First-class Isomorphic Specialization by Staged Evaluation

Alexander Slesarenko
Shannon Laboratory, Huawei Technologies, Moscow, Russia
alexander.slesarenko@huawei.com

Alexander Filippov
Shannon Laboratory, Huawei Technologies, Moscow, Russia
filippov.alexander@huawei.com

Alexey Romanov
Shannon Laboratory, Huawei Technologies, Moscow, Russia
alexey.romanov@huawei.com

Abstract

The state of the art approach for reducing complexity in software development is to use abstraction mechanisms of programming languages such as modules, types, higher-order functions etc. and develop high-level frameworks and domain-specific abstractions. Abstraction mechanisms, however, along with simplicity, introduce also execution overhead and often lead to significant performance degradation. Avoiding abstractions in favor of performance, on the other hand, increases code complexity and cost of maintenance.

We develop a systematic approach and formalized framework for implementing software components with a first-class specialization capability. We show how to extend a higher-order functional language with abstraction mechanisms carefully designed to provide automatic and guaranteed elimination of abstraction overhead.

We propose *staged evaluation* as a new method of program staging and show how it can be implemented as zipper-based traversal of program terms where one-hole contexts are generically constructed from the abstract syntax of the language.

We show how generic programming techniques together with staged evaluation lead to a very simple yet powerful method of *isomorphic specialization* which utilizes first-class definitions of isomorphisms between data types to provide guarantee of abstraction elimination.

We give a formalized description of the isomorphic specialization algorithm and show how it can be implemented as a set of term rewriting rules using active patterns and staged evaluation.

We implemented our approach as a generic programming framework with first-class staging, term rewriting and isomorphic specialization and show in our evaluation that the proposed capabilities give rise to a new paradigm to develop domain-specific software components without abstraction penalty.

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

Keywords Generic programming; polytypic programming; staging; multi-stage programming; domain-specific languages; DSL; specialization; isomorphisms

1. Introduction

Most modern software development is done in high-level languages, reducing program complexity through their built-in ab-

straction mechanisms (such as module systems, classes, interfaces, etc.). These mechanisms are often used to create domain-specific languages (DSLs) which allow a higher level of abstraction for programs in a given domain (e.g. Spark [33] can be considered as a DSL for distributed programming).

However, these mechanisms generally also introduce execution overhead (often called *abstraction regret* [4, 21] or *abstraction penalty*) and the trade-off between abstraction and performance is often difficult.

Modern advances in compilation techniques, such as just-in-time compilation and whole program optimization generally can't eliminate the overhead completely and don't scale well with the size of the program.

A recent trend is development of DSL-centric frameworks where abstractions can be introduced and software can be built without abstraction penalty [5, 23, 24], though it may require development of special tools [3].

In such frameworks, DSL compilers allow mapping of problem-specific abstractions directly to low-level architecture-specific programming models such as [12, 20]. However, the development of DSLs is difficult by itself, and adding a compilation stage considerably increases this difficulty.

While compiling DSLs is a promising approach, we believe that there is still much to be done in tackling the abstraction penalty problem.

In particular, using multiple DSLs together in a single application is a well-known problem [30]. Existing DSL-centric frameworks usually require additional efforts for integration and interoperation of multiple DSLs.

Another problem is rapid prototyping of new DSLs and applications developed with DSLs. While there is evidence that even simple techniques can lead to significant benefits [29], we think that this problem requires a more generic and systematic approach.

In general, we believe that the problems of popular parallel programming [2] and of abstraction overhead are two sides of the same coin. In other words, a generic solution of the latter will also lead to a generic solution of the former.

The present work originated from an attempt to apply the staging [32] approach proposed in Lightweight Modular Staging (LMS) [22] to the domain of nested data parallelism (NDP).

The NDP implementation in Haskell [8] led to various Haskell extensions (such as support for non-parametric polymorphism [7]) and new transformation techniques [18]. It was also noticed [6] that NDP could have a generic programming formulation.

We implemented NDP as a polytypic library in Scala first [27], and then as a deep embedding [28] in the spirit of LMS. These experiments demonstrated that staging can have some non-trivial interaction with polytypic (or generic) programming. In this paper, we show how we can combine these directions, with useful results both for staging and for generic programming.

In the present paper, we advocate that it is useful to have first-class declarations of isomorphisms between data types to implement domain-specific compilers. In particular, we observe that isomorphisms can serve as a bridge between two levels of indirection (abstraction layers) in user-defined types. This helps in translating program code from higher levels down to some *core language*, performing specialization along the way.

The way we define and use isomorphisms is where our work is different from previous approaches [14]. We are not inferring isomorphisms but instead we require a programmer to think about isomorphic representations of domain objects in his/her application. We require the isomorphisms to be explicitly specified and captured in the application domain. On the surface of the programming language we relate specifications of isomorphisms to declarations of alternative concrete representations of abstract types. This is just one of the possible implementations at the front-end of the language and should not be considered as a limitation of the presented approach.

The key point is that after the isomorphisms are identified in the application domain, they play an important role interacting with primitives of the core language. Rewriting rules capture this interaction between isomorphisms and primitives. For each polymorphic primitive of the core language there are rules which tell how the primitive composes with isomorphisms. The core language itself can be thought of as another domain-specific language. In this paper we use the Array type to represent such a domain specifically, but any other set of types and core primitives can be used as well.

In Sections 2 and 4 we explicitly consider the case of multiple concrete implementations for an abstract type. This is the key point where non-trivial interaction of staging and generic programming happens.

We define the notion of *staged evaluation* to explicitly connect staging to the evaluation semantics of the source language. This semantics implements a virtual method invocation mechanism associated with inheritance. The staged evaluation process mimics this dynamic invocation while producing a graph representation of the source program.

Thus, if we perform staged evaluation of a function call like `mvm(new DenseMatr(...), vec)` with an instance of a concrete matrix type we get a different program graph from the one produced by evaluating `mvm(new SparseMatr(...), vec)`. This is a consequence of presence of virtual method calls in the semantics of the source language.

Thus motivated and inspired by our previous results, we develop a systematic approach and a new specialization technique for implementing domain-specific abstractions in a generic programming framework with first-class staging, rewriting and specialization capabilities.[1] Our approach is based on a combination of generic programming and staging.

In particular we present the following main contributions:

1. We describe a new method of program staging (we call it *staged evaluation*) and show how it naturally arises from the evaluation semantics of the language to be staged. Our staging algorithm transforms program terms into directed acyclic graphs (DAGs) as intermediate representation (IR). We show how staged evaluation can be described as a zipper-based [15] traversal of program terms where one-hole contexts [19] are generically constructed from evaluation reduction contexts.

2. We show how generic programming techniques together with staged evaluation lead to a very simple yet powerful method of *isomorphic specialization* which utilizes first-class definitions

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

of isomorphisms between data types to guarantee abstraction elimination.

3. We give a formal description of the isomorphic specialization algorithm and show that it can be implemented as a set of graph rewriting rules using active patterns [11, 31] and staged evaluation.

4. And last but not the least, the ideas described in this paper are implemented in our DSL-centric generic programming framework with first-class staging. We show in our evaluation that a programming framework with first-class isomorphic specialization gives rise to a new paradigm and design pattern for development of both compiled DSLs and, more broadly, domain-specific software components without abstraction penalty.

The paper is structured as follows. Section 2 gives an informal yet precise description of our approach with a motivating example from linear algebra. Section 3 formally describes our language, staged evaluation and isomorphic specialization. In Section 4 we evaluate our approach by comparing performance of various specializations. In Section 5 we compare our approach with related work and in Section 6 we conclude.

2. First-class Isomorphic Specialization in a

Nutshell

In this section we demonstrate the essence of isomorphic specialization using a simple example. We consider the matrix-vector multiplication (`mvm`) problem as our working example, which is shown in Figure 1. We use a subset of the Scala language to express necessary abstract types and their various implementations.

```
trait Vec[T] {
  def length: Int
  def dotProduct(vec: Vec[T]): T
}
trait Matr[T] {
  def rows: Array[Vec[T]]
}
def mvm(m: Matr[T], v: Vec[T]): Vec[T] = {
  val rs = m.rows // array of rows
  rs map { r ⇒ r.dotProduct(v) }
}
```

Figure 1. Abstract matrix and vector. **trait** in Scala is similar to **interface** in Java

Imagine an object-oriented framework where the `mvm` algorithm can be expressed using interfaces of abstract data types like `Matr[T]` and `Vec[T]`. Then `mvm` can be executed using some concrete classes implementing these interfaces. These implementations use different data structures for the in-memory representation of the data. Suppose the following: first, for each abstract type we have several implementations which have different performance characteristics (e.g. depending on sparseness of our data); second, we want to perform dynamic selection of the best representation based on the input data; and third, we are required to constrain ourselves to using only a certain *core language*. This could be the intermediate language of some virtual machine (e.g. Java Virtual Machine byte-code), or we might have other reasons to limit the capabilities of the core language.

This is a rather standard situation, with well known drawbacks and advantages. Among the drawbacks is the overhead imposed by interface method invocations. Among the advantages are modularity via encapsulation, flexibility to add new concrete implementations and ability to make dynamic runtime choices.

In the isomorphic specialization framework we can completely eliminate the overhead of method invocations while preserving the benefits mentioned above and thus fulfilling the *abstraction without regret* promise. Object-oriented code can be transformed into the core language with a limited set of types and primitive operations. In this paper, the core language is a higher-order functional language with pairs, sums, arrays and primitives shown in Figure 4. In Section 4 we show how to combine isomorphic specialization into the core language with subsequent optimized compilation of array operations of the core language using the LMS framework.

It is important to understand that staged evaluation and thus specialization both happen at runtime. Thus, transformation into the core language may depend on sparseness analysis so that we can dynamically select the best implementation for all the abstract data types (interfaces) used in `mvm`. Then we can specialize `mvm` with respect to the selected implementation and thus produce the optimal specialized version of `mvm`.

Now coming back to our example, we define two abstract data types: matrices and vectors. They are represented by interfaces `Vec[T]` and `Matr[T]` (see Figure 1)[2] respectively, which contain all the operations necessary to implement the algorithms we are interested in. These interfaces are, in fact, our abstractions which are built above the core language: it is the user's design choice how to call them and which properties and methods they have. In this case the interface for vectors allows us to obtain the vector's length and to calculate dot product with another vector. The matrix interface allows us to retrieve rows of the matrix as an array of vectors.

