Modular Interpreters with Implicit Context Propagation

Pablo Inostroza[a], Tijs van der Storm[a]

[a] *Centrum Wiskunde & Informatica (CWI)*
Amsterdam, The Netherlands

Abstract

Modular interpreters are a crucial first step towards component-based language development: instead of writing language interpreters from scratch, they can be assembled from reusable, semantic building blocks. Unfortunately, traditional language interpreters can be 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, base interpreters can be retro-actively *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 modular and extensible language development. The technique works in mainstream object-oriented languages, does not sacrifice type safety or separate compilation, and can be easily automated, for instance using macros in Scala or dynamic proxies in Java. We illustrate implicit context propagation using a modular definition of Featherweight Java and its extension to support side-effects, and an extensible domain-specific language for state machines. We finally investigate the performance overhead of lifting by running the DeltaBlue [13] benchmark program in Javascript on top of a modular implementation of LambdaJS [16], and a dedicated micro-benchmark. The results show that lifting makes interpreters roughly twice as slow because of additional call overhead. Further research is needed to eliminate this performance penalty.

1. Introduction

Component-based language development promises a style of language engineering where languages are constructed by assembling reusable building blocks instead of writing them from scratch. This style is particularly attractive in the context of language-oriented programming (LOP) [44], 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, ex-

Email addresses: `pvaldera@cwi.nl` (Pablo Inostroza), `storm@cwi.nl` (Tijs van der Storm) pressions, 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 [35] are a design pattern that supports type-safe extensibility of both abstract 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 these interfaces over concrete types representing the semantics. Adding new syntax corresponds to modularly extending the generic interface, and any pre-existing operation. 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 or dynamic proxies [37]. Finally, two case studies concerning a simple DSL and a simplified programming language illustrate the flexibility offered by implicit context propagation in modularizing and extending languages.

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 interpreters (Section 3).
- We show the versatility of the technique by elaborating on how implicit context propagation is used with delayed evaluation, overriding, mutable context information, exception handling, continuation-passing style, languages with multiple syntactic categories, generic desugaring of language constructs and interpretations other than dynamic semantics (Section 4).
- We present a simple, annotation-based Scala macro to generate boilerplate lifting code automatically and show how lifting can be implemented generically using dynamic proxies in Java (Section 5).

- To illustrate the usefulness of implicit context propagation in a language-oriented programming setting, we provide a case study of extending a simple language for state machines with 3 new kinds of transitions (Section 6).

- The techniques are furthermore illustrated using an extremely modular implementation of Featherweight Java with state [24, 11]. This allows us to derive 127 hypothetical variants of the language, out of 7 given language fragments (Section 7).

- Interpreter lifting introduces additional runtime overhead. We investigate this overhead empirically by running the DeltaBlue [13] benchmark in Javascript on top of a modular implementation of LambdaJS (λ_{JS}) [16]. The results show that a single level of lifting makes interpreters roughly twice as slow. Executing a dedicated loop-based micro-benchmark shows that additional call overhead is the prime cause of the slow down (Section 8).

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#.

This paper extends an earlier paper [25] with additional examples of implicit context propagation in Sections 4.1 and 4.5, the implementation of lifting using dynamic proxies (Section 5.2), a new case study (Section 6), an investigation of the performance overhead of lifting (Section 8), and an expansion of the related work discussion in Section 9.

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 as "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

[1]All code examples are in Scala [34] (`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.

	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 at 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.

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 [35] style, providing the essential ingredient to solve the problem using implicit context propagation.

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.

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 nullary 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, this trait cannot be composed with EvArith because the carrier types are different: EvArith is defined over Ev whereas EvVar is defined over 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 also support modular extension with new 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.

3. Implicit Context Propagation

We have seen how the incompatibility between Object Algebras defined over different function types precludes 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.

```
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

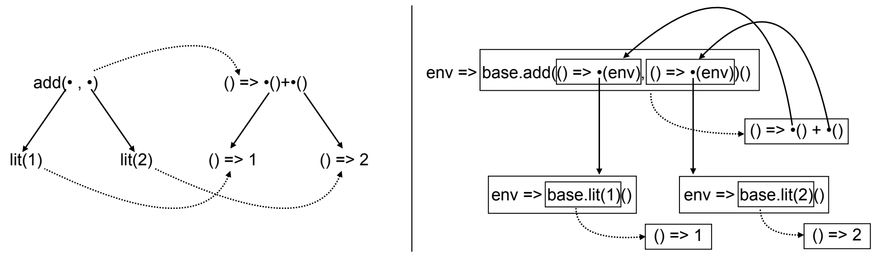

Figure 2: The left column shows how the expression add(lit(1), lit(2)) is mapped to its denotation by EvArith; the nodes in the tree are of type Ev (() => Val). On the right, the result of lifting the denotation produced by EvArith to the type EvE (Env => Val) to propagate environments. The dotted arrows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

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 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 corresponds to the argument of the closure denoted by 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 EvEArith-Binding 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 $j \in 1, ..., n$.

This 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 macro-based code generator discussed in Section 5 supports inserting the parameter anywhere in the list.

```
trait Alg$_{(T,U*)\Rightarrow V}$ extends Alg[(T, U⋆) => V] {

  val base = new Alg$_{U*\Rightarrow V}$ {}

  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.

4. Working with Lifted Interpretations

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

4.1. Delaying Evaluation

The interpreter for arithmetic expressions introduced in Section 2.2 is defined in terms of the carrier type `() => Val`. Accordingly, the algebra produces closures that need to be applied in order to obtain the final result. In this case, however, the denotations could have simply had the type `Val`, as there is no need for delayed evaluation in this language module.

