Abstract
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\(^1\).

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Design, Languages

Keywords Generic programming, Polytypic programming, Polytypic staging, Nested Data Parallelism, Multi-stage programming, Domain-specific languages, Language Virtualization

1. Introduction
A long-standing trend in software development for parallel computing is the reduction of complexity, namely the development of easy-to-use languages and libraries \([8, 9, 24]\), encapsulation of complexity in an implementation of system software \([21]\), creation of interactive working environments \([1]\).

In particular, it was shown \([26, 28]\) that a combination of a DSL approach and program staging is a promising direction of work where sufficient performance optimizations of staged code were achieved by exploiting the domain-specific semantics. So staging is a key point of the DSL optimizations.

But what if the DSL for our domain can be naturally implemented by using polytypic (generic) programming techniques? How can we stage generic code? Interestingly, this is the case when we consider nested data parallelism (NDP) as the domain. In our previous work \([27]\) we developed an embedded polytypic DSL for expressing nested data parallel algorithms in the Scala language by using generic programming \([12]\) (polytypic programming \([17]\)) techniques.

In this paper, we describe an attempt to stage our polytypic DSL, hence we term this as polytypic staging. The implementation is lightweight in a sense that it is based on an expressive type system of the Scala language.

The idea behind our approach is based on a combination between Lightweight Modular Staging (LMS) \([25]\) and polytypic (datatype-generic) programming. The idea is that by writing programs using a polymorphic embedding style \([14]\), programs can be interpreted in two modes: simulation and code generation. In the simulation mode programs are given an unoptimized (and slow), but straightforward implementation which is good for testing. In the code generation mode, a fast, optimized implementation is generated at run-time. Datatype-generic programming techniques are then applied to allow the library to be specialized with user-specific datatypes (built out of arrays, sums and products) by providing isomorphic views types \([15]\). Term rewriting techniques can be applied on the staging (code generation) phase to perform generic and domain specific optimizations.

For domain specific foundations we rely on a series of publications \([4, 19, 20]\) on the nested data parallelism model. 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 \([5]\). On the other hand, many techniques and technologies \([2, 7, 14, 23, 25]\), which we use as a foundation of our approach, appeared only in recent years so it is an interesting research question to restate the problem and implement the model in a new environment.

---

\(^1\) This work is partially supported by Russian Foundation for Basic Research project No. 12-01-00972-a
We propose our implementation of NDP as a DSL embedded in Scala using the Scala-Virtualized compiler and packaged as a library. We compare it with a Parser Combinators library which also has limited expressiveness and inherent composability, while still having a wide range of applications in different problem domains.

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.

In summary, this paper makes the following main contributions:

1. We extend our previously published work \[27\] by introducing type-directed Lightweight Polytypic Staging technique (LPS).
2. We describe how to extend the 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 our framework is able to support user-specific data types by providing isomorphic representations.
4. We show how to apply Lightweight Polytypic Staging to a special problem domain of nested data parallelism.

In this paper we also describe some aspects of the design and implementation of the Scalan library.

1.1 The DSL

We start with some examples of the DSL to illustrate the basic ideas of the NDP domain from user’s perspective.\(^2\)

Consider the definition of \(svm\) in Fig. 1. We represent a sparse vector as an array of pairs where the integer value of the pair represents the index of an element in the vector and the float value of the pair 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.

Instead of using the ordinary Array[T] type we use an abstract PArray[T] trait and by doing that, first, make the code abstract, and second, express our intent for a parallel evaluation.

When it comes to multiplying a sparse matrix with a dense vector, we can reuse our previously defined parallel function \(svm\) to define a new parallel function \(smvm\). The essence of nested data parallelism is the ability to nest one parallel map inside another parallel map. On the other hand, it supports flattening that makes it possible to automatically transform nested code into flat data parallel form which is better for execution. And that is the reason why we need staging in this domain in first place, to be able to perform transformations.

We are free, up to a family of product, sum and PArray type constructors (see Fig. 2), to define data types and in fact it is our responsibility as a programmer to define them properly. It is our choice here to represent sparse matrix as a parallel array of sparse vectors and not dense ones (as they can have considerably different memory usage characteristics). But what the polytypic DSL gives us is that for any data type we define it provides us with the specialized underlying data structure that is built in a generic way from the type definition (see section 2.4).

