Modular Interpreters for the Masses

Implicit Context Propagation using Object Algebras

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
storm@cwi.nl

Abstract

Modular interpreters have the potential to achieve component-based language development: instead of writing language interpreters from scratch, they can be assembled from reusable, semantic building blocks. Unfortunately, traditional language interpreters are hard to extend because different language constructs may require different interpreter signatures. For instance, arithmetic interpreters produce a value without any context information, whereas binding constructs require an additional environment.

In this paper, we present a practical solution to this problem based on implicit context propagation. By structuring denotational-style interpreters as Object Algebras [25], base interpreters can be retroactively *lifted* into new interpreters that have an extended signature. The additional parameters are implicitly propagated behind the scenes, through the evaluation of the base interpreter.

Interpreter lifting enables a flexible style of component-based language development. The technique works in mainstream object-oriented languages, does not sacrifice type safety or separate compilation, and can be easily automated. We illustrate implicit context propagation using a modular definition of Featherweight Java and its extension to support side-effects.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors, Code Generation, Interpreters

General Terms Languages

Keywords Modular interpreters, object algebras, implicit propagation

1. Introduction

Component-based language development promises a style of language engineering by assembling reusable building block instead of writing them from scratch. This style is particularly attractive in the context of language-oriented programming (LOP) [29], where the primary software development artifacts are multiple domain-specific languages (DSLs). Having a library of components capturing common language constructs, such as literals, data definitions, statements, expressions, declarations etc. would make the construction of these DSLs much easier and as a result has the potential to make LOP much more effective.

Object Algebras [25] are a design pattern to support type-safe extensibility of both syntax and interpretations in mainstream, object-oriented (OO) languages. Using Object Algebras, the abstract syntax of a language fragment is defined using a generic factory interface. Operations are then defined by implementing such interfaces over concrete types representing the semantics. Adding new syntax corresponds to modularly extending the generic interface, and any of pre-existing operations. New operations can be added by implementing the generic interface with a new concrete type.

Object Algebras can be seen as extensible denotational definitions: factory methods essentially map abstract syntax to semantic denotations (objects). Unfortunately, the extensibility provided by Object Algebras breaks down if the types of denotations are incompatible. For instance, an evaluation component for arithmetic expressions might use a function type $() \to \mathit{Val}$ as semantic domain, whereas evaluation of binding expressions requires an environment and, hence, might be expressed in terms of the type $\mathit{Env} \to \mathit{Val}$. In this case, the components cannot be composed, even though they are considered to represent the very same interpretation, namely *evaluation*.

In this paper we resolve such incompatibilities for Object Algebras defined over function types using implicit context propagation. An algebra defined over a function type $T_0 \times ... \times T_n \to U$ is *lifted* to a new algebra over type $T_0 \times ... \times T_i \times S \times T_{i+1} \times ... \times T_n \to U$. The new interpreter implicitly propagates the additional context information of type S through the base interpreter, which remains blissfully unaware. As a result, language components do not need to standardize on a single type of denotation, anticipating all possible kinds of context information. Instead, each semantic component can be defined with minimal assumptions about its semantic context requirements.

We show that the technique is quite versatile in combination with host language features such as method overriding, side effects and exception handling, and can be naturally applied to interpretations other than dynamic semantics. Since the technique is so simple, it is also easy to automatically generate liftings using a simple code generator. Finally, our illustrative case study using Featherweight Java shows that a large number of variants of the language can be derived using only a small number of given components.
The contributions of this paper can be summarized as follows:

- We present **implicit context propagation** as a solution to the problem of modularly adding semantic context parameters to existing interpreter functions (Section 3).
- We show the versatility of the technique by elaborating on how implicit context propagation is used with overriding, mutable context information, exception handling, languages with multiple syntactic categories, generic desugaring of language constructs and interpretations other than dynamic semantics (Section 4).

	Anticipate	Duplicate
Base Language	`trait Exp { def eval(env: Env): Val }` `class Lit(n: Int) extends Exp { def eval(env: Env) = n }` `class Add(l: Exp, r: Exp) extends Exp{ def eval(env: Env) = l.eval(env) + r.eval(env) }`	`trait Exp { def eval: Int }` `class Lit(n: Int) extends Exp { def eval = n }` `class Add(l: Exp, r: Exp) extends Exp { def eval = l.eval + r.eval }`
Extended Language	`class Var(x: String) extends Exp { def eval(env: Env) = env(x) }`	`trait Exp2 { def eval(env: Env): Val }` `class Lit2(n: Int) extends Exp2 { def eval(env: Env) = n }` `class Add2(l: Exp2, r: Exp2) extends Exp2 { def eval(env: Env) = l.eval(env) + r.eval(env) }` `class Var(x: String) extends Exp2 { def eval(env: Env) = env(x) }`

Table 1. Two attempts to adding variable expressions to a language of addition and literal expressions. On the left, the `Lit` and `Add` classes anticipate the use of an environment, without actually using it. On the right, the semantics of `Lit` and `Add` need to be duplicated.

- We present a simple, annotation-based code generator to generate boilerplate lifting code (Section 5).
- The techniques are illustrated using an extensible implementation of Featherweight Java with state. This allows us to derive 127 hypothetical variants of the language, out of 7 given language fragments (Section 6).

Implicit context propagation using Object Algebras has a number of desirable properties. First, it preserves the extensibility characteristics provided by Object Algebras, without compromising type safety or separate compilation. Second, semantic components can be written in direct style, as opposed to continuation-passing style or monadic style, which makes the technique a good fit for mainstream OO languages. Finally, the lifting technique does not require advanced type system features and can be directly supported in mainstream OO languages with generics, like Java or C#.

The full source code of our code generator and case study can be found online here:

https://github.com/cwi-swat/implicit-propagation/

2. Background

2.1 Problem Overview

Table 1 shows two attempts at extending a language consisting of literal expressions with variables in a traditional OO style[1]. The first row contains the base language implementation and the second row shows the extension. The columns represent two styles characterized by "anticipation" and "duplication" respectively. In each column, the top cell shows the "base" language, containing only literal expressions (`Lit`). The bottom cell shows the attempt to add variable expressions to the implementation.