In a certain sense using the carrier type `() => Val` for arithmetic expressions represents a form of anticipation: it is expected that arithmetic expressions will be used together with expressions that do require delayed evaluation. For instance, defining conditional expressions in an eager host language like Scala requires delayed evaluation, otherwise both branches of the conditional are evaluated. This is clearly not intended, especially in the presence of side effects.

It turns out, however, that lifting can be used to delay evaluation on top of an algebra that computes results eagerly. Consider the following implementation of `Arith` expressions:

```
trait ValArith extends Arith[Val] {
  def add(l: Val, r: Val) = IntVal(l + r)
  def lit(n: Int) = IntVal(n)
}
```

The carrier type is simply `Val` and expression evaluation is “immediate”: when expressions are constructed over this algebra, they are actually immediately evaluated without creating any intermediate closures. `ValArith` can now be lifted to produce thunks instead of values, as follows:

```
trait DelayedValArith extends Arith[Ev] {
  private val base = new ValArith {}

  def add(l: Ev, r: Ev): Ev = () => base.add(l(), r())
  def lit(n: Int): Ev = () => base.lit(n)
}
```

This version of arithmetic evaluation corresponds to the original EvArith introduced in Section 2.2 and can be composed with, for instance, implementations of conditional expressions. Lifting was characterized as interleaving the construction of the expressions on the base algebra with evaluating them. In this case, one can see that construction and evaluation actually coincide.

4.2. Overriding Interpretations: Dynamic Scoping

The propagation of environments presented in Section 3 obeys lexical scoping rules for all implicitly-propagated parameters. 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 [23], or simply dynamically scoped variables [20].

Consider the following language fragment for introducing dynamically scoped variables, similar to fluid-let in Scheme [19]:

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

The construct dynlet binds a variable x to a value in both the lexical and dynamic environment. The dynamic variable can then be referenced in the scope of dynlet using the ordinary vari of the Binding fragment (cf. Figure 1).

As an example of the dynamically scoped propagation, consider the following example term defined over the combination of Arith, Binding, and DynLet. The left column shows the abstract syntax, the right column shows the same program in pseudo concrete syntax:

```
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"))))))
```

```
dynlet x = 1 in
  let f = λ _ . x + 1 in
    dynlet x = 2 in
      let z = (dynlet x = 3 in f(1)) in
        z + x
```

This program dynamically binds x to 1, in the scope of the let which defines f as a lambda dynamically referring to x. The value of x thus depends on the dynamic scope when the closure f 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. This application is itself inside another dynamic let, yet again redefining x. So the result of this 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.

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: String, v: EvEE, b: EvEE)
    = (env, denv) => { val y = v(env, denv); b(env + (x -> y), denv + (x -> y)) }
}
```

Notice that since the static and the dynamic environment both have the same type, the signature of the carrier type makes the interpretation order-dependent. We discuss strategies for disambiguation in these scenarios when presenting automated lifting in Section 5.

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): 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): 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.

4.3. 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 context that should not be lexically scoped is a store for modeling side effects. In this case, however, the parameter should also not obey stack discipline as it did for the dynamic environment. Instead, we achieve this 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 a 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[2]:

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

To compose 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)
   }
 ...
}
```

[2]For brevity, we have elided the actual, straightforward implementation of the storage constructs.

```
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): Ev
    = () => try { l() } catch { case _:Fail => r() }

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

Figure 4: Implementing local backtracking with exception handling.

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

Since simulating mutable context information by threading immutable data breaks modularity and modularity is key to our approach, we instead depend on the support for mutable data structures in the host language in order to represent side-effects in the object language.

4.4. 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): 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):

```
type EvK = (Val => Unit) => Unit

trait EvKArith extends Arith[EvK] {
  def add(l: EvK, r: EvK): EvK = k => l(v1 => r(v2 => k(IntVal(v1+v2))))
  def lit(n: Int): EvK = k => k(IntVal(n))
}
```

Figure 5: CPS evaluator for arithmetic expressions.

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

Notice, however, that we face a limitation if we want to compose backtracking with mutable context, e.g., the store. Since the store inherits the mutable semantics from the host language, it is not trivial to customize the interaction with the backtracking behavior. For instance, in order to support transaction-reversing choice we might naively assume that we just need to lift EvChoice to EvSChoice (an interpreter that accepts the store). Unfortunately, the resulting interpreter has the wrong semantics. It does not rollback the transactions upon failure, because the captured stores are mutated at the level of the host language. An alternative is to override the semantics of the Choice interpreter in order to keep track of mutable stores at each choice point (like dynamic let required overriding lambda and apply). This leads to a contrived implementation that works around the limitations of the host language semantics. Moreover, in the case there are interactions with more mutable parameters, all of them must be considered, leading to more overriding. In summary, lifting does not automatically handle the interaction between non-local control flow extensions and extensions that require mutable parameters.

4.5. Continuation-Passing style

In the introduction, we argue that one of the benefits of our approach to language modularity is that the semantic components can be written in direct style (as opposed to continuation-passing style or monadic style). However, some language features might in fact require a different formulation of interpreters. For instance, not all non-local control flow features are conveniently expressed using exception handling. One case in point is Scheme's call-with-current-continuation (callcc) [1], which allows arbitrary capturing of the "remainder of a computation" (the continuation). We show that interpreter lifting can still be applied if all interpreters are coded in continuation-passing style (CPS) [39].

Consider, for instance, CPS evaluators for the Arith language shown in Figure 5. The carrier type EvK is defined as a function from a continuation (a function consuming a value) to the unit type (Unit in Scala). CPS interpreters never return a value, but always call the given continuation to continue evaluation. For instance, add is defined by a call to the l argument, passing a new continuation, which, when called, invokes the r argument with yet another continuation. If that continuation is invoked, the original continuation k is invoked with the result of the addition. The key aspect of CPS interpreters is that all forms of sequencing are made completely explicit in terms of function composition.

Assuming that we want to propagate an environment through the evaluation of EvKArith evaluator in order to combine it with binding expression, then the propagating strategy is the same as the one we have already observed:

