Type-Directed Language Extension for Effectful Computations

Evgenii Kotelnikov
Chalmers University of Technology
Gothenburg, Sweden
evgenyk@chalmers.se

ABSTRACT
Computation types such as functors, applicative functors and monads have become common abstractions for modeling effectful computations in functional programming languages. They are often used together with special language extensions intended to simplify the syntax of monadic expressions. We can simplify it even more by employing types rather than just mechanical syntactic transformation.

In this paper we present scala-workflow, a macro-based extension to the Scala programming language that offers uniform syntax for structuring effectful computations expressed as abstract interfaces with a stackable set of combinators. Unlike similar syntactic extensions, such as F#’s computation expressions, Haskell’s do-notation and Scala’s own for-expressions, scala-workflow allows users to transparently blend pure and effectful functions in a bounded context, as they are separated during macro expansion based on the types of subexpressions.

Categories and Subject Descriptors
D.3.2 [Programming languages]: Language Classification — Applicative (functional) languages; D.3.3 [Language Constructs and Features]: Control structures

General Terms
Languages, Algorithms

Keywords
Scala, macros, effects, monads, functors

1. INTRODUCTION
Programs that involve effectful aspects of computation, like non-determinism, concurrency, exceptions or IO, invite programming languages to seek appropriate abstractions that can capture them. Some languages prefer to have built-in 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.
Scala ’14, July 28–29, 2014, Uppsala, Sweden
Copyright 2014 ACM 978-1-4503-2868-5/14/07 ...$15.00.
http://dx.doi.org/10.1145/2637647.2637648

single-purpose primitives. Others, and most notably functional ones, employ a variety of generic algebraic structures capable of expressing computational effects.

The concept of monads, originally introduced in the context of computations by Moggi [16] gained significant attention in the Haskell community and became the de-facto standard way of structuring computations with effects in Haskell [29]. McBride and Paterson introduced applicative functors [15] (also known as sequences or idioms), that are weaker than monads, but are more widespread. Both monads and idioms are supersets of functors [11], however the latter are rarely used as effectful containers.

An entirely different approach to effectful computations are type and effect systems [25]. In the context of Scala there is ongoing work [22] on lightweight encoding of effect-polymorphic function types in Scala annotations.

To facilitate effectful computations, functional programming languages provide special extensions, aimed to simplify the syntax of the application of effectful combinators. The main idea underneath most of them is to extend the language with special syntactic forms, that are mechanically rewritten into calls of effectful combinators.

In this paper we present scala-workflow, a syntactic extension for the Scala programming language, available for download at https://github.com/aztek/scala-workflow. Unlike most similar extensions in other programming languages, scala-workflow does not simply expand the expression according to syntax-driven transformation rules, but employs type information of subexpressions to direct the rewriting process. Ultimately, scala-workflow allows users to transparently blend effectful and pure computations, which otherwise would require manual separation and more boilerplate code.

To illustrate this approach, let us give an example of composition of asynchronous computations built with Scala’s concurrent.Future class. Here, two potentially slow computations f() and g(x; Int), producing integers, are placed inside the future call to create future objects.

val f: Future[Int] = future { f() }
def g(x: Int): Future[Int] = future { G(x) }

With scala-workflow the arithmetic expression that combines the results of f and g can be expressed in direct style. The result is a future object that waits for the execution of f, supplies its incremented value to g and doubles the result.

context[Future] {  
  $(2 * g(f + 1))  
}
The arithmetic expression is put inside the $ macro that bounds the scope of rewriting and the context declaration that specifies the type of computational effect. Whereas addition and multiplication, applied to future objects, do not type check in ordinary Scala, they are correctly used with our extension, because the macro expansion takes place before the type checking. The produced well-typed expression uses calls to methods bind and map of the object future_ that implements the treatment of effects similarly to Future's own flatMap and map.

```scala
future_.bind(
  (x$1: Int) ⇒
  future_.map(
    (x$2: Int) ⇒
    2 * x$2
  ) (g(x$1 + 1))
) (f)
```

The $ macro notation is different from the usual way of expressing composition of futures in Scala with for-comprehension, as illustrated by the snippet below.

```scala
for {
  x ← f
  y ← g(x + 1)
} yield 2 * y
```

Note that one has to separate monadic bindings from pure multiplication and name intermediate results, which was not needed with scala-workflow.