The first style (left column) captures the traditional OO extension where a new AST class for variables (`Var`) is added. The extension is successful, since the base language anticipates the use of the environment. Unfortunately, the anticipated context parameter (`env`) is not used at all in the base language. Furthermore, anticipating additional context parameters, such as stores, leads to more unnecessary pollution of the evaluation interface in the base language. The main drawback of this style is that it breaks open extensibility. At the moment of writing the base language implementation the number of context parameters is fixed, and no further extensions are possible without a full rewrite.

The second style (right column) does not anticipate the use of an environment, and the implementation of `Lit` is exactly as one would desire. No environment is used, and so it is not referenced either. To allow the recursive evaluation of expressions in the extension, however, the abstract interface `Exp` needs to be replaced to require an environment-consuming `eval`. Consequently, the full logic of `Lit` evaluation needs to be reimplemented in the extension as `Lit2`. If more context parameters are needed later, the extended classes need to be reimplemented yet again. In fact, in this style, there is no reuse whatsoever.

To summarize, the traditional OO style of writing an interpreter supports extension of syntax (data variants), but only if the evaluation signatures are the same. As a result, any context parameters that might be needed in future extensions have to be anticipated in advance to realize modular extension. In the next section we reframe the example language fragments in the Object Algebras [25] style, which provides the essential ingredient to solve the problem using implicit context propagation.

[1] All code examples are in Scala (`http://www.scala-lang.org`). We extensively use Scala **trait**s, which are like interfaces that may also contain method implementations and fields. We also assume an abstract base type for values `Val` and a sub-type for integer values `IntVal`; throughout our code examples we occasionally elide some implicit conversions for readability.

2.2 Object Algebras

Using Object Algebras the abstract syntax of a language is defined as a generic factory interface. For instance, the base language abstract syntax of Table 1 is defined as the following trait:

```
trait Arith[E] {
  def add(l: E, r: E): E
  def lit(n: Int): E
}
```

Because the trait Arith is generic, implementations of the interface must choose a concrete semantic type of Arith expressions. In abstract algebra parlance, the factory interface corresponds to an algebraic *signature*, the generic type E is a syntactic *sort*, and implementations of the interface are *algebras* binding the generic sort to a concrete *carrier type*. Carrier types can be any type supported by the host language (i.e. Scala), but in this paper we only consider function types, and function objects (closures).

An evaluation algebra for the Arith language could be implemented as follows:

```
type Ev = () => Val

trait EvArith extends Arith[Ev] {
  def add(l: Ev, r: Ev) = () => IntVal(l() + r())

  def lit(n: Int) = () => IntVal(n)
}
```

The type alias Ev defines a carrier type consisting of nillary functions returning a value of type Val. Terms over the algebra are constructed by invoking the methods of the algebra:

```
def onePlusTwo[E](alg: Arith[E]): E
  = alg.add(alg.lit(1), alg.lit(2))

val eval = onePlusTwo(new EvArith {})
println(eval()) // => 3
```

The generic function onePlusTwo accepts an algebra of type Arith and constructs a term over it. Invoking this function with the evaluation algebra EvArith gives an object of type Ev which can be used to evaluate the expression add(lit(1), lit(2)).

Now let us extend Arith with variable expressions, as was attempted in Table 1. First the abstract syntax is defined using a generic trait:

```
trait Var[E] { def vari(x: String): E }
```

The syntax for both fragments can be combined using trait inheritance:

```
trait ArithWithVar[E] extends Arith[E] with Var[E]
```

For evaluating variables, we can implement the interface over a carrier type EvE which accepts an environment:

```
type EvE = Env => Val
trait EvVar extends Var[EvE] {
  def vari(x: String) = env => env(x)
}
```

Unfortunately, the two traits cannot be composed anymore because the carrier types are different: Ev and EvE. In order to compose the two syntactic interfaces, both carrier types have to be the same. In this case, however, the evaluation semantics of the language fragments require different context information, which prevents the components from being combined. We actually observe the same problem as shown in Table 1!

Fortunately, Object Algebras allow modular extension of operations. This means that it is possible to modularly define a trait for a different interpretation of the same syntax:

```
trait EvEArith extends Arith[EvE] {
  ...
}
```

This trait defines arithmetic expressions over the carrier type EvE instead of Ev. Internally this trait will delegate to the original EvArith which was defined over the type Ev. In the next section we describe this pattern in more detail.

```
trait Binding[E] {
  def lambda(x:Str, b:E): E
  def vari(x:Str): E
  def apply(e1:E, e2:E): E
  def let(x:Str, e:E, b:E): E
}

type EvE = Env => Val
type Env
  = immutable.Map[Str,Val]

class Clos(x:Str,b:EvE,e:Env)
  extends Val {
  def apply(v: Val): Val
    = b(e + (x -> v))
}
```

```
trait EvEBinding
  extends Binding[EvE] {
  def lambda(x: Str, b: EvE)
   = env => new Clos(x, b, env)

  def vari(x: Str): EvE
   = env => env(x)

  def apply(e1: EvE, e2: EvE)
   = env =>
        e1(env).apply(e2(env))

  def let(x:Str, e:EvE, b:EvE)
   = env =>
        b(env + (x -> e(env)))
}
```

Figure 1. A language fragment with binding constructs

3. Implicit Context Propagation

We have seen how the incompatibility between Object Algebras defined over different function types preclude extensibility. In this section we introduce implicit context propagation as a technique to overcome this problem, by first extending the Arith language to support variable binding, and then generalizing the pattern to support the propagation of other kinds of the context information.

3.1 Adding Environments to Arithmetic Expressions

The language fragment of expressions that require environments is shown in Figure 1. The Binding language defines four constructs: lambda (functions), vari (variables), apply (function application) and let (binding). The carrier type is EvE, a function from environments to values. To support lambdas, the Val domain is extended with closures (Clos). The interpreter on the right evaluates lambdas to closures. Variables are looked up in the environment. Function application expects that the first argument evaluates to a closure and applies it to the value of the second argument. Finally, let evaluates its third argument in the environment extended with a variable binding.

