Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 5, 2012

Slesarenko A.V.

Polytypic Staging: a new approach to an implementation of Nested Data Parallelism in Scala

Recommended form of bibliographic references: Slesarenko A.V. Polytypic Staging: a new approach to an implementation of Nested Data Parallelism in Scala. Keldysh Institute preprints, 2012, No. 5, 24 p. URL: **http://library.keldysh.ru/preprint.asp?id=2012-5&lg=e**

KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Alexander Slesarenko

Polytypic Staging: a new approach to an implementation of Nested Data Parallelism

Moscow
2012

Alexander Slesarenko

Polytypic Staging: a new approach to an implementation of Nested Data Parallelism in Scala

This paper describes *polytypic staging,* – an approach to staging of a domain-specific language (DSL) that is designed and implemented by means of polytypic (datatype-generic) programming techniques. We base our implementation on Lightweight Modular Staging (LMS) framework by extending and making it polytypic. We show how to apply it to a particular domain. The domain is nested data parallelism where data parallel programs are expressed in the DSL embedded in Scala. The paper is organized around a specific DSL, but our implementation strategy should be applicable to any *polytypic DSL* in general.
Key words: polytypic, staging, generic programming, embedded DSL, nested data parallelism, Scala

Supported by Russian Foundation for Basic Research project No. 12-01-00972-a

Александр Владимирович Слесаренко

Политиповое многостадийное программирование: новый подход к реализации вложенного параллелизма на Scala

Эта работа описывает *политиповое многостадийное программирование* – новый подход к реализации глубокого вложения (deep embedding) предметно-ориентированного языка (DSL), который реализован посредством методов политипового (обобщенного) программирования. В качестве основы используется Lightweight Modular Staging (LMS) фреймворк, который расширяется и делается политиповым. В работе показано, как применить политиповое многостадийное программирование на примере конкретной предметной области – вложенный параллелизм по данным, при этом параллельные программы записываются на DSL вложенном в язык Scala. Работа построена на примере конкретного DSL, однако, описанный подход может быть применен к любому *политиповому DSL*.
Ключевые слова: политиповое программирование, многостадийное программирование, предметно-ориентированный язык, параллельное программирование

Работа выполнена при поддержке проекта РФФИ № 12-01-00972-а

1. Introduction

It is well known that modern computing hardware is able to execute many computational threads in parallel allowing a programmer to increase performance of the program. It is especially true in such a new area as GPGPU [1] where hardware supports the execution of hundreds and thousands of program threads. However for this purpose the program must be written in a proper format, significantly different from the traditional sequential style. In other words, if the program has not been designed for parallel execution, then this program will not be able to fully use the capabilities of modern equipment.

A long-term trend in the field of development tools for parallel computing consists in lowering of the threshold of complexity, namely the development of easy to use languages and libraries [9,10,22], encapsulation of complexity in the implementation of the system software [19], creating interactive working environments [26].

In this paper, we continue to explore different methods and approaches to simplify parallel programming. We rely on a series of publications [17,4,18] on the *nested data parallelism model* (NDP), as well as on our previous work [23] on this subject.

The model of NDP was first formulated in the early 90's [3], but still is not widely used in practice, although there is a series of publications and a publicly available implementation [6].

On the other hand, many techniques and technologies [2,8,14,21,24], which we use as a foundation of our approach, have appeared only in recent years so we have attempted to restate the problem and implement the model in the new environment.

We envision our implementation of NDP as a DSL embedded in Scala-Virtualized as a host language and packaged as a library. We compare it with Parser Combinators library which also has limited expressiveness, focused target purpose and inherent composability, while still having a wide range of applications in different problem domains.

In our previous work [23] we covered a DSL view on nested data parallelism and described a design of our library. As it turned out, the implementation of the library and the DSL is most naturally expressible by using *generic programming* [11] (*polytypic programming* [16]) techniques, hence our DSL is polytypic.

From the DSL point of view, we regard our previous implementation as 'shallow embedding' as oppose to 'deep embedding' that is described in this paper and which is consistent with our previous work.

For deep embedding we use polymorphic embedding [14] and LMS [24] as its particular instance. We extend the later to account for polytypism of our DSL.

In summary, this paper makes the following main contributions:

1. We extend our previously published work [23] by introducing a "Polytypic Staging" technique (PTS).
2. We show how to extend Lightweight Modular Staging (LMS) framework by making it polytypic (datatype-generic) over a family of type constructors: sum, product and array
3. We show how to apply Polytypic Staging to a special problem domain of nested data parallelism. It turned out that Generic Programming is natural in this problem domain and leads to a modular and compositional design.
4. We briefly overview the theoretical and technical foundations of our approach such as Polymorphic Embedding, Phantom Types, Generic Programming and LMS
5. And last but not the least, we show yet another application of Virtualized Scala [2]

In this paper we also describe some aspects of the design and implementation of the Scalan library. (source code being available at http://github.org/scalan).

2. DSL view on nested data parallelism

We start with some model examples to illustrate basic ideas of NDP and what is it looks like to program against this model. Note, that we only show parts of the code relevant to our discussion and refer to our previous paper [23] for details of the library design.

Consider the definition of sparseVectorMul in Fig. 1. We represent a sparse vector as an 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 a dot-product of sparse and dense vectors as a function over arrays.

Note, that instead of using the ordinary Array[T] type we use an abstract PArray[T] trait and thus, first, make the code abstract, and second, expressing our intent for a parallel evaluation.

When it comes to multiplying a sparse matrix with a dense vector, we can reuse our parallel function sparseVectorMul to define another parallel function matrixVectorMul realizing the principle of composability inherent to nested data parallelism.

```
trait PArray[A]
type VectorElem = (Int,Float)
type SparseVector = PArray[VectorElem]
type Vector = PArray[Float]
type Matrix = PArray[SparseVector]
def sparseVectorMul(sv: SparseVector, v: Vector): Float =
      sum(sv map { case Pair(i,value) => v(i) * value })
def matrixVectorMul(matr: Matrix, vec: Vector): Vector =
      matr map {row => sparseVectorMul(row, vec)}
```

Fig. 1. Sparse matrix vector multiplication

Also note that due to polytypic nature of our DSL we have a freedom (up to a family of product, sum and PArray type constructors) to select data structures and must select them ‘properly’ (It is our choice here to represent sparse matrix as a parallel array of sparse vectors and not dense ones).