Unlike for-notations, scala-workflow is not tied to monads. An elaborate rewriting algorithm enables uniform syntax for different computation types. For example, applicative functors receive syntactic support with the same $ macro as long as the expression inside does not require handling effects that exceed their capabilities. The next snippet produces a stream, built by summing elements standing on the same position in streams x, y, and z, using zipStream, Scala's counterpart of Haskell's zipList idiom.

```scala
context(zipStream) { ...
  $(x * + yz + zs)
}
```

The fitness of the computation type is checked statically and the rewriting algorithm always rewrites the expression in a way that requires the least powerful computation type sufficient to express the computation.

In what follows, we first describe related work in Section 2. Section 3 overviews the main features of scala-workflow. Next, a number of use cases are detailed in Section 4. Section 5 addresses further improvements and Section 6 concludes the paper.

2. RELATED WORK

Perhaps the most famous syntactic extension for monadic computations is Haskell's do-notation [12]. It is based on mechanical translation of monadic bindings within a do block into calls of monadic >>> and >>. Pure and effectful bindings end up syntactically separated, as the former are defined explicitly as let-expressions.

The do-notation influenced similar syntactic extensions in other programming languages, such as perform-notation in OCaml [5] or let1-notation in Common Lisp [23]. Scala provides its counterpart known as for-expressions [17]. The principal idea remains the same, however Scala supports more versatile syntax and therefore employs a variety of combinators, such as map, flatMap, withFilter and foreach. Due to the lack of a built-in interface for monads in Scala, for-expressions expect an object to implement methods with appropriate names and types. Again, pure bindings are syntactically distinguished from effectful ones.

A generalization of the list comprehension syntax [27], known as monad comprehension [28] is available in Haskell. It allows to shrink multiple monadic bindings into a one-line expression, that can further be refined with projection and grouping functions. This can be a better alternative to do-notation for certain applications [8].

Together with the original definition of idiom, the notation of idiom brackets was introduced. They essentially unclutter the syntax of application of pure function to effectful arguments. Idiom brackets are now available in Idris [2] and Strathclyde Haskell Enhancement [14]. A proposal has been made to add support for applicative syntax resembling do-notation to GHC. scala-workflow trivially supports idiom brackets by implementing a more general notation.

Petrickev and Synne proposed a more elaborate approach to effectful syntax [20]. Their result, called computation expression, is available as part of the F# programming language [19]. The idea behind it is to extend the syntax of the language with effectful counterparts of some of the keywords, such as let, return and yield (correspondingly, let!, return! and yield!) and use them to demarcate effectful computations in a context, bounded by the scope of a computation builder. The content of the bounds is then translated, again in a purely syntactic manner, to calls of builder methods. What makes this approach more refined is that the builder is merely a stackable set of methods, that mirror the features of the language (for example, while method for loops, TryWith for exceptions and Delay for lazy computations). We borrow this idea in scala-workflow to a certain extent, but put it in a setting where no extension of the host language is needed.

A macro-based approach to the implementation of a syntactic extension for asynchronous computations has been taken in async [9] by Haller and Zaugg. The code is put inside the async macro call with the await function call denoting asynchronous subexpressions. Crockett's effectful [6] library generalizes this approach to arbitrary monads. The approach taken in scala-workflow essentially allows to get rid of explicit annotations of effectful subexpressions by inferring them automatically during macro expansion. Both async and effectful, however, support wider subset of language features of Scala, allowed in the annotated block of code, including conditional expressions, loops and pattern matching.

A somewhat generalized translation, discriminating pure from effectful expressions, was implemented in Scala's extension for delimited continuations [21]. It employs selective CPS transformation driven by type and effect system based on $cps type annotations. Effectful monadic computations can be expressed in direct style with this framework, provided that the user reflects monadic values explicitly by passing continuations to underlying bind operations. The translation procedure, presented in this work, employs a

1https://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo
similar style of type-directed transformation, however, is is driven by effectful type constructors, marking effectful expressions, rather than type annotations. This approach enables less boilerplate, but loses type safety of the input code.

Swamy et al. [24] introduced the type-directed extension to ML for monadic programming. This work implements a similar technique for Scala, but additionally employs special analysis of dependencies between effectful expressions that enables support of the whole hierarchy of computation types rather than just monads.

