Finally, a Polymorphic Linear Algebra Language

Amir Shaikhha
Department of Computer Science, University of Oxford, UK
amir.shaikhha@cs.ox.ac.uk

Lionel Parreaux
DATA Lab, EPFL, Lausanne, Switzerland
lionel.parreaux@epfl.ch

Abstract

Many different data analytics tasks boil down to linear algebra primitives. In practice, for each different type of workload, data scientists use a particular specialised library. In this paper, we present Pilatus, a polymorphic iterative linear algebra language, applicable to various types of data analytics workloads. The design of this domain-specific language (DSL) is inspired by both mathematics and programming languages: its basic constructs are borrowed from abstract algebra, whereas the key technology behind its polymorphic design uses the *tagless final* approach (a.k.a. polymorphic embedding/object algebras). This design enables us to change the behaviour of arithmetic operations to express matrix algebra, graph algorithms, logical probabilistic programs, and differentiable programs. Crucially, the polymorphic design of Pilatus allows us to use multi-stage programming and rewrite-based optimisation to recover the performance of specialised code, supporting fixed sized matrices, algebraic optimisations, and fusion.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computing methodologies → Linear algebra algorithms; Mathematics of computing → Automatic differentiation; Mathematics of computing → Graph algorithms; Theory of computation → Probabilistic computation

Keywords and phrases Linear Algebra, Domain-Specific Languages, Tagless Final, Polymorphic Embedding, Object Algebra, Multi-Stage Programming, Graph Processing, Probabilistic Programming, Automatic Differentiation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.25

Category Pearl

Acknowledgements The authors would like to thank Jeremy Gibbons and Oleg Kiselyov for their helpful comments on draft versions of this paper. The first author was partially supported by EPFL during the preparation of this paper.

1 Introduction

It is well-known that many problems can be formulated using linear algebra primitives. These problems come from various data analytics domains including machine learning, statistical data analytics, signal processing, graph processing, computer vision, and robotics.

Despite the fact that all these workloads could use a standard unified linear algebra library, in practice many different specialised libraries are developed and used for each of these workload types [9]. This is mainly due to the performance-critical nature of such data analytics workloads: in order to satisfy their performance requirement, such workloads use hand-tuned specialised libraries implemented using either general-purpose or specialised domain-specific programming languages.

In this paper, we demonstrate the Pilatus language (Polymorphic Iterative Linear Algebra, Typed, Universal, and Staged). Pilatus is a *polymorphic* domain-specific language (DSL), in the sense that it can support various workloads, such as standard iterative linear algebra tasks, graph processing algorithms, logical probabilistic programs, and linear algebra

© Amir Shaikhha and Lionel Parreaux;
licensed under Creative Commons License CC-BY

programs relying on automatic differentiation. By default, this polymorphic nature causes a significant performance overhead. We demonstrate how to remove this overhead by using safe high-level meta-programming and compilation techniques, and more specifically multi-stage programming (MSP, or *staging*) [66, 65].

This paper uses the tagless final approach [36, 10] (also known as polymorphic embedding [30] and object algebras [48]) in order to embed [32] the Pilatus DSL in the Scala programming language. This technique allows embedding an *object* language in a *host* language in a type-safe manner. In addition, this approach allows multiple semantics for the embedded DSL (EDSL). Based on this feature and by carefully choosing the abstractions involved in defining Pilatus (such as semi-ring/ring, module, and linear map structures), we provide several evaluation semantics. More specifically, we allow several variants of a linear algebra language, such as: a standard matrix algebra language, a graph language for expressing all-pairs reachability and shortest path problems, a logical probabilistic programming language, and a differentiable programming language. The polymorphic aspect of Pilatus is also essential for the seamless application of staging, and to express different optimised staged variants: fixed size matrices, deforestation [67, 25, 63, 13], and algebraic optimisations.[1]

Next, we motivate the need for Pilatus (Section 2), and we make the following contributions:

- We present Pilatus, a polymorphic EDSL in Section 3. This DSL uses the notion of semi-rings and rings (Section 3.2) in order to define operations on each individual element of a vector and a matrix. Furthermore, Pilatus uses the notion of pull arrays (Section 3.5) for defining a collection (or array) of elements.
- We present four different languages that are implemented by providing a concrete interpreter for Pilatus in Section 5: (1) a standard matrix algebra language (Section 5.1); (2) a graph DSL (Section 5.2); (3) a logical probabilistic linear algebra language (Section 5.3); and (4) a differentiable linear algebra language (Section 5.4).
- We present our use of multi-stage programming to improve the performance of Pilatus programs by creating a staged language (Section 6.2) for fixed size matrices (Section 6.5), performing algebraic optimisations (Section 6.4), and performing fusion (Section 6.6).
- We show the impact of using multi-stage programming on the performance of applications written using Pilatus in Section 7. Overall, the implementation of Pilatus consists of around 400 LoC supporting all the features presented in this paper. Pilatus uses the Squid [51] type-safe meta-programming framework for its multi-stage programming facilities, which is the only external library dependency.

Finally, we present the related work in Section 8 and conclude the paper in Section 9.

2 Motivation

Apart from standard matrix algebra tasks, many numerical workloads in various domains can be expressed using linear algebra primitives [17]. Among such examples are various graph

[1] We used Scala as the implementation language for Pilatus, but other programming languages with support for lambda expressions and multi-stage programming could be used as well; most of the techniques presented in this paper can also be implemented in Haskell, OCaml, and Java for example. For expressing rewrite-based optimisations, either the multi-stage programming framework should support code inspection (as is the case with Squid [51], which we use), or the developer is responsible for implementing/extending the intermediate representations (as with frameworks like LMS [57]).

problems such as reachability and shortest path. Figure 1 shows as example the reachability problem on both deterministic and probabilistic graphs.

Despite the expressiveness of linear algebra, there are many different libraries specialized for each particular data analytics task. This is because of two main reasons. First, most existing linear algebra libraries do not define the interfaces for extending their usage for the problems in other domains. Second, despite some efforts on providing abstract and extensible linear algebra libraries [17], such analytical tasks are performance critical. As a result, there should be hand-tuned and specialized libraries for each particular task. As an example, for graph problems, rather than having the linear-algebra-based solutions presented in Figure 1, the library developers prefer to provide specialized graph libraries for performance reasons.

This paper aims to solve both these issues by combining ideas from mathematics and programming languages. The first issue is tackled by defining a polymorphic linear algebra language by using abstractions from abstract algebra, including the ring, module, and linear map structures for expressing scalar values, a vector of values, and a matrix of values, respectively. Furthermore, for implementing these abstract interfaces, we use the tagless-final approach [10, 36], a well-known technique from the programming language community.

The examples of Figure 1 show matrices of elements of various types, for which the addition and multiplication operations can be assigned various meanings. Figure 1a shows the usage of linear algebra primitives for expressing graph reachability problems. To do so, the addition and multiplication operators are instantiated to boolean disjunction and conjunction, respectively. For expressing the reachability problems on probabilistic graphs, these two operators are instantiated with the disjunction and conjunction on boolean distributions, as shown in Figure 1b. Finally, Figure 1c shows the process of computing the derivative of an example matrix expression with respect to a given variable. To do so, each element of the matrix should be represented as a pair of numbers, known as *dual numbers*, where the first component is the actual value of that expression and the second component is the value of its derivative with respect to the given variable. As an example, the dual number representation for the element of the 2nd row and 3rd column is represented as $3 \triangleright 2$, meaning that the actual value of this element at $x = 2$ is $2x - 1 = 3$, whereas its derivative value is $(2x - 1)' = 2$. Similarly, the addition and multiplication operators are instantiated with the corresponding ones operating on dual numbers, which implement the derivative rules.

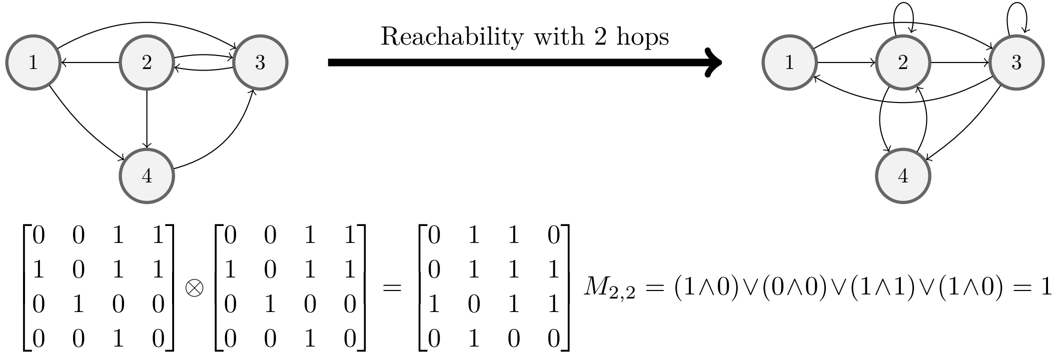
All of these use cases can be easily represented as Pilatus programs, parameterized over the meaning one wants to use for a particular domain.