We can also use a parallel function inside its own definition i.e. recursively. Fig. 2. shows how QuickSort recursive algorithm can be implemented in NDP model.

```
trait PArray[T] {
  def partition(flags: PA[Boolean]): PArray[PArray[T]]
}
def qsort(xs: PArray[Int]): PArray[Int] = {
  val len = xs.length
  if (len <= 1) xs
  else {
    val pivot = xs(len / 2)
    val less = xs map { x => x < pivot }
    val subs = xs.partition(less)
    val sorted = subs map { sub => qsort(sub) }
    concat(sorted)
  }
}
```

Fig. 2. Parallel QuickSort

Note that there are no parallel primitives in the code, as the semantics is purely functional, sequential and deterministic. Nevertheless, we express parallelism (what we want to be executed in parallel and what is not) by using types of input data (PArray in this case), intermediate data (subs: PArray[PArray[Int]]) and also by using combinators over «parallel» data types (map, partition).

Note also how we use polymorphic method concat declared as

```
def concat[A:Elem](a: PArray[PArray[A]]): PA[A]
```

It has an implicit annotation to express a requirement that type parameter A should be an instance of typeclass Elem[A]. We systematically use the techniques described in [8] to introduce polytypism in our DSL.

3. Foundations of our approach

3.1. Polymorphic Embedding of DSLs

It is well known that a domain specific language (DSL) can be embedded in an appropriate host language [15]. 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.

In *pure embedding* (or *shallow embedding*) the domain types are directly implemented as host language types, and domain operations as host language functions on these types. This approach is similar to the development of a traditional library, but DSL approach emphasizes the domain semantics: concepts and operations in the design and implementation.

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, the pure embedding approach cannot utilize domain semantics for optimization purposes because of tight coupling of the host language and the embedded one.

Recently, *polymorphic embedding* – a generalization of Hudak's approach – was proposed [14] 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.

3.2. Polymorphic Embedding in our DSL

Considering advantages of the polymorphic embedding approach we employ it in our design. For details we refer to [14]. Conside the following example

```
type Rep[A]
trait PArray[A]
type SparseVector = PArray[(Int,Float)]
type Vector = PArray[Float]
def sparseVectorMul(sv: Rep[SparseVector], v: Rep[Vector]) =
    sum(sv map { case Pair(i,value) =>  v(i) * value })
```

On the DSL level we use Scala's abstract types and type constructors as domain types. Moreover, we lift all the functions over abstract type constructor Rep. This is important because later we can provide concrete definitions yielding specific implementations.

Our sequential implementation (we call it *simulation*) is implemented by defining Rep as

```
type Rep[A] = A
```

and in our staged implementation (we call it *code generation*) is implemented by defining Rep as

```
type Rep[A] = Exp[A]
```

where Exp is a representation of terms evaluating to values of the type A. Later we will see how it is defined in LMS framework.

The ultimate goal is to expose Scalan as a polymorphically embedded DSL in the Scala language in such a way that the same code could have two different implementations with equivalent semantics. And thus we would benefit from both simulation (evaluation for debugging) and code generation (for actual data processing).

3.3. Phantom types

In addition to the polymorphic embedding technique, we also need a couple of others that were recently developed in the area of *generic programming*. We shall briefly overview them here starting with the notion of *Phantom Types* [7,12]. Consider the definition of a data type. (in a Haskell-like notation) shown in Fig. 3.

```
data Term τ =
  Zero                with τ = Int
| Succ (Term Int)     with τ = Int
| Pred (Term Int)     with τ = Int
| IsZero (Term Int)   with τ = Bool
| If (Term Bool) (Term α) (Term α) with τ = α
```

Fig. 3. Term as phantom type

Types defined this way have some interesting properties:

- `Term` is *not* a container type: an element of `Term Int` is an expression that evaluates to an integer; it is not a data structure that contains integers.
- We cannot define a mapping function `(α -> β)-> (Term α -> Term β)` as for many other data types.
- The type `Term β` might not even be inhabited: there are, for instance, no terms of type `Term String`

It has been shown [12] that phantom types appear naturally when we need to represent types as data at runtime. In our DSL we make use of phantom types to represent types of array elements (see Fig. 9) and staged values (see section 4.1).

3.4. Generic programming

Runtime type representations are a foundation of generic programming technique [11]. The idea is to define a data type whose elements (instances) represent types of data that we want to work with. *A Generic Function* is one that employs runtime type representations and is defined by induction on the structure of types.

Consider the definition of the data type `Type.`

```
data Type τ =
  RInt   with τ = Int
| RChar  with τ = Char
| RPair (Type α) (Type β) with τ = (α, β)
| RList (Type α) with τ = [α]
```

An element rt of type Type τ is a representation of τ.
For example, following is a representation of type String.

```
rString :: Type String
rString = RList RChar
```

A generic function pattern matches on the type representation and then takes the appropriate action.

```
data Bit= 0|1
compress :: forall τ.Type τ -> τ -> [Bit]
compress (RInt) i  = compressInt i
compress (RChar) c = compressChar c
compress (RList ra) [ ] = 0:[]
compress (RList ra) (a : as) =
    1 : compress ra a ++ compress (RList ra) as
compress (RPair ra rb) (a, b) =
    compress ra a ++ compress rb b
```

We assume here that compressInt::Int->[Bit] and compressChar :: Char -> [Bit] are given.

3.5. Generic programming in Scala

Functions like this can be encoded in Scala using an approach suggested in [21]. Fig. 4 shows Scala encodings for the above function compress.

```
trait Rep[A]
implicit object RInt extends Rep[Int]
implicit object RChar extends Rep[Int]
case class RProd[A,B](ra:Rep[A], rb:Rep[B])
   extends Rep[(A,B)]
implicit def RepProd[A,B](
   implicit ra:Rep[A], rb: Rep[B]) = RProd(ra, rb)
def compress[A](x:A)(implicit r:Rep[A]):List[Bit]= r match {
  case RInt => compressInt (x)
  case RChar => compressChar (x)
  case RProd(a, b) => compress(x._1)(a) ++ compress(x._2)(b)
}
```

Fig. 4. Generic function in Scala

In the DSL we use a similar technique and we also employ type representations to implement array combinators as generic functions. But because parallel arrays that we discuss here are all implemented using type-indexed types (also known as non-parametric representations) we follow a different pattern to introduce generic functions in our library.