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

Figure 2. Family of element types

We can also use a parallel function inside its own definition i.e. recursively. Fig. 3 shows how the QuickSort recursive algorithm can be expressed in the NDP model.

```
trait PArray[A]
type VectorElem = (Int,Float)
type SparseVector = PArray[VectorElem]
type Vector = PArray[Float] // dense vector
type Matrix = PArray[SparseVector] // sparse matrix

// sparse vector
def svm(sv: SparseVector, v: Vector): Float = sum(sv map { case Pair(i,value) ⇒ v(i) * value})

// sparse matrix
def smvm(matr: Matrix, vec: Vector): Vector = for (row <- matr) yield svm(row, vec)
```

Figure 3. Parallel QuickSort

\(^{2}\) The complete code is available at http://github.org/scalan to supplement the paper.

\(^{3}\) We extensively use Scala listings in the paper and assume familiarity with the language.\(^2\) We only show parts of the code relevant to our discussion and refer to our previous paper \[27\] for details of the library design and more samples.
The DSL is purely functional, sequential and deterministic. The program can be thought of as being executed by the vector virtual machine where each array primitive is executed as one data-parallel step. We express parallelism (what we want to be executed in parallel and what we don’t) by using types of an input data (PArray in this case), intermediate data (i.e. suba which has type PA[PA[Int]]) and also by using combinators over parallel data types (map, partition).

Note how partition increases the nesting level so that we can express the idea that both partitions should be executed in parallel using map. And then results are combined back in a flat array by concat which has the following type

\[
def \text{concat}[A : \text{Elem}](a : \text{PA}[\text{PA}[A]]) : \text{PA}[A]
\]

The point is that concat is a constant-time operation, and that is possible because the representation of the type PA[PA[A]] is specially chosen to support this. You can look at the Fig. 1 and probably guess how concat is implemented.

The implicit annotation A : Elem expresses a requirement that the type parameter A should be an instance of the type class Elem[A]. It means, as we will see later, that A is either built by using products, sum, and PArray constructors, or it is a user-specific data type isomorphic to some B : Elem. It is not just any user defined Scala type but any Scala type can be made into an instance of type-class Elem by providing an isomorphism.

We systematically use the techniques described in [7] to implement polytypism in our DSL. In particular, in section 2.1 we will see how to define generic functions once and for all.

### 1.2 Adding More Types

If we limit the typing capabilities of the DSL to just the types shown in Fig. 2 we will be possible to cover many practical cases. It is limited approach though, since we cannot define recursive data types in this way due to the limitations imposed by the Scala language itself. And it is not convenient for the user.

To both overcome this limitation and increase typing capabilities of the DSL we make it possible to extend the family of types shown in Fig. 2 with any user-specific types defined in Scala. The key point is to be able to make any such type U an instance of the type-class Elem. The idea is to define a canonical isomorphism (iso for short) between U and some existing instance A : Elem. This finally ensures that every user-specific type is represented by an isomorphic view type [15]. It suffices to define a function on view types (and primitive or abstract types such as Int and Boolean) in order to obtain a function that can be applied to values of arbitrary data types.

Consider as an example the definition of the Point type shown in Fig. 4. Given a user-specific type (Point in this case) all we need to do is to define an instance of Iso[A, B] type-class (see IsoPoint) witnessing that Point is canonically representable in terms of already defined instances of the Elem type-class.

\[
case \text{class Point}(x : \text{Int}, y : \text{Int})
\]

\[
\text{implicit object IsoPoint}
\begin{align*}
\text{extends Iso[(Int, Int), Point]} & \{ \\
\text{def to } & = (p : (\text{Int}, \text{Int})) \Rightarrow \text{Point}(p._1, p._2) \\
\text{def from } & = (p : \text{Point}) \Rightarrow (p.x, p.y)
\}
\end{align*}
\]

\[
def \text{distance}(p1 : \text{Point}, p2 : \text{Point}) : \text{Float} = \\
\text{val dx } = p2.x - p1.x \\
\text{val dy } = p2.y - p1.y \\
\text{sqrt(dx} * \text{dx + dy} * \text{dy})
\]

\[
def \text{minDistance}(ps : \text{PArray[Point]}) : \text{Float} = \\
\text{min(for } (p \leftarrow \text{ps}) \text{ yield distance}(\text{Point}(0,0), p))
\]

\[
case \text{class Circle}(\text{loc : Point}, r : \text{Int})
\]

\[
\text{implicit object IsoCircle}
\begin{align*}
\text{extends Iso[(Point, Int), Circle]} & \{ \\
\text{def to } & = (c : (\text{Point}, \text{Int})) \Rightarrow \text{Circle}(\text{c._1, c._2}) \\
\text{def from } & = (c : \text{Circle}) \Rightarrow (\text{c.loc, c.r})
\}
\end{align*}
\]

![Figure 4. User-specific data type](image-url)

Once the Point type is made an instance of the Elem type-class via isomorphism it can in turn be used to both define other user-specific types and participate in the isomorphisms definitions for those types as it is shown in Fig. 4. We describe the design of these features in section 3.