3. OVERVIEW

The implementation of scala-workflow is built with Scala’s compiler-time macro system. Shortly after its inception in 2012 [4], it has proved [3] to be a productive infrastructure for a number of programming techniques, such as language virtualization, deeply embedded DSLs and boilerplate generation. scala-workflow employs untyped macros, one of the macro flavors of the Macro Paradise project. This approach is different from extensions to other languages, commonly implemented at the compiler level, and allowed us to obtain a modular and extensible implementation.

The main language feature, provided by the library is workflow brackets, syntactically represented as $\langle\rangle$ macro function. The code that is put inside the brackets is freed from boilerplate generation. This freedom allows it not to pass type checking in Scala. The macro inserts all the necessary combinators and expands into correct code.

3.1 Hierarchy of workflows

The key concept of scala-workflow is a workflow, that is a structure that collects features of computation types. All workflows are instances of the Workflow[F[_]] trait where F[_] is a type constructor that encodes an effect produced during a computation. The Workflow[F[_]] trait itself does not provide any methods and is merely an initial point of extension with new methods coming from derived traits. It is the calls of these methods that get inserted during rewriting of the expression inside the workflow brackets. Every method added to the Workflow[F[_]] corresponds to a language feature that can be used in the expression.

Currently scala-workflow supports four traits, that extend the base trait. All of them make different kinds of interplay between effectful computations admissible inside the brackets.

Method point enables the brackets not to contain any effectful computations at all. In such case the expression inside the brackets simply becomes the argument of point. Note, that point corresponds to Haskell’s monadic return and applicative pure.

trait Pointing[F[_]] extends Workflow[F] { 
  def point[A] (a: => A) : F[A]
}

As a running example, consider the workflow instance of type Workflow[Option] that treats the partial nature of computations with Option as effect. Thus, the expression such as $(42)$ gets rewritten into option.point(42), because 42 is a value, and thus a pure expression.

Method map enables the brackets to contain exactly one effectful expression. As the name suggests, this method corresponds to a functor’s map.

trait Mapping[F[_]] extends Workflow[F] { 
  def map[A, B](f: A => B) : F[A] => F[B]
}

In the context of the option workflow, a safe floating point division function divide can be seen as an effectful computation.

```scala
val divide(x: Double, y: Double): Option[Double] = 
  if (y == 0) None 
  else Some(x / y)
```

To safely calculate an expression, such as $1 + \frac{m}{n}$ for any $n$ and $m$, one can simply write $(1 + \text{divide}(n, m))$. The implementation of option propagates the effect of the absence of return value when $m = 0$, so that the whole expression would be evaluated to None in such case. The produced expression is the following one.

```scala
option.map( 
  (x$1: Double) ⇒ 
  1 + x$1 
)(divide(n, m))
```

An expression with more than one effectful subexpression, such as $\text{divide}(n, m) + \text{divide}(p, q)$, cannot appear inside the brackets of a workflow that only implements map.

Method app, however, allows the brackets to contain an arbitrary positive number of independent effectful expressions. It is a generalization of map, therefore Applying extends Mapping. It corresponds to Haskell’s applicative $\llbracket\rrbracket$ operator.

trait Applying[F[_]] extends Workflow[F] with Mapping[F] { 
}

Generally speaking, Applying is capable of lifting [10] the application of a pure function over effectful arguments. Thus, the expression $(\text{divide}(n, m) + \text{divide}(p, q))$ can be rewritten with a combination of app and map.

```scala
option.app( 
  option.map( 
    (x$1: Int) ⇒ (x$2: Int) ⇒ 
    x$1 + x$2 
  )(divide(n, m)) 
)(divide(p, q))
```

To illustrate the concept of dependence between effectful computations, consider an expression divide(divide(n, m), k). Here, the result of the outermost divide can only be obtained after the result of the innermost divide is known and computational effect during its execution has been treated. This was not the case with the previous examples, where the order of evaluation of effectful expressions was not specified. Methods app and map are not capable of expressing this kind of structured evaluation.