```
trait EvEKArith extends Arith[EvEK] {
  private val base = new EvKArith {}

  def add(l: EvEK, r: EvEK): EvEK = (e, k) => base.add(k_ => l(e, k_), k_ => r(e, k_))(k)
  def lit(n: Int): EvEK = (e, k) => base.lit(n)(k)
}
```

The base add function receives functions of type EvK which call the original l and r propagating the environment e.

Note that the current continuation (k) acts just like any other context parameter. It is therefore tempting to think that lifting could be used to convert a direct style interpreter into a CPS interpreter. Unfortunately, direct style interpreters depend on the host language for their evaluation strategy. As a result, it is impossible to recover the implicit sequencing going on in base interpreters and make it explicit using CPS.

4.6. 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 a 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 directly to base.ifThen.

An alternative extension is to add 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, and thus 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.

4.7. 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 or 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 generically implemented 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 may remain unexpanded (such as for pretty printing).

4.8. Multiple Interpretations: Pretty Printing

Object Algebras support the modular extension of both syntax and operations. Thus, implicit propagation can be applied to interpretations of a language other than dynamic semantics. Examples include type checking, other forms of static analysis, and pretty printing.

Consider the example of pretty printing. Here is 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 expressions does not involve the notion of indentation. However, to pretty print the ifThen construct of Section 4.6 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 first introduce a Scala macro-based code generator for single-sorted algebras, which generates the lifting code automatically. Second, we describe how dynamic proxies in Java can be used to perform lifting at runtime.

5.1. Lift using a Scala Macro

The code generator is invoked by annotating an empty trait. In the compiled code, the code generator fills in the required lifting methods for this annotated trait. 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 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 or type safety: the generator only inspects the interfaces of the types that are specified in the annotation without needing access to the source code where these types are defined.

The @lift annotation is implemented as a Scala macro annotation [3]. Macro annotations are definition-transforming macros that can be used to generate boilerplate code at definition level. Figure 6 shows the implementation of the @lift macro annotation. The class extending StaticAnnotation defines a macro annotation and defines a method macroTransform that contains the logic of the compile-time transformation. The companion object contains the impl method referenced by macroTransform. This method receives as arguments a collection of annottees that represents all the definitions in the scope of the annotation. First, the four annotation arguments are extracted as trees and then type checked in order to obtain their types. Then, an intermediate representation is created from these types using a custom InternalImporter. The annottees are then pattern-matched in order to get the trait's name. The crucial step is calling the liftTraitTo method on the representation of the trait that corresponds to the algebra interface (instance of the class Trait, not shown). This method receives the name of the trait being transformed, the source carrier type, the trait corresponding to the base algebra implementation, and the target carrier type, and performs all the necessary transformations in order to return the representation of the lifted trait. This resulting trait is then serialized, concluding the transformation process.

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 permutation logic is encoded in the already mentioned method liftTraitTo. This is important for components that need to be "mutually lifted". Consider a component which is lifted from Sto=>Val to (Env,Sto)=>Val. To combine this component with a component of type Env=>Val, the

```
class lift[ALG, FROM, BASEALG, TO] extends StaticAnnotation{
  def macroTransform(annottees: Any*) = macro lift.impl
}

object lift{

  def impl(c: whitebox.Context)(annottees: c.Expr[Any]*) = {
    import c.universe._

    // Get the annotation arguments as trees
    val (algAST: Tree, srcFunAST: Tree, baseAlgAST: Tree, tgtFunAST: Tree)
      = c.macroApplication match{
        case q"new lift[$a, $from, $baseA, $to].macroTransform($_)" =>
          (a, from, baseA, to)
        case _ =>
          c.abort(c.enclosingPosition, "Invalid type parameters")
    }

    // Get the types of the annotation arguments
    val (algType: Type, srcType: Type, baseImplType: Type, tgtType: Type)
      = Checker.typeCheck(c)(algAST, srcFunAST, baseAlgAST, tgtFunAST)

    // Create importer that allows to create intermediate representations of traits and
    // function types based on the annotation arguments
    val (alg: Trait, srcFun: FunType, baseImpl: Trait, tgtFun: FunType)\newline
      = InternalImporter.importTypes(c)(algType, srcType, baseImplType, tgtType)}

    // At least one annottee must be a trait
    annottees.map(_.tree) match {
     case (q"$mods trait $name") :: Nil => {

       // This is the code that triggers the lifting on the intermediate representations
       val lifted: Trait = alg.liftTraitTo(name.decoded, srcFun, tgtFun, "base"+alg.name)

       // Serialize the lifted trait and return it as the result of the transformation
       val result: Expr[Any] = Render.serialize(c)(mods, lifted, alg, baseImpl, srcFun)
       result
     }
     case _ => c.abort(c.enclosingPosition, "Invalid annottee")
    }
  }
}
```

Figure 6: The lift macro annotation for lifting Object Algebra at compile time