We now discuss the implementation of the environment-passing interpreter for the Arith language using implicit context propagation. As described in Section 2, two Object Algebra interpreters can be combined if they are defined over the same carrier type. In this case, this means that EvArith needs to be lifted to an EvEArith which is defined over the carrier type EvE, i.e., Env => Val:

```
trait EvEArith extends Arith[EvE] {
  private val base = new EvArith {}

  def add(l: EvE, r: EvE): EvE
    = env => base.add(() => l(env), () => r(env))()

  def lit(n: Int): EvE
    = env => base.lit(n)()
}
```

Instead of reimplementing the semantics for the arithmetic operations, the code for each variant delegates to the base field

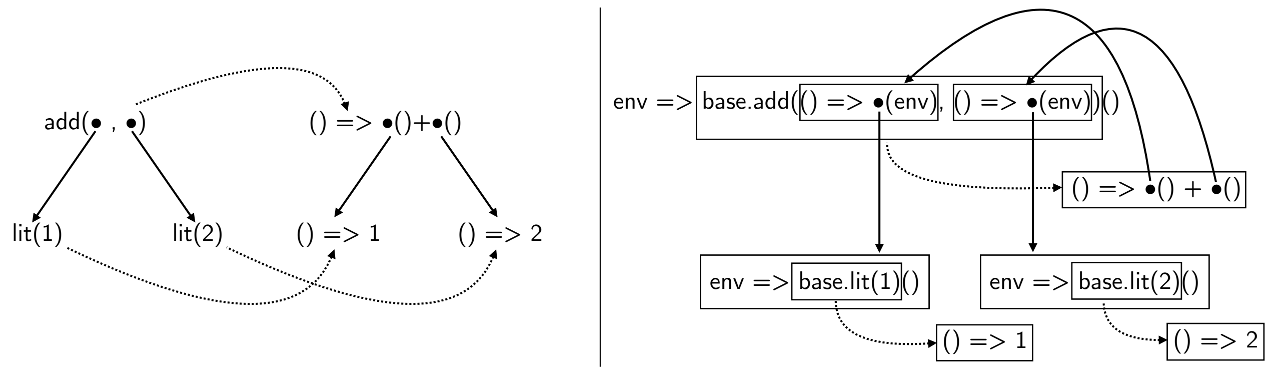

Figure 2. The left column shows how the expression add(lit(1), lit(2)) is mapped to its denotation by EvArith; the nodes in the tree are of type Ev (() => Val). On the right, the result of lifting the denotation produced by EvArith to the type EvE (Env => Val) to propagate environments. The dotted arrows indicate evaluation of Scala expressions; the solid arrows represent references.

initialized with EvArith. The interpreter EvEArith shows the actual propagation of the environment in the method for add. Invoking add on the base algebra requires passing in arguments of type Ev. This is achieved with the inline anonymous functions. Each of these closures calls the actual arguments of type EvE (l and r). Since both these arguments expect an environment, we pass in the original env that was received in the result of add.

In order to visualize lifting, Figure 2 shows the evaluation of add(lit(1), lit(2)) over EvArith (left) and over the lifted algebra EvEArith. On the left the result is a tree of closures of type Ev. The right shows how each closure is lifted to a closure of type EvE. Note that each of the closures in the denotation on the left is also present in the denotation on the right, but that they are connected via intermediate closures on the right.

The two languages can now be combined as follows:

```
trait EvEArithBinding extends EvEArith with EvEBinding
```

The following client code shows how to create terms over this language:

```
def makeLambda[E](alg: Arith[E] with Binding[E]): E = {
  import alg._
  lambda("x", add(lit(1), vari("x")))
}

val term: EvE = makeLambda(new EvEArithBinding {})
```

The method makeLambda provides a generic way of creating the example term lambda("x", add(lit(1), vari("x"))) over any algebra defining arithmetic and binding expressions. Invoking the method with an instance of the combined interpreter EvEArithBinding creates an object of type EvE.

3.2 Generating Implicit Context Propagation Code

The general pattern for generating context propagating code is shown in Figure 3. The template is written in pseudo-Scala and defines a trait $\mathsf{Alg}_{(T,U*)\Rightarrow V}$, implementing the language interface Alg over the function type (T, U*) => V. The asterisks indicate splicing of formal parameters. For instance, U* capture zero or more type parameters in the function signature (T, U*) => V. The same notation is used on ordinary formal parameters, as shown in the closure returned by constructor method C_i.

As shown in Figure 3, the base algebra is instantiated with an algebra over function type U* => V, which accepts one fewer parameter than the carrier type of $\mathsf{Alg}_{(T,U*)\Rightarrow V}$. For each constructor, C_i, the lifting code follows the pattern as shown. For presentation purposes, primitive arguments to C_i are omitted, and only arguments of the function types are shown as f_j, for $j \in 1, ..., n$.

```
trait Alg_{(T,U*)⇒V} extends Alg[(T, U*) => V] {

  val base = new Alg_{U*⇒V} {}

  def C_i(f_1: (T, U*) => V, ... , f_n: (T, U*) => V):
   (T, U*)=> V  =
     (t, u*) => base.C_i((u_1*) => f_1(t, u_1*),
                         ...,
                         (u_n*) => f_n(t, u_n*))(u*)
  . . .
}
```

Figure 3. Template for generating lifted interpreters that propagate environment-like context parameters.

The presented template concisely expresses the core mechanism of lifting. Notice, however, that it assumes that the added parameter is prepended at the front of the base signature. A realistic generation scheme would consider permutations of parameters. The code generator discussed in Section 5 supports inserting the parameter anywhere in the list.

4. Working with Lifted Interpretations

The example languages we have discussed so far only considered expressions in a purely functional framework. In the following we discuss how implicit context propagation can be used with overriding of semantics, mutable parameters to model side-effects, exception handling for non-local control, many-sorted languages, implementation by desugaring, and interpretations other than dynamic semantics.

4.1 Overriding Interpretations: Dynamic Scoping

The propagation of environments presented in Section 3 obeys lexical scoping rules for all parameters that are implicitly propagated. Some context information, however, should not be lexically scoped, but dynamically scoped. Typical examples include the binding of **self** or **this** in OO languages, dynamic contexts in context-oriented programming [17], or simply dynamically scoped variables [14].