Finally, method bind enables the brackets to contain a pair of dependent effectful expressions. It corresponds to Haskell’s $\llbracket\rrbracket$ and Scala’s flatMap.

trait Binding[F[_]] extends Workflow[F] { 
}

http://docs.scala-lang.org/overviews/macros/paradise.html
Nested divide calls are allowed in a workflow with bind.

```scala
option.bind(
  (x$1: Double) ⇒
  divide(x$1, k)
)(divide(n, m))
```

It is of course easy to compose more common abstractions of a functor, an idiom and a monad by mixing traits with appropriate methods, effectively forming a hierarchy of these structures. scala-workflow elaborates on it slightly more by adding semigroupoids to the picture. This is a concept, much like functors and monads, borrowed from category theory [1]. Informally, semi-monads and semi-idioms are respectively monads and idioms without the point method.

```scala
trait Functor[F[_]] extends Mapping[F]

trait SemiIdiom[F[_]] extends Functor[F]
  with Applying[F]

trait Idiom[F[_]] extends SemiIdiom[F]
  with Pointing[F]

  def map[A, B](f: A ⇒ B) = app(point(f))

trait SemiMonad[F[_]] extends SemiIdiom[F]
  with Binding[F]

trait Monad[F[_]] extends Idiom[F]
  with Binding[F]

  def app[A, B](f: F[A ⇒ B]) =
  bind(a ⇒ bind((g: A ⇒ B) ⇒ point(g(a)))(f))
```

Note that we are able to provide default implementations for map in Idiom and both map and app in Monad. Using these aliases we can define workflows in the same manner we would have defined idioms and monads in other functional languages. The example below illustrates possible implementation of option.

```scala
val option = new Monad[Option] {
  def point[A](a: ⇒ A) = Option(a)
  def bind[A, B](f: A ⇒ Option[B]) = {
    case Some(a) ⇒ f(a)
    case None ⇒ None
  }
}
```

The important thing from the scala-workflow perspective, however, is that the option instance has methods point, map and app (that is, it implements correspondent traits, mixed to Workflow[Option]), because this is what will be checked during expression rewriting.

### 3.2 Context declaration

While workflow brackets scope the bounds of expression rewriting, a context declaration scopes the bounds of workflow instance application.

A context declaration is a macro context, that tells workflow brackets inside of it which workflow instance will provide methods, generated after expression rewriting. The first argument of context is either a workflow instance explicitly, or a type constructor, such that there is a workflow instance available in the implicit scope. The second argument is an expression, where all workflow brackets will be rewritten to method calls of the declared workflow instance.

To trigger rewriting of the expression in the brackets, the example from the previous section should be put inside context declaration.

```scala
context(option) {
  $(1 + divide(n, m))
}
```

scala-workflow is shipped with a library of workflow instances for commonly used classes, including that of type Workflow[Option]. Moreover, most of the instances are defined with the implicit keyword. The following snippet summons option from the implicit scope and is equivalent to the previous one.

```scala
context[Option] {
  $(1 + divide(n, m))
}
```

In some cases the whole body of context is directly a block of code put in the workflow brackets. Alias macro workflow is defined specifically for that. Similarly to context it takes either the workflow instance object or the type constructor reference. Once again, the following snippet is equivalent to the previous ones.

```scala
workflow[Option] {
  1 + divide(n, m)
}
```

The choice between the context and the workflow is entirely up to the user. Examples in Section 4 feature both declarations.

### 3.3 Expression rewriting

Translation of the expression inside the $ macro into calls of workflow methods is done in two stages. First, the bindings elimination algorithm replaces all effectful subexpressions inside the expression with synthetic unique identifiers, building the graph of dependencies between subexpressions along the way. Then the rewriting algorithm generates appropriate workflow method calls based on the number and connection between collected effectful bindings.

The bindings elimination algorithm works with a given expression in a given workflow context. It takes the untyped syntax tree of the expression (it might not necessarily type check in regular Scala) and produces a graph of variable bindings together with a type-correct expression modulo the absence of effects. Steps of the algorithm:

1. Traverse nodes of the syntax tree in post-order (subnodes are visited before the root). The initial scope of effectful bindings is empty.
2. While visiting the node, check its type in the lexical scope of bindings. If the type corresponds to the type of the effect in the given workflow context (that is, it is F[A] for some A in the context Workflow[F[_]]), generate a unique identifier, save the binding between identifier and the node and replace the node with the identifier reference. Otherwise, leave the node as it is.
3. Traverse the outermost nodes until the top level of the expression is reached.
As a working example consider a computation in the `option` workflow, that, as discussed previously, can be expressed with methods `map` and `app`.

```scala
workflow(option) {
  2 * divide(1, 3) + 4 * divide(5, 6)
}
```

