Inverse Macro in Scala

Hiroshi Yamaguchi
The University of Tokyo, Japan
hiroshi_yamaguchi@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo, Japan
chiba@acm.org

Abstract

We propose a new variant of typed syntactic macro systems named *inverse macro*, which improves the expressiveness of macro systems. The inverse macro system enables to implement operators with complex side-effects, such as lazy operators and delimited continuation operators, which are beyond the power of existing macro systems. We have implemented the inverse macro system as an extension to Scala 2.11. We also show the expressiveness of the inverse macro system by comparing two versions of shift/reset, bundled in Scala 2.11 and implemented with the inverse macro system.

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

General Terms Design, Languages

Keywords Domain Specific Language, Syntactic Macro, Type, Continuation

1. Introduction

A modern programming language enables a library with language-like programming interface, which is often called an embedded domain specific language (DSL) or a library-level DSL. While this approach has an advantage that embedded DSLs are easier to develop than traditional DSLs, which require an independent compiler and a custom development environment, it also has a serious disadvantage that the power of expression of embedded DSLs is limited. In order to overcome this disadvantage, syntactic macro systems are one of the major solutions, which intercepts a compiling process to rewrite and replace parts of the compiling program. Lisp macros [5, 17, 23], Nemerle [37], and Template Haskell [35] are famous examples. Scala 2.11 also has a syntactic macro system [3], which is integrated into the type system.

However, existing macro systems still have powerful and side-effectful operators difficult to implement. A lazy operator is a typical example, which affects two or more execution points along a def-use relation. In general, classical macro systems cannot implement an operator affecting surrounding code snippets considering a control flow or a data flow. Due to this lack of ability, such operators cannot implemented as an embedded DSL even if macro systems are available. To implement lazy operators, DSL developers have to largely modify the host language compiler.

This paper proposes a new variant of typed syntactic macro systems named *an inverse macro system* for implementing more powerful and side-effectful operators. The inverse macro system has two unique features. First, while existing macros affect only local code, which consists of the macro name and the macro arguments, the inverse macros can affect not only the local code but also the surrounding code. More precisely, the inverse macros capture the code for the continuation sequence of the macro call. This feature enables to affect other execution points. Second, an inverse macro is an annotation for types. An expression typed as a type containing an inverse macro annotation is a rewritten target. This feature enables to propagate rewriting along the control flow. Due to these features, the inverse macro system enables global rewriting beyond the local scope, which is required by the implementations of several operators such as a lazy operator.

Delimited continuation operator is another operator implementable with the inverse macro system. This paper presents the implementation of shift/reset [1, 6], which is one of the typical delimited continuation operators. The inverse macro system can be used for implementing not only operators but embedded DSLs such as direct styles, which can represent functors, such as monads, applicative functors, and fork/joins, in a simple style. Although monads can be implemented with a delimited continuation operator, they can be directly implemented with an inverse macro; no intermediate operator like delimited continuation is necessary. The implementation with an inverse macro simplifies typing and is more efficient than the indirect implementation with delimited continuation.

Finally, this paper presents our implementation on top of Scala 2.11 with compiler plugin and typed syntactic macro system features. We also show the results of expressiveness and performance measurement using delimited continuation operator *shift/reset*. We compare our implantation with the implementation of shift/reset bundled with Scala.

2. Motivating Example: Lazy Operator

To show that existing macro systems have limited expressiveness, suppose that we implement a lazy operator by using a macro. A lazy operator can delay the evaluation of the argument until its value is actually consumed. Fig. 1 shows an example written in Scala 2.11. The lazy operator lzy is used in the line 2. The argument some_calc() is not evaluated here, and just wrapped up in a thunk. Since the delayed result is bound to the variable delayed and used in line 5, the delayed thunk is finally evaluated there. As similar language constructs, Scala provides lazy variables and implicit conversion as built-in constructs. A call-by-name parameter provided by Scala is also called a "lazy" construct, but it is completely different from the lazy operator here.

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3687-1/15/10...
http://dx.doi.org/10.1145/2814204.2814213

```
1 // delay some calculation
2 val delayed = lzy { some_calc() }
3   :
4 // evaluated below
5 println(delayed)
```

Figure 1. An example of a lazy operator

```
1 // -- without lazy --
2 val http = new HTTP()
3 val x = fork { http.connect(url).get() } // fork
4 default.draw() // draw the default page in parallel
5 join(x).draw() // explicitly join
6 // -- with lazy --
7 def par[T](body: => T) = {
8   val f = fork(body)
9   lzy { join(f) }
10 }
11 val x = par { http.connect(url).get() } // fork
12 default.draw() // draw in parallel
13 x.draw() // implicitly join
```

Figure 2. Parallel execution with a lazy operator

```
1 val delayed = () => { some_calc() } // definition
2   :
3 // evaluated below
4 println(delayed.apply())            // use
```

Figure 3. The lazy operator after macro expansion

There are a number of examples of the use of a lazy operator. One example is shown in Fig. 2. Line 2 to 5 shows the code without a lazy operator while line 7 to 13 shows the code using a lazy operator. Line 3 forks a subtask for fetching a web page in parallel through an HTTP connection. The thread of control of this subtask joins to the main thread in line 5 and the fetched web page is drawn. Note that the main thread explicitly waits by join until the subtask completes. On the other hand, if a lazy operator is available, we can hide the necessity of join for synchronizing the threads. The par operator defined in line 7 to 10 hides this. If a subtask is forked by par, the programmer does not have to be aware of synchronization. In line 13, draw is directly invoked on x. join is not explicitly invoked; the synchronization is automatic.

