Polymorphic Extractors for Semantic and Portable Pattern Matching (Short Paper)

Amir Shaikhha
University of Oxford
United Kingdom
amir.shaikhha@cs.ox.ac.uk

Abstract

This paper introduces polymorphic extractors, a technique for tackling two main issues with the existing pattern matching techniques in functional languages. First, this technique defines semantic pattern matching rather than a syntactic one. Second, this technique solves the portability issue when defining a set of patterns based on different underlying data-structure design choices. Furthermore, polymorphic extractors can be further improved by performing optimizations and multi-stage programming. The key technique behind polymorphic extractors is using the tagless-final technique (a.k.a. polymorphic embedding/object algebras) for defining different extraction semantics over expression terms.

CCS Concepts • **Theory of computation → Pattern matching**; *Rewrite systems*; • **Software and its engineering →** *Functional languages*.

Keywords Extractors, Tagless Final, Language Embedding

ACM Reference Format:
Amir Shaikhha. 2019. Polymorphic Extractors for Semantic and Portable Pattern Matching (Short Paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357765.3359522

1 Introduction

Pattern matching has many applications in term rewriting systems, optimizing compilers, proof systems, and verification tools. Building compilers for domain-specific languages (DSLs) benefits greatly from pattern matching; DSL developers encode domain-specific knowledge through rewrite rules for optimizing DSL expressions [9, 19, 21, 23, 25, 29].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '19, October 21–22, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359522

However, the standard pattern matching provided by functional languages suffers from two main problems. First, such pattern matching is *syntactic* and misses the *semantic* knowledge of the DSL. For example, consider the arithmetic expression $42 * (4 + 3)$. Based on mathematical knowledge it is obvious that both of the following patterns should successfully match this expression: 1) $a * (b + 3)$, and 2) $a * (3 + b)$. However, the standard pattern matching considers only the first pattern to have a successful match.

Second, there are different design decisions for defining an intermediate representation (IR) for an optimizing compiler. Each IR design choice enforces the rewrite rules to respect some forms of constraints (e.g., in ANF [6] all the subexpressions should be either a constant literal or a variable). Hence, changing the IR used in an optimizing compiler, requires a complete refactoring of all the rewrite rules. This means that these rewrite rules are not *portable*.

In this paper, we solve these two problems by combining two techniques. First, inspired from the notion of Views [30] and Scala extractors [5] (which are similar to active patterns in F# [27] and pattern synonyms in Haskell [18]), we define an extensible version of pattern matching; we define objects that examine a given pattern on a tree-structured value, and in the case of a successful match, extract some of its subtrees. We refer to such objects as *extractors* throughout this paper, and we focus on extractors for expression terms.

Second, we make the implementation of the extractors polymorphic, and thus portable, using the tagless-final [2, 12] (also known as polymorphic embedding [7] or object algebras [15]) technique. This techique has been successfully used for having different evaluation semantics for a DSL (e.g., interpretation, compilation, partial evaluation, etc.) [2, 7, 15, 21]. In this paper, we define the extraction evaluation semantics in the tagless-final embedding.

The contributions of this paper are as follows:

- After giving a background on the tagless-final approach (Section 2), we introduce extractors (Section 3) and a simple tagless-final embedding of them (Section 3.1).
- We better demonstrate the polymorphic nature of our extractors by showing how they can handle syntactic sugar constructs (Section 3.2), commutative operators (Section 4.2), associative operators (Section 4.3), and further possible improvements (Section 4.4).

- We use two different IR definitions (one in Section 2.3 and another one in Section 4) to better demonstrate the portability of our extractors.

2 Background: Tagless Final

Tagless final [2, 12] (also known as polymorphic embedding [7] or object algebras [15]) is a type-safe approach for *embedding* [8] domain-specific languages in a host programming language. This approach allows the DSL developer to make the DSL definition extensible from two aspects: 1) adding different evaluation semantics for the constructs of a DSL (e.g., interpretation, compilation, etc.), and 2) the possiblity to extend the set of constructs defined for a DSL.