The binding elimination algorithm starts with the leaves, numbers in this case. All of them are of type `Int`, so it continues. Type checking `divide(1, 3)` returns the type `Option[Double]`, that corresponds to the effectful type constructor `Option`. The newly created identifier `x$1` has the type `Double` and points to the expression `divide(1, 2)`. The rewritten expression `2 * x$1` has the type `Double`. In the same way, `divide(5, 6)` yields the new binding `x$2` of the type `Double`. The rewritten expression `4 * x$2` has the type `Double` and so does the whole expression altogether. The result of the algorithm is a partially rewritten expression and a table of bindings.

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>x$1</td>
<td>double</td>
<td><code>divide(1, 3)</code></td>
</tr>
<tr>
<td>x$2</td>
<td>double</td>
<td><code>divide(5, 6)</code></td>
</tr>
</tbody>
</table>

The synthetic variables are independent: `x$1` does not appear in the body of `x$2` and `x$2` does not appear in the body of `x$1`. The rewriting algorithm then produces a combination of methods appropriate to express the computation, `app` and `map` in this case.

```scala
option.app(
  option.map(
    (x$1: Int) ⇒ (x$2: Int) ⇒
    2 * x$1 + 4 * x$2
    )(divide(5, 6))
  )(divide(1, 3))
)
```

In the second example consider a snippet of code in the `option` workflow that involves the result of one effectful computation that is needed for another one. It requires `option` to support `bind` and `map` to rewrite this expression.

```scala
workflow(option) {
  2 * divide(divide(3, 4), 5)
}
```

The binding elimination algorithm type checks the numbers first, then generates the new identifier `x$1` of type `Double` for the node `divide(3, 4)`. Type checking the expression `divide(x$1, 5)` returns `Option[Double]`, which produces another binding `x$2`. Eventually, the whole partially rewritten expression type checks to `Double`.

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>x$1</td>
<td>Double</td>
<td><code>divide(2, 3)</code></td>
</tr>
<tr>
<td>x$2</td>
<td>Double</td>
<td><code>divide(x$1, 4)</code></td>
</tr>
</tbody>
</table>

Note that this time `x$2` depends on `x$1`, in order to evaluate `x$2` we have to evaluate `x$1` first. The notion of dependency between effectful expressions requires the bind operator and the rewriting algorithm produces a combination of bind and `map`.

```scala
val x$1: Double <- divide(3, 4)
x$2: Double <- divide(x$1, 5)
```

The error message presents the expression that failed to type check, all the bindings, their types and the expressions they point to.

```scala
error: type mismatch;
found : Double
"2".times(x$2)
where
x$1: Double <- divide(3, 4)
x$2: Double <- divide(x$1, 5)
```

3.4 Composition of workflows

It is usually the case that type constructor `f[_]` encodes a single effect produced by a computation. Multiple effects can be expressed with composition of effectful type constructors. At the same time, expressing composition of workflows might not be an easy task.

Syntactically, `scala-workflow` provides the binary operator `$` as a uniform notation for composition of workflows. That is, if `f` is `Workflow[f[_]]` and `g` is `Workflow[g[_]]`, `f $ g` is `Workflow[f[G[_.]]]`.

However, composition of workflows requires knowledge about their structure, therefore the `$` operator is defined separately for every computation type rather than generally in the `Workflow[_.]` trait.

Composition of functors is separated from the interface of a functor and defined in the `FunctorComposition` trait, that is mixed to the `Functor` trait.

3Hereinafter, the notation for composition of type constructors is deliberately shortened for typographical reasons. The correct notation would use `type level λ-function` and in this case would be `-type λ[α] = Workflow[F[G[_.]]]`.

```scala
workflow(option) {
  "2" * divide(divide(3, 4), 5)
}
```
trait FunctorComposition[F[_]] { f: Functor[F] ⇒
  def $ [G[_[_]](g: Functor[G])] =
    new Functor[F[G[_[_]]]] {
      def map[A, B](h: A ⇒ B) = f map (g map h)
    }
  def & [G[_[_]](g: Functor[G])] = g $ this
}

Method $ behaves as $ with the arguments flipped. For any functors f and g, $ f $ g is equivalent to g $ f.