`Array[T]` is a *core type* (type from the core language). In our implementation it is a plain Scala (or Java) array of values of type T. [3]

Given these abstract types, we are able to implement `mvm` as shown in Figure 1.

This code operates with a mixture of abstract types and core types (like `Array`). In order to actually execute this code we need to implement the abstract types selecting some concrete representations for matrices and vectors. For this purpose, we assume that each interface is implemented by several concrete classes as shown in Figure 2.

```
class DenseVec[T](val arr: Array[T]) extends Vec[T] {
  def length = arr.length
  def dotProduct(vec: Vec[T]) = vec match {
    case dv: DenseVec[T] ⇒ sum(arr |*| dv.arr)
    case sv: SparseVec[T] ⇒
      sum(sv.values |*| (arr(sv.indices)))
  }
}
class SparseVec[T](
    val indices: Array[Int], val values: Array[T],
    val length: Int) extends Vec[T] {
  def dotProduct(vec: Vec[T]) = vec match {
    case dv: DenseVec[T] ⇒ dv.dotProduct(this)
    case sv: SparseVec[T] ⇒
      dotProductSV(indices, values, sv.indices, sv.values)
  }
}

class DenseMatr[T](val rows: Array[DenseVec[T]]) extends Matr[T]
class SparseMatr[T](val rows: Array[SparseVec[T]]) extends Matr[T]
```

Figure 2. Dense and sparse implementation of vector and matrix types

Up to this point everything looks like the traditional object-oriented approach for designing abstractions and various implementations. And this is our design choice. We are extending our functional core language with a limited set of object-oriented abstraction mechanisms (such as interfaces, classes and methods).

[2] Of course, in practice they contain more methods. The figure shows only the methods used for `mvm`.

[3] In our implementation we use a covariant wrapper around `Array` which is invariant in Scala and a `Rep` type constructor similar to LMS. These details are out of scope of this paper.

This is a programmer's interface and point of view into our framework. In our prototype, a programmer can just use some subset of Scala, where classes and functions naturally coexist. This serves as perfect input for all subsequent processing, which we are going to discuss below.

Once we have the concrete implementations of abstract types like `DenseMatr`, we can associate them with some core types. This association or mapping is defined by means of special objects, so called isomorphisms (or *isos* for short). For simplicity we just assume that each iso is a class that implements the interface shown in Figure 3.

```
trait Iso[From,To] {
  def to(x: From): To
  def from(y: To): From
}
```

Figure 3. Interface of isomorphisms

As we will see later, isos can be automatically generated based on fields of concrete classes (e.g. `DenseMatrix` has one field `rows`), but this is just our convention to simplify the presentation. More sophisticated mechanisms can be used as well, e.g. using annotations in source code.

These *iso-functions* define transformations between values of types. Usually isos are defined in such a way that it is possible for each concrete class C to build a composition of isos which relate C with some core type τ.

What is more important, isomorphisms can also compose during staged evaluation and this composition can also be made dependent on a dynamic choice, for example based on sparseness analysis. First-class `Iso` objects might be stored in a staging-time data structure or passed around in the program in other nontrivial ways. The program might even read a configuration file at specialization time and pick either `IsoA` or `IsoB` based on its contents. That's something that plain inlining and rewriting could never achieve.

The intuition for isomorphisms here is that every time you define a concrete implementation of some abstract data type you at the same time explicitly define an isomorphic representation of instances of that type in the core language. Every time you are growing abstraction over the core language by introducing an abstract data types, you are defining a way back in all concrete implementations.

Our method of code specialization works in this setup and is enabled by such definitions of isomorphisms. It allows to automatically specialize invocations of `mvm` code into a code which is specific for the particular matrix object which happens to be an argument of the invocation at the time of staged evaluation (i.e. at runtime). Moreover, all the calls to the methods like `rows` and `dotProduct` inside the body of `mvm` are also points of dynamic choice: they are evaluated as virtual method calls, but at staging time (where the values are expressions), the result is inlining of the body of the method which is selected dynamically based on the actual types of the objects (that is how evaluation semantics of method calls is defined).

Now, let's look at how the method of isomorphic specialization will work for at least two different concrete implementations of the abstract type `Vec[T]`. For demonstration we selected dense and sparse representations.

Each concrete implementation of `Vec[T]` should implement its own versions of all abstract methods of the interface. In our example, these implementations are shown in Figure 2.

Dense vector is represented by the concrete class `DenseVec[T]`. It is mapped to the core types via iso instance generated from the constructor arguments. `DenseVec` is represented in the core language simply as an array of values of type `T`.

Sparse vector is represented by the class `SparseVec[T]`. In the core language it is represented by the vector length and a pair of arrays: `indices` which contains indices of non-zero elements and `values` which contains the corresponding values.

Dense matrix is represented by concrete class `DenseMatr[T]`. In the core language it is represented as an array of dense vectors of type `T`.

Sparse matrix is represented by concrete class `SparseMatr[T]`. In the core language it is represented as an array of sparse vectors.

```
class Array[T] {
  def length: Int
  def apply(index: Int): T // get element at index
  def apply(indices: Array[Int]): Array[T]
  def map[R](f: T ⇒ R): Array[R]
  def filter(p: T ⇒ Boolean): Array[T]
  def zip[U](other: Array[U]): Array[(T,U)]
  def |*| (other: Array[T]): Array[T] // element-wise op
}
def range(start: Int, len: Int): Array[Int]
def sum[T](arr: Array[T]): T
def unzip[T,U](pairs: Array[(T,U)]): (Array[T],Array[U])
def dotProductSV[T](
  indices1: Array[Int], values1: Array[T],
  indices2: Array[Int], values2: Array[T]): T
```

Figure 4. Core language primitives

In these implementations we use the core language primitives shown in Figure 4.

Corresponding isomorphisms for all these concrete implementations are presented in Figure 5. Note that these isomorphisms can be automatically generated for each concrete class, which we do in our implementation (see section 3.3).

```
type DVData[T] = Array[T]
class DVIso[T] extends Iso[DVData[T], DenseVec[T]] {
  def to(x: DVData[T]) = new DenseVec(x)
  def from(dv: DenseVec[T]) = dv.arr
}
type SVData[T] = (Array[Int],(Array[T], Int))
class SVIso[T] extends Iso[SVData[T], SparseVec[T]] {
  def to(x: SVData[T]) =
      new SparseVec(x._1,x._2._1,x._2._2)
  def from(sv: SparseVec[T]) =
      (sv.indices, (sv.values,sv.length))
}
type DMData[T] = Array[DenseVec[T]]
class DMIso[T] extends Iso[DMData[T], DenseMatr[T]] {
  def to(x: DMData[T]) = new DenseMatr(x)
  def from(dm: DenseMatr[T]) = dm.rows
}
type SMData[T] = Array[SparseVec[T]]
class SMIso[T] extends Iso[SMData[T], SparseMatr[T]] {
  def to(x: SMData[T]) = new SparseMatr(x)
  def from(sm: SparseMatr[T]) = sm.rows
}
```

Figure 5. Isomorphisms for matrices and vectors

Now we can illustrate how isomorphic specialization works.

In order to call the mvm function with concrete implementation of matrices and vectors we need to: 1) wrap input core data in concrete objects; 2) call mvm with created objects and 3) extract resulting data.

This three-step process is important. It doesn't matter how many objects we will create from the core data and in how many functions we use them. As long as we extract all the data from objects we can be sure that isomorphic specialization will specialize all method invocations (like rows and dotProduct) with respect to the concrete implementations that are used.

In order to simplify our further description, let's limit ourselves to dense vectors and create *wrapper functions* that just do this three-step process. The code is shown in Figure 6.

```
def dmdvm(m: Array[Array[T]], v: Array[T]): Array[T] = {
  val dm = new DenseMatr(m.map(r ⇒ new DenseVec(r)))
  val dv = new DenseVec(v)
  val res = mvm(dm, dv)
  res.arr // extract data values from Vec
}
def smdvm(m: Array[(Array[Int],(Array[T],Int))],
        v: Array[T]): Array[T] = {
  val rs = m.map((is,(vs,l)) ⇒ new SparseVec(is,vs,l))
  val sm = new SparseMatr(rs, v.length)
  val dv = new DenseVec(v)
  val res = mvm(sm, dv)
  res.arr
}
```

Figure 6. Wrapper functions

```
def dmdvm_spec(m: Array[Array[T]],
               v: Array[T]): Array[T] =
  m map { row ⇒ sum(row |*| v) }

def smdvm_spec(
    m: Array[(Array[Int], (Array[T], Int))],
    v: Array[T]): Array[T] =
  m map { r ⇒
    val indices = r._1
    val values = r._2._1
    sum(values |*| v(indices))
}
```

Figure 7. Result of isomorphic specialization

Now, if we apply isomorphic specialization to these wrappers it will generate the functions shown in Figure 7, which contain only the core language primitives that are used in a way that is specific to a concrete implementation of the abstract data types. For this particular example `mvm`, invocations in wrappers will be inlined in the wrapper body. This will bring in other invocations like `rows` and `dotProduct` which will also be inlined. All the inlining happens in dynamic fashion, following the evaluation semantics of virtual method calls.

Note how fragments of code from concrete implementations are mixed in the resulting specialized versions. Staged evaluation implements a dynamic dispatch mechanism of method calls.

Isomorphisms as first-class objects are subject to staged evaluation. They can also compose to form new isomorphisms. These are the two factors and the key to the formulation of *lifting* of isos which is described in Section 3.5.

3. Formalization

In this section we describe two languages. The first one is the target for our specialization procedure (we call it *the core language*). The second one (called FJ) is an extension of the core language with a very limited set of object-oriented constructs necessary to illustrate our algorithms and approach. FJ is inspired by Featherweight Java [16] and we try to keep similarity with their formulations.

We describe our proposed technique, which we call Staged Evaluation, for transforming FJ terms into a DAG-based intermediate representation.

We also describe an algorithm which transforms FJ programs into equivalent (with respect to user-defined isomorphisms between FJ types and core types) core language programs.

We use overline $\overline{x}$ and indexed expression $(x_i)_{i=1}^{n}$ as a shorthand notation for the list $(x_1, \ldots, x_n)$. We also allow these lists to be empty when $n = 0$ and often omit $i = 1$ part and assume it by default. Index i in this case is always bound by this notation.