Among different techniques for implementing this approach (e.g., type classes [2, 12] and the module system [2, 13]), we have chosen the mixin composition (also known as the cake pattern), as this approach has been used before for implementing DSLs in Scala [7, 21].

Throughout this paper, we consider a DSL defined for semiring-based arithmetic operations on real numbers, an extended form of which has been used for linear algebra DSLs [4, 13, 24]. This DSL consists of two binary operations for addition and multiplication, as well as a construct for creating a constant real number. The tagless-final interface for this DSL is as follows:

```
trait SemiRingDSL {
  type Rep
  def const(d: Double): Rep
  def add(a: Rep, b: Rep): Rep
  def mult(a: Rep, b: Rep): Rep
}
```

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

One can then define generic programs in the DSL:

```
class App1[B <: SemiRingDSL](val base: B) {
  import base._
  def exp1: Rep = {
    mult(const(42), add(const(4), const(0)))
  }
}
```

Next, we show different evaluation semantics for this DSL.

2.1 Interpretation

One can define an interpreter for the DSL by specifying the `Rep` type as `Double`. The following trait defines an interpreter for `SemiRingDSL`:

```
trait SemiRingDSLInter extends SemiRingDSL {
  type Rep = Double
  def const(d: Double): Rep = d
  def add(a: Rep, b: Rep): Rep = a + b
  def mult(a: Rep, b: Rep): Rep = a * b
}
```

Evaluating the example program above in the REPL with this evaluation semantics, results in:

```
scala> new App1(new SemiRingDSLInter {}).exp1
result: Double = 168.0
```

2.2 Stringification

Rather than directly *evaluating* DSL programs, one can also represent the programs as strings. Below is a trait that *stringifies* programs in our semiring-based DSL:

```
trait SemiRingDSLStringify extends SemiRingDSL {
  type Rep = String
  def const(d: Double): Rep = d.toString
  def add(a: Rep, b: Rep): Rep = s"($a + $b)"
  def mult(a: Rep, b: Rep): Rep = s"($a * $b)"
}
```

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

```
scala> new App(new SemiRingDSLStringify {}).exp1
result: String = "(42.0 * (4.0 + 0.0))"
```

2.3 Compilation

Another possibility is to have a symbolic representation of the program using algebraic data types (ADTs). The following code represents the ADT representation for the constructs of this DSL, and the tagless-final evaluation strategy for creating a symbolic representation (or compiled version of a DSL program):

```
sealed trait Exp
case class Add(a: Exp, b: Exp) extends Exp
case class Mult(a: Exp, b: Exp) extends Exp
case class Neg(a: Exp) extends Exp
case class Const(v: Double) extends Exp

trait SemiRingDSLExp extends SemiRingDSL {
  type Rep = Exp
  def add(a: Rep, b: Rep): Rep = Add(a, b)
  def mult(a: Rep, b: Rep): Rep = Mult(a, b)
  def const(d: Double): Rep = Const(d)
}
```

Evaluating the same program by using the compilation evaluation semantics results in:

```
scala> val exp1 = new App(new SemiRingDSLExp
    {}).exp1
exp1: Exp =
    Mult(Const(42.0),Add(Const(4.0),Const(0.0)))
```

This representation can later on be manipulated in order to encode domain-specific optimizations. Next, we define another evaluation semantics for extracting patterns on the compiled representation of DSL programs.

3 Extractor

In this section, we present simple extractor objects for the programs of our example DSL. Generally speaking, when an extractor is applied to a tree-shaped data structure, in the case of a successful match, it returns the subtrees of that data structure. However, it is also possible that an extractor fails the match. Thus, a generic form of extractor has the following type:

```
type GenExt[Node] = Node => Option[Seq[Node]]
```

More specifically, an extractor for the compiled representation of the DSL programs has the following type:

```
type Ext = Exp => Option[Seq[Exp]]
```