The second issue is the performance overhead caused by the polymorphic nature of the language, due to the abstractions introduced in order to solve the first issue. We use multi-stage programming (also known as *staging*) to compile away the overhead corresponding to these abstractions. Moreover, by using a staging framework with support for rewriting, we can also implement algebraic optimization rules for further improving performances.

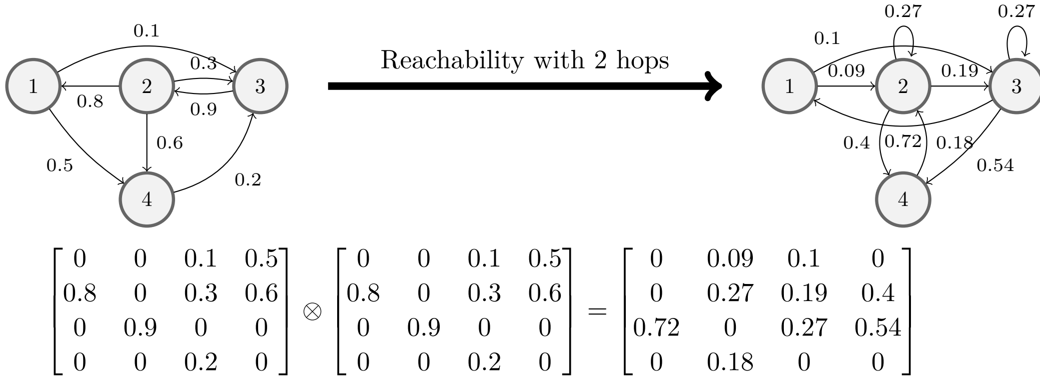
Next, we give more details on the design of Pilatus.

3 Pilatus Design

In this section, we first give an overview of the tagless final approach. Then, we define the polymorphic interface for the semi-ring and ring structures. Afterwards, we show an abstract interface for vectors and matrices using the mathematical notions of modules and linear maps. Finally, we define the interface for a functional encoding of an array of elements and control-flow constructs.



(a) The reachability problem in a graph can be expressed using matrix-matrix multiplication of the adjacency matrix of a graph. Instead of using the standard addition operator, here we use the boolean disjunction, and instead of the multiplication operator, we use the boolean conjunction.



(b) The reachability problem in a probabilistic graph can also be expressed using matrix-matrix multiplication of its adjacency matrix. Each element of the adjacency matrix represents the presence of a node with probability p. The addition and multiplication operators correspond to disjunction and conjunction of two boolean distributions, respectively.

$$f(x) = \begin{bmatrix} 0 & 0 & 1 & x+3 \\ 8 & 0 & 2x-1 & 6 \\ 0 & 3x+3 & 0 & 0 \\ 0 & 0 & x & 0 \end{bmatrix}^2 = \begin{bmatrix} 0 & 3x+3 & x^2+3x & 0 \\ 0 & 6x^2+3x-3 & 6x+8 & 8x+24 \\ 24x+24 & 0 & 6x^2+3x-3 & 18x+18 \\ 0 & 3x^2+3x & 0 & 0 \end{bmatrix}$$

$$f'(x) = \begin{bmatrix} 0 & 3 & 2x+3 & 0 \\ 0 & 12x+3 & 6 & 8 \\ 24 & 0 & 12x+3 & 18 \\ 0 & 6x+3 & 0 & 0 \end{bmatrix} \quad f(2) = \begin{bmatrix} 0 & 9 & 10 & 0 \\ 0 & 27 & 20 & 40 \\ 72 & 0 & 27 & 54 \\ 0 & 18 & 0 & 0 \end{bmatrix} \quad f'(2) = \begin{bmatrix} 0 & 3 & 7 & 0 \\ 0 & 27 & 6 & 8 \\ 24 & 0 & 27 & 18 \\ 0 & 15 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \triangleright 0 & 0 \triangleright 0 & 1 \triangleright 0 & 5 \triangleright 1 \\ 8 \triangleright 0 & 0 \triangleright 0 & 3 \triangleright 2 & 6 \triangleright 0 \\ 0 \triangleright 0 & 9 \triangleright 3 & 0 \triangleright 0 & 0 \triangleright 0 \\ 0 \triangleright 0 & 0 \triangleright 0 & 2 \triangleright 1 & 0 \triangleright 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \triangleright 0 & 0 \triangleright 0 & 1 \triangleright 0 & 5 \triangleright 1 \\ 8 \triangleright 0 & 0 \triangleright 0 & 3 \triangleright 2 & 6 \triangleright 0 \\ 0 \triangleright 0 & 9 \triangleright 3 & 0 \triangleright 0 & 0 \triangleright 0 \\ 0 \triangleright 0 & 0 \triangleright 0 & 2 \triangleright 1 & 0 \triangleright 0 \end{bmatrix} = \begin{bmatrix} 0 \triangleright 0 & 9 \triangleright 3 & 10 \triangleright 7 & 0 \triangleright 0 \\ 0 \triangleright 0 & 27 \triangleright 27 & 20 \triangleright 6 & 40 \triangleright 8 \\ 72 \triangleright 24 & 0 \triangleright 0 & 27 \triangleright 27 & 54 \triangleright 18 \\ 0 \triangleright 0 & 18 \triangleright 15 & 0 \triangleright 0 & 0 \triangleright 0 \end{bmatrix}$$

(c) The derivative of a matrix with respect to the variable x can also be expressed using linear algebra operations. The dual number technique represents each element of a matrix as the pair $v \triangleright d$ of the actual value v and the value of its derivative d. Accordingly, the addition and multiplication operators are the corresponding ones on dual numbers.

Figure 1 Example of problems expressed using different interpretations of linear algebra primitives.

3.1 Tagless Final

Tagless final [10, 36] (also known as polymorphic embedding [30] and object algebras [48] in the context of object-oriented programming languages) is a type-safe approach for *embedding* [32] domain-specific languages. This approach solves the expression problem [68] by encoding each DSL construct as a separate function, and leaving their interpretation abstract.

There are different ways of implementing this approach: (1) in languages like Haskell, one can use type classes [10, 36]; (2) in OCaml, one can use the module system [10, 37]; (3) in languages like Java, one can use the object-oriented features [48]; and (4) in Scala one can use either type classes or mixin composition (also known as the cake pattern) [30, 57].

In this paper, we follow the approach based on type classes. Consider a DSL with two constructs, one for creating an integer literal, and the other for adding two terms. The tagless final interface for this DSL is as follows:

```
trait SimpleDSL[Repr] {
  def lit(i: Int): Repr
  def add(a: Repr, b: Repr): Repr
}
```

The code above defines a *trait* (similar to an *interface* in Java or a *module signature* in ML). The `SimpleDSL` trait is parameterised with a `Repr` type, which is the type of the objects manipulated by the DSL. This trait contains one abstract method for each constructs of the DSL, here `lit` and `add`.

For convenience, we also typically define free-standing functions for writing programs in the DSL while omitting the particular DSL implementation used:

```
def lit[Repr](i: Int)          (implicit dsl: SimpleDSL[Repr]): Repr = dsl.lit(i)
def add[Repr](a: Repr, b: Repr)(implicit dsl: SimpleDSL[Repr]): Repr = dsl.add(a, b)
```

These functions require an *implicit* instance of the `SimpleDSL` trait, and redirect to the implementations of the corresponding methods in that instance. In Scala, implicit parameters need not be specified by users at each call site; indeed, they can be *filled in* automatically by the compiler, based on their expected type. Implicits are the mechanism used to implement type classes in Scala [49].

One can then define generic programs in the DSL, as follows:

```
def myProgram[Repr](implicit dsl: SimpleDSL) = add(lit(2), lit(3))
```

Which can also be written using the following shorthand syntax:

```
def myProgram[Repr: SimpleDSL] = add(lit(2), lit(3))
```

Then, one can specify a particular evaluation semantic for this program. As an example, the following type class instance defines an evaluator/interpreter for `SimpleDSL`:

```
implicit object SimpleDSLInter extends SimpleDSL[Int] {
  def lit(i: Int): Int = i
  def add(a: Int, b: Int): Int = a + b
}
```

Evaluating the example program above in the REPL with this evaluation semantics, which is automatically picked up by the compiler based on the requested type `Int`, results in:

```
scala> myProgram[Int]
result: Int = 5
```

```
trait SemiRing[R] {
  def add(a: R, b: R): R
  def mult(a: R, b: R): R
  def one: R
  def zero: R
}
trait Ring[R] extends SemiRing[R] {
  def neg(a: R): R
  def sub(a: R, b: R): R = add(a, neg(b))
}

object Pilatus {
  def add[R](a: R, b: R)(implicit sr: SemiRing[R]): R = sr.add(a, b)
  // ... other boilerplate methods elided for brevity
}
```

Figure 2 The tagless final interface for semi-rings and rings.