Consider the following language fragment for introducing dynamically scoped variables, similar to fluid-let in Scheme [13]:

```
trait DynLet[E] { def dynlet(x: Str, v: E, b: E): E }
```

The construct dynlet binds a variable x to value in both the lexical and dynamic environment. A dynamic variable can then be referenced in the scope of dynlet using the ordinary vari of the

Binding fragment (cf. Figure 1). The implementation of dynlet is straightforward by using an extra parameter of type Env representing the dynamic environment.

```
type EvEE = (Env, Env) => Val

trait EvEEDynLet extends DynLet[EvEE] {
  def dynlet(x: Str, v: EvEE, b: EvEE)
    = (env, denv) => {
      val y = v(env, denv)
      b(env + (x -> y), denv + (x -> y))
    }
}
```

To combine the lexically scoped Binding fragment with the dynamically scoped DynLet fragment, EvEBinding (cf. Figure 1) needs to be lifted so that it propagates the dynamic environment. Implicit propagation can be used to obtain EvEEBinding. Unfortunately, the dynamic environment is now inadvertently captured when lambda creates the Clos object.

To work around this problem, the implementation of lambda and apply in EvEEBinding should be overridden, to support the dynamic environment explicitly:

```
class DClos(x: String, b: EvEE, env: Env) extends Val {
  def apply(denv: Env, v: Val)
    = b(env ++ denv + (x -> v) , denv) // denv shadows env
}


trait EvEEBindingDyn extends EvEEBinding {
  override def lambda(x: Str, b: EvEE): EvEE
    = (env, denv) => new DClos(x, b, env)

  override def apply(e1: EvEE, e2: EvEE)
    = (env, denv) => e1(env, denv).apply(denv, e2(env, denv))
}
```

The closure class DClos differs from Clos only in the extra denv parameter to the apply method. The supplied dynamic environment denv is added to the captured environment, so that a dynamically scoped variable x (introduced by dynlet) will shadow a lexically scoped variable x (if any).

Although the existing lambda and apply could not be reused, one could argue that adding dynamically scoped variables to a language is not a proper extension which preserves the semantics of all base constructs. In other words, adding dynamic scoping literally changes the semantics of lambda and apply.

As an example of the dynamically scoped propagation, consider the following example term, which is defined over the combination of Arith, Binding, and DynLet:

```
dynlet("x", lit(1),
  let("f", lambda("_", add(vari("x"), lit(1))),
    dynlet("x", lit(2),
      let("z", dynlet("x", lit(3),
                  apply(vari("f"), lit(1))),
        add(vari("z"), vari("x"))))))
```

This program dynamically binds x to 1, in the scope of the let which defines a lambda dynamically referring to x. The value of x thus depends on the dynamic scope when the lambda is applied to some argument. Nested within the let is another dynamic let (dynlet) which overrides the value of x. The innermost let then defines a variable z with the value of applying f to 1, which is inside another dynamic let, yet again redefining x. So the result of the application will be 4, as the innermost dynamic scope defines x to be 3. In the body of the innermost normal let, however, the active value of x is 2, so the final addition $z + x$ evaluates to 6.

4.2 Mutable Parameters: Stores

The previous section showed how the lexical scoping of propagated parameters was circumvented through overriding the semantics of certain language constructs. Another example of a context information that should not be lexically scoped is stores for modeling side effects. In this case however, the parameter should also not obey stack discipline (which was the case for the dynamic environment). The way to achieve this is by propagating mutable data structures. Consequently, all interpreter definitions will share the same store, even when they are captured when closure objects are created.

Consider a language Storage which defines constructs for creating cells (create), updating cells (update) and inspecting them (inspect):

```
trait Storage[E] {
  def create(): E
  def update(c: E, v: E): E
  def inspect(c: E): E
}
```

The simplest approach to implement an interpreter for such expressions is to use a mutable store as a parameter to the interpreter. For instance, the following type declarations model the store as a mutable Map and the interpreter as function from stores to values:

```
type Sto = mutable.Map[Cell, Val]
type EvS = Sto => Val
```

The interpreter for Storage could then be defined as follows:

```
trait EvSStorage extends Storage[EvS] {
  def create() = st => ...
  def update(c: EvS, v: EvS) = st => ...
  def inspect(c: EvS) = st => ...
}
```

To compose, for example, the Arith language defined in Section 2.2 with Storage, the EvArith interpreter needs to be lifted in order to propagate the store. Since Sto is a mutable object, side-effects will be observable even though the propagation follows the style of propagating environments.

Unsurprisingly, perhaps, mutable data structures are an effective way of supporting side-effecting language constructs. It is interesting to contemplate whether it is possible instead to lift interpreters that thread an immutable store through the base evaluation process, without depending on mutation. We have experimented with a scheme that uses a private mutable variable, local to the traits containing the lifted methods.

The following example is a failed attempt at lifting EvArith to thread an immutable store (represented by the type ISto). Since the store is immutable, the carrier type EvS2S takes an ISto and produces a tuple containing the return value and the (possibly) updated store.

```
type ISto = immutable.Map[Cell,Val]
type EvS2S = ISto => (Val, ISto)

trait EvS2SArith extends Arith[EvS2S] {
  private val base = new EvArith {}
  private var _st: ISto = _

  def add(l: EvS2S, r: EvS2S)
    = st => { _st = st;
        (base.add(() => {val (v1, s1) = l(_st); _st = s1; v1},
                  () => {val (v2, s2) = r(_st); _st = s2; v2}
              )(), _st)
      }
  ...
}
```

At every evaluation step, the private variable _st is synchronized with the currently active store returned by sub expressions; since the

```
class Fail
  extends Exception

trait Choice[E] {
  def or(l: E, r: E): E
  def fail: E
}
```

```
trait EvChoice extends Choice[Ev] {
  def or(l: Ev, r: Ev)
    = () => try { l() }
            catch { case _:Fail => r() }

  def fail() = () => throw new Fail
}
```

Figure 4. Implementing local backtracking with exception handling.

current value of _st is also passed to the subsequent evaluation of sub terms, side effects are effectively threaded through the evaluation.

Unfortunately, this scheme breaks down when two different lifted traits have their own private _st field. As a result, expressions only see the side-effects enacted by expressions within the same lifting, but not the side-effects which originate from other lifted traits. It would be possible to share this "current store" using an ambient, global variable, allowing different traits (lifted or not) to synchronize on the same store. Such a global variable, however, compromises the modularity of the components and would complicate the code generation considerably, especially in the presence of multiple store-like context parameters.

4.3 Exception Handling: Backtracking

Many non-local control-flow language features can be simulated using exception handling. A simple example is shown in Figure 4, which contains the definition of a language fragment for (local) backtracking. The or construct first tries to evaluate its left argument l, and if that fails (i.e., the exception Fail is thrown), it evaluates the right argument r instead. Note that EvChoice does not require any context information and is simply defined over the carrier type Ev.

