2, 2, 0, 0, 0, 0)))
        ))),
        PairArray((0, 2, 4, 4), (2, 2, 0, 2) ))
    ))))
```

Figure 18 Typeclass of parallel array element

```
trait Element[A] {
  def replicate(count: Int, v: A): PA[A]
  def replicateSeg(count: Int, v: PA[A]): PA[A]
  def tabulate(len: Int)(f:Int ⇒ A): PA[A]
  def tabulateSeg(len: Int)(f:Int ⇒ PA[A]): PA[A]
  def empty: PA[A]
  def singleton(v: A) = replicate(1,v)
  def fromArray(arr: Array[A]) = tabulate(arr.length)(i ⇒ arr(i))
}

type Elem[A] = Element[A]
```

rules can be expressed in Scala, so that every instance of `PA[A]` is represented as $Rep[\![PA[A]]\!]$.

Scala [24, 23] "is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive."

As a statically typed language Scala has a very powerfull and expressive type system which, combined with a flexible syntax, can be used to employ many high level techniques and patterns such as *type classes* [7], *datatype-generic programming* [25], *type level programming* [21] etc. We refer the interested reader to these papers for details and assume familiarity with these techniques.

6.1 Typeclass `Element[A]`

If we want a type `A` to be an element-type of a parallel array (i.e. we need the `PA[A]` type), an instance of the typeclass `Element[A]` should be defined for the type `A`. To encode typeclasses in Scala we follow the *CONCEPT* pattern [7]. Fig. 18 shows the concept-interface(typeclass) `Element` and the convenience type-synonim $Elem[T]$.

For each type `A` to be able to declare type `PA[A]`, we require an implicit declaration (value, conversion or parameter) of the type `Elem[A]` to be available in the scope as an evidence that a typeclass instance exists for the typeclass `Elem[A]`. *In other words, if Scala's implicit resolution for the type `Elem[A]` cannot succeed in the scope then the typeclass instance `Elem[A]` is not defined for the type `A` in the scope.*

For each Scala's value type `A` there is an implicit definition of the type $Elem[A]$ which is declared in the library and those declarations are made available by inheriting from `Scalan` trait.

For example:

```
trait BarnesHut extends Scalan {
  type Vector = PA[Float] // requires implicit Elem[Float]

  // use PA methods over Vector datatype
  ...
}
```

These library definitions specify the base cases for inductive representation rules from Fig. 15. They are declared in the library like this:

```
  implicit val unitElement: Elem[Unit] = new UnitElement
  implicit val intElement: Elem[Int] = new StdElement[Int]
  implicit val doubleElement: Elem[Double] = new StdElement[Double]
  ...
```

The simpliest form of a parallel array is `PA[Unit]` which requires an instance of `Elem[Unit]` typeclass which is represented by `UnitElement` shown above.

Next, for each *value type* `T` of Scala we use `StdElement[T]` class that represents(implements) an instance of the typeclass `Elem[T]`. In its implementation we use the standard `Array` as the storage for array elements. For *value types* of Scala the standard arrays have unboxed representation (since they are Java arrays). We use them in our non-parametric representations of parallel arrays. As a remark: note that unboxed arrays proved to have very attractive performance characteristics due to cache locality, for which reason these unboxed representations can be regarded as one of the enabling factors of efficient parallelization.

Having defined instances of `Elem[A]` and `Elem[B]` typeclasses, we can define an instance for `Elem[(A,B)]` and `Elem[(A|B)]` like this:

```
implicit def pairElement[A,B](implicit ea: Elem[A], eb: Elem[B]): Elem[(A,B)] =
  new Element[(A,B)] {
    // implementation of methods of Element trait
    ...
  }
implicit def sumElement[A, B](implicit ea: Elem[A], eb: Elem[B]): Elem[(A|B)] =
  new Element[(A|B)]
  {
    // implementation of methods of Element trait
    ...
  }
```

And similarly, given an instance of `Elem[A]`, we can define `Elem[PA[A]]` using the definition:

```
implicit def arrayElement[A](implicit ea: Elem[A]): Elem[PA[A]] =
  new Element[PA[A]] {
    // implementation of methods of Element trait
    ...
  }
```

And last, but not the least, following this pattern we define the element type for the `Tree[A]` type:

```
implicit def treeElement[A](implicit ea: Elem[A]): Elem[Tree[A]] =
  new Element[Tree[A]] {
    // implementation of methods of Element trait
    ...
  }
```

All these implicit definitions are made available by inheriting from the `Scalan` trait. Even though their implementation is not specified in the Scalan trait and these implicit definitions remain abstract they are available for implicit resolution in Scalan code. Mixing Scalan code with concrete implementation, as shown in Fig. 1, among other things provides concrete implementation (instances) for all declared Element typeclasses.

6.2 Sequential reference implementation

In this paper we describe a reference implementation of our parallel array library. Despite the use of the word 'parallel' the implementation we describe here is in fact sequential.

This deliberate simplification allows us to concentrate on using Scala encodings to implement non-parametric representations of arrays. We call them 'parallel' arrays just by tradition. There is nothing inherently parallel in these representations by itself. As shown in [5], these representations facilitate flattening transformation which in turn leads to efficient implementation of nested parallelism.

In Fig. 19 we show the components of the Scalan library. We refer interested reader to the source code [28] for details.

7 Conclusion

It is well known that non-parametric representation of parallel arrays alone is able to reduce the running time by 50% [5] due to unboxing and cache locality. Combined with flattening transformation, array fusion and other optimization techniques along with parallel runtime, this approach can be a very promising programming model not only covering a wide range of applications but also providing a near to optimal performance (comparable to highly optimized hand-written C code).

In this paper we showed how to express nested parallelism in the Scala language using Scalan library primitives. We also showed that the Scala language is expressive enough to encode non-parametric type-indexed representations of parallel arrays – specially designed data structures that fasilitate the implementation of nested parallelism.

We also showed in Section 4 how to apply this programming model to a nontraditional application giving an implementation of a simple interpreter for lambda

Figure 19 Structure of Scalan definitions

```
package scalan.dsl
trait  ArraysBase {
  trait  Element[A] { /* see Fig. 18 */ }
  trait  PArray[T] { /* see Fig. 2 */ }
  // definitions of abstract classes:
  // UnitArray, PairArray, SumArray, NestedArray (see Fig. 14)
}
trait  Arrays extends ArraysBase { /* see Fig. 3 */ }