In our polytypic staging framework we are able to give both evaluation and staging interpretation of all the examples discussed so far. This is described in sections 3 and 4.

### 2. Foundations of our approach

#### 2.1 Polymorphic Embedding of DSLs

It is well known that a domain specific language (DSL) can be embedded in an appropriate host language [10]. 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 the DSL approach emphasizes the domain semantics: concepts and operations of the domain in the design and implementation of the library.

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 modu-
lar 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.

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

Conside the following example

```scala
type Rep[A] = Exp[A]
trait PArray[A]
type SparseVector = PArray[(Int,Float)]
type Vector = PArray[Float]
```

On the DSL level we use product, sum and PArray type constructors to express domain types (see SparseVector). We lift all the functions over abstract type constructor Rep. This is important because later we can provide concrete definitions yielding specific implementations.

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

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

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

```scala
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 develop 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).

### 2.2 Generic programming

In addition to the polymorphic embedding techniques, 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 [6, 11].

Consider the definition of a data type (in a Haskell-like notation) shown in Fig. 5.

```scala
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]
```

**Figure 5.** Type descriptors as phantom types

Types defined this way have some interesting properties:

- Type is not a container type: an element of Type Int is a runtime representation of type Int; it is not a data structure that contains integers.
- We cannot define a mapping function (α → β) → (Type α → Type β) as for many other data types.
- The type Type β might not even be inhabited: there are, for instance, no type descriptors of type Type String

It has been shown [11] 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 as runtime data (see Fig. 10) and staged values (see section 3).

Runtime type representations have been proposed as a lightweight foundation of generic programming techniques [10]. 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 by pattern matching on the type representation and then taking the appropriate action.

```scala
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) =
  i : compress ra a ++ compress (RList ra) as
compress (RPair ra rb) (a, b) =
  compress ra a ++ (compress rb b)
```

We assume here that two functions are given