Rather than directly *evaluating* DSL programs, one can also represent the programs as strings. Below is a type class instance that *stringifies* programs in our DSL:[2]

```
implicit object SimpleDSLStringify extends SimpleDSL[String] {
  def lit(i: Int): String = i.toString
  def add(a: String, b: String): String = s"$a + $b"
}
```

Evaluating the same program with the stringification evaluation semantic results in:

```
scala> myProgram[String]
result: String = "2 + 3"
```

Pilatus defines a separate type class for each category of the language constructs (e.g., semi-rings, rings, modules, linear maps, etc.), as we show next. We will introduce different evaluation semantics for this DSL by providing type class instances. These evaluation semantics are both interpretation-based (cf. Section 5) and compilation-based (cf. Section 6).

3.2 Semi-Ring and Ring

A semi-ring is defined as a set of numerical values R, with two binary operators $+$ and $\times$, and two elements 0 (additive identity) and 1 (multiplicative identity), such that for all elements a, b, and c in R the following properties hold:

- $a + 0 = a$
- $a + b = b + a$
- $(a + b) + c = a + (b + c)$
- $a \times 1 = 1 \times a = a$
- $a \times 0 = 0 \times a = 0$
- $(a \times b) \times c = a \times (b \times c)$
- $a \times (b + c) = (a \times b) + (a \times c)$
- $(a + b) \times c = (a \times c) + (b \times c)$

[2] String interpolation syntax `s"...$x..."` is equivalent to `"..."+ x + "..."`.

A ring is a semi-ring with an additional additive inverse operator $(-)$ such that for all elements a in R, $a + (-a) = 0$. The binary operator for subtraction can be easily defined as $a - b = a + (-b)$.

The tagless final encoding of semi-rings and rings is shown in Figure 2. There are six DSL constructs corresponding to addition, multiplication, negation, subtraction, one, and zero. These methods are redirected to the implementation of the corresponding operations of the `SemiRing` and `Ring` type classes. The implementation of the methods of these type classes are left abstract. These definitions will be given by each concrete semantics, which should make sure that the aforementioned properties hold for the elements of type `R`.

3.3 Module

A mathematical module is a generalization of the notion of a vector space. A module over a particular semi-ring is realised using an addition operator for two modules (similar to vector addition), and a multiplication between a semi-ring element and the module (similar to scalar-vector multiplication). For all elements a and b in a semi-ring R with the multiplicative identity 1_R, and the elements u and v in a (left-)module M, the following properties hold:

- $a \cdot (u + v) = a \cdot u + a \cdot v$
- $(a + b) \cdot u = a \cdot u + b \cdot u$
- $(a \times b) \cdot u = a \cdot (b \cdot u)$
- $1_R \cdot u = u$

Additionally, the dimension of a finite module generalises the notion of the number of basis vectors[3] representing a vector.

Figure 3 shows the tagless final interface for modules. The `Module` type class has three type parameters: (1) `V` specifies the type of the underlying vector representation; (2) `R` specifies the type of each element of the vector; and (3) `D` specifies the type of the dimension of the underlying vector. Note that all the elements of type `R` and `D` support semi-ring operations, thanks to the two type class instances `sr` and `dr`. Furthermore, the `Module` type class supports the following operations: (1) the `dim` method returns the dimension of the given module; (2) the `add` method computes the result of the addition of two module elements; and (3) the `smult` method computes the multiplication of a semi-ring element and a given module.

3.4 Linear Map

A linear map is a transformation between two modules, which preserves the addition and the scalar multiplication operations of the given module. Assume the linear map M transforming module V to module W, and both modules are over the semi-ring R. Then for all elements f in the linear map M, u and v from module V, and a from the semi-ring R, the following properties hold:

- $f(u + v) = f(u) + f(v)$
- $f(a \cdot u) = a \cdot f(u)$

Similar to functions, linear maps have two operations. First, a linear map can be applied to a module returning a transformed module, behaving similar to the function application.

[3] The basis vectors are *linearly independent* vectors (none of them can be expressed as a linear combination of the other ones) that can be used to express every vector as a *unique linear combination* of them.

```
trait Module[V, R, D] {
  implicit val sr: SemiRing[R]
  implicit val dr: SemiRing[D]
  def dim(a: V): D
  def add(a: V, b: V): V
  def smult(s: R, a: V): V
}

object Pilatus {
  // ...
  def dim[V, D](a: V)(implicit m: Module[V, _, D]): D = m.dim(a)
  // ... other boilerplate methods elided for brevity
}
```

Figure 3 The tagless final interface for modules.

```
trait LinearMap[M, V, R, D] {
  implicit val rowModule: Module[V, R, D]
  implicit val sr: SemiRing[R]
  implicit val dr: SemiRing[D]
  def apply(m: M, v: V): V
  def compose(m1: M, m2: M): M
  def add(m1: M, m2: M): M
  def dims(mat: M): (D, D)
}

object Pilatus {
  // ...
  def apply[M, V](m: M, v: V)(implicit lm: LinearMap[M, V, _, _]): V = lm.apply(m, v)
  // ... other boilerplate methods elided for brevity
}
```

Figure 4 The tagless final interface for linear maps.

Second, a linear map can be composed with another linear map resulting in another linear map, behaving similarly to function composition.

Figure 4 shows the tagless final encoding of linear maps. Here, we only consider finite linear maps transforming two finite modules, and we assume that both modules are over the same semi-ring (represented with type `R`, and the `sr` type class instance) with the same module type representation (represented with the type `V`). From a vector/matrix point of view, the `compose` and `apply` methods correspond to the matrix-matrix and matrix-vector multiplication, respectively. The `add` method corresponds to the matrix addition operator, and the `dims` construct returns the dimension of the input and output modules, which is represented as a tuple.

3.5 Pull Array and Control-Flow Constructs

Using a pull array is a well-known approach in the high-performance functional programming community for a functional encoding of arrays [64, 2, 12]. In this representation, an array is defined using two components: (1) the length of the array; and (2) a function mapping an index to the value of the corresponding element in that array.

Figure 5 demonstrates the tagless final encoding of pull arrays and looping constructs. The `build` method is responsible for constructing a pull array of size `len`, in which the ith

```
trait PullArrayOps[A, E, L] {
  def build(len: L)(f: L => E): A
  def get(arr: A)(i: L): E
  def length(arr: A): L
}
trait Looping[L] {
  def forloop[S](z: S)(n: L)(f: (S, L) => S): S
}
object Pilatus {
  // ...
  def build[A, E, L](len: L)(f: L => E)(implicit p: PullArrayOps[A, E, L]): A =
    p.build(len)(f)
  // ... other boilerplate methods elided for brevity
}
```

Figure 5 The tagless final interface for pull arrays and control-flow constructs.

element is `f(i)`, indexed from `0` to `len - 1`. The `get` method returns the ith element of the array `arr`, whereas the `length` method returns the size of the given array. Finally, the `forloop` method is meant for implementing recursion and iteration. More specifically, this function starts from the state `z`, iterates `n` times (from `0` to `n - 1`), and at the ith step, updates the state `s` with `f(s, i)`.

4 Matrix Algebra

In this section, we build the constructs of matrix algebra based on the mathematical notions explained in the previous section. First, we show the construction of vector constructs using modules and pull arrays. Then, we demonstrate the matrix constructs by using linear maps and vectors.

4.1 Vector: Module + Pull Array

A vector (more specifically, a dense vector where most elements are non-zero) can be seen as a module the elements of which are stored as a pull array. Given that each element of a vector form a semi-ring, we can define the addition, element-wise multiplication, and dot product of two vectors.

The implementation for the tagless final encoding of a vector, as well as the mentioned methods are given in Figure 6. The `V` type parameter specifies the underlying vector type representation, the `R` type parameter specifies the type of each element of the vector, and the `D` type parameter is the type of the dimension of the underlying vector.