Traits SemiIdiomComposition and IdiomComposition are defined in the same way. The fact that functors, semi-idioms and idioms form a hierarchy has a pleasant side effect here. It is possible to compose workflows of different classes; the resulting workflow will implement the weaker class of the two. The dynamic dispatch mechanism in Scala guarantees to choose the method from the right trait. For example, if f is Functor[F[_[_]]] and g is Idiom[G[_[_]]], f $ g becomes Functor[F[G[_[_]]]], since functors are weaker than idioms.

While it is possible to provide a generic implementation for composition of pairs of functors, semi-idioms and idioms, no such implementation exists for pairs of arbitrary semi-monads and monads. A common strategy in the latter case employs monad transformers [13]. To maintain a uniform language of expressions, that abstracts the variable environment and the propagation of failure of environment lookup.

4. USE CASES

We argue that scala-workflow can assist in a number of applications. In this section we present a series of case studies, employing features of the extension.

4.1 Boilerplate elimination

Following the example given in [15] for idioms brackets, we present an enhanced implementation of an evaluator for a language of expressions, that abstracts the variable environment and the propagation of failure of environment lookup.

The definition of the abstract syntax of expressions with variables and addition is given as an algebraic data type.

sealed trait Expr
case class Var(id: String) extends Expr
case class Val(value: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr

Variables are fetched from the environment of type Env. Function fetch looks up a variable in the environment and either returns an integer or fails, which is encoded as a value of type Option[Int].

type Env = Map[String, Int]
def fetch(x: String)(env: Env) = env.get(x)

The evaluator of the expression in the given environment is simply a function of type Expr ⇒ Env ⇒ Option[Int].

def eval(expr: Expr)(env: Env): Option[Int] =
  expr match {
    case Var(x) ⇒ fetch(x)(env)
    case Val(n) ⇒ Some(n)
    case Add(x, y) ⇒ for {
      lhs ← eval(x)(env)
      rhs ← eval(y)(env)
    } yield lhs + rhs
  }

This implementation can be improved. Note, that one has to explicitly pass the environment around and to have rather awkward syntax to propagate failure in the Add case. Both concerns can be addressed by encapsulation the boilerplate in an appropriate workflow. Here function[A] represents a workflow of a function with a fixed argument of type A, equivalent to Haskell’s (a ->) monad. We compose function[Env] and option to build a workflow for type constructor Env ⇒ Option[_.].
The body of the function can now be put inside context declaration.

def eval: Expr ⇒ Env ⇒ Option[Int] =
  context(function[Env] $ option) {
    case Var(x) ⇒ fetch(x)
    case Val(n) ⇒ $(n)
    case Add(x, y) ⇒ $(eval(x) + eval(y))
  }

This new notation enables separation of concerns between evaluation strategy, environment handling and failure propagation, as the latter two are hidden in the implementation of corresponding workflow objects. This effectively reduces the amount of boilerplate code in this example.

### 4.2 Domain-specific languages

scala-workflow can be used as a syntactic enhancement for embedded domain-specific languages. In this example, it assists a simple framework for demand-driven functional reactive programming [7].

The Cell trait defines a unit of data, that can be assigned with := and fetched with !. Cells can depend on each others values, much like they do in spreadsheets.

trait Cell[T] {
  def ! : T
  def := (value: T)
}

Cell has an instance of Idiom, such that point creates a source cell, that stores a value and allows to change it and app creates an observer cell, that takes the value of some other cell to compute its own. Reassigning observer cells does not make sense, hence the exception.

val frp = new Idiom[Cell] {
  def point[A](a: ⇒ A) = new Cell[A] {
    private var value = a
    def := (a: A) { value = a }
    def ! = value
  }
  def app[A, B](f: Cell[A] ⇒ B) =
    a ⇒ new Cell[B] {
      def ! = f(a!)
      def := (value: T) {
        throw new UnsupportedOperationException
      }
    }
}

A value, independent of values of any cells, put inside $, will be translated to point and effectively act as a source cell, whereas an expression that combines values of cells becomes an observer cell.

context(frp) {
  val a = $(10)
  val b = $(5)
  val c = $(a + b * 2)
  println(c!)
  b := 7
  println(c!)
}

Dereferencing a cell makes it reevaluate the whole tree of dependent cells. The snippet above will print 20 and 24.