```scala
compressInt :: Int → [Bit]
compressChar :: Char → [Bit]
```

### 2.3 Generic programming in Scala

Generic functions can be encoded in Scala using an approach suggested in [23]. Fig. 6 shows the encodings in Scala for the above function compress.

Traditionally, generic (polytypic) functions are defined for a family of types built out of sums and products. We add PArray to the family of representation types. Definition of a generic function should be given for each representation type.

---

5 We could have used a more general notion of GADT [13] but we stick with phantom types as they are simpler and well enough for our presentation.

6 The definition of compress for the case RList is straightforward and we leave it as an exercise.
trait Rep[A]
implicit object RInt extends Rep[Int]
implicit object RChar extends Rep[Char]
case class RPair[A,B](ra:Rep[A], rb:Rep[B])
  extends Rep[(A,B)]
implicit def RepPair[A,B](implicit ra:Rep[A], rb:Rep[B]) = RPair(ra,rb)
def compress[A](x:A)(implicit r:Rep[A]):List[Bit] =
  r match {
    case RInt ⇒ compressInt (x)
    case RChar ⇒ compressChar (x)
    case RPair(a, b) ⇒
      compress(x._1)(a) ++ compress(x._2)(b)
  }

Figure 6. Generic function in Scala

as shown in Fig. 6. For all other types it is usually required to give an isomorphic representation of the type in terms of the above fixed set of constructors. We give an account of isomorphic representations in section 4.

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

2.4 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 construct the representation of a parallel array (PArray) by induction on the structure of the type of its element.

As we said, we use a trait PArray[T] to represent parallel arrays in the DSL and suppose we also have convenience type synonym PA[T] defined as

trait PArray[A] // 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 the type A of the array elements. As shown in Fig. 7 we consider a family of types constructed by the limited set of type constructors.

Thus, considering each case in the definition above, we can define a representation transformation function $RT$ (see Fig. 7) that works on types. It was shown [4] how such array representations are crucial for an implementation of NDP.

In Fig. 8 we show a graphical illustration of the concrete representations induced by the function $RT$. We use Scala’s case classes to represent structure nodes of a concrete representation (UnitArray, BaseArray, etc.) and we keep the data values (data nodes) unboxed in Scala arrays (Array[A]) for each base type A. For details related to these representations we refer to [4].

Consider as an example a representation of a sparse matrix rendered by applying $RT$ to the Matrix type. It is shown graphically in Fig. 9.

2.5 Type-indexed arrays in the DSL’s implementation

To employ the above techniques in the design of our DSL lets first represent the type structure of an array element type by using the Scala encodings (shown in Fig. 10) of generic functions described above (see [27] for details).
type VectorElem = (Int, Float)
type SparseVector = PArray[VectorElem]
type Matrix = PArray[SparseVector]

Figure 9. Sparse matrix representation

type Elem[A] = Element[A] // type synonym
trait Element[A] { // type descriptor for type A
  def fromArray(arr: Array[A]): PA[A]
}
class BaseElem[T] extends Element[T] {
  def fromArray(arr:Array[T]) = BaseArray(arr)
  def replicate(len:Int, v:T) = BaseArray(Array.fill(len)(v))
}

Figure 10. 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 (pairElem). All these lead to a fully generic while still statically typed code.

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

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

And then we implement these abstract methods in concrete array classes shown in Fig. 11. Note how the implementation changes depending on the type of an array element. Each method declared in the PArray trait is a type in-
dexed function and each implementation in a concrete array class is an implementation of the function for the particular type case.

case class UnitArray(len: Int) extends PArray[Unit]{
  def length = len
  def map[R:Elem](f: Unit => R) =
    element[R].replicate(len, f(()))
}
case class BaseArray[A:Elem](arr: Array[A]) extends PArray[A] {
  def length = arr.length
  def map[R:Elem](f: A => R) =
    element[R].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](values: PA[A],
  segs: PA[(Int,Int)]) extends PArray[PA[A]] {
  def length = segs.length
  def map[R:Elem](f: PA[A] => R): PA[R] =
    element[R].tabulate(length)(i => {
      val (p,l)= segs(i); f(values.slice(p,l))
    })
}

Figure 11. Polytypic PArray methods

2.6 Lightweight Modular Staging (LMS)

Given a type A of an array element we know how to build a type-indexed representation of a parallel array using RT function thus yielding $RT[PA[A]]$ type. We have seen how to encode in our DSL these array representations together with polytypic operations over them. These techniques are used in our unstaged implementation of nested data parallelism (as described in [27]).

As it was mentioned before, the unstaged implementation is not intended to be efficient, rather, it should be simple and straightforward, as it is supposed to be used for testing and debugging (in the aforementioned simulation mode). To enable a parallel and efficient implementation, we employ a deep polymorphic embedding technique, namely a particular instance of it known as Lightweight Modular Staging (LMS) [25].

In the name, 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
a form of a program graph) and then executes that new program to produce the final result.

Consider the method `svm` in Fig. 1 and types `Matrix` and `Vector` that were used in the declaration. In the LMS framework, in order to express staging, we are required to lift the types by using the type constructor `Rep[_]` and use `Rep[Matrix]` and `Rep[Vector]`, etc. In fact, `svm` should have been declared like this to enable polymorphic embedding

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

In the case of unstaged interpretation we define `Rep` as

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

which yields a unstaged implementation of the method above with the 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]

```scala
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
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](d: Def[T]): Exp[T] =
    findOrCreateDefinition(d).sym
}
```

```
Figure 12. How Rep[T] is 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 the Scala-Virtualized compiler [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 the DSL level without providing any concrete implementation as follows

```
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 [25] for detailed explanation of how the LMS works.

3. 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 the LMS framework in particular we need to answer the question: How are we going to lift the type-indexed types along with the polytypic functions in the `Rep` world? In this section we describe the Polytypic Staging, our approach to a deep embedding of polytypic DSLs.

By design, our framework is a polytypic extension of the LMS framework, respects the type-indexed representations described before and behaves as core LMS in the non-polytypic case.

3.1 Staged Values

To be consistent with the LMS framework, 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 figure 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 Scalar
}
```

```
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. Whenever we construct a symbol we have to provide implicitly or explicitly its type descriptor. We also treat 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 the `Def` extractor defined in the core LMS framework.

```scala
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 the type `Rep[T]` is always an instance of `Sym[T]` which simplifies the implementation.