3.1 Core Language

Figure 8 summarizes the syntax of our core language. It is an explicitly typed lambda calculus enriched with pairs, sums, arrays and pattern matching **case** expressions.

$\mathcal{T} \ni \tau$	$::=$	$\texttt{Unit} \mid \texttt{Int}$	base types
	$\mid$	$(\tau_1 \times \tau_2)$	binary product type
	$\mid$	$(\tau_1 + \tau_2)$	sum type
	$\mid$	$(\tau_1 \to \tau_2)$	function type
	$\mid$	$\texttt{Array}[\tau]$	array type
$Term \ni e$	$::=$	$l \mid x : \tau$	integer literals and vars
	$\mid$	$\lambda(x : \tau).e \mid e_1 e_2$	functions
	$\mid$	$k\, \overline{e}$	constructors
	$\mid$	$\delta\, \overline{e}$	primitives
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ \{\ p_i \to e_i\ \}$	case analysis
k	$::=$	$() \mid (\_, \_)$	unit and pair
	$\mid$	$\texttt{l}[\tau_1, \tau_2] \cdot \_ \mid \texttt{r}[\tau_1, \tau_2] \cdot \_$	injections
δ	$::=$	$\texttt{fst}\ \_ \mid \texttt{snd}\ \_$	projections
	$\mid$	$\_ \oplus \_$	binary operations
p	$::=$	$l \mid k\, \overline{x}$	patterns
v	$::=$	$l \mid k\, \overline{v} \mid \lambda(x : \tau).e$	values

Figure 8. Syntax of the core language

We assign types to the terms in a standard way following typing judgments shown in Figure 9.

Although type annotations play an important role in staged evaluation we don't describe any sort of type checking to ensure that functions are applied to arguments of appropriate types. In other words we consider only well-typed terms.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash l : \texttt{Int}} \qquad \frac{}{\Gamma \vdash () : \texttt{Unit}}$$

$$\frac{\oplus : (\tau_1 \times \tau_2) \to \tau_3 \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \oplus e_2 : \tau_3}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{l}[\tau_1, \tau_2] \cdot e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{r}[\tau_1, \tau_2] \cdot e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\texttt{l}[\tau_1, \tau_2] \cdot x_1 \to e_1;\ \texttt{r}[\tau_1, \tau_2] \cdot x_2 \to e_2\} : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{Int} \quad \Gamma \vdash e_i : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{l_i \to e_i\} : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Figure 9. Typing judgments of the core language

Note that each well-typed term has exactly one type (or later, a single most-specific type). This will allow us to reify types as part of terms.

Call-by-value reduction contexts

$$\begin{array}{lcl} \mathcal{E} & ::= & \Box \mid k\ \overline{v}\ \mathcal{E}\ \overline{e} \mid \delta\ \overline{v}\ \mathcal{E}\ \overline{e} \mid \mathcal{E}\ e \mid (\lambda x.e)\mathcal{E} \\ & \mid & \textbf{case}\ \mathcal{E}\ \textbf{of}\ \{\ p_i \to e_i\ \} \end{array}$$

Call-by-value evaluation relation

$$[(\lambda x.e)\ v]\mathcal{E} \quad \mapsto \quad [[v/x]e]\mathcal{E} \qquad (1)$$

$$
\begin{array}{llll}
[\textsf{\textbf{case}}\; k\,\overline{v}\; \textsf{\textbf{of}}\; \{\; k_i\,\overline{x_i} \to e_i\;\}]\mathcal{E} & \mapsto & [[\overline{v}/\overline{x_j}]e_j]\mathcal{E}, \text{if } k = k_j & (2)\\
[\textsf{\textbf{case}}\; l\; \textsf{\textbf{of}}\; \{\; l_i \to e_i\;\}]\mathcal{E} & \mapsto & [e_j]\mathcal{E}, \text{if } l = l_j & (3)\\
[\delta\,\overline{v}]\mathcal{E} & \mapsto & [l]\mathcal{E}, \text{if } l = [\![\delta]\!]\overline{v} & (4)
\end{array}
$$

Figure 10. Evaluation semantics of the core language

The standard call-by-value evaluation semantics of the core language, which is shown in Figure 10, doesn't use reified types.

3.2 FJ Language

Figure 11 summarizes the extensions to the core language with additional object oriented constructs. We use Scala-like syntax for fields and methods. We denote types from the core language with τ and we use σ to denote interfaces and classes. We will refer to classes and interfaces as *object types* and to the types constructed from them as *FJ types*.

$$
\begin{array}{llll}
\tau & ::= & \cdots \mid \sigma & \text{extended types}\\
\sigma & ::= & I \mid C & \text{object types}\\
p & ::= & \overline{cd}\; e & \text{program}\\
cd & ::= & & \text{declaration}\\
 & \mid & \textsf{\textbf{trait}}\; I\; \{\overline{ms}\} & \text{interface}\\
 & \mid & \textsf{\textbf{class}}\; C\; \textsf{\textbf{extends}}\; I\{\overline{fd}\;\overline{md}\} & \text{class}\\
ms & ::= & \textsf{\textbf{def}}\; m(\overline{x : \sigma}) : \sigma & \text{method signature}\\
fd & ::= & \textsf{\textbf{val}}\; f : \sigma & \text{field}\\
md & ::= & \textsf{\textbf{def}}\; m(\overline{x : \sigma}) : \sigma = e & \text{method definition}\\
e & ::= & \cdots & \text{extended expressions}\\
 & \mid & e.f & \text{field selection}\\
 & \mid & e.m(\overline{e}) & \text{method invocation}
\end{array}
$$

	$\mid$	$\textsf{\textbf{new}}\ C(\overline{e})$	instance
v	$::=$	$\cdots$	extended values
	$\mid$	$\textsf{\textbf{new}}\ C(\overline{v})$	objects

Figure 11. FJ language syntax. C, f, m are class, field, and method names respectively

We assume that a correct program induces a number of utility functions that we will use in the typing rules. First, we assume the function $fields(\sigma)$ returns a sequence $\overline{\textsf{\textbf{val}}\ f : \sigma}$ pairing a field of a class or interface with its type, for all the fields declared in type σ. Second, we assume the partial function $ftype$, which is a map from an FJ type and a field name to a type. Thus $ftype(\sigma, f)$ returns the type of the field f in the class or interface σ. Third, we assume a partial function $mtype$ that is a map from an object type and a method name to a type signature. For example, we write $mtype(C, m) = \overline{\sigma} \to \phi$ when class C contains a method m with formal parameters of type $\overline{\sigma}$ and return type ϕ. Similarly, the body of the method m of the object type σ, written $mbody(\sigma, m)$, is a pair $(\overline{x}, e)$ of a sequence of parameters $\overline{x}$ and an expression e.

Furthermore, we assume that for each interface I there exists at least one class C which implements I. A field `f` of a class C can implement the (argument-less) method `f()` in the interface I.

There are no assignments, inheritance, super calls, object identity, exceptions, or access control in FJ. Each class has exactly one constructor which takes all the fields as arguments, in the order specified in the class declaration. There are no statements and method body is simply an expression. We use integers, arithmetic operations, case pattern matching instead of conditional expressions. All references to `this` are explicit. Overloaded method references are resolved statically by including argument types as part of the method name. FJ permits recursive class dependencies with the full generality of Java. A class can refer to types and call constructors of any other class. But note that recursion in methods is not supported.

The typing and semantics extensions for FJ with respect to the core language are given by extended evaluation contexts, two additional primitive reduction rules and three typing judgments shown in Figure 12. The FJ type system is sound and decidable. Please see [16] for further details.

3.3 Isomorphisms

For each class an isomorphic representation is defined based on its fields. Given a class C with the fields $\{\textsf{\textbf{val}}\ f_i\ :\ \sigma_i\}^n$, the

Subtyping

$$C <: C \quad \frac{C <: D \quad D <: E}{C <: E} \quad \frac{\textsf{class}\ C\ \textsf{extends}\ I\{\ldots\}}{C <: I}$$

Expression typing

$$\frac{\Gamma \vdash e_0 : C_0 \quad fields(C_0) = \overline{\textsf{val}\ f : C}}{\Gamma \vdash e_0.\mathtt{f}_i : C_i}$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{e} : \overline{E} \quad \overline{E} <: \overline{D}}{\Gamma \vdash e_0.\mathtt{m}(\overline{e}) : C}$$

$$\frac{fields(C) = \overline{\textsf{val}\ f : D} \quad \Gamma \vdash \overline{e} : \overline{E} \quad \overline{E} <: \overline{D}}{\Gamma \vdash \textsf{new}\ C(\overline{e}) : C}$$

Call-by-value evaluation contexts

$$\begin{array}{lll} \mathcal{E} & ::= & \cdots \\ & | & \mathcal{E}.f \mid \mathcal{E}.m(\overline{e}) \mid v.m(\overline{v}\ \mathcal{E}\ \overline{e}) \mid \textsf{new}\ C(\overline{v}\ \mathcal{E}\ \overline{e}) \end{array}$$

Call-by-value evaluation relation

$$\begin{array}{lll} [\textsf{new}\ C(\overline{v}).f_i]\mathcal{E} & \mapsto & [v_i]\mathcal{E}, \text{if } fields(C) = \overline{\textsf{val}\ f : C} \quad (5) \\ [\textsf{new}\ C(\overline{v}).m(\overline{u})]\mathcal{E} & \mapsto & [[\textsf{new}\ C(\overline{v})/\mathtt{this}; \overline{u}/\overline{x}]e_0]\mathcal{E}, \quad (6) \\ & & \text{if } mbody(m, C) = (\overline{x}, e_0) \end{array}$$

Figure 12. FJ typing and evaluation semantics

function $reptype(C)$ returns the type `Unit` if $n = 0$, σ_1 if $n = 1$, $(\sigma_1, (\ldots, (\sigma_{n-1}, \sigma_n)))$ otherwise. For example

$$reptype(\mathtt{SparseVec[T]}) = \mathtt{(Array[Int],(Array[T],Int))}.$$

For each FJ class C there exists a special class Iso_C which implements the interface `Iso`$[reptype(C), C]$ (see Figure 3). Note that `Iso` is not a generic (polymorphic) interface as FJ doesn't support generics. Rather, it is a template which produces an interface when instantiated with type parameters.

Thus, given any class C, the function $iso(C)$ returns an instance of Iso_C. This instance represents an *isomorphism* between a core type τ and the FJ class C.[4]

The isomorphisms returned by iso are called *primary*. They can be composed to form other *composite* isomorphisms. This is discussed in Section 3.5.