3.1 Simple Tagless Final Extractor

Next, we present the generic tagless-final interface for the extractor evaluation semantics:

```
trait SemiRingDSLExtGen extends SemiRingDSL {
  type Node
  type Rep = GenExt[Node]
  def hole: Rep
  final def const(d: Double): Rep = (e: Node) =>
      constGen(e) match {
    case Some(d2) if d == d2 => Some(Seq())
    case _ => None
  }
  def constGen: Node => Option[Double]
}
```

The `hole` method matches with every instance of the input tree-shaped data structure, returning a sequence of one element containing the whole tree. The `constGen` method extracts the double value from the double constant literal. Thus, the `const` method uses the `constGen` and checks if the double value is the same as the expected one. In the case of a success match, the `const` method returns an empty sequence; otherwise, the match should fail (by returning a `None` value).

For the previously mentioned compiled reprsentation (cf. Section 2.3), the tagless extractor interface is as follows:

```
trait SemiRingDSLExt extends RingDSLExtGen {
  type Node = Exp
  def add(a: Rep, b: Rep): Rep = (e: Exp) => e
      match {
    case Add(av, bv) => (a(av), b(bv)) match {
      case (Some(s1), Some(s2)) => Some(s1 ++ s2)
      case _ => None
    }
    case _ => None
  }
  def mult(a: Rep, b: Rep): Rep =
    // similar to addition
  def constGen = (e: Exp) => e match {
    case Const(d) => Some(d)
    case _ => None
  }
  def hole: Rep = (e: Exp) => Some(Seq(e))
}
```

All constructs are implemented as follows. First, an input expression is matched against the corresponding pattern. If the match fails, then the extractor also returns `None`. If the match is successful, then the extractors for the parameters of the construct (referred to as *subextractors*) should be applied to the subexpressions. If even one of these matches fails, the extractor must fail again. However, if all matches are successful, then the extractor concatenates the sequences returned by all subextractors.

Evaluating the previous example program by using the simple extractor evaluation semantics results in a function. Applying it to the compiled representation of the expression results in:

```
scala> val pat1 = new App(new SemiRingDSLExt
    {}).exp1
scala> pat1(exp1)
result: Option[Seq[Exp]] = Some(List())
```

This means that the match is successful. However, this extractor does not extract any of the subexpressions. The following example shows the usage of the `hole` method for retrieving a subexpression:

```
scala> val pat2 = {
        val srExt = new SemiRingDSLExt {}
        import srExt._
        mult(hole, add(const(4), hole))
      }
scala> pat2(exp1)
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Const(0.0)))
```

As this example shows, the first `hole` extracts the constant literal `Const(42.0)`, and the second one extracts the constant literal `Const(0.0)`. Note that although in this example,

the extracted terms are all constant terms, they could be any arbitrary expression term, as long as the pattern matches successfully. We have used these constant terms to make the examples more concise and easier to follow.

The next example considers the case where the extractor does not match with the given expression:

```
scala> val pat3 = {
         val srExt = new SemiRingDSLExt {}
         import srExt._
         mult(hole, add(const(0), hole))
       }
scala> pat3(exp1)
result: Option[Seq[Exp]] = None
```

Even though we know that the addition operator is commutative (the order of operands does not matter), the extractor is still syntactic; this semantic knowledge is needed to be encoded in the extractor. We will encode the commutative property into the addition operator in Section 4.2.

3.2 Syntactic Sugar Constructs

Let us extend our semiring DSL with the additive inverse construct, resulting in a ring-based DSL. The tagless-final interface for the extended DSL is as follows:

```
trait RingDSL extends SemiRingDSL {
  def neg(a: Rep): Rep
  final def sub(a: Rep, b: Rep): Rep =
    add(a, neg(b))
}
```

As one can see, the subtraction operator is defined as *syntactic sugar* in terms of the addition and negation operators. Thus, for the symbolic representation of this DSL it is sufficient to define an ADT case for the negation operator, and the subtraction operator is *desugared* into the representation for the addition and negation operators. The tagless-final evaluation semantics for compilation is as follows:

```
trait RingDSLExp extends SemiRingDSLExp with
    RingDSL {
  def neg(a: Rep): Rep = Neg(a)
}
```