### 3.2 Staged Type Descriptors

In the staged context the descriptors of types of array elements shown in Fig. [10] remain unchanged. This means that we can keep our type representation schema with one adaptation: we need to *lift* all the methods of the `Element[T]` trait.

```scala
type Elem[A] = Element[A]
trait Element[T] {
  def replicate(count: Rep[Int], v: Rep[A]): PA[A]
  def fromArray(arr: Rep[Array[A]]): PA[A]
}
class BaseArray[T] extends Element[T] {
  def replicate(len: Rep[Int], v: Rep[A]) =
    BaseArray(ArrayFill(len, v))
}
class BaseElem[T] extends Element[T] {
  def fromArray(arr: Rep[Array[A]]): PA[A] =
    BaseArray(arr)
}
```

3.3 Staged 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[A]` type has a type-indexed representation that is built by induction on the structure of A. We also extensively use a convenience type synonym `PA` defined as follows

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

Thus, in a staged context, `PA` is no longer a synonym of `PArray` and now it is a synonym of a lifted `PArray`. In other words `PA[T]` is a lifted value of array with elements of type T. It is not a key point in our implementation but the introduction of `PA[A]` simplifies our presentation (and in fact greatly simplifies the code of the library).

Let us use the code in Fig. [13] to describe how values of the type `PArray` are staged (or lifted) in our polytypic staging framework. First, notice that the `replicate` method of `pairElem` produces a value of the `PA[(A,B)]` 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

```scala
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
  }
}
```
of type \( PA[(A,B)] \) we can think of it as a value of some symbol of type \( Sym[PA[(A,B)]] \). Next, recall that in LMS we get lifted values of the type \( Rep[T] \) by the following implicit conversion (recall also that \( Rep[T] = Exp[T] \))

\[
\text{implicit def toExp[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d).sym}
\]

The conversion is automatically inserted by the compiler, it 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 parallel arrays from \( Def[T] \) with appropriate \( T \) so that they can be first, converted to symbols and second, added to the graph as array construction nodes. As an example see Fig. 13 where \( PairArray \) is returned by the method \( replicate \). The definitions to represent arrays are shown in Fig. 15.

**abstract class** \( PADef[A] \) **extends** \( Def[PAArray[A]] \)

**case class** UnitArray(len: Rep[Int])

**extends** \( PADef[Unit] \) {
  def map[R:Elem](f: UnitRep ⇒ Rep[R]): PA[R] = element[R].replicate(len, f(toRep(())))
}

**case class** BaseArray[A:Elem](arr: Rep[Array[T]])

**extends** \( PADef[(A,B)] \) {
  def map[R:Elem](f: Rep[(A,B)] ⇒ Rep[R]): PA[R] = element[R].tabulate(arr.length)(i ⇒ f(a(i),b(i)))
}

**case class** NArray[A:Elem](arr: PA[A], segs: PA[(Int,Int)])

**extends** \( PADef[PAArray[A]] \) {
  def map[R:Elem](f: PA[A] ⇒ Rep[R]): PA[R] = element[R].tabulate(length)(i ⇒ f(a(i),b(i)))
}

**Figure 15.** Array classes as graph nodes (Defs)

Compare these classes with those shown in Fig. 11 and note how class signatures became lifted either explicitly by using the \( Rep[T] \) constructor or implicitly by redefining the \( PA[T] \) synonym as \( Rep[PAArray[A]] \). Moreover, the type representation transformation function \( TR \) shown in Fig. 7 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 \( PAArray[A] \) 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 \( PAArray[T] \) we both create the data structure using our concrete array classes and at the same time we build nodes of the program graph. This is another key difference from the LMS framework. In the LPS design some nodes of the graph can have a behavior.

The staged representation transformation (\( SRT \)) is shown in Fig. 16. The function \( L \) is a mapping of types of concrete arrays to the types of staged values.