The `zipMap` method, receives two vectors `v1` and `v2` as input and creates a vector of the same size,[4] for which each element is constructed by applying the binary operator `op` on the corresponding elements from `v1` and `v2`. The `add` and `elemMult` are constructed by passing the addition and multiplication functions of the underlying semi-ring of elements to the `zipMap` method. The `map` method applies a given function to each element of the input vector and produces a vector of the same size with the transformed elements as output. The `smult` method is implemented using this method. Finally, the `dot` method computes the dot

[4] We assume that the input vectors have the same size for the sake of simplicity. In practice, this property can be enforced statically using Scala's powerful implicit programming capabilities, and singleton types.

```
trait Vector[V, R, D] extends Module[V, R, D] {
  implicit val pa: PullArrayOps[V, R, D]
  implicit val looping: Looping[D]
  def dim(a: V): D = pa.length(a)
  def add(v1: V, v2: V): V = zipMap(v1, v2, sr.add)
  def smult(s: R, a: V): V = map(a, e => sr.mult(s, e))
  def map(v: V, op: R => R): V = pa.build(pa.length(v))(i => op(pa.get(v)(i)))
  def zipMap(v1: V, v2: V, op: (R, R) => R): V =
    pa.build(pa.length(v1))(i => op(pa.get(v1)(i), pa.get(v2)(i)))
  def elemMult(v1: V, v2: V): V = zipMap(v1, v2, sr.mult)
  def dot(v1: V, v2: V): R =
    looping.forloop(sr.zero)(pa.length(v1))((acc, i) =>
      sr.add(acc, sr.mult(pa.get(v1)(i), pa.get(v2)(i))))
  /* sum and norm are omitted for brevity */
}
```

Figure 6 The tagless final implementation for (dense) vectors.

product of two vectors `v1` and `v2` by first computing the element-wise multiplication of these vectors, and then adding the elements of this intermediate vector.

Next, we use the mentioned vector data structure together with linear maps in order to define a matrix data-structure.

4.2 Matrix: Linear Map + Vector

Figure 7 shows the implementation of matrices (more specifically, dense matrices) using linear maps and vectors. The `M` type parameter specifies the type of the underlying matrix representation, `V` represents the type of each row-vector and column-vector of the matrix, `R` denotes the type of each element of the matrix, and `D` specifies the type of the dimension of each row and each column of the matrix.

In order to facilitate usages of the generic library, we have implemented several helper methods. The `get` method returns the corresponding element in the rth row and c^{th} column of the matrix.[5] The `numRows` and `numCols` methods return the number of rows and columns of a matrix, respectively. The `getRow` method returns the vector representing the rth row of the given matrix, whereas `getCol` returns a vector containing the elements in the cth column of the given matrix. Finally, the `zipMap` and `map` methods have similar behaviour to the methods with the same name from the vector data type.

The `add` method returns the result of the addition of two matrices, which is implemented using the `zipMap` method. The `mult` method returns the matrix-matrix multiplication of two matrices. This method is implemented by performing a vector dot-product of each row of the first matrix with each column of the second matrix. Finally, the `transpose` method returns the transpose of the given matrix.

4.3 Putting It All Together

Before showing different evaluation semantics in the upcoming sections, we need a way to print the result values. To do so, we define the `Printable` type class which converts the value of a particular type into a string. Figure 8 shows the corresponding tagless final definition.

[5] As we will see in Section 6, `r` and `c` can have types other than `Int`.

```
trait Matrix[M, V, R, D] extends LinearMap[M, V, R, D] {
  implicit val paMat: PullArrayOps[M, V, D]
  implicit val vector: Vector[V, R, D]
  /* Other implicit values: paRow, rowModule, looping, dr, sr */
  /* apply and compose methods use the mult method, elided for brevity. */
  def get(mat: M, r: D, c: D): R =
    paRow.get(paMat.get(mat)(r))(c)
  def numRows(mat: M) = dims(mat)._1
  def numCols(mat: M) = dims(mat)._2
  def getRow(mat: M, i: D): V = paMat.get(mat)(i)
  def getCol(mat: M, j: D): V = getRow(transpose(mat), j)
  def zipMap(m1: M, m2: M, bop: (R, R) => R): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m1))(j =>
        bop(get(m1, i, j), get(m2, i, j))))
  def add(m1: M, m2: M): M = zipMap(m1, m2, sr.add)
  def mult(m1: M, m2: M): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m2))(j =>
        vector.dot(getRow(m1, i), getCol(m2, j))))
  def transpose(mat: M): M =
    paMat.build(numCols(mat))(i =>
      paRow.build(numRows(mat))(j => get(mat, j, i)))
  /* map, eye, fill, and zeros are omitted for brevity */
}
```

Figure 7 The tagless final implementation for (dense) matrices.

```
trait Printable[T] {
  def string(e: T): String
}
object Pilatus {
  // ...
  def getString[T](e: T)(implicit p: Printable[T]) = p.string(e)
}
```

Figure 8 The tagless interface for the stringification of the values of different evaluation semantics.

▶ **Example 1.** Throughout this paper, we use the following example matrix program, where we change the values for matrix `m` based on the evaluation semantic that we are interested in:

```
def example[M, D](m: M)(implicit mev: Matrix[M,_,_,D], pev: Printable[M]): Unit = {
  import mev._
  val I = eye(numRows(m))
  val m2 = mult(m, m)
  val res = add(I, add(m, m2))
  println(getString(res))
}
```

This program accepts the matrix `m`, the value of which differs based on the evaluation semantic that we would like to use. The result of this program is the addition of the identity matrix (represented using the `eye` method), the given input matrix, and the second power of it.

In the next sections, we give several concrete interpretations for Pilatus, and we show the output of the example program above for each of the interpretations.

```
implicit object RingInt extends Ring[Int] {
  def add(a: Int, b: Int) = a + b
  def mult(a: Int, b: Int) = a * b
  def one: Int = 1
  def zero: Int = 0
  def neg(a: Int): Int = -a
}
implicit object RingDouble extends Ring[Double] {
  /* Similar to RingInt */
}
```

Figure 9 The tagless final interpreter (a.k.a. type class instances) for a ring of integer and double values.

```
class PullArrayArrayOps[E: ClassTag] extends PullArrayOps[Array[E], E, Int] {
  def build(len: Int)(f: Int => E): Array[E] =
    Array.tabulate(len)(f)
  def get(arr: Array[E])(i: Int): E =
    arr(i)
  def length(arr: Array[E]): Int =
    arr.length
}
class LoopingInt extends Looping[Int] {
  def forloop[S](f: (S, Int) => S)(z: S)(n: Int): S =
    (0 until n).foldLeft(z)(f)
}
object Semantics {
  implicit def pullArrayArrayOps[E: ClassTag] = new PullArrayArrayOps[E]
  implicit val loopingInt = new Looping[Int]
}
```

Figure 10 The tagless final interpreter for a pull array, represented as a list of elements, and the control-flow constructs.

5 Interpreted Languages

In this section, we first show an evaluation strategy which results in a standard matrix algebra library. Then, we show how we can define an alternative interpretation which leads to treating Pilatus as a graph library. Afterwards, we show a linear algebra library for logical probabilistic programming. Finally, we demonstrate how Pilatus can behave as a library for differentiable programming.

5.1 Standard Matrix Algebra

In order to define a standard matrix algebra library for Pilatus, we start by defining a normal interpreter for rings. Figure 9 shows the interpretation for a ring of integer and double values. In both cases, the addition and multiplication operations are defined using the primitive operations provided by the Scala language.

Figure 10 shows an interpreter for pull arrays, where every constructed pull array is materialised into an array of elements. Hence, retrieving an element and returning the size of the pull array is achieved by returning the corresponding element in the materialised array and the length of the array, respectively. Finally, the implementation of `forloop` is achieved by performing a `foldLeft` on the range of elements from `0` to `n-1`, and passing the initial state and the accumulator function.

```
case class PullArrayInter[E](len: Int, f: Int => E)

class PullArrayInterOps[E] extends PullArrayOps[PullArrayInter[E], E, Int] {
  def build(len: Int)(f: Int => E): PullArrayInter[E] =
    PullArrayInter(len, f)
  def get(arr: PullArrayInter[E])(i: Int): E =
    arr.f(i)
  def length(arr: PullArrayInter[E]): Int =
    arr.len
}
object Semantics {
  // ...
  implicit def pullArrayInterOps[E] = new PullArrayInterOps[E]
}
```

Figure 11 The tagless final interpreter for a pull array, represented as a pair of length and the element constructor function.

An alternative way of interpretation for pull arrays, which avoids the materialisation of the intermediate arrays into a sequence, keeps a data structure which holds the length and the constructor function of each element. This representation is given in Figure 11.

▶ **Example 2.** The standard matrix algebra interpreter evaluates the example program as follows:

```
import Semantics.{ pullArrayInterOps, loopingInt, ringInt }
val adj = Array(Array(0, 0, 1, 5),
                Array(8, 0, 3, 6),
                Array(0, 9, 0, 0),
                Array(0, 0, 2, 0))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
[ [ 1, 9, 11, 5 ]
, [ 8, 28, 23, 46 ]
, [ 72, 9, 28, 54 ]
, [ 0, 18, 2, 1 ] ]
```

Note that the `build` method is redirected to the `build` method of the `PullArrayOps` type class (cf. Figure 5).