Existing macro systems cannot implement this lazy operator because the implementation requires global rewriting. A lazy operator affects two kinds of execution points, one where a lazily-evaluated expression is constructed, and one where the expression is evaluated for consumption. The two execution points are related by the def-use relation on the value that the expression results in. At the former point, the lazy operator lzy is placed. It constructs a thunk object wrapping the expression as its argument. At the latter point, the thunk object is referred to to obtain the resulting value. The latter point will be located near the former one but they are different. A macro of existing systems cannot rewrite the code at the latter point. It can only *locally* rewrite the code at the former point. This code consists of only the macro name and its arguments. For example, Fig. 1 should be macro-expanded into Fig. 3. The expansion in line 2 is straightforward. The macro name lzy and its argument { ... } are locally replaced with a new expression. However, the macro cannot replace the code in line 5 so that apply should be called on delayed. Line 5 is far from line 2. It is outside the macro call to lzy.

```
1 lzy_dsl {
2   val delayed = lzy { some_calc() }
3     :
4   println(delayed)
5 }
```

Figure 4. A workaround for existing macro systems

A workaround for implementing a lazy operator by using an existing macro system is to combine two macros. See Fig. 4. Here, we introduce a new macro lzy_dsl as well as lzy. The argument to lzy_dsl is the whole code in line 2 to 5. It contains a macro call to lzy in line 3. Since the two execution points that the lazy operator affects are included in the macro argument to lzy_dsl, the code at the two execution points can be replaced during the expansion of the lzy_dsl macro. The occurrence of lzy is only used as the indicator of the execution point. Note that the two execution points are local ones from the viewpoint of lzy_dsl. However, this workaround has two problems. First, lzy_dsl is verbose and the appearance of the code is not desirable. Furthermore, both lzy_dsl and lzy must be placed in the same method body. This workaround cannot deal with the case like Fig. 2 since the lzy operator is in the body of par but the lazy expression is evaluated outside the body of par.

3. Proposal: Inverse Macro

To address the problem mentioned in the previous section, we propose a new language construct named an *inverse* macro, which is implemented on top of Scala 2.11. The inverse macro system has two unique features. First, to enable global rewriting, the inverse macro system captures the *continuation* sequence of a macro call. In other words, it captures the syntax tree representing the code sequence following a macro call. For example, in Fig. 1, if lzy { some_calc() } is a macro call, its continuation sequence is the syntax tree representing the following lines including println(delayed), where some_calc() is lazily evaluated. Hence the inverse macro can traverse this syntax tree and rewrite delayed in line 5 into delayed.apply(). Note that the inverse macro also captures the syntax tree of the macro call lzy { some_calc() }, which consists of the macro name and the macro arguments, as existing macro systems do. Furthermore, an inverse macro is represented by a type annotation. In Fig. 1, if the return type of the lzy method is set to T@lzyAnn and the annotation lzyAnn is an inverse macro, lzy is interpreted as a call to the inverse-macro lzyAnn. The block { some_calc() } following lzy is a macro argument. Like the existing macro system in Scala, an expression interpreted as an inverse-macro call has to be a syntactically-valid expression in Scala and it also has to be well typed before macro expansion.

3.1 Expansion Flow

The work flow for expanding an inverse macro consists of seven stages: enumeration, detection, normalization, capture, invocation, recursion, and splice. Fig. 5 shows an overview of the work flow. The example program reads two integers from different URLs in parallel and prints the sum of those integers. lzy looks like a lazy operator but it is an inverse-macro call. fork starts a subtask running in parallel and returns a future object. The resulting value of the subtask is obtained by get on the future object.

Enumeration First, the macro system decomposes the syntax tree of a given block into the direct sub-trees of the root, each of which corresponds to an expression. They are enumerated in the execution order. Here, a block is not only an ordinary block in Scala but also a method body, a by-name argument, or a block of a control expression such as the then clause and the else clause of an if expression,

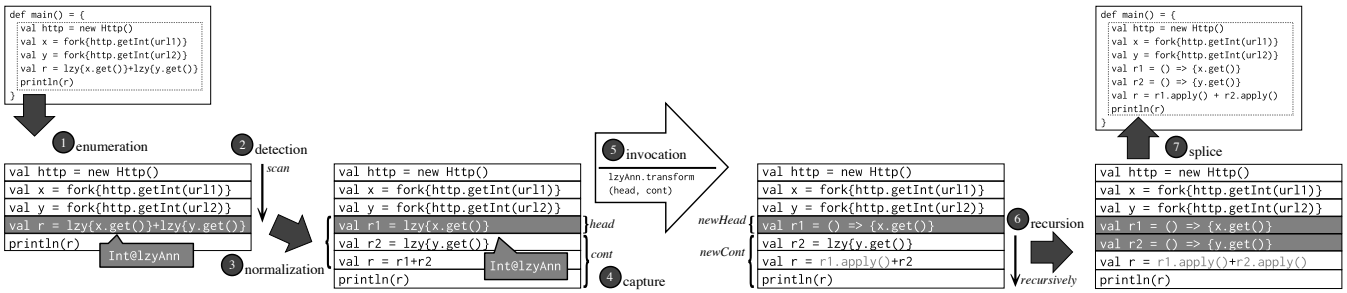

Figure 5. Work flow for macro expansion

```
1 lzy{ x.get() } + lzy{ y.get() }
```

```
1 val r1 = lzy{ x.get() }
2 val r2 = lzy{ y.get() }
3 r1 + r2
```

Figure 6. A-normalization

```
1 if (cond) {
2   val delayed = lzy (...): Int@lzyAnn
3   println(delayed) // included
4 }
5 println("outside") // not included
```

Figure 7. Capturing the *cont* in an if expression

and the case bodies of a match expression. In the figure, the body of main is a block and the system decomposes it. Its sub-trees are val http = new Http(), val x = fork{http.getInt(url1)}, and so on.

Detection Second, the system detects an inverse macro in the enumerated sub-trees. It computes the type of each sub-tree and determines whether it is an inverse-macro call. If an inverse-macro call is found, the system proceeds to the next step. In the figure, the system examines each sub-tree in the execution order. It first examines val http = new Http() and then val r = lzy{x.get()} + lzy{y.get()}. The call to lzy is typed as Int@lzyAnn. Since the annotation lzyAnn is an inverse macro, the system expands the call in the following steps. Note that a block following a method name is a method-call argument in Scala. It is called a *by-name* argument.

Normalization Before expanding a macro call, the system normalizes the sub-tree containing a macro call. In the figure, val r = lzy{x.get()} + lzy{y.get()} is normalized. To simplify, the system considers only the return type of the right side when determining whether a variable declaration is expanded or not. Under this strategy, without normalizing, val r = lzy{x.get()} + lzy{y.get()} is not expanded since the return type of the right side is inferred to Int and contains no inverse macro. There are two kinds of normalization.

For complex method calls, A-normalizing [14] is applied. For example, the system normalizes an expression lzy{x.get()} + lzy{y.get()} shown in the upper part of Fig. 6 so that the resulting values of the sub-expressions will be stored in temporary variables r1 and r2. The code after the normalization is shown in the lower part of Fig. 6. The variables are substituted for the corresponding occurrence of the sub-expression.