If EvChoice is lifted to EvEChoice to implicitly propagate environments, the exception handling still provides a faithful model of backtracking, because the environments are simply captured in the closures l and r. In other words, upon backtracking – when the Fail exception is caught – the *original* environment is passed to r.

```
trait EvEChoice extends Choice[EvE] {
  private val base = new EvChoice {}
  def or(l: EvE, r: EvE)
    = env => base.or(() => l(env), () => r(env))()

  def fail() = env => base.fail()()
}
```

For instance, evaluating the following term using this algebra, results in the correct answer (1):

```
let("x", lit(1), or(let("x", lit(2), fail()), vari("x")))
```

4.4 Many-sorted Languages: If-Statements

Up till now, the language components only have had a single syntactic category, or *sort*, namely expressions. In this section we discuss the propagation in the presence of multiple syntactic categories, such as expressions and statements.

In the context of Object Algebras, syntactic sorts correspond to type parameters of the factory interfaces. For instance, the following trait defines a language fragment containing if-then statements:

```
trait If[E, S] { def ifThen(c: E, b: S): S }
```

The ifThen construct defines a statement, represented by S, and it contains an expression E as condition.

The interpreter for ifThen makes minimal assumptions about the kinds of expressions and statements it will be composed with. Therefore, E is instantiated to Ev (() => Val; see above), and S is instantiated to the type Ex:

```
type Ex = () => Unit

trait EvIf extends If[Ev, Ex] {
  def ifThen(c: Ev, b: Ex): Ex = () => if (c()) b()
}
```

The type Ex takes no parameters, and produces no result (Unit). The ifThen construct simply evaluates the condition c and if the result is true, executes the body b.

A first extension could be the combination with statements that require the store, like assignments. Statements that require the store are defined over the type ExS = Sto => Unit. As a result, EvIf needs to be lifted to map type Ex to ExS. Since the only argument of ifThen that has type Ex is the body b, lifting is only applied there. In the current language, expressions do not have side-effects, so they do not require the store, and consequently do not require lifting:

```
type ExS = Sto => Unit

trait ExSIf extends If[Ev, ExS] {
  private val base = new EvIf {};

  def ifThen(c: Ev, b: ExS)
    = st => base.ifThen(c, () => b(st))()
}
```

Note that the argument c is passed unmodified to base.ifThen.

An alternative extension to consider is adding expressions which require an environment. In Section 3.1 such expressions were defined over the type EvE = Env => Value. In this case, EvIf needs to be lifted so that ifThen can be constructed with expressions requiring the environment. In other words, c: Ev needs to be lifted to c: EvE. However, since an actual environment is needed to invoke a function of type EvE, the result sort Ex also needs to be lifted to accept an environment:

```
type ExE = Env => Unit

trait EvEIf extends If[EvE, ExE] {
  private val base = new EvIf {}

  def ifThen(c: EvE, b: ExE)
    = env => base.ifThen(() => c(env), () => b(env))
}
```

In this lifting code, the invocation of c requires an environment; hence, the closure returned by ifThen needs to be of type ExE to accept the environment and pass it to c.

The context parameters propagate outwards according to the recursive structure of the language. At the top level, the signature defining the semantics of a combination of language fragments will accept the union of all parameters needed by all the constructs that it could transitively contain.

4.5 Desugaring: Let

Desugaring is a common technique to eliminate syntactic constructs ("syntactic sugar") by rewriting them to more basic language constructs. As a result, the implementation of certain operations (like compilation, interpretation) becomes simpler, because there are fewer cases to consider.

Desugaring in Object Algebras is realized by directly calling another factory method in the algebra. Note that methods in traits in Scala do not have to be abstract. As a result, desugarings can be implemented generically, directly in the factory interface. The same, generic desugaring can be reused in any concrete Object Algebra implementing the syntactic interface.

As an example, recall the Binding language of Figure 1. It defines a let constructor which was implemented directly in the right column of Figure 1. Instead, let can be desugared to a combination of lambda and apply:

```
trait Let[E] extends Binding[E] {
  def let(x:Str, e:E, b:E) = apply(lambda(x, b), e)
}
```

This trait generically rewrites let constructs to applications of lambdas, binding the variable x in the body b of the let. Since the desugaring is generic, it can be reused for multiple interpreters, including the ones resulting from lifting. If EvEBinding (Figure 1) is lifted to propagate the store, for instance, the desugaring would automatically produce lifted lambda and apply denotations.

```
type EvES = (Env, Sto) => Val

trait EvESBinding extends Binding[EvES]  {
  ... // store propagation code
}

trait EvESBindingWithLet extends EvESBinding with Let[EvES]
```

Generic desugarings combined with traits (or mixins) provide a very flexible way to define language constructs irrespective of the actual interpretation of the constructs themselves. Keeping such desugared language constructs in separate traits also makes them optional, so that they are not expanded if this is undesired, for example in the case of pretty printing.

4.6 Multiple Interpretations: Pretty Printing

Object Algebras support the modular extension of syntax as well as operations. There is no reason for implicit propagation not to be applied to interpretations of a language other than the dynamic semantics. Examples could include type checking, other forms of static analysis or pretty printing.

Consider the example of pretty printing. Here is an example of a pretty printer for Arith expressions, PPArith, defined over the carrier type PP (() => String):

```
type PP = () => Str // "Pretty Print"

trait PPArith extends Arith[PP] {
  def add(l: PP, r: PP) = () => l() + " + " + r()

  def lit(n: Int) = () => n.toString
}
```

Pretty printing of arithmetic expression does not involve the notion of indentation. However, to pretty print the ifThen construct of Section 4.4 we would like to indent the body expression. This is realized with a context parameter i that tracks the current indentation level:

```
type PPI = Int => Str

trait PPIIf extends If[PPI,PPI] {
  def ifThen(c: PPI, b: PPI)
    = i => "if " + c(0) + "\n" +
          " " * i + b(i + 2)
}
```

Both modules can be combined after lifting PPArith to propagate the parameter representing the current indentation:

```
trait PPIArith extends Arith[PPI] {
  private val base = new PPArith {}
  def add(l: PPI, r: PPI) = i => base.add(() => l(i), () => r(i))()
  def lit(n: Int) = i => base.lit(n)()
}
```

5. Automating Lifting

We have introduced implicit context propagation and illustrated how the liftings work in diverse scenarios. Although the liftings can be written by hand, they represent a significant amount of error-prone boilerplate code. We have developed a macro-based code generator for single-sorted algebras, which generates the lifting code automatically.