5.2 Graph DSL for Reachability and Shortest Path

A directed graph can be represented using its adjacency matrix. More specifically, a graph with n vertices can be represented using a matrix of size $n \times n$, in which all elements are Boolean. If the element in the ith row and j^{th} column is `true`, this means that there is an edge between the ith and j^{th} vertices in the graph.

In order to support such adjacency matrices, we need to use the Boolean semi-ring for the matrix elements. Figure 12 shows the implementation of the Boolean semi-ring, in which addition performs disjunction, and multiplication performs conjunction.

Using the Boolean semi-ring for the elements of a matrix leads to a graph library. This instantiation of Pilatus is appropriate for expressing reachability computations among all vertices of a graph: given an adjacency matrix M, each element of $M \times M$ shows the existence of a path of length 2 between the two vertices in the corresponding graph.

```
class SemiRingBoolean extends SemiRing[Boolean] {
  def add(a: Boolean, b: Boolean) = a || b
  def mult(a: Boolean, b: Boolean) = a && b
  def one: Boolean = true
  def zero: Boolean = false
}
object Semantics {
  // ...
  implicit val semiRingBoolean = new SemiRingBoolean
}
```

Figure 12 The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph reachability problems.

The graph algorithms that can be implemented on top of Pilatus are not limited to reachability ones. By adding other types of semi-rings, one can express other graph computation problems. As an example, Tropical semi-rings can express shortest-path graph problems [45, 17]. Figure 13 shows the tagless final encoding of Tropical semi-rings. The `ShortestPath` data type represents the path between two nodes of a graph, where `Unreachable` specifies a path of length $+\infty$, and `Distance(v)` specifies a path of length `v`. The Tropical semi-ring computes the minimum length of two paths as the addition operator of the semi-ring, and adds the length of two paths as the multiplication operator. We omit the definition for other semi-rings for graph and other similar problems (e.g., linear equations, data-flow analysis, petri nets, etc., which are already explored in the literature [17]).

▶ **Example 3.** When one uses the Boolean semi-ring, the example program is actually computing the existence of paths with maximum length two among all the nodes. When we provide the adjacency matrix of the graph of Figure 1a, the example program evaluates to:

```
import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolean }
val adj = Array(Array(false, false, true, true),
                Array(true, false, true, true),
                Array(false, true, false, false),
                Array(false, false, true, false))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
[ [ T, T, T, T ]
, [ T, T, T, T ]
, [ T, T, T, T ]
, [ F, T, T, T ] ]
```

5.3 Probabilistic Linear Algebra Language

Probabilistic models are used in many applications including artificial intelligence, machine learning, cryptography, and economics. Probabilistic programming languages have proven to be successful for expressing such stochastic models in a declarative style without worrying about computational aspects [11, 28, 27, 39]. As an example, an important computer vision application was recently expressed in only 50 lines of code in the Picture probabilistic programming language [40].

In this paper, our aim is not to make Pilatus a full-fledged probabilistic programming language. Instead, we show how we can encode Boolean probability distributions in Pilatus in the form of a semi-ring. This means that we support the conjunction and disjunction

```
sealed trait ShortestPath {
  def add(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => Unreachable
    case (x, Unreachable) => Unreachable
    case (Distance(v1), Distance(v2)) => Distance(v1 + v2)
  }
  def min(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => x
    case (x, Unreachable) => x
    case (Distance(v1), Distance(v2)) => Distance(math.min(v1, v2))
  }
}
case class Distance(v: Int) extends ShortestPath
case object Unreachable extends ShortestPath

class SemiRingTropical extends SemiRing[ShortestPath] {
  def add(a: ShortestPath, b: ShortestPath) = a.min(b)
  def mult(a: ShortestPath, b: ShortestPath) = a.add(b)
  def one: ShortestPath = Distance(0)
  def zero: ShortestPath = Unreachable
}

object Semantics {
  // ...
  implicit val semiRingTropical = new SemiRingTropical
}
```

Figure 13 The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph shortest-path problems.

between two Boolean distributions. Also, the zero and one elements of the semi-ring correspond to the distribution with the probability of one for false and true, respectively. As a side effect of the compositional design of Pilatus, we can support vectors and matrices of such distributions as well, virtually for free. Thus, Pilatus supports probabilistic graphs and the associated path queries, similar to systems such as ProbLog [15].

Figure 14 shows the tagless final implementation for Boolean distributions. The `BoolProb` data type has a list of probabilities assigned to each Boolean value. This data type is actually a *probability monad* [26, 19]. As is customary with monad implementation in Scala, the `flatMap` method represents the bind operator of the monad, and the `apply` method of the companion object represents the unit operator. The `normalise` method makes sure that the list of probabilities associated to each Boolean value has distinct Boolean values, and that the probabilities sum up to one.

There are many alternative implementations for the probability monad such as lazy trees [39] with the possibility to support distributions for values other than Booleans. Furthermore, in this context one can use various optimisations such as variable elimination [16]. Finally, it is possible to explore other inference mechanisms [42]. All these aspects are orthogonal to the purposes of this work, and Pilatus can be extended to support all these features, which we leave as exercises to the reader.

```
case class BoolProb(l: List[(Boolean, Double)]) {
  def flatMap(f: Boolean => BoolProb): BoolProb = {
    val ll = for(x <- l; y <- f(x._1).l) yield { y._1 -> (y._2 * x._2) }
    BoolProb(ll).normalise()
  }
  def normalise(): BoolProb = {
    val sum = l.map(_._2).sum
    val nl = l.groupBy(_._1).mapValues(_.map(_._2).sum / sum)
    BoolProb(nl.toList)
  }
}
object BoolProb {
  def apply(v: Boolean): BoolProb = BoolProb(List(v -> 1.0))
}

class SemiRingBoolProb extends SemiRing[BoolProb] {
  def add(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) one else b)
  def mult(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) b else zero)
  def one: BoolProb = BoolProb(true)
  def zero: BoolProb = BoolProb(false)
}

object Semantics {
  // ...
  implicit val semiRingBoolProb = new SemiRingBoolProb
}
```

Figure 14 The tagless final implementation using the Boolean probability monad for semi-ring operations.

▶ **Example 4.** When we give the adjacency matrix of the probabilistic graph of Figure 1b as the input to the example program, the evaluation is as follows:

```
  import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolProb }
  def flip(p: Double): BoolProb = BoolProb(List(true -> p, false -> (1 - p)))
  val adj = Array(Array(flip(0), flip(0), flip(0.1), flip(0.5)),
                  Array(flip(0.8), flip(0), flip(0.3), flip(0.6)),
                  Array(flip(0), flip(0.9), flip(0), flip(0)),
                  Array(flip(0), flip(0), flip(0.2), flip(0)))
  val m = build(4)(i => build(4)(j => adj(i)(j)))
  example(m)
  // output:
  [ [ 1, 0.1, 0.2, 0.5 ]
  , [ 0.8, 1, 0.4, 0.8 ]
  , [ 0.7, 0.9, 1, 0.5 ]
  , [ 0, 0.2, 0.2, 1 ] ]
```

As in the previous section, the example program computes the all-pairs path with maximum length of two. Hence, the result matrix is the probability of the existence of a path by traversing at most one intermediate node.

```
class DualSemiRing[R](implicit val sr: SemiRing[R])
extends SemiRing[(R, R)] {
  type Dual = (R, R)
  def add(a: Dual, b: Dual) = (sr.add(a._1, b._1), sr.add(a._2, b._2))
  def mult(a: Dual, b: Dual) =
    (sr.mult(a._1, b._1), sr.add(sr.mult(a._1, b._2), sr.mult(a._2, b._1)))
  def one: Dual = (sr.one, sr.zero)
  def zero: Dual = (sr.zero, sr.zero)
}
class DualRing[R](implicit val ring: Ring[R])
extends DualSemiRing[R] with Ring[(R, R)] {
  def neg(a: Dual) = (ring.neg(a._1), ring.neg(a._2))
}
object Semantics {
  // ...
  implicit def dualSemiRing[R: SemiRing] = new DualSemiRing[R]
  implicit def dualRing[R: Ring] = new DualRing[R]
}
```

Figure 15 The tagless final implementation using dual numbers for ring operations.

5.4 Differentiable Linear Algebra DSL

Many applications in machine learning such as training artificial neural networks require computing the derivate of an objective function. In many cases, the manual derivation of analytical derivatives is not a practical solution, as it is error prone and time consuming. Hence, several techniques were developed for automating the derivation process.