For constructs taking blocks, the system transforms each construct in an ad-hoc way. Such constructs are affected by child blocks whether their result types contain inverse macros or not. The system recursively transforms the blocks and then, if the return types of the blocks contain an inverse macro, the construct is substituted by an expression considering the inverse macro. For a simpler expression, such as an if, a match, and a try expression, the system recursively transforms the blocks.

However, our prototype of the inverse macro system can not normalize return expressions, while expressions, or finally clauses well. When the return type of a child block contains an inverse macro, the system fails to normalize. Normalizing such constructs is our future work.

Capture Then the macro system captures two sub-trees, *head* and *cont*. The *head* represents the inverse macro call (and the assignment of the result value). In Fig. 5, the *head* is the tree representing val r1 = lzy{x.get()}. The *cont* represents the continuation sequence of the head. In Fig. 5, the *cont* is the tree representing val r2 = lzy{y.get()} and the following lines, val r = r1 + r2 and println(r).

The captured *cont* is delimited within the current block. For example, if an inverse-macro call is in a method body, the *cont* does not include the syntax tree representing the outside of the method body. If a macro call is in the else clause of an if expression, the *cont* includes only the syntax tree representing (part of) that clause. In Fig. 7, the inverse macro @lzyAnn can capture only println(delayed); line 1 and 5 are not included. Although this restriction improves the locality of macro expansion, it prohibits inter-procedural rewriting during macro expansion. A technique for doing inter-procedural rewriting for an inverse macro is mentioned later in the applications section.

Invocation After capturing *head* and *cont*, the macro system invokes the transformation method for the macro. *head* and *cont* are passed as the arguments. The transformation method returns modified *head* and *cont*, which will substitute for the originals. We call them *newHead* and *newCont*, respectively.

The transformation method can traverse and transform the trees given as *head* and *cont*. For example, it can identify a place where an expression specified by a lazy operator is evaluated and then it can transform the code at the place into appropriate code.

Recursion *newCont* returned by the transformation method is recursively processed by the inverse-macro system whereas *newHead* is not. A macro call included in *newHead* is not expanded. The aim of this design is to avoid an infinite expansion loop and an expansion unexpected by the programmer. Therefore, in Fig. 5, newHead, which is val r1 = () => {x.get()}, is not expanded any more. On the other hand, newCont is recursively transformed. Hence a macro call to lzy in val r2 = lzy{y.get()} is expanded.

By default, the recursive application of macro expansion to the continuation sequence is performed in this stage after the Invocation stage. However, it is sometime desirable to apply the recursive

```
1 def lzy[A](body: => A): A@lzyAnn = ??? // dummy
2 class lzyAnn extends IMAnnotation
3 object lzyAnn extends IMTransformer {
4  def transform(...)(head: Tree, cont: List[Tree]) =
5  { head match {
6   case ValDef(mods, name, tpt, Apply(_, body)) =>
7    // produce an intermediate object
8    val newHead = typecheck(
9     q"val␣$name␣:␣(()␣=>␣$tpt)␣=␣()␣=>␣$body")
10   // traverse and replace
11   val newCont = for (t <- cont) yield
12    replace(t, head.symbol,
13     q"${head.symbol}.apply()")
14   List(newHead) -> newCont
15  }}}
```

Figure 8. Implementation of a lazy operator

macro expansion to the continuation sequence *cont*, not *newCont*, in the Invocation stage. The inverse-macro system allows programmers to do this if needed.

Splice Finally, the macro system substitutes the syntax trees *newHead* and *newCont* for the original trees *head* and *cont* in the compiled program, respectively.

3.2 Example: Implementing a Lazy Operator

We describe how to implement an inverse macro for a lazy operator. Note that this implementation does not support inter-procedural macro expansion and hence it does not work for the example in Section 2. The complete implementation is shown in Section 5.

The pseudo code of the implementation is shown in Fig. 8. In line 1, the lazy operator is implemented as a method whose return type is A@lzyAnn. Since its method body is never invoked, it is defined as ???. This indicates in Scala that the body is not implemented. lzyAnn is an annotation. An annotation extending IMAnnotation is considered as an inverse macro. The corresponding transformation method is defined in the companion object, which is a singleton object with the same name as the class. That corresponds to a static method in Java.

The transformation method transform takes two arguments head and cont as well as some other arguments. In Fig. 5, head is the syntax tree of val r1 = lzy{x.get()}. head is already normalized before being passed as already mentioned. cont is the syntax tree of val r2 = lzy{y.get()} and the following lines. It is not guaranteed that cont is normalized before being passed unlike head. The transform method returns a tuple of newHead and newCont. They are modified head and cont, respectively.

In the body of the transform method, the tree transformation API of the existing Scala macro system is available. For example, a tree-node class Tree and its subclasses such as ValDef and Apply are available. Pattern matching by match { case ... => ... } is also available. The API provides quasi quotes, which is used in line 9 and 12 in Fig. 8. It also provides the typecheck method for typing. It is used in line 8.

The transform method for the lazy operator performs as follows. First, it replaces a call to lzy with an expression producing a thunk object wrapping the argument to lzy, for example, {x.get()}. Then the method examines cont and replaces all occurrences of the variable bound to the result of lzy included in head. Those occurrences are replaced with a call to apply on the variable so that the wrapped thunk will be invoked. The transform method finally returns newHead and newCont.

3.3 Typing with Inverse Macros

The inverse macro system exploits type annotations. In Scala, a type t with an annotation is identical to the type t without an annotation. However, the inverse macro system changes this typing rule; it distinguishes a type with an annotation from one without an annotation.

```
1 def withLzy(body: => Int@lzyAnn)
2 def withoutLzy(body: => Int)
3
4 // valid cases
5 withLzy { lazied(): Int@lzyAnn }
6 withLzy { raw(): Int }                  // add
7 def _1(): Int@lzyAnn = { lazied(): Int@lzyAnn }
8 def _2(): Int@lzyAnn = { raw(): Int } // add
9
10 // invalid cases
11 withoutLzy { lazied(): Int@lzyAnn }      // omit
12 def _3(): Int = { lazied(): Int@lzyAnn } // omit
```

Figure 9. The valid/invalid examples for Completion

```
1 def by_value(x: Int)
2 by_value (lazied(): Int@lzyAnn)    // coerced
3 val x : Int = lazied(): Int@lzyAnn // coerced
```

Figure 10. The valid examples for Coercion

Completion Let t be a type. t without an inverse macro is compatible with t with an inverse macro. However, the opposite is not true. t is a subtype of t with an inverse macro. For example, Int is compatible with Int@lzyAnn whereas Int@lzyAnn is not compatible with Int. Without this rule, an annotation designating macro expansion might be accidentally lost.