Now suppose that we have the following definitions

$$\textsf{\textbf{class}}\ C_1\ \textsf{\textbf{extends}}\ I\{\ldots\};\ \textsf{\textbf{class}}\ C_2\ \textsf{\textbf{extends}}\ I\{\ldots\}$$

We will refer to the interface I as an *abstract type* and implementing classes C_1 and C_2 as *concrete implementations* or *concrete classes* of this abstract type. By using this *abstract vs. concrete* terminology we will always implicitly assume this connection with classes, interfaces and isomorphism instances defined above.

For any abstract type I, every concrete implementation of I defines both an *alternative representation* of instance data and concrete implementation of the methods declared in I.

We assume that all abstract types are closed, that is, all their concrete implementations are known in advance. Our implementation doesn't have this limitation but we assume it here to simplify discussion.

If we want all concrete representations of an abstract type I to be inter-convertible, we need to impose an additional *convertibility*

[4] Strictly speaking, $Iso_C[\tau, C]$ defines an isomorphism not between τ and C (for example not every `(Array[Int],(Array[T],Int))` corresponds

to a SparseVector[T]) but between some subset $T \subseteq \tau$ and C. This in particular means that isos are postulated and cannot be inferred. We allow ourselves to be a bit sloppy and implicitly assume such subset T in further discussion.

requirement on concrete classes. Namely, all of their fields must implement some *property-method* in I. E.g. `Vec[T]` will have to include methods `arr` (from `DenseVec[T]`), `indices` and `values` (from `SparseVec[T]`). But this is not required for isomorphic specialization to work.

3.4 Staged Evaluation

We have already mentioned that *staged evaluation* can be formalized as zipper-based traversal of program terms where one-hole contexts are generically constructed from evaluation reduction contexts of the language. We refer an interested reader to related publications [15, 19].

This section describes a staged evaluation algorithm for FJ. We represent programs for staged evaluation as finite mappings $\{\overline{\alpha \to node}\}$ from identifiers (or *addresses*) to *nodes* where the set of nodes is defined by the grammar given in Figure 13[5]. Every address referenced by a node must itself be mapped to a node. We consider this mapping as a graph where the incoming edges of each node are given by the addresses it contains. This graph must be acyclic (i.e. recursive definitions aren't allowed). In this paper we use DAG (directed acyclic graph), graph and program graph as synonyms.

$$
\begin{array}{lcll}
Node & := & x & \text{variable} \\
 & | & d\,\overline{\alpha} & \text{definition node} \\
 & | & \lambda\alpha.\beta & \text{function} \\
 & | & \textbf{case}\ \gamma\ \textbf{of}\ \{p_i \rightarrow \beta_i\} & \text{pattern matching} \\
d & := & l & \text{constant} \\
 & | & k & \text{constructor} \\
 & | & \delta & \text{primitive} \\
 & | & \textbf{new}\ C & \text{class constructor} \\
 & | & \texttt{I.m} & \text{interface method} \\
 & | & \texttt{C.m} & \text{class method} \\
\delta & := & \ldots & \text{as in the core language} \\
 & | & app & \text{function application} \\
 & | & mcall & \text{method call} \\
 & | & mdef & \text{method definition} \\
p & := & l \mid k\,\overline{\alpha} & \text{case patterns}
\end{array}
$$

Figure 13. DAG nodes

We assume $\mathcal{K}$ is a set of constructors, $\mathcal{L}$ is a set of constants, $\mathcal{V}$ is a set of term variables, $\mathcal{P}$ is a set of primitives, $\mathcal{CLS}$ is a set of classes and C ranges over $\mathcal{CLS}$, $\mathcal{ABS}$ is a set of abstract types and I ranges over $\mathcal{ABS}$.

Δ ranges over the set Dag of all DAGs, Greek lower case letters α, β, γ and ν range over addresses in a given DAG, $node$ (or n for short) ranges over nodes. `I.m` ranges over interface methods, `C.m` ranges over class methods (as well as fields, which are treated as zero-argument methods), and d ranges over the set $\mathcal{D} = \mathcal{K} \cup \mathcal{L} \cup \mathcal{P} \cup \mathcal{CLS} \cup \{\texttt{I.m}\} \cup \{\texttt{C.m}\}$. The primitive $app(\alpha, \beta)$ denotes application of function referenced by α to β. $mcall(\gamma, \mu, \overline{\alpha})$ denotes invocations of the method referenced by the address μ on the instance γ with arguments $\overline{\alpha}$. $mdef(\texttt{C.m}, \phi)$ denotes the method definition for `C.m`, where ϕ is the lambda-abstraction for the body of the method.

Terms are defined as in Figure 8 except variables are replaced by addresses, so that nodes are basically terms of depth 0 and 1 (except for method calls and definitions, as described above).

We call a pair of a DAG Δ and an address α in its domain a *marked DAG* and denote it by $\Delta\langle\alpha\rangle$. α serves as a pointer into Δ and is called *the marked address*. $MDag$ is the set of all marked DAGs.

Pattern-matching function $(\!|\_|\!)$ can be used to extract nodes from DAGs and we write patterns $\Delta\langle(\!|pattern|\!)\rangle$ to bind the DAG with Δ and the node at marked address with $pattern$. To simplify

[5] For $d = l$, $\texttt{I.m}$ or $\texttt{C.m}$ $\overline{\alpha}$ in $d\,\overline{\alpha}$ is always empty.

handling of nested patterns, $\Delta\langle(\!|(\alpha, (\beta, \gamma))|\!)\rangle$ is syntactic sugar for $\Delta\langle(\!|(\alpha, (\!|(\beta, \gamma)|\!))|\!)\rangle$. We also use patterns as predicates.

We define *binding scope* as the set of nodes which depend on variables introduced by lambda-expressions and case-expressions. It will be convenient to represent it by a term called the *scope body* and calculated by the function $scope$ defined in Figure 14.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{scope : \overline{Addr} \times MDag \to Term} \\
\ scope(\overline{\alpha}, \Delta\langle\beta\rangle) & \mapsto & scope'(\overline{\alpha}, \beta, \Delta\langle\beta\rangle) \\
\\
\multicolumn{3}{l}{scope' : \overline{Addr} \times Addr \times MDag \to Term} \\
\multicolumn{3}{l}{\ scope'(\overline{\alpha}, \beta, \Delta\langle\gamma\rangle)} \\
\quad \text{if } \gamma \in free(\overline{\alpha}, \beta) & \mapsto & \gamma \\
\ scope'(\_, \beta, \Delta\langle\alpha@(\!|x|\!)\rangle) & \mapsto & \alpha \\
\ scope'(\overline{\alpha}, \beta, \Delta\langle(\!|d(\gamma_i)^n|\!)\rangle) & \mapsto & d(scope'(\overline{\alpha}, \beta, \Delta\langle\gamma_i\rangle))^n
\end{array}
$$

```
$scope'(\overline{\alpha}, \beta, \Delta\langle(\!|\lambda\alpha'.\gamma|\!)\rangle)$    $\mapsto$  $\lambda\alpha'.scope(\alpha', \Delta\langle\gamma\rangle)$
$scope'(\overline{\alpha}, \beta, \Delta\langle(\!|node|\!)\rangle)$
  if $node =$ case $\gamma$ of $\{$       $\mapsto$  case $scope'(\overline{\alpha}, \beta, \Delta\langle\gamma\rangle)$ of $\{$
    $k_i\overline{\alpha_i} \to \gamma_i$                     $k_i\overline{\alpha_i} \to scope(\overline{\alpha_i}, \Delta\langle\gamma_i\rangle)$
  $\}$                                  $\}$
```

Free addresses of a binding scope

$$free(\overline{\alpha}, \Delta\langle\beta\rangle) \mapsto \{\nu \mid \exists\gamma \in S \;\; \gamma \multimap \nu\} \setminus S$$

where
$$\begin{aligned} S &= \{\beta\} \cup (Up \cap Down) \\ Up &= \{\gamma \mid \beta \overset{*}{\multimap} \gamma\} \\ Down &= \{\gamma \mid \exists i \;\; \gamma \overset{*}{\multimap} \alpha_i\} \end{aligned}$$

Dependency relation ($\overset{*}{\multimap}$ denotes its transitive closure)

$$\frac{\Delta\alpha = d\;(\beta^n)}{\alpha \multimap \beta_i} \forall i \le n$$

$$\frac{\Delta\gamma = \lambda\alpha.\beta \;\; \nu \in free(\alpha, \Delta\langle\beta\rangle)}{\gamma \multimap \nu}$$

$$\frac{\Delta\gamma = \textsf{\textbf{case}}\; \gamma_0 \;\textsf{\textbf{of}}\; \{p_i\; \overline{\alpha_i} \to \beta_i\}^n}{\gamma \multimap \gamma_0}$$

$$\frac{\Delta\gamma = \textsf{\textbf{case}}\; \gamma_0 \;\textsf{\textbf{of}}\; \{p_i\; \overline{\alpha_i} \to \beta_i\}^n \;\; \nu \in free(\overline{\alpha_j}, \Delta\langle\beta_j\rangle)}{\gamma \multimap \nu} \forall j \le n$$

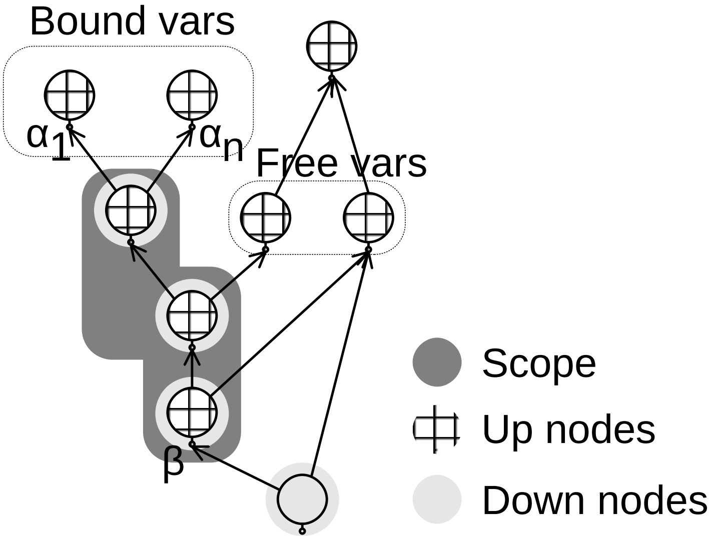

Figure 14. Scope body and auxiliary definitions

We consider two lambda nodes to be α-equivalent when there exists an address substitution which makes their scope bodies equal. This is formalized by the following definition:

Definition 1 (α-equivalence)**.** $\forall \Delta, \gamma_1, \gamma_2$, *if* $\Delta\gamma_1 = \lambda\overline{\alpha_1}.\beta_1, \Delta\gamma_2 = \lambda\overline{\alpha_2}.\beta_2$ *and* $scope(\overline{\alpha_1}, \Delta\langle\beta_1\rangle) = [\overline{\alpha_1}/\overline{\alpha_2}]scope(\overline{\alpha_2}, \Delta\langle\beta_2\rangle)$ *then* $\lambda\overline{\alpha_1}.\beta_1 \overset{\alpha}{\equiv} \lambda\overline{\alpha_2}.\beta_2$.