Similarly, for the extraction tagless-final interface, there is no need to define the extractor for the subtraction construct; implementing the negation construct is sufficient:

```
trait RingDSLExt extends SemiRingDSLExt with
    RingDSLExtGen {
  def neg(a: Rep): Rep = (e: Exp) => e match {
    case Neg(av) => a(av)
    case _ => None
} }
```

```
class App2[B <: RingDSL](val base: B) {
  import base._
  def exp2: Rep =
    mult(const(42), sub(const(0), const(4)))
  def exp3: Rep =
    add(const(42), add(const(4), const(3)))
  def exp4_1: Rep =
    add(add(const(42), const(41)), add(const(3),
        const(4)))
  def exp4_2: Rep =
    add(add(const(42), add(const(41), const(3))),
        const(4))
  def exp4_3: Rep =
    add(add(add(const(42), const(41)), const(3)),
        const(4))
}
```

Figure 1. Several example generic programs defined in the ring-based DSL.

Consider the generic programs represented in Figure 1. By applying the previously defined extractors on `exp2`, we get the following results:

```
scala> pat1(exp2)
result: Option[Seq[Exp]] = None
scala> pat2(exp2)
result: Option[Seq[Exp]] = None
scala> pat3(exp2)
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Neg(Const(4.0))))
```

The first two extractors, fail matching with the given expression, as they are expecting the first operand of addition to be `Const(4.0)`. However, the last extractor successfully matches the expression thanks to desugaring subtraction into addition and negation. Also, when the `exp2` is evaluated by the extraction semantic, it matches with its corresponding compiled version:

```
scala> val pat4 = new App2(new RingDSLExt
    {}).exp2()
scala> pat4(exp2)
result: Option[Seq[Exp]] = Some(List())
```

In this section, we have shown the power of the simple extractors for pattern matching over expression terms, even for syntactic sugar constructs. Next, we investigate more advanced forms of extractors.

4 Deep Extractor

In this section, we show how extractors can encode more semantic knowledge of the DSL constructs. In other words, we show how to shift from syntactic pattern matching into semantic pattern matching. First, we show how to make the extractors aware of the commutative property of a construct

in Section 4.2. Then, we make the extractors for associative constructs in Section 4.3. Finally, we show future directions for advanced extractors.

4.1 Deep Embedding for Extractors

In Section 3.1, we have introduced a simple form of extractors. This form *directly* (or *shallowly*) evaluates the extractors given expression terms. However, similar to the limitations associated with the direct embedding of DSLs [10, 26], in order to enrich extractors with more features, such as domain-specific rules (Section 4.2 and Section 4.3) and optimizations (Section 4.4), we get inpsiration from deep embedding of DSLs, and introduce the *deep* extractors.

Similar to deep embedding of DSLs in a host language, we have to define the symbolic representation for each of the extractor cases. To reduce the number of cases, we introduce two cases for both expression terms and extractors: 1) constant values, and 2) n-ary functions.[1] Additionally, the extractors require a case for holes.

Figure 2 shows the tagless interface for both compilation and extractor evaluation semantics. The `apply` method of the subclasses of the `ExtDeep` trait specifies the extractor behaviour. The most interesting case is the `FuncExt` case for extracting n-ary functions. In this case, after checking if the expression is an n-ary function with the same expected symbol, all the subextractors are matched with all the subexpressions. If all of them successfully match, the concatenation of the returned sequences of all subextractors is returned.

4.2 Commutative Rules

Let us now encode the commutative property of addition. To do so, we define the following case for deep extractors:

```
case class ExtCom(ext: FuncExt) extends ExtDeep {
  def apply(e: Exp): Option[Seq[Exp]] = ext(e)
      match {
    case Some(seq) => Some(seq)
    case None => FuncExt(ext.sym, Seq(ext.pats(1),
        ext.pats(0)))(e)
} }
```

For a given extractor of a binary operator (2-ary function), the defined extractor first checks if it matches with the input expression. If not, it tries again with an extractor where the order of subextractors is swapped. This way, it tries to match the given extractor irrespective of the order of its input operands.