The code generator is invoked by annotating an empty trait, and the code generator will fill in the required lifting methods in the compiled code. Here is an example showing how to lift the EvArith interpreter to propagate the environment and the store:

```
@lift[Arith[_], ()=> Val, EvArith, (Env, Sto) => Val]
trait EvESArith
```

The @lift annotation receives four type parameters: the trait that corresponds to the generic Object Algebra interface representing the language's syntax (Arith), the carrier type of the base implementation (()=>Val), the trait that provides the base level implementation (EvArith), and finally, the target carrier type ((Env,Sto)=>Val).

The @lift annotation is implemented as a Scala macro [2]. The annotated trait produces an implementation for the lifted trait that extends the factory interface instantiating the type parameter to the extended carrier type. The compiled code will contain the lifted methods that delegate to the specified base implementation. Note that the code generation does not break independent compilation nor type safety, as the generator just inspects the interfaces of the types that are specified in the annotation, without needing access to the source code where these types are defined.

Notice too that the code generator does not simply prepend a new parameter at the front of the parameter list (as in the template of Figure 3), but performs the necessary permutations to appropriately lift the base signature to the target. This is important for components that need to be "mutually lifted". Consider the example of a component which is lifted from Sto=>Val to (Env,Sto)=>Val. To combine this component with a component of type Env=>Val, the latter should be lifted to (Env,Sto)=>Val as well, but in this case, the parameter is added at the end.

The current version @lift does not disambiguate parameters with the same type. However, it is conceivable to implement this by requiring the user to provide the disambiguation information in the annotation. This is an opportunity for future work.

6. Case Study: Modularizing Featherweight Java

To examine how implicit context propagation helps in modularizing a programming language implementation, we have performed a case study using *Featherweight Java* (FJ) [18]. The case study consists of a modular interpreter for FJ, and its extension to a variant that supports state (SFJ), inspired by [10].

The case study serves to answer two questions:

- What is the flexibility that implicit context propagation provides to support the definition of languages by assembling language fragments?
- How much boilerplate code is avoided by implicit context propagation?

These questions are answered below by analyzing the number of hypothetical languages that can be defined from the combination of SFJ fragments, and counting how many liftings are possible.

6.1 Definition of FJ and SFJ

FJ was introduced as a minimal model of a Java-like language, small enough to admit a complete formal semantics. In FJ, there are no side-effects and all values are objects; it supports object creation, variables, method invocation, field accessing and casting. To study

		Syntax	Signature
FJ	Field access	e.f	CT=>Obj
	Object creation	**new** C(e,...)	()=>Obj
	Casting	(C) e	CT=>Obj
	Variables	x	(Obj,Env)=>Obj
	Method call	e.m(e,...)	(Obj,CT,Env)=>Obj
SFJ	Sequencing	e ; e	()=>Obj
	Field assignment	e.f = e	(CT,Sto)=>Obj
	Object creation	**new** C(e,...)	(CT, Sto)=>Obj
	Variables	x	(Obj,CT,Env,Sto)=>Obj

Table 2. Signatures per (S)FJ language construct

Signature	Base	Liftings	Derived	Total
CT=>Obj	2	O/E/S	14	16
(Obj,Env)=>Obj	1	C/S	3	4
(Obj,CT,Env)=>Obj	1	S	1	2
()=>Obj	2	C/O/E/S	30	32
(CT,Sto)=>Obj	2	O/E	6	8
(Obj,CT,Env,Sto)=>Obj	1		0	1
				63

Table 3. Number of Base interpreters per signature, possible Liftings (C = CT, O = Obj, E = Env, S = Sto), number of possible Derived interpreters and Total number of possible interpreters.

how to extend a language to a variant that requires more context information, we introduce SFJ, which also features field updating and sequencing.

We have modularly implemented FJ and its extension to SFJ defining one language module per alternative in the abstract grammar. Each language construct is represented as a single Object Algebra interface to allow for maximum flexibility. As a consequence, the semantics of each construct is defined in its own trait assuming only the minimal context information necessary for the evaluation of that particular construct.

A complete definition of SFJ requires four kinds of context information:

- An Obj that represents the object being currently evaluated (i.e., **this**). In FJ, the Obj simply contains the object's class name and the list of arguments that are bound to its fields.
- The class table CT which contains the classes defined in an FJ program. The classes contain the meta information about objects, in particular, how the ordering of constructor arguments maps to the object's field names.
- The environment Env which maps parameters to Objs.
- The store Sto modeling the heap (just needed in the case of SFJ).

As shown in Table 2, six different signatures are used to implement nine constructs. For presentation purposes, we solely focus on the expression constructs. Object creation does not require any context information. Field access and casting only require the class table to locate fields in objects by offset. Variables require the current object to evaluate the special variable **this**, and the environment to lookup other variables. Similarly, method calls require the class table to find the appropriate method to call; the current object is needed to (re)bind the special variable **this** and the environment is needed to bind formal parameters to actual values. Sequencing does not depend on any context parameters. Field assignment uses the class table to locate fields and the store to modify the object. Finally, notice that the cases for object creation and variable referencing had to be redefined in SFJ over the signatures(CT,Sto) => Obj and (Obj,CT,Env,Sto) => Obj in order to allocate storage for the newly created object and inspecting the referenced object in the store, respectively. In particular, variable referencing needs all the context parameters as it needs to "reconstruct" the object structure by inspecting the store and finding the information about the order of arguments in the class table.

For implementing FJ, four of the base interpreters (for variables, field access, object creation and casting) are lifted to the function type (Obj,CT,Env) => Obj. Combining these lifted interpreters results in an implementation of basic FJ.

In order to obtain a full implementation of SFJ, the FJ interpreters need to be lifted to also propagate the store and the stateful fragments need to be lifted to propagate the environment, class table and current object, where needed. The result is a set of interpreters defined over the "largest" signature (Obj,CT,Env,Sto) => Obj.

6.2 Analyzing Hypothetical Subsets of SFJ

The previous subsection detailed how the implementation of FJ and SFJ can be constructed by modularly assembling the language fragments of Table 2. Here we discuss hypothetical subsets of the language. Note that these subsets might not (and probably will not) be meaningful in any practical sense. However, they illustrate the flexibility obtained using implicit context propagation.