```
interface ExpAlg<E> {
    E lit(int n);
    E add(E l, E r);
}

@FunctionalInterface
interface IEval {
    int eval();
}
```

```
interface EvalExp extends ExpAlg<IEval> {
    default IEval lit(int n) {
        return () -> n;
    }

    default IEval add(IEval l, IEval r) {
        return () -> l.eval() + r.eval();
    }
}
```

Figure 7: Arithmetic expression evaluation using Java 8 functional interfaces and default methods

latter should be lifted to (Env,Sto)=>Val as well, but in this case, the parameter is added at the end.

The current version of @lift does not disambiguate parameters with the same type. It is always possible to create artificial wrapping types to distinguish between two context objects of the same type. It is, however, also conceivable to implement this by requiring the user to provide the disambiguation information in the annotation. This is an opportunity for future work.

5.2. Dynamic Lifting in Java

The Scala macro approach generates method implementations for a trait introduced by the programmer. Java does not have macro facility that supports a similar style of code generation. One would think the Java annotation processing framework [38] could be used for this, but, unfortunately, annotation processing only allows the generation of new classes or types, but not filling in the implementations of existing (abstract) classes or interfaces. As a result, client code becomes dependent on generated types which, in turn introduces temporal dependencies within the build cycle of the code.

Fortunately, it is also possible to perform lifting dynamically using the concept of dynamic proxies [37]. Since Java 8, function types are represented using *functional interfaces*: interfaces with a single method that acts as the "apply" method of functions. Dynamic proxies can be used to provide a generic implementation of such interfaces. Thus, instead of generating methods that encode the lifting of an interpreter, the context parameters are implicitly propagated at runtime.

Figure 7 shows a simple expression evaluator in Object Algebra style using Java 8 functional interfaces and interface default methods. Note how the closure notation automatically creates objects that are instances of the IEval interface.

Of course, another language fragment could require additional context parameters, which are not reflected in the type IEval. For instance, denotations requiring a environment could be represented by the following interface:

```
@FunctionalInterface
interface IEvalEnv {
    int eval(Env env);
}
```

```
interface EvalExpEnv extends ExpAlg<IEvalEnv> {
    static final ExpAlg<IEval> base = new EvalExp() {};

    default IEvalEnv lit(int n) {
        return lift(env -> base.lit(n));
    }

    default IEvalEnv add(IEvalEnv l, IEvalEnv r) {
        return lift(env -> base.add(lower(l, env), lower(r, env)));
    }

    static IEvalEnv lift(Function<Env, IEval> f) {
        return (IEvalEnv) Proxy.newProxyInstance(IEvalEnv.class.getClassLoader(),
                new Class<?>[] { IEvalEnv.class }, (p, m, as) -> f.apply((Env) as[0]).eval());
    }

    static IEval lower(IEvalEnv e, Env env) {
        return (IEval) Proxy.newProxyInstance(IEval.class.getClassLoader(),
                new Class<?>[] { IEval.class }, (p, m, as) -> e.eval(env));
    }
}
```

Figure 8: Manually lifting arithmetic expression evaluation in Java to propagate environments

We need an implementation of ExpAlg over the carrier type IEvalEnv. Figure 8 shows a slightly contrived implementation of expression evaluation with dynamic environment propagation using dynamic proxies. If this code would be implemented by hand there would be no need for dynamic proxies: the implementation would simply follow the pattern of the manual Scala lifting in Section 3.1. However, the example illustrates how dynamic proxies can be used to simulate functions in Java.

The lifting is realized using the helper methods lift and lower. Both methods return proxies created using java.lang.reflect.Proxy::newProxyInstance. This method takes a class loader, an array of interfaces the proxy is supposed to export, and an instance of the java.lang.reflect.InvocationHandler interface to handle method requests on the proxy. Since InvocationHandler is a functional interface in Java 8, one can directly provide closures as invocation handlers. The returned object will behave as an instance of all the provided interface types, but all method invocations will be routed to the invocation handler.

The lift method takes a function from the extra parameter (Env) to IEval and uses it to create proxy objects of type IEvalEnv. The returned proxy object provides the extra argument (as[0]) to the closure f to obtain an IEval object and then calls eval() on it. The lower function has the inverse effect of lift: it turns objects of a "larger" type (e.g., IEvalEnv) into objects of a smaller type (e.g., IEval), again using proxies.

The closure f provided to lift abstracts over how to turn a base interpreter into the lifted type given the extra parameter which should be propagated. For instance, in the case of add, the provided function to lift calls the base.add constructor with lowered versions of l and r. Lowering is realized by the helper method lower, which turns IEvalEnv objects into IEval objects. Whenever eval is called on such an IEval, it delegates back to the original IEvalEnv providing the extra parameter env that was captured when lower was invoked.

The interface EvalExpEnv can be composed with other interfaces over the same carrier type, similar in style to Scala trait inheritance. However, the propagation code is specific for two carrier types (i.e. IEval and IEvalEnv). This problem is solved by introducing yet another level of dynamic proxies, this time at the level of the algebras themselves.

Since signatures of Object Algebras are represented by Java interfaces, dynamic proxies can also be used to simulate Object Algebras themselves. The following lifter method creates such "lifting" algebras automatically:

```
static <F, S, T> F lifter(Class<F> ialg, Class<S> source, Class<T> target, F base) {
    return (F) Proxy.newProxyInstance(ialg.getClassLoader(), new Class<?>[] {ialg},
            new Lifter<>(source, target, base));
}
```

Intuitively, this method turns an Object Algebra of type F<S> into an algebra of type F<T>, assuming that both S and T are functional interfaces, where the single method in T has one extra parameter. Method invocations on the resulting algebra of type F<T> are handled by the invocation handler Lifter which will create proxy objects delegating to the base algebra (of type F<S>) and propagating the extra parameter behind the scenes.

The Lifter class is shown in Figure 9. The entry point of the Lifter class is always an invocation of a factory method (e.g., add, lit, etc.) which will be handled by the invoke method from Java's InvocationHandler interface. The invoke method receives the current proxy object, the called method and the method's arguments (kids). It immediately returns the result of calling lift which produces the desired target type T.

The lift method follows the same pattern as the lift method of Figure 8. The main difference is that the argument f now accepts both a method object representing the method supporting the extra parameter in addition to the extra argument itself. Whereas the closure provided to lift in Figure 8 directly called the appropriate factory method, in this case the base interpreter is created using reflection, and the arguments are lowered using a loop in lowerKids: for every constructor argument in kids that is an instance of the target type T, a proxy is created in the lower method. This proxy will call the method extEval on the original T object (kid), extending the list of arguments with the extra argument originally received in the closure provided to lift; the extend helper method creates a copy of args with the extra object appended to it.

After the evaluator object is obtained from the f closure in lift, the corresponding method in type S is looked up using reflection on source. We simply look for a method with the same name, but with one fewer parameter, and then invoke it on the base evaluator ignoring the extra parameter. Finally, this eval method is invoked on the base evaluator object, ignoring the last element of the extended argument array (extArgs).

Using the lifter method any Object Algebra F<S> defined over a functional interface **interface** S { U m(C_1 c_1, ..., C_n c_n); } can now be converted to an algebra F<T> defined in terms of **interface** T { U m(C_1 c_1, ..., C_n c_n, C_{n+1} c_{n+1}); }. The extra parameter c_{n+1} will be dynamically propagated.

Implicit context propagation using dynamic proxies operates at the level of runtime objects, whereas the Scala macro operated at the trait level. In other words: the result of lifter is a runtime object, not a trait or class. As a result, it is not possible anymore to compose language fragments using trait/interface inheritance. Once again, dynamic proxies can be used to mitigate the problem.

```
class Lifter<S, T, F> implements InvocationHandler {
    private final Class<S> source;
    private final Class<T> target;
    private final F base;

    Lifter(Class<S> s, Class<T> t, F base) {
        this.source = s;
        this.target = t;
        this.base = base;
    }

    public T invoke(Object proxy, Method constructor, Object[] kids) {
        return lift((extEval, extra) -> (S)constructor.invoke(base, lowerKids(extEval, kids, extra)));
    }

    private T lift(BiFunction<Method, Object, S> f) {
        return proxy(target, (p, extEval, extArgs) -> {
            S evaluator = f.apply(extEval, extArgs[extArgs.length - 1]);

            // Get the method in S corresponding to extEval in T
            Method eval = source.getMethod(extEval.getName(),
                    Arrays.copyOf(extEval.getParameterTypes(), extArgs.length - 1));

            // Invoke it on the base interpreter ignoring the extra parameter.
            return eval.invoke(evaluator, Arrays.copyOf(extArgs, extArgs.length - 1));
        });
    }

    private S lower(Method extEval, T kid, Object extra) {
        return proxy(source, (p, eval, args) -> extEval.invoke(kid, extend(args, extra)));
    }

    private Object[] lowerKids(Method eval, Object[] kids, Object extra) {
        Object lowered[] = Arrays.copyOf(kids, kids.length);
        for (int i = 0; i < kids.length; i++)
            if (target.isInstance(kids[i]))
                lowered[i] = lower(eval, (T)kids[i], extra) ;
        return lowered;
    }
}
```

Figure 9: The Lifter class for lifting Object Algebra using dynamic proxies

The following code defines a union combinator which multiplexes method invocations from a single Object Algebra interface between actual implementations of subsets of the interface (see [36, 15] for similar incarnations of this idea):

```
static <T> T union(Class<T> ialg, Object ...algs) {
    return (T) Proxy.newProxyInstance(ialg.getClassLoader(), new Class<?>[] { ialg },
            (x, m, args) -> {
                for (Object alg : algs)
                    try { return m.invoke(alg, args); } catch (Exception e) { continue; }
                throw new UnsupportedOperationException("no such method"); });
}
```

Let's assume we have two language fragments, the simple arithmetic expressions and a language for binding constructs. At the interface level these fragments can be combined using interface extension:

```
interface ExpBindAlg<E> extends ExpAlg<E>, BindAlg<E> { }
```

The dynamically lifted version of EvalExp can now be combined with an implementation of the binding fragment (say, EvalBind):

```
ExpAlg<IEvalEnv> evalExp = lifter(ExpAlg.class, IEval.class, IEvalEnv.class, new EvalExp() {});
BindAlg<IEvalEnv> evalBind = new EvalBind() {};
ExpBindAlg<IEvalEnv> evalExpBind = union(ExpBindAlg.class, evalExp, evalBind);
```

The algebra evalExpBind can now be used to create expressions just like any ordinary algebra.

5.3. Discussion

We have presented two approaches to automate lifting: Scala macros and Java dynamic proxies. As the macro transformation is a compile-time mechanism, all the type information is available when generating the code for the lifted algebras, and thus all the generated code is type safe. On the other hand, in the case of the dynamic proxies, the lifting is realized at runtime and requires casts in order to make the proxied object conform to the lifted interfaces. All the code at the points where explicit casting occur (e.g., the cast that returns a proxied lifted object from the lifter method) is unsafe. Having said that, this unsafety is limited to code that is provided by the framework as a reusable mechanism for dynamic lifting. Provided that the framework is correctly implemented, the end user is not affected by it.

6. Case study 1: Extending a DSL for State Machines

In this section we present a case study, based on an extensible DSL for state machines, inspired by the example DSL introduced in [12]. The exploration of this DSL is motivated by the following considerations.

First, state machines emphasize the DSL perspective: state machines are different from expression-oriented or even statement-oriented programming languages. The contexts involved are not just environments or stores, but may contain arbitrary interfaces to some unspecified outside world.

Second, whereas the examples of Section 4 are mostly single sorted expression languages, state machines are inherently many sorted, since such a language typically