Our usage of DAGs instead of trees for defining programs was originally motivated by the fact that sharing and many other optimizations are better achieved using graphs.

But specifically for this paper the key point is that DAGs (empowered by active patterns) allow us to express rewriting rules implementing isomorphic specialization locally instead of having to look arbitrarily deeply into the tree.

This works in concert with the staged evaluation algorithm where the DAGs are constructed from inputs to output in a breadth-first way while the front is kept in evaluation stack. Thus, the structure of the resulting DAG is unknown until the graph is fully constructed. This process is hard to describe using just terms with let-bindings.

The staged evaluation algorithm is defined by two functions: injection and staged evaluation, which are parametrized by an additional function RW. By default RW is identity, otherwise it specifies some rewriting rules that are applied while building the graph (and may be mutually recursive with injection and staged evaluation). We will see an example of non-identity RW in Section 3.5.

The *injection* function $\Delta \leftarrow n$ defined in Figure 15 adds a node n to a DAG Δ. It returns $\Delta'\langle\alpha\rangle$ where α is the address of n. This is a *collapsing injection* [26] which means that every different term is represented by a unique sub-DAG. In particular, when Δ already contains a node equivalent to n, $\Delta' = \Delta$ and $\alpha \in \mathcal{D}om\ \Delta$.

$$\begin{array}{ll}
\_ \leftarrow \_ : Dag \times Node \rightarrow MDag & \\
\Delta \leftarrow \gamma & \mapsto\ \Delta\langle\gamma\rangle \\
\Delta \leftarrow x & \mapsto\ (\Delta \cup \{\gamma \mapsto x\})\langle\gamma\rangle \text{ where } \gamma \text{ is fresh in } \Delta
\end{array}$$

$$
\begin{array}{lll}
\Delta \leftarrow \texttt{C.m} & \mapsto & RW(\Delta\langle\mu\rangle) \text{ if } \exists\mu : \Delta\mu = mdef(\texttt{C.m}, \_) \\
& \mapsto & AddNode(\Delta', mdef(\texttt{C.m}, \phi)) \text{ where} \\
& & \quad \begin{array}{ll}
(\overline{x}, e) & = mbody(C, m) \\
\Delta_0\langle\alpha_0\rangle & = \Delta \leftarrow \texttt{this} \\
(\Delta_i\langle\alpha_i\rangle & = \Delta_{i-1} \leftarrow x_i)_{i=1}^{n} \\
\beta & = SE[\![[\alpha_0/\texttt{this}; \overline{\alpha}/\overline{x}]e]\!]\ \Delta_n \\
\Delta'\langle\phi\rangle & = \Delta_n \leftarrow \lambda\{\alpha\}_{i=0}^{n}.\beta
\end{array} \\
\Delta \leftarrow \lambda\overline{\alpha}.\beta & \mapsto & RW(\Delta\langle\gamma\rangle) \text{ if } \exists\gamma : \Delta\gamma \overset{\alpha}{\equiv} \lambda\overline{\alpha}.\beta \\
& \mapsto & AddNode(\Delta, \lambda\overline{\alpha}.\beta) \text{ otherwise} \\
\Delta \leftarrow node & \mapsto & RW(\Delta\langle\alpha\rangle) \text{ if } \exists\alpha : \Delta\alpha = node \\
& \mapsto & AddNode(\Delta, node) \text{ otherwise} \\
\\
AddNode(\Delta, n) & \mapsto & RW((\Delta \cup \{\gamma \mapsto n\})\langle\gamma\rangle) \\
& & \text{where } \gamma \text{ is fresh in } \Delta
\end{array}
$$

Figure 15. Injection function

The *staged evaluation* function $SE[\![e]\!]\ \Delta$ defined in Figure 16 takes a term e and a DAG Δ and returns a marked DAG $\Delta'\langle\alpha\rangle$ where α corresponds to the value of the term e.

The key intuition for staged evaluation is that it behaves similarly to the evaluation relation defined in Figures 10 and 12, but with values in the evaluation contexts replaced by addresses, and with a stack of contexts instead of just one. Any address α produced by injection is considered a partially evaluated value of the term, and SE' is recursively applied to that value until the context stack becomes empty and the evaluation finishes (this is the last case in Figure 16).

In other words, staged evaluation uses and calculates new addresses in the same way as standard evaluation uses and calculates new values.

Note also that when we use staged evaluation for a method definition or for a lambda we create fresh addresses for the arguments and substitute them in the body. This reflects the call-by-value semantics of FJ.

The last but not the least piece of the puzzle is to explain what SE algorithm is doing with respect to the FJ language and its evaluation semantics. Let's write the algebraic type of FJ expressions (defined in Figures 8 and 11) as a regular recursive data type using the notation of [19].

$SE[\![\_]\!] : Term \to Dag \to MDag$
$SE[\![e]\!]\ \Delta$

$SE'[\![\_]\!] : Term \to Stack\ \partial E \to Dag$
$SE'[\![l]\!]\ \mathcal{R}\ \Delta$
$SE'[\![d\ e_0 \overline{e}]\!]\ \mathcal{R}\ \Delta$
$SE'[\![\textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}]\!]\ \mathcal{R}\ \Delta$
$SE'[\![e_1\ e_2]\!]\ \mathcal{R}\ \Delta$
$SE'[\![\lambda x.e]\!]\ \mathcal{R}\ \Delta$

$SE'[\![e.\mathtt{m}(\overline{e})]\!]\ \mathcal{R}\ \Delta$
$SE'[\![\alpha]\!]\ (d\ \overline{\beta}\Box e_0 \overline{e_1} :: \mathcal{R})\ \Delta \quad \mapsto$
$SE'[\![\alpha]\!]\ (d\ \overline{\beta}\Box :: \mathcal{R})\ \Delta \quad \mapsto$
$SE'[\![\alpha]\!]\ (\textbf{case}\ \Box\ \textbf{of}\ \{p_i\ \overline{x_i} \to e_i\} :: \mathcal{R})\ \Delta_0 \quad \mapsto$

$$SE'[\![\alpha]\!]\ (\Box\ e_2 :: \mathcal{R})\ \Delta \qquad \mapsto$$
$$SE'[\![\alpha]\!]\ (\beta\ \Box :: \mathcal{R})\ \Delta \qquad \mapsto$$
$$SE'[\![\alpha : \sigma]\!]\ (\Box.\mathtt{m}() :: \mathcal{R})\ \Delta \qquad \mapsto$$

$$SE'[\![\alpha]\!]\ (\Box.\mathtt{m}(e_0\overline{e}) :: \mathcal{R})\ \Delta \qquad \mapsto$$
$$SE'[\![\alpha]\!]\ (\beta.\mathtt{m}(\overline{\gamma}\Box e_0\overline{e}) :: \mathcal{R})\ \Delta \qquad \mapsto$$
$$SE'[\![\alpha]\!]\ ((\beta : \sigma).\mathtt{m}(\overline{\gamma}\Box) :: \mathcal{R})\ \Delta \qquad \mapsto$$

$$SE'[\![\alpha]\!]\ \epsilon\ \Delta \qquad \mapsto$$

$$\mapsto \quad SE'[\![e]\!]\ \epsilon\ \Delta$$

$$\rightarrow MDag$$
$$\mapsto \quad SE'[\![\alpha]\!]\ \mathcal{R}\ \Delta' \text{ where } \Delta'\langle\alpha\rangle = \Delta \leftarrow l$$
$$\mapsto \quad SE'[\![e_0]\!]\ (d\ \Box\overline{e} :: \mathcal{R})\ \Delta$$
$$\mapsto \quad SE'[\![e]\!]\ (\textbf{case}\ \Box\ \textbf{of}\ \{p_i \rightarrow e_i\} :: \mathcal{R})\ \Delta$$
$$\mapsto \quad SE'[\![e_1]\!]\ (\Box\ e_2 :: \mathcal{R})\ \Delta$$

$$\begin{array}{rl} \mapsto & SE'[\![\gamma]\!]\ \mathcal{R}\ \Delta_3 \text{ where} \\ & \quad \begin{array}{ll} \Delta_1\langle\alpha\rangle & = \Delta \leftarrow x \\ \Delta_2\langle\beta\rangle & = SE'[\![[\alpha/x]e]\!]\ \epsilon\ \Delta_1 \\ \Delta_3\langle\gamma\rangle & = \Delta_2 \leftarrow \lambda\alpha.\beta \end{array} \\ \mapsto & SE'[\![e]\!]\ (\Box.\mathtt{m}(\overline{e}) :: \mathcal{R})\ \Delta \end{array}$$

$$SE'[\![e_0]\!]\ (d\,\overline{\beta}\alpha\Box\overline{e_1} :: \mathcal{R})\ \Delta$$

$$SE'[\![\alpha']\!]\ \mathcal{R}\ \Delta' \text{ where } \Delta'\langle\alpha'\rangle = \Delta \leftarrow d\,\overline{\beta}\alpha$$

$$\begin{array}{l} SE'[\![\alpha']\!]\ \mathcal{R}\ \Delta'' \text{ where} \\ \quad \begin{array}{ll} (\Delta_i\langle\overline{\gamma_i}\rangle & = \Delta_{i-1} \leftarrow \overline{x_i})^n; \Delta'_0 = \Delta_n \\ (\Delta'_i\langle\beta_i\rangle & = SE'[\![[\overline{\gamma_i}/\overline{x_i}]e_i]\!]\ \epsilon\ \Delta'_{i-1})^n \\ \Delta''\langle\alpha'\rangle & = \Delta'_n \leftarrow \textbf{case}\ \alpha\ \textbf{of}\ \{p_i\ \overline{\gamma_i} \rightarrow \beta_i\} \end{array} \end{array}$$