Table 3 shows per interpreter signature how many interpreters can be derived using implicit context propagation. The second column lists the number of given base interpreters over a specific signature. The third column indicates the number of lifting opportunities. Finally, the last column shows the total number of possible interpreters, including the base interpreters. Note that the next-to-last row shows two base interpreters because the interpreter for object construction needed to be rewritten to allocate storage.

Lifting opportunities are described using a shorthand indicating which types of parameters could be added to the signatures using implicit context propagation (C = CT, O = Obj, E = Env, S = Sto). For instance, the string "O/E/S" in the first row means that an interpreter over CT=>Obj can be lifted to any of the following 7 signatures:

```
(Obj,CT)=>Obj, (Env,CT)=>Obj, (Sto,CT)=>Obj,
(Obj,CT, Env)=>Obj, (Obj,CT, Sto)=>Obj,
(Env,CT,Sto)=>Obj, (Obj,Env,CT,Sto)=>Obj
```

The number of possible lifted interpreters given n base interpreters can be computed using the following formula $n \times (2^k - 1)$, where k represents the number of possibly added context parameters. Since there are $n = 2$ base interpreters in the first row for which $k = 3$, 7 opportunities apply to each of them, and thus the total number of derivable interpreters is 14.

Summing the last column in Table 3 gives an overall total of 63 possible interpreters, of which only 9 are written by hand. The other 54 can be derived automatically using implicit context propagation. It is thus possible to observe that our technique eliminates considerable amount of boilerplate when deriving new variants of languages from base language components.

The 63 interpreters include 7 over the "largest" signature (Obj,CT,Env,Sto) => Obj for each of the 7 language constructs. These 7 fragments allow $2^7 - 1 = 127$ combinations representing hypothetical subsets of SFJ (excluding the empty language). The interpreter for full SFJ is just one of these 127 variants. This gives an idea of the flexibility that implicit context propagation provides in defining multiple language variants from assembling the different language modules.

7. Discussion and Related Work

In this section we provide a qualitative assessment of implicit context propagation as a technique and discuss related work.

7.1 Discussion

Although all the code in this paper, as well as the case study, is written in Scala, it is easy to port implicit context propagation to other languages as well. For instance, Java 8 introduce default methods in interfaces, which can be used for trait-like multiple inheritance. Without a trait-like composition mechanism, the technique can still be of use, except that extensibility would be strictly linear. This loses some of the appeal for constructing a library of reusable semantic building blocks, but still enjoys the benefits of type safety and modular extension.

Unlike in other work on modular interpreters (see below for a discussion of related work), implicit context propagation is simpler than, for instance, extensible effects in the sense of [20]. For context information other than read-only, environment-like parameters, we depend on the available mechanisms of the host language. For instance, read-write effects (stores) are modeled using mutable data structures (cf. Section 4.2). Other effects, such as error propagation, local backtracking (Section 4.3), non-local control flow (`break`, `continue`, **return** etc.), and gotos and co-routines [1] can be simulated using the host language's exception handling mechanism. Support for concurrency or message passing can be directly implemented using the host language's support for threads or actors (cf. [12]).

A drawback of implicit context propagation is that, even though the boilerplate code can be automatically generated, the user still has to explicitly specify which liftings are needed, and compose the fragments herself. Instead of using the annotations, it would be convenient if one could simply extend a trait over the right signature, and that the actual implementation would be inferred completely. For instance, instead of writing the `@lift` annotation described in Section 5, one would like to simply write:

```
trait Combined extends EvEBinding with Arith[EvE]
```

The system would then find implementations of `Arith[_]` to automatically define the required lifting methods right into the `Combined` trait. If multiple candidates would exist, it would be an error. This is similar to how Scala implicit parameters are resolved[2]. We consider this as a possible direction for future work.

7.2 Related Work

Component-Based Language Development The vision of building up libraries of reusable language components to construct languages by assembling components is not new. An important part of the Language Development Laboratory (LDL) [15] consisted of a library of language constructs defined using recursive function definitions. Heering and Klint considered a library of reusable semantic components as a crucial element of Language Design Assistance [16]. Our work can be seen as a practical step in this direction. Instead of using custom specification formalisms, our semantic components are defined using ordinary programming languages, and hence, are also directly executable.

More recently, Cleenewerck investigated reflective approaches to component-based development [5, 6]. In particular he investigated the different kinds of interfaces of various language aspects and how they interact. Implicit context propagation can be seen as a mechanism to address one such kind of feature interaction, namely the different context requirements of interpreters.

Directly related to our work is Mosses' work on component-based semantics [4]. Languages are defined by mapping abstract syntax to fundamental constructs (*funcons*), which in turn are defined using I-MSOS [24], an improved, modular variant of Structural Operational Semantics (SOS) which also employs implicit context propagation. The modular interpreters of this paper can be seen as the denotational, executable analog of I-MSOS modules. In fact, our implicit context propagation technique was directly inspired by the propagation strategies of I-MSOS.

Finally, first steps to apply Object Algebras to the implementation of extensible languages have been reported in [11]. In particular, this introduced Naked Object Algebras (NOA), a practical technique to deal with the concrete syntax of a language using Java annotations. We consider the integration of NOA to the modular interpreter framework of this paper as future work. In particular, we want to investigate designs to support multiple concrete syntaxes for an abstract semantic component.

Modular Interpreters The use of monads to structure interpreters is a well-known design pattern in functional programming. Monads, as a general interface for sequencing, allow idioms such as environments, stores, errors, and continuations to be automatically propagated. However, monads themselves do not allow these different effects to be combined.

Liang et al. [23] consolidated much of the earlier work (e.g., [9, 26]) on how *monad transformers* can be used to solve this problem. Monad transformers *lift* one type of monad into another, richer one. This allows one to define extensible interpreters. Unfortunately, the order of transforming one monad into the richer monad influences the semantics of the result. This means that a monadic interpreter supporting the store first, and then non-determinism, leads to a different semantics than when the monads are transformed the other way round.

Duponcheel [8] extended the work of [23] by representing the abstract syntax of a language as algebras, and interpreters as catamorphisms over such algebras to cater for extensible syntax. This is similar to the Object Algebra style employed in this paper.