3.6. Type-indexed data types

A *type-indexed data type* is a data type that is constructed in a generic way from an argument data type. It is a generic technique and we briefly introduce it here adapted for our needs. For a more thorough treatment the reader is referred to [13].

In our example, in the case of parallel arrays, we have to define an array type by induction on the structure of the type of an array element.

Suppose we have a trait PArray[T] (to represent parallel arrays) and convenience type synonym PA[T] defined as

```
trait PArray[A]  // here PArray stands for Parallel Array
type PA[A] = PArray[A]
```

For this abstract trait we want to define concrete representations depending on the underlying structure of type A.

First, let us define what types can be used as types of array elements. As shown in Fig. 5 we consider a family of types constructed by the recursive definition:

```
A,B = Unit | Int | Float | Boolean // base types
| (A,B)                 // product (pair of types)
| (A|B)                 // sum type where (A|B) = Either[A,B]
| PArray[A]             // nested array
```

Fig. 5. Family of element types

Thus, considering each case in the definition above, we can define a representation transformation function RT (see Fig. 6) that works on types. It was shown [4] how such array representations enable nested parallelism to be implemented in a systematic way

```
RT: * -> *
RT⟦PArray[Unit]⟧  = UnitArray(len:Int)
RT⟦PArray[T]⟧    = BaseArray(arr:Array[T])
                    where T = Int | Float | Boolean
RT⟦PArray[(A,B)]⟧= PairArray(a:RT⟦PArray[A]⟧,
                             b:RT⟦PArray[B]⟧)
RT⟦PArray[(A|B)]⟧ = SumArray(
               flags: RT⟦PArray[Int]⟧,
               a: RT⟦PArray[A]⟧,
               b: RT⟦PArray[B]⟧)
RT⟦PArray[PArray[A]]⟧ = NArray(
               values  : RT⟦PArray[A]⟧,
               segments: RT⟦PArray[(Int,Int)]⟧)
```

Fig. 6. Representation Transformation

Below we show how to use Scala's *case classes* to represent structure nodes of a concrete representation (UnitArray, BaseArray, etc.) and how to keep the data values (data nodes) unboxed in Scala arrays (Array[A]). A graphical illustration of these representations is shown on in Fig. 7. For details related to these representations we refer to [4].

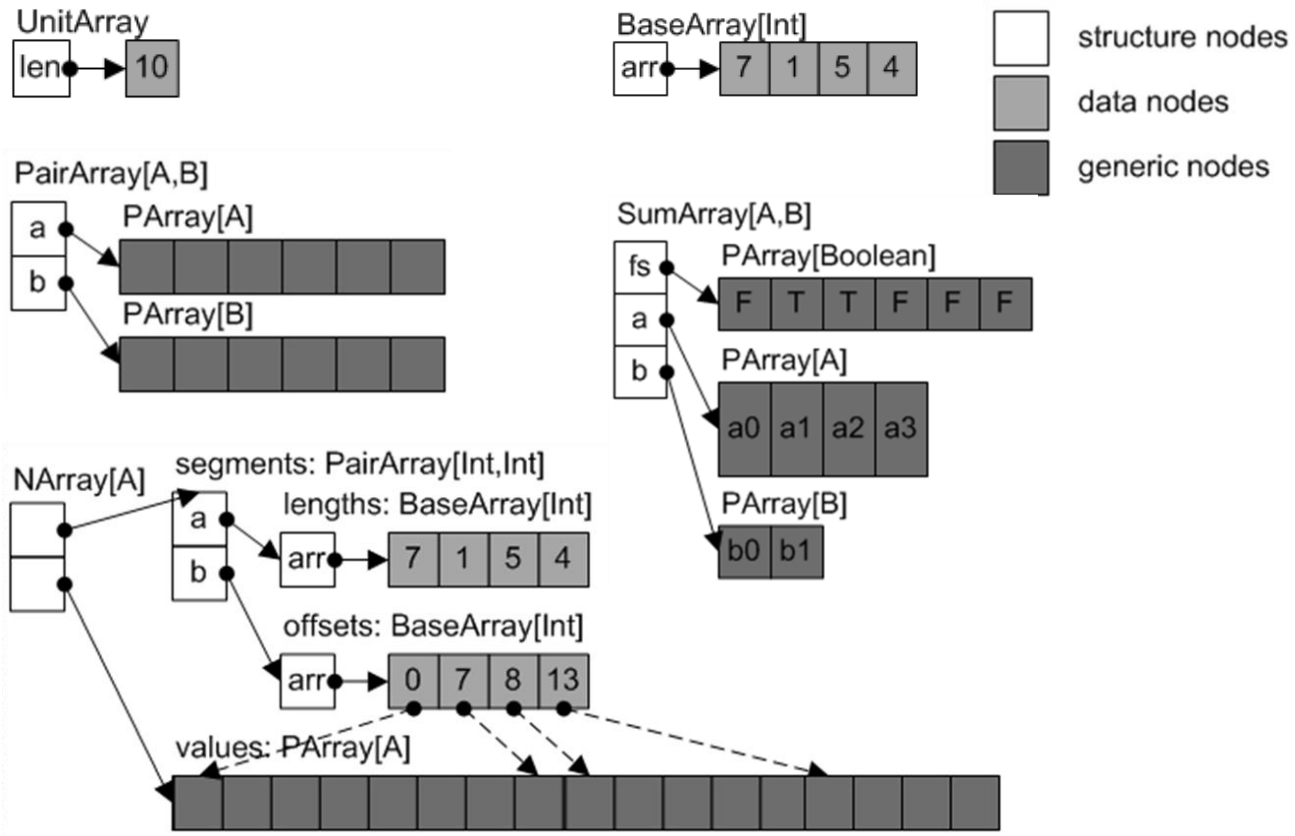

Fig. 7. Type-indexed representations of PArray

Consider as an example a representation of a sparse matrix rendered by applying RT function to Matrix type. It is shown graphically in Fig. 8.

```
type VectorElem = (Int,Float)
type SparseVector = PArray[VectorElem]
type Matrix = PArray[SparseVector]
```