### 4.3 Transparent exception handling

Consider an example of a program that parses XML document fetched from the web. Each step of the program can end in an error. The provided address might not be a valid URL, we might not be able to fetch the page over the network or the contents of the page might not be correct XML. One way to implement these failing functions is through the use of exceptions, that guarantee that the execution will not go further than the point of the error.

```scala
def fetchXML(address: String): XML = {
  val url = URL.fromString(address)
  val page = Page.fetch(url)
  val contents = page.getContents
  XML.fromString(contents)
}
```

Unlike Java, Scala does not support checked exceptions, therefore the compiler is unable to distinguish between functions that do and do not throw exceptions (in other words, effectful and pure). Following the spirit of statically typed functional programming, one might want to promote the effect of the exception to the type level. For that reason Scala 2.10 offers the scala.util.Try monad, that stores either the result of a computation or a Throwable object that represents a failure. Achieving the lofty goal of making exception effects explicit, however, stumbles on the syntactic awkwardness of monadic exception handling.

In the following snippet, effectful methods URL.fromString, Page.fetch and XML.fromString are assumed to return values wrapped in Try instead of throwing an exception.

```scala
def fetchXML(address: String): Try[XML] = {
  for {
    url ← URL.fromString(address)
    page ← Page.fetch(url)
    contents ← page.getContents
    xml ← XML.fromString(contents)
  } yield xml
}
```

It takes more that just a syntactic transformation to handle Try. One has to inspect the types of val expressions to separate effectful binds from pure values assignments and insert either ← or = operators. Moreover, one has to excessively name the intermediate results of the computation. Putting the code in the workflow[Try] context makes the code syntactically equivalent to its exception-based version.

```scala
def fetchXML(address: String): Try[XML] = workflow[Try] {
  val url = URL.fromString(address)
  val page = Page.fetch(url)
  val contents = page.getContents
  XML.fromString(contents)
}
```

The workflow definition workflow[Try] syntactically resembles try keyword. One can further apply Try's recover method to mirror the whole try/catch expression. The result of the function, however, would still be wrapped in Try.

### 4.4 Imperative functional programming

Programs in functional languages can gain clarity with imperative syntax using scala-workflow. This example shows how computations, interspersed with logging can be expressed in a purely functional manner and at the same
time have the same syntactical structure as imperative core with log writes as side effects.

We represent logging as a workflow for type constructor (\_\_, List[\text{String}]), where the first argument of the pair is the result of a computation and the second one is a list of log messages. A built-in implementation of such workflow is accumulator\[M\]. It is similar to Haskell’s writer monad.

```
val logging = accumulator[List[String]]
```

Every function that intends to write to a log inside a logging workflow should produce a pair of the result value and a list of messages, generated during its computation.

```
def mult(x: Int, y: Int) = (x * y, List("Calculating $x * $y"))
```

A function that does not perform any computation and simply writes to the log has the type (Unit, List[\text{String}]).

```
def info(message: String) = (Unit, List(message))
```

Now, wrapping the code in a logging workflow allows to intersperse pure and effectful functions.

```
val (result, log) = workflow(logging) {
    info("Assigning x to 2")
    val x = 2
    info("We are about to multiply by 3")
    val triple = mult(3, x)
    info("We are about to calculate something complex")
    val cube = mult(mult(x, x), x)
    val sum = cube + triple
    info("Actually, only half of it was needed")
    sum / 2
}
```

The outcome of the evaluation is a pair of the result value 7 and the accumulated list of log messages.

```
List("Assigning x to 2",
    "We are about to multiply by 3",
    "Calculating 3 * 2",
    "We are about to calculate something complex",
    "Calculating 2 * 2",
    "Calculating 4 * 2",
    "Actually, only half of it was needed")
```

Note, that we got imperative-looking code implemented in a purely functional subset of the language. No mutable state or side effects were involved in this example.

5. FUTURE WORK

There is a number of directions, where \text{scala-workflow} can be further improved. They are left for future work.