package scalan.sequential
trait  SeqImplementation extends ArraysBase {
  // definitions of Element instances:
  // stdElement, unitElement, pairElement, sumElement, arrayElement
  // (see Section 6)
  trait  SeqPArray[T] extends PArray[T] { /* common implementations */ }
  case class SeqUnitArray(override val len: Int)
      extends UnitArray(len) with SeqPArray[Unit] {
    // implementation of PA[Unit] (see Fig. 15 rule R0)
  }
  case class SeqStdArray[T](arr: Array[T])(implicit t: Elem[T], z:Zero[T])
      extends StdArray[T](arr) with SeqPArray[T] {
    // implementation of PA[T] (see Fig. 15 rule R1)
  }
  case class SeqPairArray[A, B](
      override val a: PArray[A], override val b: PArray[B])
      (implicit  e:Elem[(A,B)])
      extends PairArray[A,B](a,b) with SeqPArray[(A,B)] {
    // implementation of PA[(A,B)] (see Fig. 15 rule R2)
  }
  case class SeqSumArray[A, B](
      override val flags: PArray[Boolean],
      override val a: PArray[A],
      override val b: PArray[B])
      (implicit  ea: Elem[A], eb: Elem[B], e: Elem[(A|B)])
      extends SumArray[A, B](flags, a, b) with SeqPArray[(A|B)] {
    // implementation of PA[(A|B)] (see Fig. 15 rule R3)
  }
  case class SeqNestedArray[A](
      override val arr: PA[A], override val segments: PA[(Int,Int)])
      (implicit  ea: Elem[A], epa: Elem[PA[A]])
      extends NestedArray[A](arr,segments) with SeqPArray[PA[A]] {
    // implementation of PA[PA[A]] (see Fig. 15 rule R4)
  }
}
```

calculus. Moreover, we believe that it is possible to apply this approach (in general and our library in particular) to a whole range of so-called *language-oriented* applications.

This is a work-in-progress report and a lot more has to be done to make it applicable to real-world cases.

7.1 Limitations and further development

Our approach has some limitations, so that lifting them can be regarded as a goal of future investigation and development.

1. First and the most important limitation is that our implementation is sequential (see more explanations in the subsection below).

2. The approach doesn't directly support (mutually) recursive datatypes as PA elements and the only way to encode parallel arrays of recursive datatypes is to use the `Tree[A]` type constructor. We show in Section 4 how it can be done using `Expr` datatype in combination with Scala *extractors*.

3. It may be possible to include exponential datatypes to define instances `Elem[A⇒ B]` of array elements, and this extension seems to be straightforward and can employ techniques described in [20]

4. Each declaration of `PA[T]` requires the type `T` to be constructed as an element type (see Fig. 4). This can be considered a limitation when integrating parallel functions with the rest of the program. For each user defined class (or case class) `U` a mapping to some element type `T` should be defined to convert instances of `Array[U]` to `Array[T]` and then to `PA[T]`. A systematic approach described in [25] can be used to alleviate this.

7.2 Notes on parallel implementation

Our sequential implementation can be regarded as just a test for our approach. But it can also be used to simulate parallel evaluation for debuging purposes. This simulaion can equally be used both to validate the correctness of particular application and to validate a particular parallel implementation of the library itself.

Considering possible parallel implementations, we see a range of alternatives that can be regarded as directions for future investigation and development.

1. The most natural and logical choice is to use Scala's parallel collections framework [27] as a basis for representing parallel arrays and operations from Fig. 2.

2. We can exploit the transformational nature of the nested parallel approach and generate parallel code for particular back-end language (including Scala in this role).

3. Another option is to regard Scalan as a high-level embedded DSL and automatically generate parallel code for some target heterogeneous architecture like Open CL, CUDA, FPGA, etc [19].

Acknowledgements

The author expresses his gratitude to Sergei Romanenko, Ilya Klyuchnikov, Andrei Klimov and to all participants of Refal seminar at Keldysh Institute for useful comments and fruitful discussions of this work.

References

[1] Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[2] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990.

[3] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, 1995.

[4] Manuel M. T. Chakravarty and Gabriele Keller. More Types for Nested Data Parallel Programming. In *In Proceedings ICFP 2000: International Conference on Functional Programming*, pages 94–105. ACM Press, 2000.

[5] Manuel M. T. Chakravarty and Gabriele Keller. An Approach to Fast Arrays in Haskell, 2002.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. In *In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[7] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes as Objects and Implicits. In *n Proceedings of the 25th ACM International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH/OOPSLA)*, October 2010.

[8] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. MapReduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation.* USENIX Association, 2004.

[9] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. MPI: The Complete Reference (Vol. 2). Technical report, The MIT Press, 1998.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 148–174, 2004.

[11] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.

[12] Paul Hudak. Building domain-specific embedded languages. *ACM COMPUTING SURVEYS*, 28, 1996.

[13] Paul Hudak. Modular domain specific languages and tools. In *IN PROCEEDINGS OF FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE*, pages 134–142. IEEE Computer Society Press, 1998.

[14] Patrik Jansson. Polytypic programming. In *2nd Int. School on Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.

[15] Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[16] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell, 2008.

[17] Gabriele Keller and Manuel M.T. Chakravarty. Flattening Trees, 1998.

[18] Ilya Klyuchnikov and Sergei Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions, 2011. Accepted to PSI 2011.

[19] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. GPU Kernels as Data-Parallel Array Computations in Haskell, 2009.

[20] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher Order Flattening. In *International Conference on Computational Science (2)*, pages 920–928, 2006.

[21] Rúnar Óli. Type-Level Programming in Scala. `http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/`, June 2010.

[22] NVIDIA. NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2011.

[23] Martin Odersky. The scala language specification. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, November 2010.

[24] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[25] Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for generic programmers. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 25–36, New York, NY, USA, 2008. ACM.

[26] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[27] Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, and Martin Odersky. A generic parallel collection framework, 2010.

[28] Alexander Slesarenko. Scalan. `http://github.com/scalan`, April 2011.