A different approach to extensible interpreters was pioneered by Cartwright and Felleisen [3]. They present *extended direct semantics*, allowing orthogonal extensions to base denotational definitions. In this framework, the interpreters execute in the context of a global authority which takes care of executing effects. A continuation is passed to the authority to continue evaluation after the effect has been handled. In extended direct semantics, the semantic function $\mathcal{M}$ has a fixed signature $Exp \to Env \to C$ where C is an extensible domain of computations. The fixed signature of $\mathcal{M}$ allows definitions of language fragments to be combined.

Kiselyov et al. [20] generalized the approach of [3], allowed the administration functions to be modularized as well, and embedded the framework in Haskell using open unions for extensible syntax. In particular, the extensible effects approach does not suffer from the ordering problem of monad transformers.

Implicit Propagation Implicit propagation has been researched in many forms and manifestations. The most related treatment of implicit propagation is given by Lewis et al. [22], who describe implicit parameters in statically typed, functional languages. A difference to our approach is that implicit parameters cannot be retro-actively added to a function: a top-level evaluation function would still need to declare the extra context information, even though its value is propagated implicitly.

Another way of achieving implicit propagation in functional languages is using extensible records [21]. Functions consuming records may declare only the fields of interest. However, if such a function is called with records containing additional fields, they will be propagated implicitly.

Implicit propagation bears similarity to dynamic scoping, as for instance, found in CommonLisp or Emacs Lisp. Dynamic scoping is

[2] `http://docs.scala-lang.org/tutorials/tour/implicit-parameters.html`

a powerful mechanism to extend or modify the behavior of existing code [14]. For instance, it can be used to implement aspects [7] or context-oriented programming [17].

Another area where implicit propagation has found application is in language engineering tools. For instance, [28] introduced scoped dynamic rewrite rules to propagate down dynamically scoped context information during a program transformation process. Similarly, the automatic generation of copy rules in attribute grammars is used to propagate attributes without explicitly referring to them [19].

Finally, the implicit propagation conventions applied in the context of I-MSOS [24], have been implemented in DynSem, a DSL for specifying dynamic semantics [27]. In both I-MSOS and DynSem, propagation is made explicit by transforming semantic specifications.

8. Conclusion

Component-based language engineering would bring the benefits of reuse to the construction of software languages. Instead of building languages from scratch, they can be composed from reusable building blocks. In this work we have presented a design for modular interpreters that support a high level of reuse and extensibility. Modular interpreters are structured as Object Algebras, which support modular, type safe addition of new syntax as well as new interpretations. Different language constructs, however, may have different context information requirements (such as environments, stores, etc.), for the same semantic interpretation (evaluation, type checking, pretty printing, etc.).

We have presented **implicit context propagation** as a technique to eliminate this incompatibility by automatically *lifting* interpretations requiring n context parameters to interpretations accepting $n + 1$ context parameters. The additional parameter is implicitly propagated, through the interpretation that is unaware of it. As a result, future context information does not need to be anticipated in language components, and opportunities for reuse are increased.

Implicit context propagation is simple to implement, does not require advanced type system features, fully respects separate compilation, and works in mainstream OO languages like Java. We have shown how the pattern operates in the context of overriding, mutable context information, exception handling, languages with multiple syntactic categories, generic desugaring and interpretations other than dynamic semantics. Furthermore, the code required for lifting can be automatically generated using a simple annotation-based code generator. Our modular implementation of Featherweight Java with state shows that the pattern enables an extreme form of modularity, bringing the vision of a library of reusable language components one step closer.

One aspect that requires future work is to evaluate the performance impact of lifting. Since lifting is based on creating intermediate closures, lifted interpreters could be significantly slower than directly implemented base interpreters. Other directions for further research include the integration of concrete syntax (cf. [11]), and the application of implicit context propagation in the area of DSL engineering. We expect that DSL interpreters require a much richer and diverse set of context parameters, apart from the standard environment and store idioms. Finally, we will investigate the design of a library of reusable interpreter components as a practical, mainstream analog of the library of fundamental constructs of [4].

References

[1] L. Allison. Direct semantics and exceptions define jumps and coroutines. *Information Processing Letters*, 31(6):327–330, 1989.

[2] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *SCALA*, pages 3:1–3:10. ACM, 2013.

[3] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994.

[4] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*, pages 132–179. Springer, 2015.

[5] T. Cleenewerck. Component-based DSL development. In *GPCE*, pages 245–264. Springer, 2003.

[6] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universteit Brussel, 2007.

[7] P. Costanza. Dynamically scoped functions as the essence of AOP. *ACM SIGPLAN Notices*, 38(8):29–36, 2003.

[8] L. Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. 1995. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.7093`.

[9] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[10] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pages 171–183. ACM, 1998.

[11] M. Gouseti, C. Peters, and T. v. d. Storm. Extensible language implementation with Object Algebras (short paper). In *GPCE*, 2014.

[12] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3): 202–220, 2009.

[13] L. T. Hansen. Syntax for dynamic scoping, 2000. URL `http://srfi.schemers.org/srfi-15/srfi-15.html`. SRFI-15.

[14] D. R. Hanson and T. A. Proebsting. Dynamic variables. *ACM SIGPLAN Notices*, 36(5):264–273, 2001.

[15] J. Harm, R. Lämmel, and G. Riedewald. The language development laboratory (LDL). In *Selected papers from the 8th Nordic Workshop on Programming Theory (NWPT'96)*, pages 77–86, 1997.

[16] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.

[17] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[18] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, pages 132–146. ACM, 1999.

[19] U. Kastens. The GAG-system: A tool for compiler construction. 1984. URL `http://digital.ub.uni-paderborn.de/hs/download/pdf/41963`.

[20] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices*, 48(12):59–70, 2013.

[21] D. Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.

[22] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118. ACM, 2000.

[23] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343. ACM, 1995.

[24] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.

[25] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: practical extensibility with Object Algebras. In *ECOOP*, pages 2–27. Springer, 2012.

[26] G. L. Steele Jr. Building interpreters by composing monads. In *POPL'94*, pages 472–492. ACM, 1994.

[27] V. Vergu, P. Neron, and E. Visser. DynSem: A DSL for dynamic semantics specification. In *RTA*, LIPICS, 2015.

[28] E. Visser. Scoped dynamic rewrite rules. *Electronic Notes in Theoretical Computer Science*, 59(4):375–396, 2001.

[29] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.