Fig. 9 shows an example. The withLzy method in line 1 takes a by-name parameter. The return type of the by-name parameter is Int@lzyAnn, an Int type annotated with an inverse macro lzyAnn. The withoutLzy method in line 2 also takes a by-name parameter but its return type is Int, a type without an inverse macro. It is valid to pass to withLzy not only a block with a return type Int@lzyAnn but also a block with a return type Int (line 5, 6). It is valid to call a method returning a value of type Int when computing a return value of type Int@lzyAnn (line 8). However, it is not valid to pass to withoutLzy a block with a return type Int (line 11). A value of type Int@lzyAnn cannot be a return value for a function _3 if the return type of _3 is Int (line 12).

Coercion At the position evaluated by the call-by-value strategy, a type annotated with an inverse macro is coerced into the type without an inverse macro. As shown in Fig. 10, if a method by_value takes a by-value parameter x of type Int (line 1), it is valid to pass to by_value a value of type Int@lzyAnn (line 2). The value is coerced. This is also true for other eagerly-evaluated positions such as computing an initial value of a variable (line 3). The value returned by the lazied method is coerced since the type of x is not Int@Lzy but Int.

Merge A conditional branch, such as if, match, and try expressions, requires to merge multiple types with an inverse macro. First, each type is decomposed into a raw type and an annotation type. For example, Int@lzyAnn is decomposed into Int and lzyAnn. Then each part is merged; the minimal common supertype of the types is computed according to the standard typing rule of Scala. After being merged, the two parts are combined to construct the merged type. If some types are not annotated with an inverse macro, their annotation type is considered as Nothing, which is a subtype of all types, that is, the bottom type.

For example, as shown in Fig. 11, an if expression is typed as follows. if(x) (b: B@Ba) else (c: C@Ca) is typed to T@Ta if and

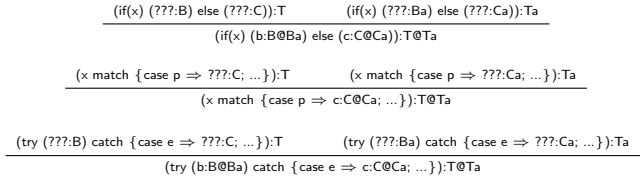

Figure 11. The rules for Merging

```
1 foo() : Int @lzyAnn @lzyAnn
2 bar() : Int @lzyAnn @lzyAnn1
3 baz() : Int @lzyAnn @lzyAnn1 @lzyAnn2
```

Figure 12. Invalid annotations

```
1 (if (cond) {
2   foo() : Int @A @B
3 } else {
4   bar() : Int @C @D
5 }) : ???
```

```
1 (if (cond) {
2   foo() : Int @A @B
3 } else {
4   bar() : Int @B @A
5 }) : ???
```

Figure 13. Corner cases

only if if(a) (???: B) else (???: C) is typed to T by the standard typer of Scala and if(x) (???: Ba) else (???: Ca) is typed to Ta by the standard typer of Scala. match and try expressions are in the same way.

3.4 Single Annotation Restriction

The current inverse macro system has one restriction, named the single annotation restriction. Any types can contain at most one inverse macro annotation. All of the examples shown in the Fig. 12 are invalid.

There are two reasons why this restriction is introduced. The first reason is to avoid an unexpected rewriting result. In practice, a consistent composition of macro expansion is difficult to program. The second reason is that it is difficult to merge two types with annotations. This problem occurs at conditional branches. For example, two corner cases are shown in Fig. 13. In the first case, which is a natural type of if, Int @A @B @C @D or Int @C @D @A @B? What is natural in the second case? This composition problem is interesting but difficult to solve. We hence currently prohibit a type with two or more inverse-macro annotations.

4. Implementation

We implemented a prototype of the inverse macro system on top of Scala 2.11. The system consists of two components: a macro and a compiler plugin, which are meta-programming features of Scala 2.11. The macro is the main component of the inverse macro system, which expands inverse macros as described in the previous section. The compiler plugin intercepts registration of method definitions and then inserts a call for inverse-macro expansion as shown in Fig. 14. This is what existing macro systems cannot do. It should

```
1  // before
2 def foo() = {...}
3 // after
4 def foo() = transform { ... }
```

Figure 14. Inserting the inverse-macro expander

```
1 // before
2 if(cond) 10 else (20: Int@lzyAnn)
3 // after adding type ascription
4 (if(cond) 10 else (20: Int@lzyAnn)): Int@lzyAnn
```

Figure 15. Type inference of if expression

be noted that the plugin was developed as an extension to Macro Paradise Plugin[1].

This implementation technique causes a few limitations. In constructors and the unapply method, an inverse macro is not expanded. A call for expansion is not inserted since assertions by the Scala compiler disturb the insertion. An inverse macro is not expanded at the right side of a field declaration since a field declaration and a local variable declaration are difficult to distinguish.

The inverse-macro expander checks and inserts type ascriptions since the inverse macro annotation does not follow Scala's default rule. Conditional branches, such as if, match, and try constructs, are typical examples. For example, the type of line 2 in Fig. 15 is inferred to Int, but the desirable result is Int@lzyAnn. Therefore, the macro expander adds a type ascription Int@lzyAnn as line 4.

However, this addition is not a perfect solution. For example, reset of shift/reset, whose signature is

```
1 def reset[A,C](body: => A@cpsParam[A,C]): C
```

is not inferred properly; C is always inferred to Nothing. This failure is difficult to recover in the macro expansion phase since the Scala compiler sometimes reports a type error and aborts the compilation in the former phase. Due to this restriction, the power of type inference by the current implementation is somewhat poor and macro users are forced to add type ascriptions manually.

To make it easy to develop an inverse macro, our macro system provides a library helping decomposition and reconstruction of abstract syntax trees. It also provides a hygiene system. As described in section 3, to provide that library, we reused an existing macro library of Scala since it already provides necessary functionality.