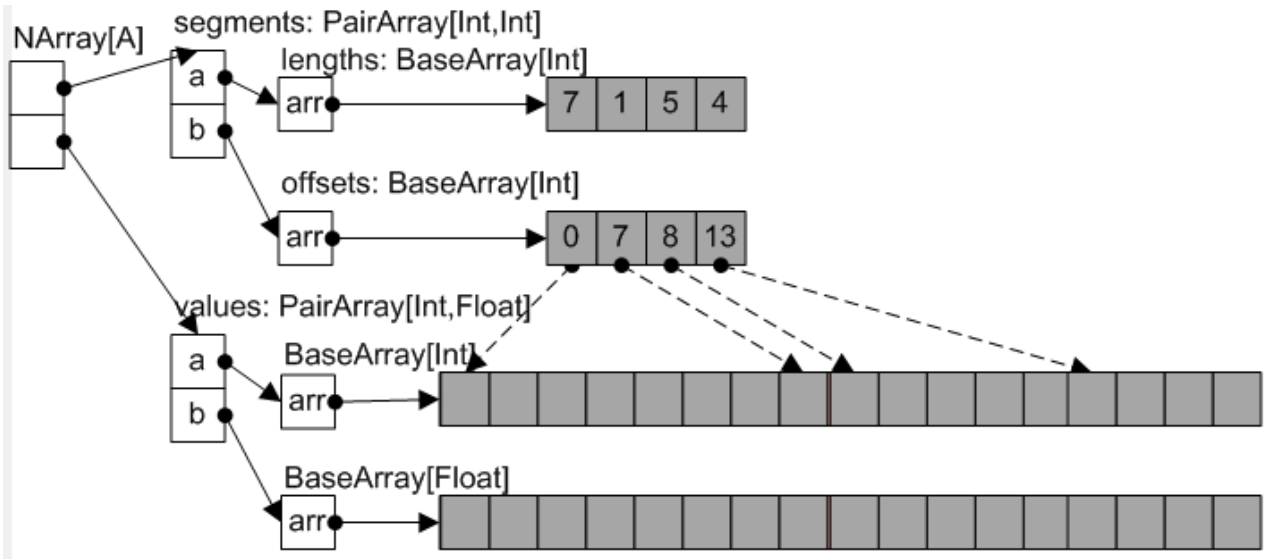

Fig. 8. Sparse matrix representation

3.7. Type-indexed arrays in the DSL's implementation

To employ the above techniques in the design of our DSL lets first represent a type structure of an array element type by using the Scala encodings of generic functions described above (see [23] for details).

```
type Elem[A] = Element[A] // convenience type synonim
trait Element[A] { // abstract type descriptor for type A
  def replicate(count: Int, v: A): PA[A] // factory methods
  def fromArray(arr: Array[A]): PA[A]
}
class BaseElement[T] extends Element[T] {
  def fromArray(arr:Array[T]) = BaseArray(arr)
  def replicate(len:Int,v:T) = BaseArray(Array.fill(len)(v))
}
implicit val unitElement: Elem[Unit] = new UnitElement
implicit val intElement:  Elem[Int]  = new BaseElement[Int]
implicit val floatElement: Elem[Float] =
                                    new BaseElement[Float]
implicit def pairElement[A,B]
  (implicit ea: Elem[A], eb: Elem[B]) = new Element[(A,B)] {
  def replicate(count: Int, v: (A,B)) =
        PairArray(ea.replicate(count, v._1),
                  eb.replicate(count, v._2))
}
```

Fig. 9. Representation of the types of array elements

Note, that in Scala we can equip type representations with generic functions (`replicate` in this sample) by using inheritance. Moreover, we can use a concrete array representation (PairArray) in the implementation for a particular type case (pairElement). All these lead to a fully generic while still statically typed code.

Next, we need to represent arrays as type-indexed data types thus implementing the RT function defined in previous section. Consider the code shown in Fig. 10.

```
type PA[A] = PArray[A]  // convenience type synonim
trait PArray[A]
case class UnitArray(len: Int) extends PArray[Unit]
case class BaseArray[A:Elem](arr: Array[A])
   extends PArray[A]
case class PairArray[A:Elem,B:Elem](a: PA[A], b:PA[B])
   extends PArray[(A,B)]
case class NArray(values:PA[A], segments:PA[(Int,Int)])
   extends PArray[PA[A]]
```

Fig. 10. Concrete array classes

To define generic (polytypic) functions over our arrays we first declare them in the PArray trait:

```
trait PArray[A] {
  def length: Int
  def map[B:Elem](f: A => B): PA[B]
  /* and other methods */
}
```

And then we implement these abstract methods in concrete array classes. Note how the implementation changes depending on the type of an array element.

```
case class UnitArray(len: Int) extends PArray[Unit] {
  def length = len
  def map[B:Elem](f: Unit=>B) =
     element[B].tabulate(len)(i => f())
}
case class BaseArray[A:Elem](arr: Array[A])
   extends PArray[A] {
  def length = arr.length
  def map[B:Elem](f: A => B) =
     element[B].tabulate(arr.length)(i => f(arr(i)))
}
case class PairArray[A:Elem,B:Elem](a:PA[A], b:PA[B])
   extends PArray[(A,B)] {
  def length = a.length
  def map[R:Elem](f: ((A,B)) => R) =
     element[R].tabulate(length)(i => f(a(i),b(i)))
}
case class NArray[A:Elem](arr:PA[A],segments: PA[(Int,Int)])
   extends PArray[PArray[A]] {
  def length = segments.length
  def map[R:Elem](f: PA[A] => R): PA[R] =
     element[R].tabulate(length)(
           i => {val (p,l) = segments(i); f(arr.slice(p,l))}
         )
}
```

Fig. 11. Polytypic PArray methods

3.8. Lightweight Modular Staging (LMS)

So far, given a type A of an array element we know how to build a type-indexed representation of the array using RT function thus yielding RT⟦PA[A]⟧ type. Next, we have seen how to encode in our DSL these array representations together with polytypic operations on them. These techniques form the basis of our sequential reference implementation of nested data parallelism (as described in [23]).