Automatic differentiation (or algorithmic differentiation) is one of the most well-known techniques to systematically compute the derivative of a program. This technique systematically applies the chain rule, and evaluates the derivatives for the primitive arithmetic operations (such as addition, multiplication, etc.) [4].

Among different implementations of automatic differentiation, here we show the forward mode technique using *dual numbers*. In this implementation, every number is augmented with an additional component, which maintains the computed derivative value. Correspondingly, all primitive operations should be augmented with the appropriate derivation computation.

Figure 15 demonstrates the generic tagless final interface for the dual number representation of a ring. This interface uses the pair representation for dual numbers, in which the first component is the normal value, whereas the second component is the derivative value. The second component in the implementation of the addition operator reflects the addition rule of derivation ($d(a + b) = da + db$), whereas the one in multiplication reflects the multiplication rule ($d(a \times b) = da \times b + a \times db$).

▶ **Example 5.** Let us consider again the example matrix given in Figure 1c. By representing this input matrix using dual numbers, our running example is evaluated as follows:

```
import Semantics.{ pullArrayInterOps, loopingInt, ringInt, dualRing }
val adj = Array(Array(0 -> 0, 0 -> 0, 1 -> 0, 5 -> 1),
                Array(8 -> 0, 0 -> 0, 3 -> 2, 6 -> 0),
                Array(0 -> 0, 9 -> 3, 0 -> 0, 0 -> 0),
                Array(0 -> 0, 0 -> 0, 2 -> 1, 0 -> 0))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
```

```
[ [ 1 -> 0, 9 -> 3, 11 -> 7, 5 -> 1 ]
, [ 8 -> 0, 28 -> 27, 23 -> 8, 46 -> 8 ]
, [ 72 -> 24, 9 -> 3, 28 -> 27, 54 -> 18 ]
, [ 0 -> 0, 18 -> 15, 2 -> 1, 1 -> 0 ] ]
```

More specifically, computing the square of this matrix results in:

```
val m2 = compose(m, m)
println(getString(m2))
// output:
[ [ 0 -> 0, 9 -> 3, 10 -> 7, 0 -> 0 ]
, [ 0 -> 0, 27 -> 27, 20 -> 6, 40 -> 8 ]
, [ 72 -> 24, 0 -> 0, 27 -> 27, 54 -> 18 ]
, [ 0 -> 0, 18 -> 15, 0 -> 0, 0 -> 0 ] ]
```

This output is the same as what we have observed in Figure 1c.

6 Staging and Optimisation

In this section, we show how to use multi-stage programming (MSP, or just *staging*) to improve the performance of PILATUS programs, by removing the abstraction overhead incurred by the high-level programming features we use to make our DSL polymorphic. We use Squid [51, 52], a type-safe meta-programming framework that supports MSP. Squid is implemented in Scala as a macro library, making its usage straightforward and user-friendly.

6.1 Preliminaries on Squid and Multi-Stage Programming

Squid makes use of *quasi-quotes* to manipulate program fragments; this way, one can both compose programs together (multi-stage programming) and pattern-match on them to perform rewritings, thereby achieving *quoted staged rewriting* [50].

Quasi-quotes. Given a Scala expression `e` of type `T`, the quasi-quote expression `code"e"` has type `Code[T]` and represents a program fragment whose representation can be manipulated programmatically. Crucially, quasi-quotes may contain *holes*, delineated by the `${...}` escape syntax.[6] When constructing a program fragment, code inside of a hole is evaluated and inserted in place of the hole. For example, `val part = code"List(1,2,3)"; code"2 * $part.length"`, which has type `Code[Int]`, evaluates to `code"2 * List(1,2,3).length"`.

Runtime Compilation. After composing a program at run time using quasi-quotes, one can then either dump a stringified version of the code inside a file to be compiled and run later, or runtime-compile it on the fly, using the `.compile` method, which will produce bytecode that can then be run efficiently. After the one-off cost of runtime compilation, a definition such as `val f = code"(x: Int)\[ => x + 1".compile` will be as efficient as `val f = (x: Int)\] => x + 1`.

Multi-Stage Programming (MSP). The goal of MSP is to turn a program which contains abstractions and indirections into a *code generator*: instead of producing the program's result directly, this staged program will produce *code* that is straightforward and free of

[6] When the escaped expression is a simple identifier, one can leave out the curly braces.

abstractions, to compute the program's result more efficiently. To achieve this, one annotates the non-static parts of the program (those that should be executed later) using quasi-quotes and `Code` types. Thanks to runtime compilation, the staged program effectively partially evaluates the fixed parts of a program even if they depend on values obtained at runtime.

Code Pattern Matching and Rewriting. Squid extends the classical MSP ability with code pattern matching and rewriting (quasi-quotes are allowed in patterns), which lets programmers inspect already-composed code fragments in a type-safe way. As we will see in Section 6.4, in practice this saves programmers the trouble of having to define their own inspectable program representations before turning them into code.

6.2 Staging Pilatus

Thanks to the polymorphic nature of Pilatus, it is quite straightforward to turn a given semantics into a multi-stage program. All we need to do is to provide an evaluation semantics which manipulates program fragments instead of normal values, and which composes these fragments together instead of directly evaluating the results of each operation.

Figure 16 shows the staged versions of some of the Pilatus interfaces. A `RingCode[T]` is a ring implementation[7] that manipulates `Code[T]` ring elements (the `RingCode[T]` class extends `Ring[Code[T]]`). The `CodeType[T]` type class is used to automatically infer runtime type representations, which is necessary for Squid program manipulation. Notice that the implicit `ringCode` definition takes an implicit argument of type `Code[Ring[T]]`. This works out of the box, because Squid can turn an implicit `Ring[T]` into a `Code[Ring[T]]` automatically, lifting the code used for generating the implicit.

As an example, consider the following polymorphic Pilatus program:

```
def polymorphicProgram[R: Ring](a: R, b: R): R = mult(add(a, one), b)
```

And the following two usages, one with a direct `R = Int` interpretation, and one with a staged `R = Code[Int]` one:

```
import Semantics.{ ringInt }, StagedSemantics.{ ringCode }
Console.print("Enter an integer number: ")
val k = Console.readInt
val f_slow = (x: Int) => polymorphicProgram(x, k)
val f_code = (x: Code[Int]) => polymorphicProgram(x, Const(k))
val f_fast = code"(x: Int) => ${f_code}(x)".compile
```

The `Const` constructor turns a primitive value (here an `Int`) into a code value (here a `Code[Int]`). Notice that we insert `f_code` into a quasi-quote even though it is not a code value, but a function from code to code; in fact, it is implicitly lifted by Squid [51].

Assuming the user enters the number `27` on the console, the code generated at runtime for `f_fast` will be equivalent to `(x: Int)\[ => Semantics.ringInt.mult(Semantics.ringInt.add(x, 1), 27)` which, after inlining of the statically-dispatched `ringInt` methods, corresponds to `(x: Int)\] => (x + 1)* 27`. To understand why this is much more efficient than the `f_slow` version, consider that the evaluation of `f_slow` has to go through virtual dispatch of all the ring operations; moreover, it also has to use boxed representations of the manipulated

[7] Strictly speaking, this implementation does not form a ring, because for example `code"2+1"` is not the same as `code"1+2"` – though they are "morally" equivalent as they represent equivalent programs.

integer values due to the generic context in which ring operations are defined, which requires repeated allocations and unwrapping of boxed integers. As a result, in a realistic workload, even the just-in-time compiler will typically not manage to make that code as fast as the straightforward primitive operations performed by `f_fast`.[8]

This kind of overhead easily compounds as we introduce more abstractions, to the point where non-staged abstract programs end up being *orders of magnitude* slower than the staged versions [72], as we will see in Section 7.

6.3 Staged Representation Optimisations

An interesting aspect of MSP is that it lets us define data structures made of partially-staged data. For example, if we want to partially evaluate the allocation of pairs and the selection of their components, we can use representations of type `(Code[A],Code[B])` instead of `Code[(A,B)]`.

This comes in useful when representing dual numbers in our staged interpreter. We can implement an alternative to `DualRing` that is specialised for handling code values, and define its operations accordingly, for example:

```
def mult(a: (Code[R], Code[R]), b: (Code[R], Code[R])) =
  (code"$sr.mult(${a._1}, ${b._1})",
   code"$sr.add($sr.mult(${a._1}, ${b._2}), $sr.mult(${a._2}, ${b._1}))")
```

Note that in the code above, we use program fragments `a._1` and `b._1` *several times*. This is fine, because the default intermediate representation that Squid uses to encode program fragment is based on the A-normal form [20], which let-binds every subexpression to a local variable, and thus avoids code duplication [51, 50]; in other words, by inserting a given code value in several places, we only duplicate variable references.