\[
L, SRT:\ * → *
\]

\[
L[\text{UnitArray}(len: Rep[Int])] = PA[Unit]
\]

\[
L[\text{BaseArray}(arr: Rep[Array[T]])] = PA[T]
\]

where \( T = \text{Int|Float|Boolean} \)

\[
L[\text{PairArray}(a: PA[A], b: PA[B])] = PA[(A,B)]
\]

\[
L[\text{SumArray}(flags: PA[Boolean],
\hspace{1em}a: PA[A], b: PA[B])] = PA[(A|B)]
\]

\[
L[\text{NArray}(
\hspace{1em}values: PA[A],
\hspace{1em}segs: PA[(Int,Int)])] = PA[PAArray[A]]
\]

\[
SRT[PAArray[Unit]] = \text{UnitArray}(len: Rep[Int])
\]

\[
SRT[PAArray[(A,B)]] = BaseArray(arr: Rep[Array[T]])
\]

\[
SRT[PAArray[(A|B)]] = \text{PairArray}(a: L[SRT][PAArray[A]],
\hspace{1em}b: L[SRT][PAArray[B]])
\]

\[
SRT[PAArray[PAArray[A]]] = \text{NArray}(values: L[SRT][PAArray[A]],
\hspace{1em}segs: L[SRT][PAArray[(Int,Int)])]
\]

**Figure 16.** Staged Representation Transformation

A graphical illustration of these representations in a form of a program graph is shown in Fig. 17 where we use the following methods to construct new arrays:

\[
def fromArray[T:Elem](x: Rep[Array[T]]): PA[T] = element[T].fromArray(x)
\]

\[
def replicate[T:Elem](len: Rep[Int], v: Rep[T]): PA[T] = element[T].replicate(len, v)
\]

In a staged context (when \( type Rep[A] = Exp[A] \)) it is possible to achieve an effect of constant propagation and a limited form of partial evaluation by applying domain-specific rewrites (see Section 3.5). Our experiments show that if all the input data of the function is known at staging time, our rewriting method, while simple enough, is able to fully evaluate the function. It is illustrated in Fig. 17 where the array building expressions are evaluated to a type-indexed representation of the resulting arrays and that representation only contains data arrays in Const nodes and concrete array nodes form Fig. 15 that represent \( PAArray[A] \) values.
3.4 Staged Polytypic Functions

The same way as we lift the methods in the type descriptors (types derived from `Element[T]` and shown in Fig. 13) we can lift the methods in the concrete array classes (those derived from `PArray[T]` and shown in Fig. 15).

Compare this code with the non-staged version in Fig. 11 and note how the signatures are all lifted over `Rep` and the bodies of the methods remain literally unchanged. It is interesting that polymorphic embedding allows to share the same code for unstaged and staged implementation even in the library itself which makes the design very flexible.

As a not very trivial example of staging, we show in Fig. 18 a program graph that we get by staging of the function `svm`. The `Lambda(x,exp)` is a representation in the graph of a lambda abstraction where `x` is a symbol that represents the variable and `exp` is a symbol that represent the body of the lambda term.

3.5 Domain specific transformations

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 the staging time rewrites. Our method of rewriting is very simple and is based on the one proposed in [25].

The method uses the fact that every staged operation, which is represented by a graph node, in terms of the Scala language is represented by descendants of the `Def` class. Every time a new definition is created it is converted to the corresponding `Exp` by the implicit method `toExp` shown in Fig. 19.

The rewriting is performed by using the following algorithm. If we can find the definition in the graph, we just return its symbol. Otherwise, we try to rewrite the Def. If the result of rewrite is not defined then there is no rules that can be applied so the definition is added to the graph. If the rewrite comes back with a new symbol then we extract its definition from the graph (by using `Def`) and go recursively with the new definition.

We have found that this iterative rewriting has to be continued until there is no applicable rewrite rule, since one rewrite (that is applied) often happens to create a new symbol that leads to the possibility of another rewrite. In partic-
4. User-Specific Data Types

In this section we describe how to add any user-specific data type to our framework. The key point is to be able to make any such type \( U \) an instance of type-class \( \texttt{Elem} \). The idea is to define isomorphism between \( U \) and some existing instance \( \texttt{A:Elem} \). We extend our family of array element types as it is shown in Fig. 20.

\[
A, B = \text{Unit} \mid \text{Int} \mid \text{Float} \mid \text{Boolean} \quad \text{// base types} \\
(A, B) \quad \text{// product (pair of types)} \\
(A | B) \quad \text{// sum type where (A | B) = Either[A,B]} \\
P\text{Array}[A] \quad \text{// nested array} \\
U \text{ if exist Iso}[A, U] \quad \text{for some } A : \text{Elem}
\]