5. Applications

To show the expressiveness of inverse macros, this section presents three examples. They are an inter-procedural variant of lazy operator, the delimited continuation operator, and a monad in the direct style. All the three examples are side-effectful operators.

5.1 An Inter-procedural Operator

The first example is an inter-procedural lazy operator. Although the implementation shown in Fig. 8 does not perform inter-procedural rewriting, the inverse macro system enables inter-procedural rewriting. The inter-procedural lazy operator was used for showing the *fork/join* example shown in Fig. 2. In that example, the par method was implemented with an inter-procedural lazy operator.

To implement an inter-procedural lazy operator, two problems have to be addressed. One is the caller-site problem, how to prop-

[1] https://github.com/scalamacros/paradise

```
1 def lzy[A](body: => A): A@lzyAnn =  // add @lzyAnn
2   throw new Lzy(() => body)         // return
3 class Lzy[A](run: () => A)
4   extends scala.util.control.ControlThrowable {
5     def apply(): A = run()
6   }
```

```
1 val delayed: T = lzy { some_calc() } // definition
2   :
3 // evaluated below
4 println(delayed)                     // use
```

```
1 val delayed: Lzy[T] = try {          // definition
2   lzy { some_calc() }                // exceptional
3   null
4 } catch {
5   case e: Lzy[T] => e
6 }
7   :
8 // evaluated below
9 println(delayed.apply())             // use
```

Figure 16. The behavior of the inter-procedural lazy operator

agate rewriting out of the method body of par. Although the lazy operator lzy is executed within the body of par in line 9 in Fig. 2, the delayed value is evaluated in line 13, which is out of the body. The other is the callee-site problem, how to return an intermediate object representing the delayed computation from par to its caller site (line 11). A thunk object has to be returned by par and passed to line 13 through a variable x. Since Scala is a typed language, a naïve approach may cause a type error. Note that the par operator is supposed to return the value returned by join, not a thunk object.

The caller-site problem is naturally addressed by the inverse macro system since rewriting by the system is invoked according to type information. In Fig. 2, if the return type of par is typed to T@lzyAnn, rewriting is invoked at the caller side (line 11). Moreover, even if the return type is omitted as in Fig. 2, it is properly inferred by Scala's typing system.

The callee-site problem is addressed by using a runtime exception system, which is somewhat tricky, but this technique is commonly used for implementing typed continuation operators. With this technique, an intermediate object is not *returned* but *thrown* in the method body of par. It is immediately *caught* at the caller site (line 11). By this code rewriting, the type incompatibility problem is avoided. Although this technique needs only local rewriting, throwing an exception is relatively slow and hence the overall performance might be significantly degraded. This performance problem is revisited later in Section 6.

The summary of the behavior is shown in Fig. 16. The upper program defines the lzy library, the middle program shows the program before an inverse macro expansion, and the lower program shows the program after the expansion. In the callee site of lzy method, the program *throws* the thunk object Lzy instead of *returning*. In the caller site, the call of lzy method invokes an inverse macro expansion due to its return type A@lzyAnn. The inverse macro lzyAnn inserts try-catch expression to catch the thrown thunk object.

5.2 Delimited Continuation Operators

The delimited continuation operator is a language construct for capturing the *current continuation* as a closure. The current continuation means the current remaining computation, in other words, the current stack frame. Similarly to the lazy operator, the delimited continuation operator affects an execution point different from the point where the operator is invoked. The implementation of this operator captures the continuation sequence and wraps it up to be a closure. The continuation sequence is passed by the inverse macro system to the transformation method for the macro. In principle, this implementation technique is equivalent to the type-directed selective CPS transformation [28, 32].

```
1 class ControlContext[+A,-B,+C](ctl: (A=>B)=>C)
2  extends scala.util.control.ControlThrowable {
3   def map[A1](ctn1: A=>A1) =
4     ControlContext(
5      (ctn2: A1=>B)=>ctl((x: A) => ctn2(ctn1(x))))
6   def flatMap[A1,B1,C1<:B]
7   (ctn1: A=>A1@cpsParam[B1,C1]) = {
8     ControlContext((ctn2: A1=>B1) =>
9     ctl((x: A) => try {
10      ctn2(ctn1(x)).asInstanceOf[B]
11    } catch {
12      case ctx: ControlContext[A1,B1,C1] =>
13        ctx.ctl(ctn2)
14    }))}}
15 class cpsParam[-B,+C] extends IMAnnotation
16 object cpsParam extends IMTransformer {
17   def transform(...)(head: Tree, cont: List[Tree]):
18    (List[Tree], List[Tree]) = {
19     head match {
20       case ValDef(mods, name, tpt, rhs) =>
21        // wrap continuation to an inner method
22         val func = makeFunc(head, cont, api)
23         val block =
24           if (isPure(func))
25             // use map if pure
26             q"${func.symbol}␣(try␣{$rhs}␣catch␣{
27 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣case␣$ex␣=>
28 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣throw␣$ex.map(${func.symbol})␣})"
29           else
30             // use flatMap if impure
31             q"${func.symbol}␣(try␣{$rhs}␣catch␣{
32 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣case␣$ex␣=>
33 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣throw␣$ex.flatMap(${func.symbol})␣})"
34         List(func, block) -> Nil
35     }}}
36 def shift[A, B, C](fun: (A=>B)=>C): A@cpsParam[B,C]
37  = throw ControlContext(fun)
38 def reset[A, C](body: => A@cpsParam[A,C]):C =
39  try { body } catch {
40     case ctx: ControlContext[A,A,C] =>
41      ctx.ctl(identity) }}
```

Figure 17. Implementation of shift/reset

Fig. 17 shows the implementation of the shift/reset operators [1, 6]. They are typical delimited continuation operators. The shift operator is a continuation operator and the reset is a delimiting operator. First, we implement an intermediate object, which has two methods: map and flatMap. These methods are used to implement the continuation passing style. The map method is used when the continuation is pure, in other words, when the return type does not contain an inverse macro. The flatMap method is used when the continuation is impure. It is used for implementing the runtime exception technique described in section 5.1. The transformation by the inverse macro is as follows. First, the continuation is wrapped up to be an inner method. Then, a function call in the continuation passing style is constructed considering whether it is pure or impure. Finally, the transformation method returns a newly built syntax tree. The implementation of shift/reset operators also uses the runtime exception technique described in section 5.1. The intermediate object is an exception object and hence its class extends ControlThrowable. The shift operator is translated into throw while the reset operator is into try-catch.