As it was mentioned before, our sequential implementation is straightforward, inefficient and is supposed to be used for debugging (in the aforementioned simulation mode). To enable a parallel and efficient implementation, we employ a polymorphic embedding technique, namely its particular instance of it known as *Lightweight Modular Staging (LMS)* [24]

In the name LMS, *Lightweight* means that it uses just Scala's type system. *Modular* means that we can choose how to represent intermediate representation (IR) nodes, what optimizations to apply, and which code generators to use at runtime. And *Staging* means that a program instead of executing a value, first, produces other (optimized) program (in form of a program graph) and then executes that new program to produce the final result.

Consider the method matrixVectorMul in Fig. 1. and types Matrix, Vector that were used in its declaration. That is how we usually write methods in our programs. Instead of this, in LMS framework, in order to express staging, we are required to **lift**

some types using the type constructor Rep[_] and use Rep[Matrix], Rep[Vector], etc. In fact, sparseVectorMul should have been declared like this to enable polymorphic embedding

```
def sparseVectorMul(
      sv: Rep[SparseVector], v: Rep[Vector]): Rep[Float] =
    sum(sv map { case Pair(i,value) =>  v(i) * value })
```

In the case of sequential implementation we define Rep as

```
type Rep[A] = A
```

and provide sequential implementation using a usual evaluation semantics of the host language (i.e. Scala).

On the other hand, LMS is a staging framework and we want to build IR instead of just evaluating the method. To achieve this, LMS defines Rep as shown in Fig. 12.

```
trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
}
trait Expressions {
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]
  abstract class Def[T] // operations (defined in subtraits)

  class TP[T](val sym: Sym[T], val rhs: Def[T])
  var globalDefs: List[TP[_]] = Nil
  def findDefinition[T](d: Def[T]): TP[T] =
        globalDefs.find(_.rhs == d)
  def findOrCreateDefinition[T](d: Def[T]): TP[T] =
        findDefinition(d).getOrElse{
          createDefinition(fresh[T],d)
        }
  implicit def toExp[T](x: T): Exp[T] = Const(x)
  implicit def toExp[T](d: Def[T]): Exp[T] =
      findOrCreateDefinition(d).sym
}
```

Fig. 12. How Rep[T] defined in LMS

This, in effect, enables lifting of the method' bodies too, so that its evaluation yields a program graph.

Lifting of expressions is performed when the code is compiled using Virtualized Scala [2]. For example, consider the following lines of code:

```
val x: Rep[Int] = 1
val y = x + 1
```

There is no method '+' defined for Rep[Int], but we can define it on DSL level without providing any concrete implementation as follows:

```
trait IntOps extends Base {
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

When such a declaration is in the scope of `x+1` then '+' is replaced by Scala compiler with `infix_+(x, toExp(1)).`

In a staging context infix_+ is defined so that it generates an IR node of the operation

```
trait IntOpsExp extends BaseExp with IntOps {
  case class IntPlus(x:Exp[Int],y:Exp[Int]) extends Def[Int]
  def infix_+(x: Exp[Int], y: Exp[Int]) = IntPlus(x,y)
}
```

Here IntPlus is an IR node that represents '+' in the program graph. Note that infix_+ should return Rep[Int] while IntPlus extends Def[Int], so implicit conversion

```
implicit def toExp[T](d: Def[T]): Exp[T] =
      findOrCreateDefinition(d).sym
```

which is defined in Expressions trait is called here thus providing graph building machinery.

We refer to [24] for detailed explanation of how LMS works.

4. Polytypic Staging

We have shown that for each type A of array element we use the type representation function RT to build type-indexed representation of PArray[A] type. We also showed how we define PArray's methods using polytypic techniques so that once defined they work for all types in the family. Thus, emphasizing the domain-specific nature of our library and considering its polytypic design we can think of it as a *polytypic DSL*.

If we want to *deeply* embed our Polytypic DSL in Scala by applying Polymorphic Embedding techniques in general and LMS framework in particular we need to answer the question: "*How are we going to lift type-indexed types along with polytypic functions in the Rep world?*"

In this section we describe Polytypic Staging, – our approach to "deep embedding" of Polytypic DSLs. By design, our framework:

- is an extension of LMS
- respects type-indexed representations described before
- adds additional dimension of flexibility to LMS by making it polytypic
- behaves as core LMS in the non-polytypic case

4.1. Staged Values

To be consistent with LMS, we do not change the original definition of Rep, but we need to make some extensions to account for a polytypic case, they are shown in the following listing in italicized bold.

```
type Rep[T] = Exp[T]
abstract class Exp[+T] {
  def Type: Manifest[T] = manifest[T]  // in LMS
  def Elem: Elem[T] // added in Scalan
}
case class Sym[T:Elem](val id: Int) extends Exp[T] {
  override def Elem = element[T]
}
case class Const[+T:Manifest](x: T) extends Def[T]
def element[T] = implicitly[Element[T]]
```

These additions ensure that each staged value has a runtime type descriptor that we use to implement polytypism. We also regard constants as definitions (more precisely as operations of arity 0), and we can do it without a loss of generality since given a symbol we can always extract its right-hand-side definition by using Def extractor [20] defined in core LMS

```
object Def {
  def unapply[T](e: Exp[T]): Option[Def[T]] = e match {
    case s @ Sym(_) => findDefinition(s).map(_.rhs)
    case _ => None
  }
}
```

Treating constants as definitions in our implementation of LMS means that any lifted value of type Rep[T] is always an instance of Sym[T] which simplifies our implementation.

4.2. Type Descriptors

The descriptors of types of array elements shown in Fig. 9 remain unchanged. This means that we can keep our type representation schema with one adaptation: we need to lift all the methods of Element[T] trait.