By using the third example extractor and constructing a commutative extractor for it, and then applying it to the first example expression, the result is as follows:

```
scala> pat3(exp1)
```

[1]The expression `'foo` is a constant literal of `Symbol` type in Scala.

```
sealed trait Exp
case class Const(v: Double) extends Exp
case class FuncExp(sym: Symbol, es: Seq[Exp])
    extends Exp
trait RingDSLExpDeep extends RingDSL {
  type Rep = Exp
  def neg(a: Rep): Rep = FuncExp('Neg, Seq(a))
  def add(a: Rep, b: Rep): Rep =
    FuncExp('Add, Seq(a, b))
  def mult(a: Rep, b: Rep): Rep =
    FuncExp('Mult, Seq(a, b))
  def const(d: Double): Rep = Const(d)
}

sealed trait ExtDeep extends Ext
case class FuncExt(sym: Symbol, pats: Seq[Ext])
    extends ExtDeep {
  def apply(e: Exp): Option[Seq[Exp]] = e match {
    case FuncExp(sym2, es) if sym == sym2 =>
      pats.zip(es).foldLeft(Option(Seq[Exp]()))((acc,
          cur) => (acc, cur._1(cur._2)) match {
        case (Some(v1), Some(v2)) => Some(v1 ++ v2)
        case _ => None
      })
    case _ => None
} }
case object ConstExt extends (Exp=>Option[Double]){
  def apply(e: Exp): Option[Double] = e match {
    case Const(d) => Some(d)
    case _ => None
} }
case object HoleExt extends ExtDeep {
  def apply(e: Exp): Option[Seq[Exp]] = Some(Seq(e))
}
trait RingDSLExtDeep extends RingDSLExtGen {
  type Node = Exp
  def add(a: Rep, b: Rep): Rep =
    FuncExt('Add, Seq(a, b))
  def mult(a: Rep, b: Rep): Rep =
    FuncExt('Mult, Seq(a, b))
  def neg(a: Rep): Rep = FuncExt('Neg, Seq(a))
  def constGen = ConstExt
  def hole: Rep = HoleExt
}
```

Figure 2. Tagless final interface for the deep embedding of both compilation and extractor evaluation semantics.

```
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Const(4.0)))
```

As opposed to what we have observed in Section 3.1, this extractor successfully matches the given expression. This is thanks to encoding the commutative property of addition in its corresponding extractor.

4.3 Associative Rules

Another important algebraic property for the addition and multiplication operators is associativity. As opposed to a commutative operator, the order of operands is important in an associative operator. However, the pairing of operators for evaluating them is not important. The following case handles associative operators:

```
case class ExtAsc(pats: Seq[Ext], sym: Symbol)
    extends ExtDeep {
  def apply(e: Exp): Option[Seq[Exp]] = e match {
    case FuncExpN(s, es) if s == sym =>
      pats.zip(es).foldLeft(Option(Seq[Exp]()))((acc,
          cur) => (acc, cur._1(cur._2)) match {
        case (Some(v1), Some(v2)) => Some(v1 ++ v2)
        case _ => None
      })
    case _ => None
} }
```

The evaluation of an associative extractor is very similar to an n-ary function. The key difference is using the FunExpN (the implementation of which is elided due to space constraints) which returns the list of the operands, no matter how their paranthesisation looks like.

Let us use the last three example expressions of Figure 1 that are equivalent modulo the fact that the association of the addition operator is different among them.

```
scala> val pat4 = {
         val rExt = new RingDSLExtDeepAsc {}
         import rExt._
         add(add(hole, hole), add(const(3), hole))
       }
scala> pat4(exp4_1)
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Const(41.0), Const(4.0)))
scala> pat4(exp4_2)
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Const(41.0), Const(4.0)))
scala> pat4(exp4_3)
result: Option[Seq[Exp]] = Some(List(Const(42.0),
    Const(41.0), Const(4.0)))
```

The extractor pattern has matched all the expression terms returning the same subexpressions, no matter how their addition operators were associated.