Unlike the implementation of lazy operator, the shift/reset operators need macro expansion on their *cont* in advance since the return type of the continuation has to be known for the following

```
open(n1).foreach { f1: File =>
  :
 open(n2).foreach { f2: File =>
   :
  open(n3).foreach { f3: File =>
    :
    // deeper and deeper
...}}}
```

```
val f1 = reflect(open(n1)): File@monad[AsyncIO[File]]
 :
val f2 = reflect(open(n2)): File@monad[AsyncIO[File]]
 :
val f3 = reflect(open(n3)): File@monad[AsyncIO[File]]
 :
```

Figure 18. Asynchronous I/O in callback style and direct style

transformation. As mentioned before, the inverse macro system allows to explicitly control the order of macro expansion. Similarly, *newCont* generated by the translation is empty since no further macro expansion is necessary.

This implementation does not return a pure value by throwing a runtime exception. It directly returns the value. This avoids the generation of many intermediate objects and it reduces runtime overhead due to throwing a runtime exception.

Note that the shift/reset operators are bundled with Scala 2.11 as part of the CPS plugin. However, the Scala language plans to remove them from Scala 2.12. The inverse macro system allows programmers to use the shift/reset operators in Scala 2.12 and later.

5.3 Monads in Direct Styles

A monad in the direct style [11–13] was proposed by Filinski. This example does not implement an operator but it shows that an inverse macro is useful to implement an implicit side-effectful DSL. Task-parallelizing DSL such as async/await and coroutines or generators are useful applications. A monad in the direct style allows programmers to use a monadic library through a simple programming interface in the semantic level. A similar construct is the Do-notation [24] in Haskell. Since the Do-notation is a syntactic interface, it requires special keywords do and <-. On the other hand, a monad in the direct style does not require such keywords but semantic information, typically types.

To illustrate a benefit of the direct style, we show the asynchronous I/O library of node.js[2]. The library adopts the callback style, or the explicit continuation passing style, to allow a program to continue running without blocking till a requested I/O finishes. Although the library is widely used in practice because of its efficiency, programmers using the library often see a problem called *callback hell*. They often have to describe deeply nested closures, which are troublesome to read and write as shown in the upper half in Fig. 18. The open method takes a file name and returns an AsyncIO[File] object. This object has a continuation-passing-style method foreach, which takes a closure taking a File object as its argument. The level of nesting in this program will be deeper as more files are opened. A monad in the direct style addresses this problem by implicitly performing the CPS transformation by using reflective language constructs. It allows programmers to write a simpler program shown in the lower half in Fig. 18. There are no nested closures. The reflect operator hides the continuation passing style from the programmers. It invokes the CPS transformation at compile time so that the program will be transformed into a program equivalent to the upper half.

[2] https://nodejs.org/

The reflect operator can be implemented with an inverse macro that transforms the code following the operator into a closure. Hence the approach is similar to the implementation for delimited continuation. First, a type annotation monad is defined. Then the transformation method for monad is written so that it will generate the code that creates a closure by wrapping up the continuation sequence passed to it.

It is known that, if a delimited continuation operator is available, the reflect operator can be implemented on top of that operator.[11] However, the implementation of the reflect operator by using an inverse macro (without implementing a delimited continuation operator) is more straightforward and it has a few advantages.

First, the implementation involves simpler types. For example, the delimited continuation operators shift/reset take three type parameters and one of them exploits return-type polymorphism. In Fig. 17, the type parameter C is for return-type polymorphism. Such a type parameter is not inferred in Scala. Thus the implementation using shift/reset may cause the reflect operator with an unnecessarily complicated type. Although the reflect operator or other monadic operators do not need return-type polymorphism, if their implementation uses the shift/reset operator, their user programmers have to add type ascriptions to the reflect operator and monadic operators. If they are implemented with an inverse macro, their user programmers do not have to add unnecessary type ascriptions.

Another advantage is that runtime overhead will be small since the implementation with an inverse macro does not need closures representing controllers and continuations, which the implementation with delimited continuation needs. Furthermore, the approach using an inverse macro is applicable to the implementation of the direct style of other functors, such as fork/joins and applicative functors [27]. For example, we can implement *fork/join in the direct style* and *applicative functors in direct style*, which is similar to applicative-do [25] in Haskell.

6. Experiments

We show the expressiveness and performance of an inverse macro by implementing a delimited continuation operator *shift/reset*. Shift/reset is also provided by Scala 2.11 as the CPS plugin [32], which is used for comparison.

6.1 Expressiveness

First, we show the expressiveness of inverse macros. For this objective, we tested our implementation with the unit test suites[3] for the CPS plugin, which suggests how compatible the shift/reset implementation with the inverse macro system is with the implementation by the CPS plugin for Scala 2.11. This unit test consists of 10 suites: Functions, IfReturn, IfThenElse, PatternMatching, Suspendable, TryCatch, HigherOrder, Inference, Return, and While. It includes 35 test cases. The suite named Misc was removed since it consists of test cases depending on the internal implementation.

Table. 1 lists the result obtained by using Scala 2.11.7 and OpenJDK 1.8.0_45, 64 bit version. 6 test suites containing 23 test cases were successfully passed with only small modification. The modification we had to do was as follows. First, we had to add type ascriptions. As described in the implementation section, the power of the type inference of the inverse macro system is poorer than that of the CPS plugin. We thus had to add type ascriptions to shift and reset as shown in Fig. 19. We also had to modify try-catch expressions. Since they were wrongly catching a ControlThrowable object internally used by the implementation using the inverse

[3] These suites were downloaded from https://github.com/scala/scala-continuations.

Status	Name of test suites	#Cases
Passed	Functions, IfReturn, IfThenElse, PatternMatching, Suspendable, TryCatch	23
Failed(1)	Return, While	12
Failed(2)	HigherOrder, Inference	

Table 1. Test result

```
// before
reset { shift {k => ... } }
// after
reset[Unit,Unit] { shift {(k: Unit=>Unit) => ... } }
```

```
// before
catch { case ex: Throwable => 9 }
// after
catch { case ex: Throwable
 if !ex.isInstanceOf[ControlThrowable] => 9 }
```

Figure 19. Modification added to test cases

macro, we had to modify the pattern match rule in the catch clause. See also Fig. 19.