$$SE'[\![e_2]\!]\ (\alpha\ \Box :: \mathcal{R})\ \Delta$$

$$SE'[\![\gamma]\!]\ \mathcal{R}\ \Delta' \text{ where } \Delta'\langle\gamma\rangle = \Delta \leftarrow app(\beta, \alpha)$$

$$\begin{array}{l} SE'[\![\nu]\!]\ \mathcal{R}\ \Delta_2 \text{ where} \\ \quad \begin{array}{ll} \Delta_1\langle\mu\rangle & = \Delta \leftarrow \sigma.m \\ \Delta_2\langle\nu\rangle & = \Delta_1 \leftarrow mcall(\alpha, \mu, []) \end{array} \end{array}$$

$$SE'[\![e_0]\!]\ (\alpha.\mathtt{m}(\Box\overline{e}) :: \mathcal{R})\ \Delta$$

$$SE'[\![e_0]\!]\ (\beta.\mathtt{m}(\overline{\gamma}\alpha\Box\overline{e}) :: \mathcal{R})\ \Delta$$

$$\begin{array}{l} SE'[\![\nu]\!]\ \mathcal{R}\ \Delta_2 \text{ where} \\ \quad \begin{array}{ll} \Delta_1\langle\mu\rangle & = \Delta \leftarrow \sigma.m \\ \Delta_2\langle\nu\rangle & = \Delta_1 \leftarrow mcall(\beta, \mu, \overline{\gamma} +\!\!+ [\alpha]) \end{array} \end{array}$$

$\Delta\langle\alpha\rangle$

Figure 16. Staged Evaluation (call-by-value). $\mathcal{R}$ stands for a stack of evaluation contexts, ϵ for the empty stack and $(\mathcal{E} :: \mathcal{R})$ for a stack with $\mathcal{E}$ on top and $\mathcal{R}$ underneath. d stands for either k or δ in $Term$. Fields are treated as zero-argument methods.

$$\begin{array}{rcl} E & = & \mu e.(\mathcal{L} \times e^0 + \mathcal{V} \times e^0 \\ & + & \mathcal{K} \times e^n + \mathcal{P} \times e^n \\ & + & \lambda(x:\tau).e_1 \times e^0 + e^2 \\ & + & \square.\mathtt{f} \times e \\ & + & (\mathtt{I.m} + \mathtt{C.m}) \times e \times e^n \\ & + & \textbf{new}\ C \times e^n \\ & + & \textbf{case}\ \square\ \textbf{of}\ \{p_i \to e_i\} \times e) \end{array}$$

Now by formally differentiating right side as a function of variable e we get

$$\begin{array}{rcl} \partial E & = & \mu e.(\\ & & Fin\, n \times \mathcal{K} \times e^{n-1} + Fin\, n \times \mathcal{P} \times e^{n-1} \\ & + & Fin\, 2 \times e \\ & + & \square.\mathtt{f} \\ & + & Fin\, (n+1) \times (\mathtt{I.m} + \mathtt{C.m}) \times e^n \\ & + & Fin\, n \times \textbf{new}\ C \times e^{n-1} \\ & + & \textbf{case}\ \square\ \textbf{of}\ \{p_i \to e_i\}) \end{array}$$

Here $Fin\ n$ is a type which contains exactly n different values, every value of this type can represent an index $i \in \{1 \ldots n\}$. Thus a value of $Fin\ n \times e^{n-1}$ can be represented as $e_1 \ldots e_{i-1} \square e_{i+1} \ldots e_n$ i.e. a list of length n with a hole, and using our overline notation as $\overline{e}\square\overline{e}$.

Now if we replace the holes with letter $\mathcal{E}$ and additionally require expressions before $\mathcal{E}$ to be values (i.e. addresses) we get exactly the data type of the evaluation contexts of FJ, which according to Danvy [9] is isomorphic to the data type of defunctionalized continuations of an evaluation function of the language.

At the same time ∂E is a type of one-hole contexts for terms of the language, which we use to represent a state of SE' algorithm (see context patterns in Figure 16). Note that SE' moves the hole forward in the list by replacing terms with evaluated values (addresses).

This observation makes it clear that the stack of zipper contexts corresponds to the defunctionalized continuation and represents the work which remains to be done by SE' while evaluating the term.[6]

Finally, it turns out that applying staged evaluation to a scope body returns the same address it was extracted from. Formally:

Proposition 1. *Let RW be identity, then* $\forall \Delta, \overline{\alpha} \in \mathcal{V}, \beta \in \mathcal{D}om\ \Delta\ \ SE[\![scope(\overline{\alpha}, \Delta\langle\beta\rangle)]\!]\ \Delta = \Delta\langle\beta\rangle$

Proof (sketch). By induction on the structure of $scope(\overline{\alpha}, \Delta\langle\beta\rangle)$. □

3.5 Isomorphic Specialization

The generic nature of staged evaluation leads to a generic formulation of the isomorphic specialization transformation. The idea is to use the ability of SE to handle terms with DAG addresses as values, i.e. to evaluate applications $(\mu\ \gamma)\ \alpha$ where μ, γ, α are addresses. This feature allows us to integrate local term rewriting rules in a process of global DAG construction.

Rewriting rules transform a marked DAG $\Delta\langle\gamma\rangle$ into a new marked DAG. This means that each rewriting can change either the DAG or the address or both. Many new nodes can be added to the DAG as part of rewriting, but already existing nodes can't change. DAG is an immutable data structure.

Function RW applies rewriting rules iteratively until reaching a fixed point where no more rewrites are possible. This is important because one rewriting often opens possibilities for another. Note that each new node is subject for rewriting *after* it is added to the DAG by the injection function. As a consequence, it can be forgotten (in which case it will not be a part of the final binding scope). The intuition is that the DAG Δ represents the *universe* (or

[6] Notably, this construction of zipper-style traversal doesn't work for call-by-name evaluation contexts because ∂E contains $(\lambda x.e)\mathcal{E}$ and $k\ \overline{v}\ \mathcal{E}\ \overline{e}$ which aren't CBN contexts.

```
class Iso$_\times$[$A_1, A_2, B_1, B_2$](
    val $iso_1$: Iso[$A_1, B_1$], val $iso_2$: Iso[$A_2, B_2$])
  extends Iso[$A_1 \times A_2, B_1 \times B_2$] {
  def to(a:$A_1 \times A_2$) = ($iso_1$.to(fst(a)), $iso_2$.to(snd(a)))
  def from(b:$B_1 \times B_2$)= ($iso_1$.from(fst(b)), $iso_2$.from(snd(b)))
}
class Iso$_+$[$A_1, A_2, B_1, B_2$](
    val $iso_1$: Iso[$A_1, B_1$], val $iso_2$: Iso[$A_2, B_2$])
  extends Iso[$A_1 + A_2, B_1 + B_2$] {
  def to(a:$A_1 + A_2$) = case a of {
    l·$a_1$ → l·$iso_1$.to($a_1$); r·$a_2$ → r·$iso_2$.to($a_2$)
  }
  def from(b:$B_1 + B_2$) = case b of {
    l·$b_1$ → l·$iso_1$.from($b_1$); r·$b_2$ → r·$iso_2$.from($b_2$)
  }
}
class Iso$_{arr}$[$A, B$](val $iso$: Iso[$A, B$])
  extends Iso[Array[$A$],Array[$B$]] {
  def to(as: Array[$A$]) = as.map(iso.to)
  def from(bs: Array[$B$]) = bs.map(iso.from)
}
class Iso$_\rightarrow$[$A_1, A_2, B_1, B_2$](
    val $iso_1$: Iso[$A_1, B_1$], val $iso_2$: Iso[$A_2, B_2$])
  extends Iso[$A_1 \rightarrow A_2, B_1 \rightarrow B_2$] {
  def to(f:$A_1 \rightarrow A_2$) = b ⇒ $iso_2$.to(f($iso_1$.from(b)))
  def from(g:$B_1 \rightarrow B_2$)= a ⇒ $iso_2$.from(g($iso_1$.to(a)))
}
```

Figure 17. Compositions of isomorphisms

sea) of nodes, the marked address points to some particular node and everything else happens relative to the marked address.

RW works by pattern-matching the node corresponding to the marked address, and each case can be regarded as one rewrite rule. Each rule first extracts some subgraph using active patterns which we described in Section 3.4. The result is a term with addresses of the nodes as the values of the variables bound by the pattern. Thus we have a term on the left-hand-side with variables that can be used on the right-hand-side. So we can define some term on the right using those variables bound on the left and inject it back to the DAG using staged evaluation by handing the term to SE, which makes $\_ \leftarrow \_$, SE and RW mutually recursive.

Thus, we can define graph transformation by specifying a set of term rewriting rules. Not every transformation can be defined in this way, but we claim that isomorphic specialization can be defined as just a set of specially selected rewrite rules in this framework. These rules are shown in Figure 18.

Let's look at these rules. The idea is to use a special constructor which we call *view* and denote as $e \lhd iso$ with the following typing rule:

$$\frac{\Gamma \vdash e : \tau, iso : \texttt{Iso}[\tau, \sigma]}{\Gamma \vdash e \lhd iso : \sigma}$$

The intuition is that each node of this view type represents an isomorphic connection between a value of the core type τ and a value of the FJ type σ. So, if we define rewriting rules that systematically move these views along the edges of the DAG, from bound variables towards the root of the binding scope, then the DAG, which remains after rewriting is complete, contains only the core language nodes and thus it represents a program in the core language. This resulting program is equivalent to the original program because every rewriting step preserves semantics.

To define these rules we need to be able to compose primary isomorphisms and to create new isomorphisms associated with view nodes. We define one composite isomorphism for each type constructor of the core language, which allows us to *lift* views over types. Composite isomorphisms are shown in Figure 17 and the method of isomorphic specialization is implemented by the rewriting rules shown in Figure 18.