4.4 Further Improvements

There are several directions in which the current deep extractors can be improved.

AC Extractors. We have shown how to define commutative and associative extractors separately. However, we have not shown how to define extractors which are simultaneously associative and commutative (referred to as AC rewrite rules [3]). One possibility to support them is to define a more generic form of commutative extractors, which does not only handle binary operators, but rather n-ary functions. This is feasible by defining an order for subexpressions based on their symbols and operands.

Normal Form Extractors. Many optimizing compilers are using specific normal forms such as CPS [1], SSA [22], and ANF [6] in order to simplify data-flow analysis on the expression terms. It is possible to provide extractors which are aware of the underlying normal form, without the need to worry if the subexpressions are bound to a variable or not.

Optimizing Extractors. By looking at the implementation of the `apply` function of the extractors, one can realise that there is too much overhead of function calls and iterations using `foldLeft`. One possible way to remove this overhead is to use multi-stage programming [28] to perform partial evaluation and loop fusion [14].

Guarded Extractors. Up to now, we have only considered extractor patterns without any particular guard. This means that there is no support for *conditional* rewrite rules [11]. An an example of conditional rewrite rules, one may want to check if two holes extract the same expression or not. Non-linear pattern matching [20] is an approach to solve this problem, which can be integrated into our library.

Named Holes. The pattern holes presented in this paper are all unnamed; an extractor returns a sequence of extracted values, the order of which is specified by the evaluation order, which makes their usage unintuitive. To avoid this issue, one can use named holes; an extractor returns a map associating hole names with the corresponding extracted value.

Quasiquotes. An alternative way for improving the usability of polymorphic extractors is using quasiquotes to express expression terms in the pattern side. These pattern expressions are converted using macro-based frameworks such as Yin-Yang [10] or Squid [16, 17] into their tagless-final representation. The example extractors defined in Section 3 are represented as follows using quasiquotes:

```
// pat1
e match { case code"42 * (4 + 0)" => ... }
// pat2
e match { case code"$a * (4 + $b)" => ... }
// pat3
e match { case code"$a * (0 + $b)" => ... }
```

In these examples, the `*` and `+` operators are converted into `mult` and `add` method invocations, respectively. Furthermore, the meta variables (`a` and `b`) are converted into the `hole` method invocation. Then, based on the call-by-value evaluation strategy, the expression extracted by each hole is associated with the corresponding meta variable.

References

[1] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[2] 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.

[3] Nachum Dershowitz. A taste of rewrite systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 199–228. Springer, 1993.

[4] 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.

[5] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, pages 273–298. Springer, 2007.

[6] 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.

[7] 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.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[9] Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN.

[10] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-Yang: Concealing the deep embedding of DSLs. GPCE 2014, pages 73–82. ACM, 2014.

[11] Stéphane Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175 – 193, 1984.

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

[13] Oleg Kiselyov. Reconciling abstraction with high performance: A metaocaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101, 2018.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A Eisenberg. Pattern synonyms. In *ACM SIGPLAN Notices*, volume 51, pages 80–91. ACM, 2016.

[19] Markus Puschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[20] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *J. ACM*, 39(2):295–316, April 1992.

[21] 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.

[22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. POPL '88, pages 12–27. ACM, 1988.

[23] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient Differentiable Programming in a Functional Array-processing Language. *Proc. ACM Program. Lang.*, 3(ICFP):97:1–97:30, July 2019.

[24] Amir Shaikhha and Lionel Parreaux. Finally, a Polymorphic Linear Algebra Language. In *Proceedings of the 33rd European Conference on Object-Oriented Programming*, ECOOP'19, 2019.

[25] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.

[26] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for edsl. In *International Symposium on Trends in Functional Programming*, pages 21–36. Springer, 2012.

[27] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *ACM SIGPLAN Notices*, volume 42, pages 29–40. ACM, 2007.

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

[29] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP'98, pages 13–26, 1998.

[30] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM, 1987.