On the other hand, 4 test suites containing 12 test cases failed. We saw two kinds of failure. Some tests failed since the current version of the inverse macro system does not support while and return expressions. This is one of our future work. The other tests failed due to the incompleteness of our shift/reset implementation. For example, the owners of symbols are not properly set in our current implementation. Although this would not directly mean an inherent problem of the expressiveness of the inverse macro system, better library supports for implementing the transformation method is desirable. This is also our future work. Overall, the results show the idea of inverse macros is with good potential but we still need efforts to make it practical.

6.2 Performance

We measured the performance of shift/reset implemented with the inverse macro system. We used two micro benchmarks shown in Fig. 20. The upper program includes reset but does not actually capture the current continuation by shift. During macro expansion, this program is transformed into the continuation passing style (CPS) but no object is created during runtime for representing a continuation. The lower program captures the current continuation by shift. It is transformed into the CPS and an object is created during runtime. We changed the number of the repetition of line 5 to 8 from 1 to 50. The benchmark programs were run 1,000,000 times and we measured the total execution time. For comparison, we also ran the micro benchmarks compiled with the CPS plugin. In this experiment, in order to suppress the affect of JIT compiling, we discarded the first 10 runs and took the average of the next 10 runs. The machine we used for the experiment had Intel Core i7-4770S 3.10GHz with 4 cores and 8 physical threads. The JVM was OpenJDK 1.8.0_45, 64 bit version. Scala compiler was version 2.11.7. We used an optimizing option -optimise.

Fig. 21 shows the result. The number of repetition indicates how many times line 5 is repeated. The left graph shows the execution time of the upper program while the right graph shows that of the lower one. The upper graph reveals that the overhead due to the conversion to the CPS is comparable between our inverse macro and Scala's CPS plugin. According to the lower graph, the runtime penalty for handling the current continuation is large if implemented with our inverse macro. This is because our implementation

```
1 val a = Array[Int](0)
2 // start benchmark
3 reset[Unit, Unit]{
4   // n times
5   a(0) += (1 : @cpsParam[Unit, Unit])
6   a(0) += (1 : @cpsParam[Unit, Unit])
7   :
8   a(0) += (1 : @cpsParam[Unit, Unit])
9   ()
10 }
11 // end benchmark
```

```
1 val a = Array[Int](0)
2 // start benchmark
3 reset[Unit, Unit]{
4   // n times
5   a(0) += shift((k: Int => Unit) => k(1))
6   a(0) += shift((k: Int => Unit) => k(1))
7   :
8   a(0) += shift((k: Int => Unit) => k(1))
9   ()
10 }
11 // end benchmark
```

Figure 20. Micro benchmarks for shift/reset

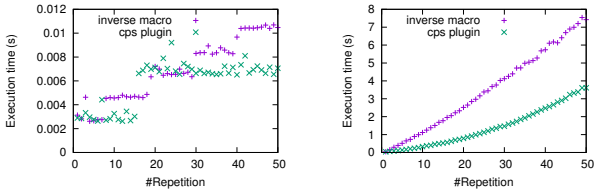

Figure 21. The execution time of micro benchmarks for shift/reset

heavily uses a runtime exception to capture the current continuation. This implementation approach is different from that of the CPS plugin. The CPS plugin modifies the method signatures and directly returns captured continuation objects. Our inverse macro system cannot provide this feature for preserving the locality of rewriting.

We also measured the compilation time of a program using an inverse macro. Fig. 22 shows the compilation time of the program listed above the graph. The machine was the same that we used when measuring the execution performance. Although the program size was small, the compilation time was apparently long. This is because the inverse macro system repeatedly traverses a syntax tree.

7. Related Work

The inverse macro system is similar to the typed syntactic macro system in Scala [3]. Both systems construct typed abstract syntax trees by using the standard parser and typer of Scala. They capture syntax trees, apply macro expansion to them, and splice them to the original context. However, the inverse macro system allows global rewriting; it can capture the syntax tree representing the continuation sequence as well as the tree representing the macro call.

Besides Scala's macro system, a large number of macro systems have been proposed. The C preprocessor, Lisp macros [5, 17, 23], Dylan macros [2], Nemerle macros [37], TemplateHaskell [35], Scala macro [3] are well known macro systems. These macro systems can be categorized with respect to inputs; the C preprocessor is a lexical macro system while Lisp macros, Dylan macros,

```
1 object Main {
2   def main(args: Array[String]) =
3     reset[Unit, Unit] {
4       val a = Array[Int](0)
5       // n times
6       a(0) += shift((k: Int => Unit) => k(1))
7       a(0) += shift((k: Int => Unit) => k(1))
8        :
9       a(0) += shift((k: Int => Unit) => k(1))
10      ()
11    }
12  }
13 }
```

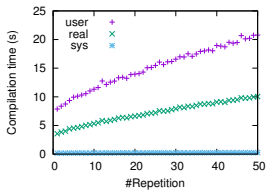

Figure 22. The compilation time of shift/reset implemented by the inverse macro system

Nemerle macros, and TemplateHaskell are syntactic macro systems. Scala macro is a typed syntactic macro system. Some syntactic macro systems like TemplateHaskell can use type information. Macro systems can also be categorized with respect to transformation methods; the C preprocessor uses a template substitution method while the other macro systems use a multi-staged method, where transformation rules are written in the host languages. The inverse macro system is a variant of typed syntactic macro systems and uses a multi-staged method. However, the macro systems except the inverse macro system cannot capture the continuation sequence. Moreover, macro calls are designated by placing an explicit token or a macro name like method calls whereas the inverse macro uses type annotation.

The multi-stage computation systems are also related work. They include MataML [39], MetaOCaml [4] or BER MetaOCaml [22], Scala virtualized system [33], LMS [31], and Delite [38]. These systems can capture and modify code snippets embedded in a source code. While they capture code snippets explicitly designated, the inverse macro system captures code snippets implicitly determined as the continuation sequence.

Delimited continuation operators such as *shift/reset* [1, 6], are used to capture the current continuation as a closure. They are similar to inverse macros that capture the continuation sequence. *control/prompt* [10], *fcontrol* [36], *set/cupto* [16], and *splitter* [30] are also delimited continuation operators. Delimited continuation operators are different from inverse macros since the current continuation is a runtime value but the continuation sequence is the code snippet corresponding to the continuation.