Together these rules implement isomorphic specialization in such a way that the following property holds: if a closed FJ expres-

sion has a core type then staged evaluation with specializing rewriting rules will produce a core language expression which doesn't contain any FJ constructs or vestiges of the `to/from` functions of isomorphisms. This is formalized in the next conjecture:

Conjecture 1. *Let RW be $RW_{spec}, e \in Term_{FJ}$ such that e is closed and $e : \tau$ for some $\tau \in \mathcal{T}_{Core}$. Then $\exists e' : \tau, \theta : Addr \to Addr$ such that $e' \in Term_{Core}$ and $SE[\![e]\!]\,\{\} = \theta(SE[\![e']\!]\,\{\})$ (where $\{\}$ is the empty DAG).*

This holds even in the presence of multiple concrete implementations of any abstract type. This is achieved by 1) applying systematic rewriting during staged evaluation; 2) providing domain specific rules that lift view nodes towards the output of the DAG; and 3) applying staged evaluation to the first-class `Iso` instances.

The idea is that `to/from` pairs are not just compiled away by applying the identity rule. Instead, staged evaluation is applied to `Iso` instances themselves (as can be seen from the rules). Each `from` implementation accesses the properties of the concrete class. And this is where virtual method invocation semantics of staged evaluation takes place. It leads to inlining of the concrete implementation code selected at evaluation time (i.e. at runtime).

We don't claim that the conjecture will hold for any pair of languages. Domain specificity is important here, as the rules are domain-specific. Rather, it is an important property of the core language to be *friendly to isomorphisms* and to serve as a target of isomorphic specialization. In our experiments we observed a multitude of such friendly languages, which supports our conjecture.

4. Evaluation

In a framework with a first-class isomorphic specialization we can automatically specialize a program written in terms of abstract data types with respect to any concrete representations of those abstract data types translating to a given core language. In our approach this looks like programming with classes and interfaces of object oriented-programming, with the difference that we are able to automatically eliminate all abstractions and accompanying overhead.

In Section 2 we illustrated our approach using `mvm` example. We showed in Figure 7 two resulting specializations generated for dense and sparse representations of matrices. In both those cases the vector has dense representation. If we consider also sparse representation of vectors then we can generate another two specializations of original `mvm` example two of which we show in Figure 19.

In this section we describe results of our experiments and explain why we think isomorphic specialization is important.

In practice, the choice of a particular representation of data depends on what kind of input data we have. For example, if input data is sparse then it would be reasonable to use sparse representation and if the data is dense then dense format would be more efficient. This is a common trade-off and typically, in order to make a justified choice, all representations should be tried out. This is exactly the case where our isomorphic specialization would be very useful. Because it is first-class you just need to implement all concrete representations of abstract types you are interested in and the framework will generate necessary specialized versions of your program for you. And importantly, you can think about each concrete implementation independently from the other implementations. You don't need to worry about how they will be mixed into generated specialized versions. This also means that if you invent a new representation you just need to implement it and add to the system. The system will generate new specialized variants automatically.

To show how important it is to eliminate abstraction overhead we measure performance of all specialized versions of `mvm` and compare them with the original version without specialization and

$$
\begin{array}{ll}
RW_{spec} : MDag \to MDag & \\
RW_{spec}\, \Delta\langle(\!|\textbf{case}\ k_j \overline{\gamma}\ \textbf{of}\ \{k_i \overline{\alpha_i} \to \beta_i\}|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\delta\ \overline{l}|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\texttt{fst}\ (\gamma_1, \gamma_2)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\texttt{snd}\ (\gamma_1, \gamma_2)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|mcall(\gamma @ (\!|\textbf{new}\ C\ \overline{\beta}|\!), \texttt{C.m}, \overline{\alpha})|\!)\rangle & \mapsto \\
 & \\
RW_{spec}\, \Delta\langle(\!|mcall(\gamma @ (\!|\textbf{new}\ C\ \overline{\beta}|\!), \texttt{I.m}, \overline{\alpha})|\!)\rangle & \mapsto \\
 & \\
RW_{spec}\, \Delta\langle(\!|mcall(\gamma, mdef(\_, \phi), \overline{\alpha})|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|app(\lambda\alpha.\beta, \gamma)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|(a_1 \lhd iso_1, a_2 \lhd iso_2)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|(a \lhd iso, x)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|(x, a \lhd iso)|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\texttt{fst}\ ((a_1, \_) \lhd \textbf{new}\ Iso_{\times}(iso_1, \_))|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\texttt{snd}\ ((\_, a_2) \lhd \textbf{new}\ Iso_{\times}(\_, iso_2))|\!)\rangle & \mapsto \\
RW_{spec}\, \Delta\langle(\!|\textbf{case}\ \texttt{l} \cdot a \lhd \textbf{new}\ Iso_{+}(iso_1, \_)\ \textbf{of}\ \{\texttt{l} \cdot b \to \beta\}|\!)\rangle & \mapsto
\end{array}
$$

$RW_{spec}\ \Delta\langle(\!|\textsf{\textbf{case}}\ \texttt{r}\cdot a\lhd\textsf{\textbf{new}}\ Iso_{+}(\_,iso_2)\ \textsf{\textbf{of}}\ \{\texttt{r}\cdot b\rightarrow\beta\}|\!)\rangle \quad \mapsto$

$RW_{spec}\ \Delta\langle(\!|\texttt{l}\cdot(a\lhd iso)|\!)\rangle \quad \mapsto$
$RW_{spec}\ \Delta\langle(\!|\texttt{r}\cdot(a\lhd iso)|\!)\rangle \quad \mapsto$
$RW_{spec}\ \Delta\langle(\!|as.map(f).map(g)|\!)\rangle \quad \mapsto$
$RW_{spec}\ \Delta\langle(\!|(as\lhd iso).map(f)|\!)\rangle \quad \mapsto$
$RW_{spec}\ \Delta\langle(\!|as.map(f) : Array[\sigma]|\!)\rangle \quad \mapsto$

$RW_{spec}\ \Delta\langle\gamma\rangle \quad \mapsto$

$SE[\![[\overline{\gamma}/\overline{\alpha_j}]scope(\overline{\alpha_j},\Delta\langle\beta_i\rangle)]\!]\ \Delta$
$\Delta\leftarrow l$ where $l=[\![\delta]\!]\ \overline{l}$
$SE[\![\Delta\langle\gamma_1\rangle]\!]\ \Delta$
$SE[\![\Delta\langle\gamma_2\rangle]\!]\ \Delta$
$SE[\![(\phi\ \gamma)\ \overline{\alpha}]\!]\ \Delta'$ where
    $\Delta'\langle(\!|mdef(\texttt{C.m},\phi)|\!)\rangle \quad = \Delta\leftarrow\texttt{C.m}$
$SE[\![(\phi\ \gamma)\ \overline{\alpha}]\!]\ \Delta'$ where
    $\Delta'\langle(\!|mdef(\texttt{C.m},\phi)|\!)\rangle \quad = \Delta\leftarrow\texttt{C.m}$
$SE[\![(\phi\ \gamma)\ \overline{\alpha}]\!]\ \Delta$
$SE[\![[\gamma/\alpha]scope(\alpha,\Delta\langle\beta\rangle)]\!]\ \Delta$
$SE[\![(a_1,a_2)\lhd\textsf{\textbf{new}}\ Iso_{\times}(iso_1,iso_2)]\!]\ \Delta$
$SE[\![(a,x)\lhd\textsf{\textbf{new}}\ Iso_{\times}(iso,id)]\!]\ \Delta$
$SE[\![(x,a)\lhd\textsf{\textbf{new}}\ Iso_{\times}(id,iso)]\!]\ \Delta$
$SE[\![a_1\lhd iso_1]\!]\ \Delta$
$SE[\![a_2\lhd iso_2]\!]\ \Delta$
let $\Delta'\langle\gamma\rangle = SE[\![iso_1.to(a)]\!]\ \Delta$
in $SE[\![[\gamma/b]scope(b,\Delta\langle\beta\rangle)]\!]\ \Delta'$
let $\Delta'\langle\gamma\rangle = SE[\![iso_1.to(a)]\!]\ \Delta$

in $SE[\![[\gamma/b]scope(b, \Delta\langle\beta\rangle)]\!]\ \Delta'$
$SE[\![(\mathtt{l} \cdot a) \lhd \textbf{new}\ Iso_{+}(iso, id)]\!]\ \Delta$
$SE[\![(\mathtt{r} \cdot a) \lhd \textbf{new}\ Iso_{+}(id, iso)]\!]\ \Delta$
$SE[\![as.map(a \Rightarrow g(f(a)))]\!]\ \Delta$
$SE[\![iso.to(as).map(f)]\!]\ \Delta$
let $iso^{\sigma}_{\tau} = iso[\sigma]$
in $SE[\![as.map(a \Rightarrow iso^{\sigma}_{\tau}.from(f(a))) \lhd \textbf{new}\ Iso_{arr}(iso^{\sigma}_{\tau})]\!]\ \Delta$
$\Delta\langle\gamma\rangle$

Figure 18. Rewriting rules for specialization

```
def dmsvm_spec(m: Array[Array[T]],
               v: (Array[Int], (Array[T],Int))): Array[T] = {
  val indices = v._1
  val values = v._2._1
  m.map { row ⇒ sum(row(indices) |*| values) }
}
def smsvm_spec(m: Array[(Array[Int],(Array[T],Int))],
               v: (Array[Int], (Array[T],Int))): Array[T] = {
  val indices = v._1
  val values = v._2._1
  m.map { (is,(vs,_)) ⇒
    dotProductSV(is, vs, indices, values)
  }
}
```

Figure 19. Result of isomorphic specialization

optimization of array operations. First, we applied isomorphic specialization to wrapper functions. The DAGs of specialized versions (shown in Figures 7 and 19) were injected into LMS to produce optimized Scala code. Then for each experiment we did the following steps:

1. Randomly generate input matrix and vector data of desired size and *sparseness* (percentage of zero values).

2. Convert the input vector and matrix into sparse and dense representations, as described in Section 2.
3. Run all versions of generated Scala code.

We used the Scalameter benchmarking library to measure execution time. It performs preliminary warm-up and then executes given code repeatedly in order to calculate the average execution time. The final times in milliseconds for all the experiments are given in Table 1 and Table2.

All matrices have the same size: $10^4 \times 10^4$. Accordingly, the length of input vectors is also 10^4. S_m is matrix sparseness and S_v is vector sparseness. The remaining columns show evaluation time for the original version and each specialized version.