6.4 Algebraic Optimisations

Thanks to the staged interpretations of Pilatus, which allows us to manipulate program fragments as first-class values, we can leverage the algebraic properties of ring structures to perform optimisations. To do so, we can *extend* the staged ring implementation, so that we use the normal staged method implementations by default, and *override* those methods where there is a potential for algebraic optimisations. The goal of the overridden methods is to return simplified program fragments based on the shape of their inputs.

This technique is similar to the original tagless final [10] and polymorphic embedding [30] approaches to algebraic optimisation. The main difference is that thanks to Squid's analytic capabilities, we do not need to create our own intermediate symbolic representation of programs, and instead we can pattern-match on code values directly.

An implementation of this optimised staged semantics for rings is given in Figure 17. When used in pattern position, traditional quasi-quote escapes `${...}`, which *insert* code values into bigger expressions, are written `$${...}` instead.

---

[8] Runtime systems like the CLR for C# avoid boxing by performing runtime specialisation of generic code, but that only achieves a small part of all the optimisation and partial evaluation we are interested in here. C++ templates can perform advanced compile-time specialisation, which could get us closer to our goal (though this means specialisation could not rely on runtime values), but they are difficult and heavyweight, yet much less flexible because they do not allow for first-class manipulation of code values.

```
import squid.IR.Predef._ // import the ‘Code’, ‘CodeType’ and ‘code’ functionalities

class SemiRingCode[T: CodeType](val sr: Code[SemiRing[T]]) extends SemiRing[Code[T]] {
  def add(a: Code[T], b: Code[T]) = code"$sr.add($a, $b)"
  def mult(a: Code[T], b: Code[T]) = code"$sr.mult($a, $b)"
  def one: Code[T] = code"$sr.one"
  def zero: Code[T] = code"$sr.zero"
}
class RingCode[T: CodeType](val ring: Code[Ring[T]])
extends SemiRingCode[T](ring) with Ring[Code[T]] {
  def neg(a: Code[T]) = code"$ring.neg($a)"
}
class PullArrayCodeOps[E: CodeType]
extends PullArrayOps[Code[PullArrayInter[E]], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): Code[PullArrayInter[E]] =
    code"PullArrayInter($len, $f)"
  def get(arr: Code[PullArrayInter[E]])(i: Code[Int]): Code[E] = code"$arr.f($i)"
  def length(arr: Code[PullArrayInter[E]]): Code[Int] = code"$arr.len"
}

object StagedSemantics {
  implicit def semiRingCode[T: CodeType]
    (implicit cde: Code[SemiRing[T]]): SemiRing[Code[T]] = new SemiRingCode(cde)
  implicit def ringCode[T: CodeType]
    (implicit cde: Code[Ring[T]]): Ring[Code[T]] = new RingCode(cde)
  // other similar definitions elided...
}
```

Figure 16 The tagless final encoding of compiled rings, pull arrays, and control-flow constructs.

Many more algebraic rewritings can be added to perform partial evaluation and normalization of program fragments. We have omitted them for the sake of brevity. Furthermore, one can encode the algebraic properties of modules (cf. Section 3.3) and linear maps (cf. Section 3.4) as rewrite rules, which we leave for the future.

6.5 Fixed-Size Matrix DSL

In some applications, such as computer vision, the matrices or vectors have a small size and sometimes their size are statically known (e.g. a vector of size 3 to show a point in the 3D space). In these cases the necessary memory for the corresponding arrays can be allocated at compile time (or even stack allocated), leading to better performance and memory consumption at run time.

Pilatus can be instantiated with an evaluator that makes sure that the length of arrays is known during the compilation time. In this case, the representation of a pull array is a sequence of the symbolic representation for each element. Furthermore, the representation for its length is an integer, instead of a symbolic representation. Interestingly, this representation is the same as the one shown in Figure 10, but with the `E` type instantiated to multi-stage code types.

```
class SemiRingOptCode[T: CodeType](sr: Code[SemiRing[T]]) extends SemiRingCode[T](sr) {
  override def add(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => a
    case (code"$$sr.zero", _) => b
    case _ => super.add(a, b)
  }
  override def mult(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => code"$sr.zero"
    case (code"$$sr.zero", _) => code"$sr.zero"
    case (_, code"$$sr.one") => a
    case (code"$$sr.one", _) => b
    case _ => super.mult(a, b)
  }
}

class RingOptCode[T: CodeType](ring: Code[Ring[T]]) extends RingCode[T](ring) {
  override def neg(a: Code[T]) = a match {
    case code"$$ring.zero" => code"$ring.zero"
    case _ => super.neg(a)
  }
}
```

Figure 17 The tagless final encoding of the compiled library of PILATUS, which applies algebraic optimisations for the elements of matrices.

6.6 Fused DSL

Deforestation [67, 25, 63, 13] is a well-known technique used in functional languages in order to remove the unnecessary intermediate data structures. This removal has a positive effect on both memory consumption and run-time performance, thanks to the removal of unnecessary memory allocations and avoidance of unnecessary computations.

One of the key advantages of using pull arrays is providing deforestation. However, to benefit from this feature, one should provide an appropriate representation for pull arrays which avoids materialisation. This can be achieved by symbolically maintaining the length and the constructor function. Whenever the array is indexed or the length of array is needed, instead of creating a symbolic representation for them, we can use the maintained length and constructor function.

Figure 18 represents the implementation of fused pull array, and a compiler allowing deforestation for PILATUS.

7 Evaluation

In this section, we show how multi-stage programming and rewriting can make PILATUS faster than the high-level implementation, while being competitive with a handwritten low-level implementation. We use several micro benchmarks consisting of a pipeline of vector operations such as addition, dot product, and norm. Each benchmark is tested with five different approaches:

- PILATUS by using a native array without optimisation
- PILATUS by using a pull array without optimisation
- PILATUS by using a pull array with staging
- PILATUS by using a pull array with staging and fusion
- A handwritten low-level optimised implementation

```
case class PullArrayCode[E](len: Code[Int], f: Code[Int] => Code[E])

class PullArrayCodeFusedOps[E]
      extends PullArrayOps[PullArrayCode[E], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): PullArrayCode[E] =
    PullArrayCode(len, f)
  def get(arr: PullArrayCode[E])(i: Code[Int]): Code[E] =
    arr.f(i)
  def length(arr: PullArrayCode[E]): Code[Int] =
    arr.len
}
```

Figure 18 The tagless final encoding of the compiled library of PILATUS, which removes all unnecessary intermediate arrays.

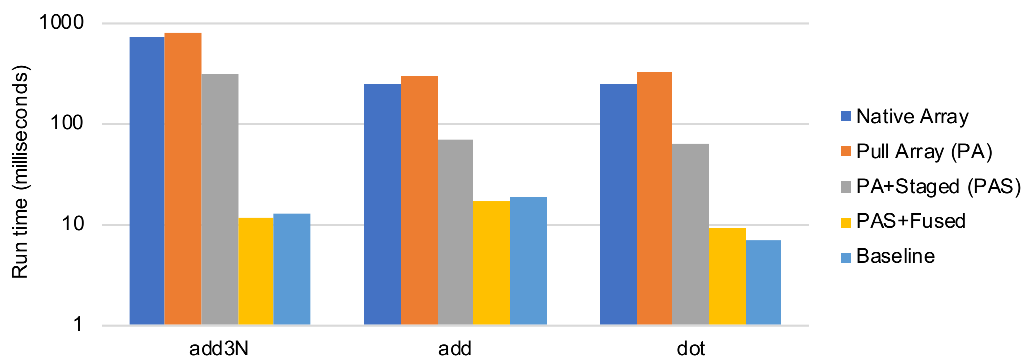


Figure 19 Performance comparison between PILATUS with different configurations and a baseline low-level implementation.

The experiments are performed on a six-core Intel Xeon E5-2620 v2 processor with 256GB of DDR3 RAM (1600Mhz), with Scala version 2.12.8 running on the OpenJDK 64-Bit Server VM (build 24.95-b01) with Java 1.7.0_101.

Figure 19 shows the performance results. For these experiments, the input vectors are all stored in a native JVM array, consisting of one million integer elements. Based on these experiments we make the following observations. First, changing the usage of native array representation to a pull array causes a minor performance overhead. This is because the JIT of JVM is unable to remove the overhead caused by the lambdas used in a pull array. Second, the overhead of lambdas as well as several other overheads are removed by using staging. This performance improvement is between 2.5x to 5x. Finally, the intermediate arrays are successfully removed by benefiting from the fusion of pull arrays (cf. Section 6.6). The improvement varies between 4x to 26x depending on the number of removed intermediate arrays. This makes the staged and fused PILATUS competitive with the baseline low-level implementation.