The JIT (just-in-time) macro of Lancet [34] can be regarded as delimited continuation operators. Lancet's JIT macro captures an internal representation of the delimited continuation. It does not capture a closure representing a continuation. Unlike typical macros, Lancet's JIT macro is invoked by the JIT compiler during runtime. Lancet's JIT macro is different from our inverse macro since the former needs an explicit delimiting operator whereas the latter implicitly delimits by exploiting block structures. Furthermore, Lancet's JIT macro is applied to the running program after the linker runs. The inverse macro is applied at compile time to an individual compilation unit. It is more suitable for static transformation under separate compilation.

The type-directed selective CPS transformation [28] is a technique adopted by several statically-typed languages such as Scala [32]. The idea of this technique is similar to our inverse macros. Although this technique was developed for CPS transformation, the inverse macro can be used to implement other transformations.

The inverse macro system uses annotations for types. Annotation processing is a well-known programming technique. For example, in Java, JUnit4 and Lombok are popular annotation processors.[4] However, most of them are dedicated annotation processors and do not provide syntactic macro system. Scala's Macro Paradise plugin[5] provides syntactic macro system using annotations, not including annotations for types with this system. A type coercion triggered macro system like our inverse macro system is very rare.

Our inverse macros can be used to implement embedded DSLs but they do not enable syntax extensions to a host language. A macro call has to be syntactically valid in Scala. For extending syntax, extensible parsers such as CamlP4 [7], SugarJ [9], ProteaJ [18], and TSL [29] are needed. The inverse macro system could be used as a back-end system of those extensible parsers so that a richer embedded DSL can be implemented.

Aspect-oriented programming (AOP) systems [20] enables global rewriting as our inverse macros do. In a typical AOP language AspectJ [21], an aspect can instruct the compiler to "rewrite" the code designated by a pointcut so that a thread of control will be dispatched there to an advice body. The cflow (control flow) pointcut and the dflow (data flow) pointcut [26] designate the rewritten code similarly to the type annotations for inverse macros. However, since control flow and data flow are runtime information, the designation by cflow and dflow implies runtime penalties. Type annotations are compile-time information and thus an inverse macro does not imply any runtime overhead due to designation.

The language workbench [15], such as Spoofax [19] and MetaProgrammingSystem [8], is a system enabling us to develop a DSL and an IDE (integrated development environment) for that language. A inverse macro would be a component for building these systems.

8. Conclusion

This paper presented a new language construct named an inverse macro, which enables global rewriting of syntax trees. The inverse macro system has two unique features. An inverse macro can rewrite the continuation sequence and it is designated by type annotations. The inverse macro system enables to implement interprocedural and side-effectful operators such as a lazy operator and a delimited continuation operator. Moreover, the system also enables to implement direct styles, which would simplify a number of domain specific languages.

This paper also showed our prototype implementation of the inverse macro system on top of Scala 2.11. Although the performance of the delimited continuation operator implemented by that system was not sufficiently good, the function of that implemented operator was fairly compatible to the operator included in the CPS plugin of Scala.

References

[1] K. Asai and O. Kiselyov. Introduction to programming with shift and reset. *ACM SIGPLAN Continuation Workshop 2011*, 2011.

[2] J. Bachrach, K. Playford, and C. Street. D-expressions: Lisp power, dylan style. *Style DeKalb IL*, 1999.

[3] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In

[4] `http://junit.org/`, `https://projectlombok.org/`

[5] `http://docs.scala-lang.org/overviews/macros/annotations.html`

Proceedings of the 4th Workshop on Scala (SCALA '13), pages 3:1–3:10, Montpellier, France, July 2013.

[4] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, pages 57–76, Erfurt, Germany, September 2003.

[5] W. Clinger and J. Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*, pages 155–162, Orlando, Florida, USA, January 1991.

[6] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming (LFP '90)*, pages 151–160, Nice, France, June 1990.

[7] D. de Rauglaudre. Camlp4 reference manual, 2003. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`.

[8] S. Dmitriev. Language oriented programming: The next programming paradigm, 2004. `https://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`.

[9] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, pages 391–406, Portland, Oregon, USA, October 2011.

[10] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*, pages 180–190, San Diego, CA, USA, January 1988.

[11] A. Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*, pages 446–457, Portland, Oregon, USA, December 1994.

[12] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 175–188, San Antonio, Texas, USA, January 1999.

[13] A. Filinski. Monads in action. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*, pages 483–494, Madrid, Spain, January 2010.

[14] C. Flanagan, A. Sabry, B. F. Duba, and M. 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, Albuquerque, New Mexico, USA, June 1993.

[15] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. `http://martinfowler.com/articles/languageWorkbench.html`.

[16] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 12–23, La Jolla, California, USA, June 1995.

[17] T. P. Hart. Macro definitions for lisp. 1963.

[18] K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*, pages 13–24, Lugano, Switzerland, April 2014.

[19] L. C. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, pages 444–463, Reno/Tahoe, Nevada, USA, October 2010.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001.

[22] O. Kiselyov. MetaOCaml — an OCaml dialect for multi-stage programming, 2010. `http://okmij.org/ftp/ML/MetaOCaml.html`.

[23] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*, pages 151–161, Cambridge, Massachusetts, USA, August 1986.

[24] S. Marlow. The haskell 2010 language report, 2009. `http://www.haskell.org/onlinereport/haskell2010/`.

[25] S. Marlow. Applicative do-notation, 2013. `https://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo`.

[26] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer Berlin Heidelberg, 2003.

[27] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

[28] L. R. Nielsen. A selective CPS transformation. In *MFPS 2001, Seventeenth Conference on the Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 311–331. Elsevier, 2001.

[29] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP 2014 — Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer Berlin Heidelberg, 2014.

[30] C. Queinnec and B. Serpette. A dynamic extent control operator for partial continuations. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*, pages 174–184, Orlando, Florida, USA, January 1991.

[31] T. Rompf and M. 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, Eindhoven, The Netherlands, October 2010.

[32] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 317–328, Edinburgh, Scotland, September 2009.

[33] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):1–43, 2013.

[34] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 41–52, Edinburgh, United Kingdom, July 2014.

[35] T. Sheard and S. P. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[36] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[37] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in nemerle. *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, 2004.

[38] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transaction on Embedded Computing*, 13(4s):134:1–134:25, 2014.

[39] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '97)*, pages 203–217, Amsterdam, The Netherlands, June 1997.