These results more or less reflect our intuition about performance of different representations. But not all was obvious in advance: we see that when sparseness of both vectors and matrices

S_m	S_v	**dmdv**	**dmsv**	**smdv**	**smsv**
0%	0%	11389	14740	14827	53348
10%	10%	11408	13326	13253	44376
50%	50%	12944	7443	7428	21788
90%	90%	13093	1682	1546	3548
99%	99%	13251	227	167	280
0%	50%	11483	7466	14758	27987
50%	0%	12946	14852	7462	42471
10%	90%	13907	1659	14029	9728
90%	10%	14304	14581	1608	27586

Table 1. Execution times of original versions of `mvm`

S_m	S_v	**dmdv**	**dmsv**	**smdv**	**smsv**
0%	0%	309	354	366	760
10%	10%	311	323	332	1002
50%	50%	310	202	187	924
90%	90%	307	104	42	172
99%	99%	307	18	8	18
0%	50%	308	198	373	1134
50%	0%	310	359	187	986
10%	90%	311	118	335	497
90%	10%	311	323	42	345

Table 2. Execution times of specialized and optimized versions of `mvm`

is close to 50%, `smdvm` or `dmsvm` perform much better than `dmdvm` and `smsvm` (see line 3), and that `smsvm` is generally quite slow. Thus *specialization for free* really does help with selecting the best representation instead of merely confirming expected results.

Our implementation is a Scala library which is rather small and generic. It uses first class type descriptions and works for any data types defined by the user in his/her application.

The proposed isomorphic specialization cannot and should not replace other optimization techniques. In particular here we show how it works in concert with LMS. In terms of optimization, the main benefit comes from deforestation and loop fusion of array operations which are implemented in LMS. It turned out that for all specialized versions LMS was able to automatically generate Scala code eliminating unnecessary intermediate arrays and fusing all the loops. If you were writing it manually you would write very similar code. This means that we can use LMS is an efficient implementation of our core functional language with arrays. All we have to do is to translate domain-specific abstractions to this core language. This separation of concerns works very well in practice, as it allows to build a solution from the *best-in-class* components.

Thus, we have shown how to develop an object-oriented functional language in which you can create abstractions without worrying about their performance overhead.

5. Related work

This work is based on our previous attempts [28] to combine generic (aka polytypic) programming techniques and Lightweight Modular Staging [22] in the context of Scala language. That work was mostly focused on technical details of deep DSL embedding using some tricks of Scala language. The main idea is that by writing programs using a polymorphic embedding style [13], they can be interpreted in at least two modes: evaluation and code generation. In the evaluation mode programs are immediately executed using runtime of the host language (Scala). In the code generation mode *the same code* yields a graph-based intermediate representation. This was the main motivation for explicit formulation of staged evaluation as it is presented in this paper.

Our current implementation of staged evaluation and isomorphic specialization is derived from Scalan [28]. We removed everything related to NDP and generalized the Scalan library in such a way that NDP could be considered as an application. Similar to LMS we use `Rep[T]` based embedding in Scala, but we don't use Scala-virtualized [3] for such embedding. Instead we use dynamic proxies for embedding of user-defined types and to implement method invocation behavior of staged evaluation. And this is where the staged evaluation is different from LMS. First, we put global *smart constructors* of LMS into user-defined classes where they become just methods. Second, we allow class instances as DAG nodes. Third, the only implementation of `Exp[T]` during staging is `Sym[T]`. This makes `Exp[T]` instances always behave like typed references to DAG nodes, so that if `e : Exp[Matr[Float]]` then semantically `e` is the address of a node of type `Matr[Float]`. Another difference from LMS is that rewriting doesn't happen in smart constructors. Instead it is defined by a separate set of rules applied until fixed point is reached. We found this technical difference very convenient in practice as the staging mechanism is separated from rewriting.

In spite of the differences, thanks to flexibility of Scala as the host language, we can combine Scalan and LMS into an end-to-end solution. After the final graph is created we create an instance of an LMS context and inject all the core language primitives produced by isomorphic specialization into it. Then the Scala code corresponding to this instance is generated, leveraging code generation and optimizations implemented in LMS.

Sacher [26] is the main source of inspiration for the notation of marked DAGs and the collapsing injection. DAG rewriting based on pattern matching using extractors (or active patterns) is due to [22]. We found the combined notation of marked DAGs and active patterns very convenient for our formulation.

The idea of not rebuilding already-present nodes can be traced back to Sassa and Goto [25], who described what is usually called *hash consing*. Kahrs [17] showed how it can be used to implement fully-collapsed jungles.

We describe the staged evaluation algorithm as a zipper-based traversal. This formulation is closely related to Danvy's research [9, 10] on inter-deriving semantic artifacts.

Collapsing injection of terms into DAGs that we perform as part of staged evaluation corresponds to the approach to common sub-expression elimination in Rompf [24]. Similarly dead code

elimination is automatically achieved by respecting the dependency relation during calculation of binding scopes.

We use pattern matching of DAGs in a way similar to a method described in [21]. But because we work in a purely functional context without effects our formulation is different and is based on dynamic recognition of binding scopes in DAGs and extracting them using active patterns. This greatly simplifies formulation of and reasoning about rewriting transformations.

The object-oriented extension of the core language is inspired by Featherweight Java [16] but we adapted their formulation to our core language.

Isomorphic representations of types have a long history in the generic programming community (see [14] for an overview), but the main question is usually how to automatically generate isomorphisms for user defined types. We, on the contrary, emphasize *user-defined isomorphisms* as a bridge between an abstraction and some concrete representation in the core language. Thus our approach is the opposite: we require the user to explicitly specify how he wants each concrete implementation to be represented in the core language. Because staged evaluation happens at runtime we can also bind this isomorphism specification with a runtime configuration framework (e.g. using dependency injection).

6. Conclusion

The potential of a new approach can be judged by theoretical generality and practical simplicity. We described isomorphic specialization for an enriched simply-typed lambda calculus extended with a very limited set of object oriented constructs just to capture the essence of the approach and simplify our presentation. But it is by no means limited by this language characteristics. Our implementation works with polymorphic types and functions, supports inheritance hierarchies and multiple inheritance of abstract types.

At the same time the behavior of specialization transformation is robust, predictable and very efficient in practice. It worked for all data types we used in an implementation of machine learning algorithms such as Logistic Regression and SVM, and it also shines in defining various representations of graphs and specializing abstract graph algorithms.

Under not too strict and quite reasonable conditions, isomorphic specialization is automatic and provides strong guarantees which we formulated as a conjecture in Section 3.5. We didn't prove this statement formally, but we have a strong evidence supported by many examples that it always holds.

Isomorphic specialization as a particular transformation is based on the machinery established by staged evaluation. This is a new formulation of staging which allows the use of term rewriting to simplify graph construction and transformations. The specialization transformation presented in this paper is just one possible application of staged evaluation.

We also showed that generic programming techniques together with staging can lead to a very simple yet powerful specialization method which can be made first-class in a high-level functional language.

We gave a formalized description of isomorphic specialization algorithm and showed that it can be implemented as a set of simple rewriting rules over graph-based IRs.

In our formalization we rely on acyclic graphs assuming that we deal only with non-recursive programs. This may sound like quite a limitation but in practice it greatly simplifies implementation and reasoning about it while still allowing us to support many practical data structures. This is the main limitation of presented algorithms but not the approach itself and we consider it as future research.

Besides recursion, we have not covered many questions about formal properties of the presented methods and we didn't prove our conjecture. We presented staged evaluation for a call-by-value language. Following the work of Danvy [9], the technique can probably be made independent of the evaluation order. We rely on full reification of types but it may be interesting to further investigate type usage and characterize some minimal requirements regarding reification of types. This and similar questions are also directions of future research.

Finally, we limit ourselves to pairs and binary sums of types in formalization and, in fact, in our prototype implementation. But extending to arbitrary tuples and tagged unions would be useful and could be done using e.g. the Shapeless [1] library for Scala.

Acknowledgments

The authors express their gratitude to Tiark Rompf and the anonymous reviewers of WGP 2014 for numerous useful comments and suggestions for improvements of this paper.

References

[1] Shapeless: Generic Programming for Scala. `http://typelevel.org/`.

[2] Failure is not an option: Popular parallel programming. Technical report, Workshop on Advancing Computer Architecture Research (ACAR-1), 2010.

[3] Philipp Haller Adriaan Moors, Tiark Rompf and Martin Odersky. Tool Demo: Scala-Virtualized, 2011.

[4] Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Shonan challenge for generative programming: short position paper. In *PEPM*, pages 147–154, 2013.

[5] Kevin J. Brown, Arvind K. Sujeeth, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. volume 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011.

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

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

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

[9] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. *ACM SIGPLAN Workshop on Continuations*, 2004.

[10] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In Wilfrid Hodges and Ruy de Queiroz, editors, *Logic, Language, Information and Computation*, volume 5110 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[11] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 273–298, Berlin, Heidelberg, 2007. Springer-Verlag.

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

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

148, New York, NY, USA, 2008. ACM.

[14] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *Tarmo Uustalu, editor, Proceedings 8th International Conference on Mathematics of Program Construction, MPC'06, volume 4014 of LNCS*, pages 209–234. Springer-Verlag, 2006.

[15] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[16] Atsushi Igarashi, BC Pierce, and Philip Wadler. Featherweight Java : A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming ...*, 23(3):396–450, 2001.

[17] Stefan Kahrs. Unlimp uniqueness as a leitmotiv for implementation. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 115–129. Springer Berlin Heidelberg, 1992.

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

[19] Conor Mcbride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

[20] NVIDIA. NVIDIA CUDA C Programming Guide, 2011.

[21] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne, 2012.

[22] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[23] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*, pages 497–510, 2013.

[24] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan

Chafi, Martin Odersky, and Kunle Olukotun. Building-Blocks for Performance Oriented DSLs. In *DSL*, pages 93–117, 2011.

[25] Masataka Sassa and Eiichi Goto. A hashing method for fast set operations. *Inf. Process. Lett.*, 5(2):31–34, 1976.

[26] Jens Peter Secher. Driving-based program transformation in theory and practice, 2002.

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

[28] Alexander V. Slesarenko. Lightweight Polytypic Staging of DSLs in Scala. In S.A. Romanenko A.V. Klimov, editor, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages pp.228–256. Ailamazyan University of Pereslavl, July 2012.

[29] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *GPCE*, pages 145–154, 2013.

[30] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, pages 52–78, 2013.

[31] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. ACM.

[32] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, December 1997.

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, 2011.