```
type Elem[A] = Element[A]
trait Element[A] {
  def replicate(count: Rep[Int], v: Rep[A]): PA[A]
  def fromArray(arr: Rep[Array[A]]): PA[A]
}
class BaseElement[T] extends Element[T] {
  def fromArray(arr: Rep[Array[A]]) = BaseArray(arr)
  def replicate(len: Rep[Int], v: Rep[A]) =
           BaseArray(ArrayFill(len, v))
}
implicit val unitElement: Elem[Unit] =new UnitElement
implicit val intElement:  Elem[Int] = new BaseElement[Int]
implicit val floatElement:Elem[Float]=new BaseElement[Float]
implicit def pairElement[A,B]
  (implicit ea: Elem[A], eb: Elem[B]) = new Element[(A,B)] {
  def replicate(count:Rep[Int], v:Rep[(A,B)]):PA[(A,B)] =
         PairArray(ea.replicate(count, v._1),
                   eb.replicate(count, v._2))
}
```

Fig. 13. Staged representation of types

Note that even after the lifting of the methods their bodies remain literally the same. This is achieved by systematic use of Rep[T] type constructor in signatures of classes and methods. We also employ Scala idiom known as the "pimp my library" pattern to add methods that work with values lifted over Rep[T]. For example, consider expressions v._1 and v._2 in Fig. 13, whose implementation is shown in Fig. 14.

```
def unzipPair[A,B](p: Rep[(A,B)]): (Rep[A],Rep[B]) = p match {
  case Def(Tup(a, b)) => (a, b)
  case _ => (First(p), Second(p))
}
class PairOps[A:Elem,B:Elem](p: Rep[(A,B)]) {
  def _1: Rep[A] = { val (a, _) = unzipPair(p); a }
  def _2: Rep[B] = { val (_, b) = unzipPair(p); b }
}
implicit def pimpPair[A:Elem,B:Elem](p: Rep[(A,B)]) = new PairOps(p)

case class Tup[A,B](a: Exp[A], b: Exp[B])
   extends Def[(A,B)]
case class First[A,B](pair: Exp[(A,B)])
   extends Def[A]
case class Second[A,B](pair: Exp[(A,B)]) extends Def[B]
```

Fig. 14. Staging methods using Pimp my Library pattern

Note how we use the core LMS's Def extractor to implement staging logic. Given lifted pair (`p: Rep[(A,B)]`) we either successfully extract Tup(a,b) constructor and return original constituents of the pair, or we emit new IR nodes thus deferring tuple deconstruction until later stages.

Figures above show how we implement our polytypic staging framework on top of core LMS, but as we will see in the next section, to lift type-indexed representations of PArray[A] over Rep[_] and to stage polytypic array methods we still need to introduce some extensions above core LMS.

4.3. Staging type-indexed data types

Polytypism in our DSL is focused around the PArray[A] trait (which on the DSL level represents parallel arrays) and every value of the PArray[T] type has a type-indexed representation that is built by induction on the structure of T. We also extensively use a convenience type synonym PA defined as follows:

```
trait PArray[A]
type PA[A] = Rep[PArray[A]]
```

Thus, PA is no longer a synonym of PArray and now it is a synonym of lifted PArray. In other words PA[T] is a lifted value of array with elements of type T.

Let us use the example in Fig. 13 to illustrate how values of the type PArray are staged (or lifted) in our polytypic staging framework. First, notice that `replicate` method of pairElement produces a value of `PA[(A,B)]` type which is a synonym of `Rep[PArray[(A,B)]]` and so it is a lifted `PArray[(A,B)]` and in LMS such values are represented by symbols of type `Sym[PArray[(A,B)]]`. Thus having a

value of type `PA[(A,B)]` we can think of it as a value of some symbol of type `Sym[PArray[(A,B)]]`.

Next, recall that in LMS we get lifted values of the type Rep[T] by implicit conversion (recall that Rep[T] = Exp[T]):

```
implicit def toExp[T](d: Def[T]): Exp[T] =
                findOrCreateDefinition(d).sym
```

which is automatically inserted by the compiler, converts any definition to a symbol and builds a program graph as its side effect. We employ this design by deriving all classes that represent arrays from Def[T] with appropriate T. As an example see Fig. 13 where PairArray is returned by the method `replicate`. Definitions to represent arrays are shown in Fig. 15.

```
abstract class PADef[A] extends Def[PArray[A]]
                            with PArray[A]
case class UnitArray(len: Rep[Int]) extends PADef[Unit]
case class BaseArray[A:Elem](arr: Rep[Array[A]])
   extends PADef[A]
case class PairArray[A:Elem,B:Elem](a: PA[A], b: PA[B])
   extends PADef[(A,B)]
case class SumArray[A:Elem,B:Elem]
                   (flags: PA[Boolean], a: PA[A], b: PA[B])
   extends PADef[(A|B)]
case class NArray[A:Elem](arr:PA[A], segments:PA[(Int,Int)])
   extends PADef[PArray[A]]
```

Fig. 15. Array classes as graph nodes (Defs)

Compare these classes with those shown in Fig. 10. and note how class signatures became lifted either explicitly by using Rep[T] constructor or implicitly by redefining PA[T] synonym as `Rep[PArray[A]]`. Moreover, the type Representation Transformation function TR shown in Fig. 6 also remains almost the same, but works with lifted types (see Fig. 16). This similarity is due to the polymorphic embedding design of our approach where we want to give different implementations to the same code.

Note, how we mix-in the PArray[A] trait into every graph node of the type PADef[A]. In this way, when we stage (or lift over Rep) a type-indexed representation of PArray[T] we both create structure nodes using our concrete array classes and at the same time we build program graph nodes.

The Representation Transformation in a staged context is shown in Fig. 16.