Up-to-date implementation. As it was mentioned before, the current implementation of \text{scala-workflow} employs untyped macros. Their distinctive feature is that the macro expansion is performed before type checking, which is crucial for the type-driven algorithm. Unfortunately for us, untyped macros were discontinued\(^{4}\) and are no longer supported in Macro Paradise. A sensible replacement should be \textit{macro annotations}, that are also expanded before type checking. However, being on the cutting edge of Macro Paradise development, they are not mature enough to support \text{scala-workflow} as of now. Specifically, there are problems with visibility of declarations from the current scope of the macro, that present an obstacle for the implementation. They are expected to be solved in the future.\(^{5}\)

An annotation-based implementation will replace context and \text{workflow} macros with synonymous annotation macros.

```
@workflow[List] val x = List(1, 2) * List(4, 5)
```

The syntax of composition of workflows will remain the same, since a workflow object could be passed explicitly as an annotation argument.

```
@context(function[Env] $ option) def eval: Expr ⇒ Env ⇒ Option[Int] = {
    case Var(x) ⇒ fetch(x)
    case Val(n) ⇒ $n
    case Add(x, y) ⇒ $(eval(x) + eval(y))
}
```

Wider subset of Scala. The current implementation is rather conservative with respect to Scala features it supports. This is a deliberate choice, however in the future we hope to support a wider subset of Scala features, such as loops, exceptions and lazy evaluations. A possible implementation would include adding more basic workflow traits with corresponding rewriting rules, similarly to F#.

Context inference. \text{scala-workflow} generally reduces the amount of boilerplate code needed to support effectful computations. It can be reduced even more in the case of simple one-line expressions. For example, it could have been inferred that the expression List(1, 2, 3) + 4 should be evaluated in the \text{workflow}[List] context and avoid wrapping $\{List(1, 2, 3) * 4\}$ in the context declaration explicitly. However, a few problems need to be solved.

It is generally not clear which type constructor should be chosen in case it is a subtype of another type constructor. For example, the expression \text{Some}(42) would be expected to be evaluated in the \text{Workflow}[\text{Option}], yet it has the type \text{Some}[\text{Int}], which is a subtype of \text{Option}[\text{Int}].

Moreover, when the type of the expression is built from a type constructor applied to more than one argument it might be ambiguous, which of them should be fixed. For example, an expression of type Either[\text{A}, \text{B}] could be evaluated both in the workflow of Either[\text{A}, \text{B}] and Either[\text{B}, \text{A}].

Workflow inference. As noted before, Scala's standard library does not explicitly have a Monad interface, but rather the expansion of \textit{for-expression} expects an object to have map, flatMap and filter methods. Hence, \text{scala-workflow} has to have a trivial workflow implementation for every class with a monadic interface that follows this convention, which affects extensibility of the library. An alternative approach could be to automatically infer a SemiMonad instance in case the map and flatMap methods are present in the class.

Other formalisations of effects. Despite the focus on the abstract methods of the Workflow trait rather than specific algebraic structures, \text{scala-workflow} is still heavily influenced by the famous functor/idiom/monad hierarchy of computation types. It would be interesting to see if our approach to syntactic extensions could be adapted to other formalisations of effects, particularly to comonads\(^{25}\) and code-notation\(^{25}\), and also if both formalisations can be unified in one extension.

\(^{4}\)http://scalamacros.org/news/2013/08/05/macro-paradise-2.0.0-snapshot.html

6. CONCLUSIONS
The main contribution of this work is scala-workflow, a macro-based syntactic extension for Scala.

The choice of an abstract workflow interface instead of predefined interfaces for computation types enables a uniform syntax for all computation types that can be expressed with methods available for extension. It also leaves space for further extensions, that can facilitate support of Scala features, not available with the current implementation.

The type-directed expression rewriting algorithm allows users to transparently blend effectful and pure functions in a bounded context. This offers a trade-off between an explicit structure of effects and syntactic flexibility. We argue that the latter can provide better separation of concerns, because the program maintains its logical structure, while effect handling ends up being hidden inside a workflow instance implementation.

The most similar approach was taken in F#'s computation expressions. The key distinction of scala-workflow is that it does not need an extended set of keywords for marking effectful computations. Moreover, it is implemented not as a compiler extension, but rather as a macro-based library that operates with syntax trees and the type checker API.

scala-workflow primarily aims to enhance the syntax of effectful computations. At the same, a type constructor in the given workflow implementation is not required to semantically capture computational effects, it can simply represent a generic data structure. This makes scala-workflow suitable for a wide range of monadic and applicative domain-specific languages.

7. ACKNOWLEDGMENTS
This work was partially supported by the Swedish VR grant D0497701. The author would like to thank Philipp Haller and anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper.

8. REFERENCES