Figure 20. User-specific type as array element type

In other words, type \( U \) can be regarded as belonging to the type-class \( \texttt{Elem} \) if there is an isomorphism between \( U \) and some \( A : \texttt{Elem} \). Type \( \text{Iso}[A,B] \) is defined like this

\[
\text{trait Iso}[A,U] \{
\text{  def } eA : \texttt{Elem}[A] \quad \text{// type descriptor for } A \\
\text{  def } eU : \texttt{Elem}[U] \\
\text{  def from: } U \Rightarrow A \quad \text{// unstaged morphisms} \\
\text{  def to: } A \Rightarrow U \\
\text{  def fromStaged: } \texttt{Rep}[U] \Rightarrow \texttt{Rep}[A] \\
\text{  def toStaged: } \texttt{Rep}[A] \Rightarrow \texttt{Rep}[U] 
\}
\]

The reason we have separate versions for unstaged and staged isomorphism is that in a staged context we need to have an unstaged version of iso too.

Next, we need to extend the representation transformation for both unstaged (defined in Fig. 16) and staged (defined in Fig. 21) versions. Corresponding extensions are shown in Fig. 21.

\[
\begin{align*}
RT, L, SRT : \ast & \Rightarrow \ast \\
RT[\texttt{PA}[U]] &= \texttt{ViewArray}(\texttt{arr}: \texttt{RT}[\texttt{PA}[A]], \text{iso: Iso}[A,U]) \\
&\quad \text{if exists unique Iso}[A,U] \text{ for some } A : \texttt{Elem} \\
L[\texttt{ViewArray}(\texttt{arr}: \texttt{PA}[A], \text{iso: Iso}[A,U])] &= \texttt{PA}[U] \\
SRT[\texttt{PA}[U]] &= \texttt{ViewArray}(\texttt{arr: L}[SRT[\texttt{PA}[A]]], \text{iso: Iso}[A,U]) \\
&\quad \text{if exists unique Iso}[A,U] \text{ for some } A : \texttt{Elem}
\end{align*}
\]

Figure 21. Representation transformation for user-specific types

Remember, that for every type \( A \) we need a runtime type descriptor \( \texttt{Elem}[A] \) to be able to create arrays of type \( \texttt{PA}[A] \). For the case of user-specific data type \( U \) the type descriptor is shown below

\[
\text{implicit def viewElement}[A,U](\text{implicit iso: Iso}[A,U]): \texttt{Elem}[U] = \texttt{new} \quad \texttt{Element}[U] \\
\text{def replicate}(l: \texttt{Rep}[\texttt{Int}], v: \texttt{Rep}[U]): \texttt{PA}[U] = \texttt{ViewArray}(
\quad \text{iso.eA}.\text{replicate}(1, \text{iso.fromStaged}(v)), \text{iso})
\]

We use the type descriptor of an representation type \( \text{iso.eA} \) to build an actual array and wrap it with \( \texttt{ViewArray} \) to get type-indexed representation for \( \texttt{PA}[U] \) (see Fig. 21).

To complete our presentation of user-specific types we show an implementation of function \( \text{map} \) below. Notice the usage of the isomorphism in the body of the function.

\[
\begin{align*}
\text{case class } \text{ViewArray}[A,U](\text{arr: } \texttt{PA}[A], \text{iso: Iso}[A,U]) \text{ extends } \texttt{PA}[U] \{
\text{def } \text{map}[R: \texttt{Elem}](f: \texttt{Rep}[U] \Rightarrow \texttt{Rep}[R]): \texttt{PA}[R] = \{
\text{element}[R].\text{tabulate}(\text{length})
\quad (i \Rightarrow f(\text{iso.toStaged}(\text{arr}(i))))
\}\}
\end{align*}
\]

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 a 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 until the running time.

Another interesting feature of NDP is compositional (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 have initially chosen an approach and the development platform that are different from those of our predecessors and we put an emphasis on limiting the degree of generality by formulating the problem as development of the DSL.

There are reasons to believe that by limiting the degree of generality, we can more easily use the domain semantics for an implementation of deeper and more significant optimizations. In addition, the LMS platform gives us some opportunities for integration with other DSLs \[2, 28\], which, in turn, will allow us, by combining their semantics, to improve the depth of optimizations and performance.

The staging approach as it is described here is a front-end of the compiler tool-chain. In a polytypic context it opens up many questions both for research and software.
engineering. That is also true for the rewriting rules. Our experiments with the rewritings in the NDP domain show that even simple rewriting strategy combined with domain knowledge can exhibit radical optimizations not possible in the context of general purpose language. We regard this questions as directions of the future research.

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

References