```
L⟦UnitArray(len: Rep[Int])⟧       = Sym[PArray[Unit]]
L⟦BaseArray[T](arr:Rep[Array[T]])⟧ = Sym[PArray[T]]
                                    where T=Int|Float|Boolean
L⟦PairArray(a:PA[A], b:PA[B])⟧    = Sym[PArray[(A,B)]]
L⟦SumArray(flags:PA[Boolean],
           a:PA[A],b:PA[B])⟧      = Sym[PArray[(A|B)]]
L⟦NArray(values:PA[A],
         segments:PA[(Int,Int)])⟧ = Sym[PArray[PArray[A]]]

TR⟦PArray[Unit]⟧ = UnitArray(len:Rep[Int])
TR⟦PArrya[T]⟧   = BaseArray(arr:Rep[Array[T]])
                   where T = Int|Float|Boolean
TR⟦PArray[(A,B)]⟧ = PairArray(a:L⟦TR⟦PArray[A]⟧⟧,
                              b:L⟦TR⟦PArray[B]⟧⟧)
TR⟦PArray[(A|B)]⟧ = SumArray(
                flags: L⟦TR⟦PArray[Int]⟧⟧,
                a: L⟦TR⟦PArray[A]⟧⟧,
                b: L⟦TR⟦PArray[B]⟧⟧)
TR⟦PArray[PArray[A]]⟧ = NArray(
           values  : L⟦TR⟦PArray[A]⟧⟧,
           segments: L⟦TR⟦PArray[(Int,Int)]⟧⟧)
```

Fig. 16. Staged Representation Transformation

A graphical illustration of these representations in a form of program graph is shown in Fig. 17, where we use methods defined like this

```
def fromArray[T:Elem](x: Rep[Array[T]]): PA[T] =
        element[T].fromArray(x)
def replicate[T:Elem](count: Rep[Int], v: Rep[T]):PA[T]=
        element[T].replicate(count, v)
```

Note also that we can achieve effects of constant propagation and partial evaluation by applying domain-specific rewritings. Our experiments show that if all input data are known at staging time, our rewriting method, while simple enough, is still able to fully evaluate intermediate graph nodes resulting in a type-indexed representation that only contains data arrays in Const nodes and structure nodes that represent PArray values. (see Fig. 17)

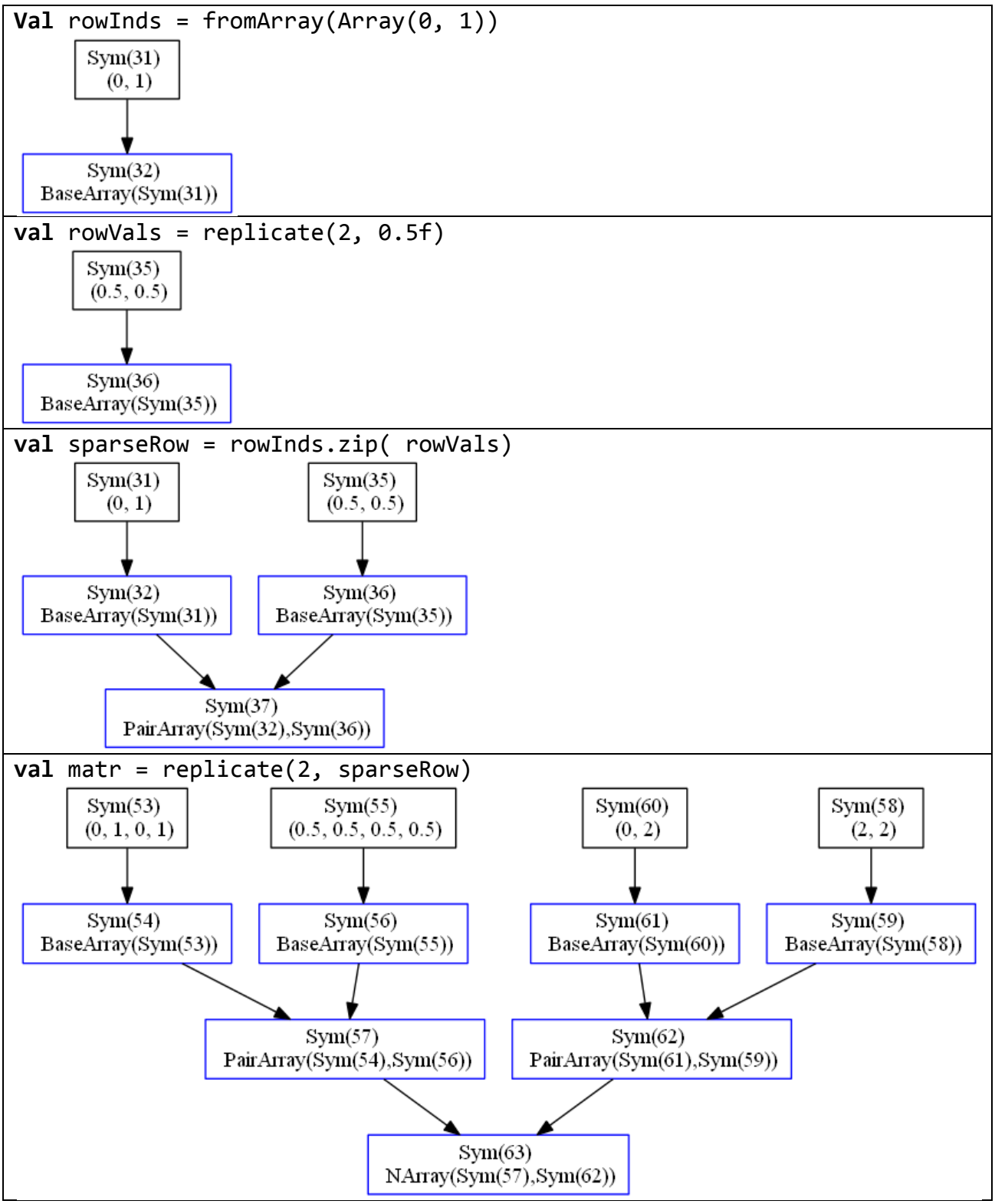

Fig. 17. Array constructors and the resulting graph

4.4. Staging polytypic functions

The same way as we lift methods in type descriptors (types derived from Element[T]) we can lift methods in concrete array classes (those derived from PArray[T]). We've already shown in Fig. 15 how we lift signatures of array classes, here is how we stage polytypic method `map` (see Fig. 18)

```
case class UnitArray(len: Rep[Int]) extends ... {
  def map[R:Elem](f: Rep[Unit] => Rep[R]): PA[R] =
      element[R].tabulate(len)(i => f(toRep(())))
}
case class BaseArray[A:Elem](arr: Rep[Array[T]]) extends ... {
  def map[B:Elem](f: Rep[A] => Rep[B]) =
      element[B].tabulate(arr.length)(i => f(arr(i)))
}
case class PairArray[A:Elem,B:Elem](a:PA[A], b:PA[B])
   extends ... {
  def map[R:Elem](f: Rep[(A,B)] => Rep[R]): PA[R] = {
    element[R].tabulate(length)(i => f(a(i),b(i)))
  }
}
case class NArray[A:Elem](arr:PA[A],segments:PA[(Int,Int)])
   extends ... {
  def map[R:Elem](f: PA[A] => R): PA[R] =
       element[R].tabulate(length)(i => {
     val Pair(p,l) = segments(i); f(arr.slice(p,l))
       })
}
```

Fig. 18. Staged polytypic method `map`

Compare this code with the non-staged version in Fig. 11 and note how the signature lifted over Rep and bodies of the methods remain literally unchanged.

As an example of staging something not very trivial, we show in Fig. 19 a program graph that we get by staging the following function:

```
val svm = (sv: Rep[SparseVector])=>(v: Rep[Vector]) =>
               sparseVectorMul(sv, v)
```

4.5. Generalizing Domain-Specific Rewrites

One of the benefits that we can get out of deep embedding is the ability to perform domain-specific optimizations. For instance we can use staging time rewrites. Our method of implementing rewrites is very simple and is based on the one proposed in [24]. We just slightly improve it by making it more general while still simple.

Recall that when a definition (an instance of Def class) should be converted to Rep, an implicit conversion defined by core LMS is inserted by the compiler to do the job. Fig. 20 shows how we redefine this conversion to enable rewritings.

Fig. 19. Program Graph

```
implicit def toExp[T:Elem](d: Def[T]): Exp[T] =
  findDefinition(d) match {
  case Some(TP(s, _)) => s
  case None =>
    val newD = rewrite(d)
    if (newD == d) {
        val TP(newSym, _) = createDefinition(fresh[T], d)
        newSym
    } else
        toExp(newD.asInstanceOf[Def[T]])
  }
def rewrite[T](d: Def[T]): Def[_] = d
```

Fig. 20. Generalized Rewritings

If we can find the definition in the graph, we just return its symbol. Otherwise, we try to rewrite the Def yielding a new one. If it is the same (which means that we cannot apply any rewriting rule), then we add the Def to the graph. If the rewriting returns something new, then we drop the original Def and go recursively with the new.

We have found that this iterative rewriting has to be continued until a fixed point is reached, since rewrite often happens to create a new definition that leads to the possibility of another rewrite. In particular, it is necessary in order to achieve the effects of partial evaluation.

We use stackable overrides to implement domain-specific rules in a modular way

```
trait StagedArithmetic extends  … {
  override def rewrite[T](d: Def[T]): Def[_] = d match {
    case FractionalMod(Def(Const(x)), Def(Const(y)), i) =>
      Const(i.quot(x, y))
    case _ => super.rewrite(d)
  }
}
trait StagedStdArrayOps extends … {
  override def rewrite[T](d: Def[T]): Def[_] = d match {
    case ArrayLength(Def(ArrayScan(arr, _))) =>
      ArrayLength(arr)
    case arrDef@ArrayFill(Def(Const(len)), Def(Const(v))) =>
      Const(Array.fill(len)(v))
    case _ => super.rewrite(d)
}
```

5. Conclusion

There are some interesting features of the nested data parallelism that make it attractive to research. First, – it has been shown (at least theoretically) that it admits an efficient implementation, second, - this model covers a wide class of algorithms of practical importance [3], and third, – it has a purely functional and deterministic semantics of the language and, as a consequence, enables programs to be written in declarative style.

Declarative languages are usually easier to use, because they allow the programmer to directly formulate what is to be done without specifying how it has to be done, while some decisions may be postponed even until the runtime.

Another interesting feature of NDP is compositionality (or, more generally, modularity). Once a program has been written, it can be repeatedly reused as a subroutine without modification.

In this paper we have made another step towards the effective implementation of the NDP model. We initially chose an approach and a development platform that is different from our predecessors and we put emphasis on limiting the degree of generality by formulating the problem as the development of DSL.

There are reasons to believe that by limiting the degree of generality, we can more easily use the domain semantics for implementing deeper and more significant optimizations. In addition, the LMS platform gives us some opportunities for integra-

tion with other DSLs [2,25], which, in turn, will allow us, by combining their semantics, to improve the depth of optimizations and performance.

Although the proposed method of *polytypic staging* is formulated in terms of the NDP domain, still, the implementation strategy described here should be applicable to any polytypic DSL in general.

6. Acknowledgements

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

7. References

[1] *General-Purpose Computation on Graphics Hardware*. http://gpgpu.org/.

[2] Philipp Haller Adriaan Moors, Tiark Rompf and Martin Odersky. *Tool Demo: Scala-virtualized*, 2012.

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

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

[5] 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.

[6] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. *Data Parallel Haskell: a status re- port.* In In DAMP 2007: Workshop on Declarative Aspects of Multicore Programming. ACM Press, 2007.

[7] James Cheney and Ralf Hinze. *Phantom types*, 2003.

[8] 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: Soft- ware for Humanity (SPLASH/OOPSLA), October 2010.

[9] 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.

[10] 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.

[11] Ralf Hinze. *A new approach to generic functional programming.* In Pro- ceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '00, pages 119–132, New York, NY, USA, 2000. ACM.

[12] Ralf Hinze. *Fun with phantom types.* The Fun of Programming, pages 245– 262, 2003.

[13] Ralf Hinze, Johan Jeuring, and Andres L¨oh. *Type-indexed data types.* In SCIENCE OF COMPUTER PROGRAMMING, pages 148–174, 2004.5

[14] 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.

[15] Paul Hudak. *Building domain-specific embedded languages.* ACM COM- PUTING SURVEYS, 28, 1996.

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

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

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

[19] NVIDIA. *NVIDIA CUDA C Programming Guide*., 2011.

[20] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Second Edition*. Artima, 2010.

[21] 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.

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

[23] Alexander Slesarenko. *Scalan: polytypic library for nested parallelism in Scala.* Preprint 22, Keldysh Institute of Applied Mathematics, 2011.

[24] Martin Odersky Tiark Rompf. *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls*, 2010.

[25] Arvind Sujeeth and HyoukJoong Lee and Kevin Brown and Tiark Rompf and Hassan Chafi and Michael Wu and Anand Atreya and Martin Odersky and Kunle Olukotun. *OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning.* ICML '11

[26] Eclipse. http://eclipse.org