```
trait Stm[M, S, T] {
  def machine(name: String, states: Seq[S]): M
  def state(name: String, transitions: Seq[T]): S
  def transition(event: String, target: String): T
}
```

Figure 10: The abstract syntax of the state machine language

involves at least the syntactic categories of state machines, states and transitions. Thus, this case study illustrates more clearly that adding context parameters to nested AST types requires lifting surrounding AST types, similar to how statement interpreters required lifting in Section 4.6.

Third, the state machine example strongly illustrates that implicit context propagation can be seen as a form of "scrapping your boilerplate" [28]. In a sense, implicit context propagation supports the creation of *structure shy* [32] language extensions: adding language features deep down within the syntactic structure does not require to change the definitions of the semantics of surrounding, context unaware node types. Similar concerns were addressed, for instance, by implicit parameters (as described in [30]): many cases in the definition of a recursive function do not use the context information, but some leaves of the recursion need the information.

Finally, instead of building upon the assumption that a language is constructed by assembling the smallest possible building blocks, the state machine case study is presented from the perspective of language extension. Given an existing definition of a state machine language, we will extend it with three new types of transitions: conditional transitions, transitions with token output, and conditional transitions with token output.

6.1. State Machines

A state machine consists of a list of named state definitions. Each state contains a list of outgoing transitions. A transition fires on an event (a string) and transitions to a target state (identified by name). The abstract syntax of the state machine language is shown in Figure 10.

An interpreter for the base language is shown in Figure 11. The carrier type EvM captures functions from the current state and event to the next state, if any. A machine simply finds the first state with a firing transition. A transition may fire if it is defined in the state we are in (st), and if the event ev matches the event in the transition. If a transition fires, the target state is returned.

As an example, consider a simple state machine controlling the opening and closing of doors:

```
def doors[M, S, T](alg: Stm[M, S, T]): M =
  alg.machine("doors", Seq(
     alg.state("closed", Seq(alg.transition("open", "opened"))),
     alg.state("opened", Seq(alg.transition("close", "closed")))))
```

This state machine can be used as follows:

```
type EvM = (String, String) => Option[String]

trait EvalStm extends Stm[EvM, EvM, EvM] {
  override def machine(name: String, states: Seq[EvM]): EvM
    = (st, ev) => states.map(_(st, ev)).find(_.isDefined).flatten

  override def state(name: String, transitions: Seq[EvM]): EvM
    = (st, ev) => if (name == st) transitions.map(_(st, ev)).find(_.isDefined).flatten else None

  override def transition(event: String, target: String): EvM
    = (st, ev) => Option(if (ev == event) target else null)
}
```

Figure 11: A simple interpreter for state machines. The carrier type EvM captures functions from current state and event to next state (if any).

```
trait TokenTrans[T] {
  def transition(event: String, target: String, tokens: Set[String]): T
}

trait CondTrans[E, T] {
  def transition(cond: E, event: String, target: String): T
}

trait CondTokenTrans[E, T] {
    def transition(cond: E, event: String, target: String, tokens: Set[String]): T
}
```

Figure 12: Three language extensions defining the abstract syntax transitions with token output, transitions with conditions, and transitions with both token output and conditions.

```
val stm = doors(new EvalStm {})
val Some(st1) = stm("closed", "open")
println(st1);
val Some(st2) = stm(st1, "close")
println(st2);
```

Executing the code will print out:

```
opened
closed
```

6.2. Modular Extension of State Machines

The abstract syntax of the three state machine extensions is shown in Figure 12. Each extension is defined in its own trait. TokenTrans defines transitions that output a set of tokens. CondTrans defines conditional transitions, introducing an additional E sort representing expressions. Finally, CondTokenTrans, combines both extensions to support conditional transitions with token output.

Transitions with Token Output. The definition of transitions with token output is as follows:

```
type EvTT = (String, String, Writer) => Option[String]

trait EvalTokenTrans extends TokenTrans[EvTT] {
  def transition(event: String, target: String, tokens: Set[String]): EvTT
    = (st, ev, w) => if (ev == event) {
        tokens.foreach(t => w.append(t + "\n")); Some(target)
      }
      else None
}
```

The type EvTT describes that transitions with token output require another context parameter, in this case a Writer object that the output tokens will be written to. The transition method then returns a function that outputs the given tokens whenever the transition fires.

To combine this language extension with the base interpreter of state machines shown in Figure 11, the latter has to be lifted to propagate the Writer parameter. The lifted version of EvalStm has the following structure:

```
trait LiftEvalStm extends Stm[EvTT, EvTT, EvTT] {
  private val base = new EvalStm {}

  def machine(name: String, states: Seq[EvTT]): EvTT
    = (st, ev, w) => base.machine(...)

  def state(name: String, transitions: Seq[EvTT]): EvTT
    = (st, ev, w) => base.state(...)

  def transition(event: String, target: String): EvTT
    = (st, ev, w) => base.transition(...)
}
```

Note that all three type parameters of Stm (M, S, T) are bound to the extended signature EvTT since the writer object needs to be provided from the top in order to be propagated down to the level of transitions.

Given the syntax traits Stm and TokenTrans, state machines may now contain transitions that declare tokens that will be output upon firing:

```
def doorsTokens[M, S, T](alg: Stm[M, S, T] with TokenTrans[T]): M =
  alg.machine("doors", Seq(
    alg.state("closed", Seq(alg.transition("open", "opened", Set("TK1OP", "TK2OP")))),
    alg.state("opened", Seq(alg.transition("close", "closed", Set("TK1CL", "TK2CL"))))))
```

To execute such extended state machines, the interpreters LiftEvalStm and EvalTokenTrans need to be combined:

```
val stm = doorsTokens(new LiftEvalStm with EvalTokenTrans {})
val w = new StringWriter()
val Some(st1) = stm("closed", "open", w)
...
```

Printing out the contents of the writer object w will contain the tokens as sequentially output by the new kind of transitions.

Conditional Transitions. The extension with conditional transitions follows a similar pattern as the extension with transitions outputting tokens. In this case, however, the extension introduces a new syntactic category for expressions. As a result, this extension also requires a separate language fragment defining the syntax and semantics of expressions. We assume this language is defined in its own trait Cond and that its interpreter EvalCond is defined over the type EvE = Env => Boolean. As a result, the propagated context parameter is the environment used to evaluate a transition's condition.

The semantics of conditional transitions are then defined as follows:

```
type EvET = (Env, String, String) => Option[String]

trait EvalCondTrans extends CondTrans[EvE, EvET] {
  override def transition(cond: EvE, event: String, target: String): EvET
    = (env, st, ev) => Option(if (ev == event && cond(env)) target else null)
}
```

Creating state machines with conditional transitions is now defined over the algebra interfaces Stm, Cond and CondTrans. Executing such state machines combines the lifted base interpreter to propagate the environment, with EvalCond and EvalCondTrans.

Conditional Transitions with Token Output. The final extension combines both conditions and token output in transitions. Although this extension can be considered in isolation, it makes intuitively more sense to allow this kind of transition to coexist with the two extensions described above. As a result, adding conditional transitions with token output requires two-level lifting of the base interpreter. In other words, one of the lifted interpreters for the previous extensions is lifted once again to propagate the additional context (i.e. writer object or environment).

Additionally, to combine this extension with the previous extensions, both interpreters EvalTokenTrans and EvalCondTrans need to be lifted. EvalTokenTrans needs to propagate the environment, and EvalCondTrans needs to propagate the writer object. Note however, that the interpreter for conditions (EvalCond) does not require lifting.

The only code that remains to be written is the interpreter for the new kind of transitions itself:

```
type EvETT = (Env, String, String, Writer) => Option[String]

trait EvalCondTokenTrans extends CondTokenTrans[EvE, EvETT] {
  override def transition(cond: EvE, event: String, target: String, tokens: Set[String]): EvETT
    = (env, st, ev, w) => if (ev == event && cond(env)) {
        tokens.foreach(t =>w.append(t + "\n"))
        Some(target)
      }
      else None
}
```

Note that this definition duplicates the logic of ordinary transitions, conditional transitions and transitions with token output. This may seem unfortunate, but understandable: the new kind of transition represents feature interaction between transition firing, condition evaluation and token output, which can never be automatically derived from the given interpreters.

Summary. To summarize, given the manually written language modules EvalStm, EvalTokenTrans, EvalCondTrans and EvalCondTokensTrans and an additional module defining conditional expressions (EvalCond), we can derive the following 7 language variants:

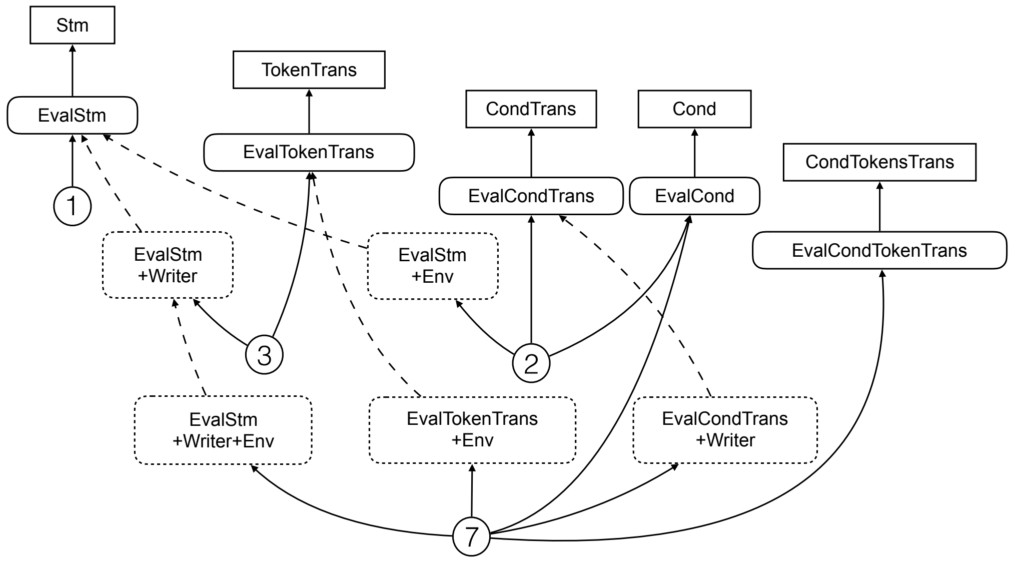

Figure 13: Modular extension of a state machine DSL with conditions and/or token output. Rectangles define syntactic constructs. Rounded rectangles are interpreters; dotted borders indicate lifted interpreters. Solid arrows represent trait inheritance and dashed arrows represent delegation inherent in lifting. Each circle represent an executable composition of modules.

1. Stm
2. Stm **with** CondTrans
3. Stm **with** TokenTrans
4. Stm **with** CondTokenTrans
5. Stm **with** TokenTrans **with** CondTokenTrans
6. Stm **with** CondTrans **with** CondTokenTrans
7. Stm **with** CondTrans **with** TokenTrans **with** CondTokenTrans

Compositions 1, 2, 3, and 7 make the most sense and are depicted graphically in Figure 13. Each solid rectangle defines a syntactic trait, the semantics of which is implemented in the rounded rectangles (interpreters); the solid arrows represent trait inheritance or extension. The dashed rounded rectangles represent liftings of interpreters, and the dashed arrows represent delegation to the base language. The circles represent compositions of language fragments.

7. Case Study 2: Modularizing Featherweight Java

To examine how implicit context propagation helps in modularizing a programming language implementation, we present a second case study using *Featherweight Java* (FJ) [24]. The case study consists of a modular interpreter for FJ and its extension to a variant that supports state (SFJ), inspired by [11].

The case study addresses two questions:

		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

- 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?

In this section, these questions are answered by analyzing the number of hypothetical languages that can be defined from the combination of SFJ fragments, and by counting the possible liftings.

7.1. 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 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 variables 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.
- 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.

Notice too 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. We have implemented the lifting using the @lift macro annotation discussed in Section 5.1. The relevant code is shown in Figure 14.

7.2. Analyzing Hypothetical Subsets of SFJ

The previous subsection detailed how the implementation of a complete language, in this case FJ and SFJ, can be constructed from assembling language fragments. Here we discuss hypothetical subsets of such languages. Even though these subsets might not (and probably will not) be meaningful in any practical sense, they illustrate the flexibility that implicit context propagation promote.

Table 3 shows how many interpreters can be derived per interpreter signature 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.