8 Related Work

8.1 Linear Algebra Languages and Libraries

The R programming language [56] is widely used by statisticians and data miners. It provides a standard language for statistical computing that includes arithmetic, array manipulation, object oriented programming and system calls. It is a Turing-complete language. By contrast,

with our language we chose to focus solely on linear algebra operations. This minimalistic approach results in a language that is not Turing Complete, but is nevertheless polymorphic in various dimensions.

The Spiral [55] project introduces the languages SPL [71], OL [22], and more recently LL [62] which mainly captures the non-iterative matrix operations of PILATUS. Furthermore, the intermediate languages Σ-SPL [23] and Σ-LL [62] expose opportunities to perform loop fusion. One interesting direction is to use the search-based techniques to perform global optimisations offered by Spiral. Furthermore, Spiral in Scala [47] supports loop unrolling and fixed size matrices by using the staging facilities offered by LMS [57] and abstracting over data layout. Kiselyov [37] has used the tagless final approach and staging facilities of MetaOCaml in order to implement a linear algebra DSL based on rings and pull arrays (but not modules and linear maps) and has implemented many of the optimisations that we have presented in this paper. However, to the best of our knowledge, none of these projects consider graph algorithms, probabilistic programming, and automatic differentiation of linear algebra.

In the Haskell programming language, Dolan [17] implements a linear algebra library which uses different semi-ring configurations for expressing graph algorithms, as well as several other algorithms. We can easily extend PILATUS in order to support the additional semi-ring configurations used in that work. However, even though Elliot [18] implements a library for forward automatic differentiation in Haskell, Dolan [17] does not consider automatic differentiation. In addition, as both these two libraries are implemented without using any multi-stage programming facilities, none of them can support the staged libraries provided by PILATUS.

8.2 Deforestation and Array Fusion

Deforestation [67] and the corresponding short cut techniques [25, 63, 13] were introduced for functional languages for removing the unnecessary intermediate collections. Recently, these techniques have been implemented as a library using multi-stage programming [33, 38, 59].

On the other hand, in the high-performance functional array programming there are the two well-known array representations, which also achieve deforestation: pull arrays and push arrays [2, 12]. Each one of these two array representations comes with its own benefits, for which [64] combines the benefits of these two complementary representations. Pull arrays have been used for various DSLs [3, 37, 60] to produce efficient low-level code from the high-level specification of linear algebra programs. However, none of these systems consider other domains presented in this paper.

8.3 Automatic Differentiation and Differentiable Programming

Many techniques for finding optima of a given objective function (such as gradient-descent-based techniques) require the derivative of that function. Automatic differentiation (AD) [35] is one of the key techniques for automatically computing the derivative of a given program. Thus, these tools are an essential component of many machine learning frameworks. There is a large body of work on AD frameworks for imperative programming languages such as Tapenade [29] for C and Fortran, ADIFOR [7] for Fortran, and Adept [31] and ADIC [46] for C++, ADiMat [8], ADiGator [70], and Mad [21] performs AD for MATLAB programs, whereas AutoGrad [41], Theano [6], Tensorflow [1] performs AD for a subset of Python programs. There are also AD tools developed for functional languages such as DiffSharp [5] for F#, d$\widetilde{\text{F}}$ [61] for a subset of F#, Stalingrad [53] for a dialect of Scheme, as well as the

work by Karczmarczuk [34] and Elliott [18] for Haskell. The most similar work to ours is Lantern [69], which uses the multi-stage programming features provided by LMS [57] in order to perform AD for numerical programs written in a subset of Scala. A key feature provided by Lantern is supporting reverse-mode AD by using *delimited continuations* [14], which can also be supported by Pilatus, which we leave it for the future. All the presented frameworks are only for differentiable programming, whereas Pilatus supports graph computations and probabilistic programming, while providing algebraic-based and compiler optimisations for improving performance.

8.4 Probabilistic Programming

Probabilistic programming languages (PPL) can express stochastic models in a productive manner without worrying about the low-level details [28]. Infer.NET [44], Picture [40], and probabilistic C are examples of imperative PPLs, whereas, BUGS [24], STAN [11], and Church [27] are functional PPLs. Figaro [54] is an object-oriented probabilistic programming language which is defined as an EDSL on top of Scala. PRISM [58], BLOG [43], and ProbLog [15] are examples of Logical PPLs. Pilatus is inspired by both logical PPLs and the embedding of functional PPLs [39]. Although Pilatus has a more limited expressivity power in comparison with other logical probabilistic programs, it supports various other domains such as differentiable programming for linear algebra workloads.

9 Conclusions

In this paper, we have presented Pilatus, a polymorphic linear algebra language. This language is embedded in Scala and can have several interpretations supporting various domains such as standard matrix algebra, all-pairs reachability and shortest-path computations for graphs, logical probabilistic programming, and differentiable programming. In order to compensate the performance penalty caused by the abstraction overheads, Pilatus uses multi-stage programming. Furthermore, thanks to the mathematical nature of Pilatus, we use algebraic rewrite rules to further improve the performance.

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.
2. Johan Anker and Josef Svenningsson. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*, pages 1–12, 2013.
3. Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.
4. Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint*, 2015. `arXiv:1502.05767`.
5. Atilim Gunes Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. DiffSharp: Automatic Differentiation Library. *arXiv preprint*, 2015. `arXiv:1511.07727`.
6. James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

7 Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

8 Christian H Bischof, HM Bucker, Bruno Lang, Arno Rasch, and Andre Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 65–72. IEEE, 2002.

9 Jacques Carette and Oleg Kiselyov. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. *Sci. Comput. Program.*, 76(5):349–375, May 2011.

10 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

11 Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

12 Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, NY, USA, 2012. ACM.

13 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

14 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

15 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

16 Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Learning in graphical models*, pages 75–104. Springer, 1998.

17 Stephen Dolan. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 101–110, New York, NY, USA, 2013. ACM.

18 Conal M. Elliott. Beautiful Differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 191–202, New York, NY, USA, 2009. ACM.

19 Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

20 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

21 Shaun A Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.

22 Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

23 Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal Loop Merging for Signal Transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 315–326, 2005.

24 Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, pages 169–177, 1994.

25 Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA, pages 223–232. ACM, 1993.

26 Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

27 Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint*, 2012. `arXiv:1206.3255`.

28 Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

29 Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

30 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*, pages 137–148. ACM, 2008.

31 Robin J. Hogan. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

32 Paul Hudak. Building Domain-specific Embedded Languages. *ACM Comput. Surv.*, 28(4es), December 1996.

33 Manohar Jonnalagedda and Sandro Stucki. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2015.

34 Jerzy Karczmarczuk. Functional differentiation of computer programs. *ACM SIGPLAN Notices*, 34(1):195–203, 1999.

35 Gershon Kedem. Automatic Differentiation of Computer Programs. *ACM Trans. Math. Softw.*, 6(2):150–165, June 1980.

36 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

37 Oleg Kiselyov. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101, 2018.

38 Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 285–299, New York, NY, USA, 2017. ACM.

39 Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009.

40 Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

41 Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, 2015.

42 Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 603–616. ACM, 2018.

43 Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. *Statistical relational learning*, page 373, 2007.

44 Tom Minka, John Winn, John Guiver, and David Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge, 2014.

45 Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

46 Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010.

47 Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 125–134, New York, NY, USA, 2013. ACM.

48 Bruno C.d.S Oliveira and William R Cook. Extensibility for the Masses. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.

49 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

50 Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 131–145, New York, NY, USA, 2017. ACM.

51 Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Squid: Type-safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 56–66, New York, NY, USA, 2017. ACM.

52 Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying Analytic and Statically-typed Quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL):13:1–13:33, December 2017.

53 Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

54 Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical Report 137, Charles River Analytics, 2009.

55 Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

56 R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL: `http://www.R-project.org/`.

57 Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

58 Taisuke Sato. A Glimpse of Symbolic-Statistical Modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, October 2008.

59 Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming*, 28:e10, 2018.

60 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. ACM.

61 Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon Peyton Jones, and Christoph Koch. Efficient Differentiable Programming in a Functional Array-Processing Language. *arXiv preprint*, 2018. `arXiv:1806.02136`.

62 Daniele G. Spampinato and Markus Püschel. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32. ACM, 2014.

63 Josef Svenningsson. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132. ACM, 2002.

64 Bo Joel Svensson and Josef Svenningsson. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, NY, USA, 2014. ACM.

65 Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

66 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

67 Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

68 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

69 Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *Advances in Neural Information Processing Systems*, pages 10200–10211, 2018.

70 Matthew J Weinstein and Anil V Rao. Algorithm 984: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading. *ACM Trans. Math. Softw*, 2016.

71 Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

72 Jeremy Yallop. Staged Generic Programming. *Proc. ACM Program. Lang.*, 1(ICFP):29:1–29:29, August 2017.