```
@lift[Field[_], CT => Obj, EvCT2ObjField, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjField

@lift[New[_], (CT, Sto) => Obj, EvCtSt2ObjNew, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjNew

@lift[Cast[_], CT => Obj, EvCt2ObjCast, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjCast

@lift[Call[_], (Obj, CT, Env) => Obj, EvSlfCtEnv2ObjCall, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjCall

@lift[Seq[_], () => Obj, Ev2ObjSeq, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjSeq

@lift[SetField[_], (CT, Sto) => Obj, EvCtSt2ObjSetField, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjSetField
```

Figure 14: Automatic lifting of SFJ language components using the @lift macro annotation

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.

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. Thus, 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.

8. Performance Overhead of Lifting

A lifted interpreter exhibits more runtime overhead than its equivalent non-lifted version. This is because the lifting works creating new closures at runtime to adapt the signatures of the arguments and adds additional call overhead to go from one closure to the other.

8.1. Benchmarking Realistic Code: Executing the DeltaBlue Benchmark

In order to have an idea of the impact of lifting, we performed an experiment using the DeltaBlue benchmark [13]. DeltaBlue represents an incremental constraint solver and is used to benchmark programming languages. DeltaBlue was originally developed in Smalltalk; in our experiment we use the Javascript version[3]. To execute the benchmark we developed a modular interpreter of λ_{JS}, based on the semantics described in [16].

Using the desugaring framework published at [17] the DeltaBlue Javascript code was desugared to λ_{JS} and input to our interpreter. To assess the performance impact of lifting we ran the benchmark using two interpreters: one that employed lifting, and one that propagated the context parameters explicitly, i.e., written in the "anticipation style" discussed earlier.

The interpreter that propagates all the context information explicitly represents our baseline, as we consider it to be a straightforward implementation that follows the interpreter pattern [14] (except it uses closures as AST objects).

In order to make the comparison as fair as possible regarding to the impact of lifting, both interpreters were modularized in logically-related units representing sublanguages of λ_{JS}, namely: *Binding*, *Control*, *Mutability*, *Core*, and *Exceptions*. This means that both styles of interpreters have the same trait inheritance structure.

In the case of the lifted interpreter, each module is defined using carrier signatures that consider only the needed context. For the non-lifted interpreter, however, every module is defined over the maximal signature, anticipating all required context information even though not all of it is required in each and every module. For instance, the *Control* module does not need any context because it only deals with syntactic forms

[3]See `https://github.com/xxgreg/deltablue`

Component	Expression Carrier Type
Binding	Env => Val
Control	() => Val
Core	Sto => Val
Exceptions	Env => Val
Mutable	Sto => Val

Table 4: Signatures of expression carrier types for the base components of the λ_{JS} implementation that depends on lifting

that represent control flow. While the lifted interpreter's carrier type `()->Val` reflects this, the non-lifted version is defined over the largest signature: the signature with both environment and store as context parameters (`(Sto, Env)->Val`).

Table 4 shows the signatures of the carrier type that corresponds to expressions, in the base components of the λ_{JS} implementation that depends on lifting. Since there is no single module defined over the largest signature, all modules have to be lifted to either propagate the store or to propagate the environment, or both. As a result, all modules exhibit some level of lifting, and thus all nodes created by the desugared DeltaBlue program will be affected by the performance overhead of lifting.

The benchmarks were executed on a MacBook Pro with OS X Yosemite (version 10.10.3) and 8 GB RAM, running on an Intel Core i5 CPU (2.7 GHz). We use Oracle's JVM (JDK 8u51), executed in server mode. The DeltaBlue benchmark can be run for a variable number of constraint solving iterations. We executed the program for an increasing number of iterations from 5 to 30 in steps of 5, taking 15 time measurements per number of iterations using Java `System.nanoTime()`.

Figure 15 shows a box plot of the execution times. For each number of iterations, we show the execution time for the non-lifted interpreter (white fill) and the lifted interpreter (gray fill). Calculating the difference in slowdown relative to the size, we observe that the slowdown is in the range of 63% and 81% with a median of 77.87%. We conjecture that the slowdown is due to the allocation of new closures at runtime and additional call overhead between the lifted closure and base closures. To zoom in on these effects, we now describe a micro-benchmark that partially confirms this hypothesis.

8.2. Micro-benchmark: Executing Lifted Interpreters in a Loop

To isolate the effect of lifting we created a micro-benchmark based on a simple expression language, containing literals, addition and a sum construct. The sum construct receives an integer n and an argument expression and sums the evaluated arthis argument n times in a loop. For $n \in [0 \ldots 1{,}000{,}000]$ with step size $10{,}000$ the benchmark executes the expression sum(n, add(lit(1), lit(2))). We report the average running time measured using `System.nanoTime()` over 100 executions per n.

The benchmark is executed for three versions of the expression interpreter: one without lifting (explicitly propagating an environment), one that is lifted to implicitly propagate and environment, and an optimized lifted interpreter that we discuss below.

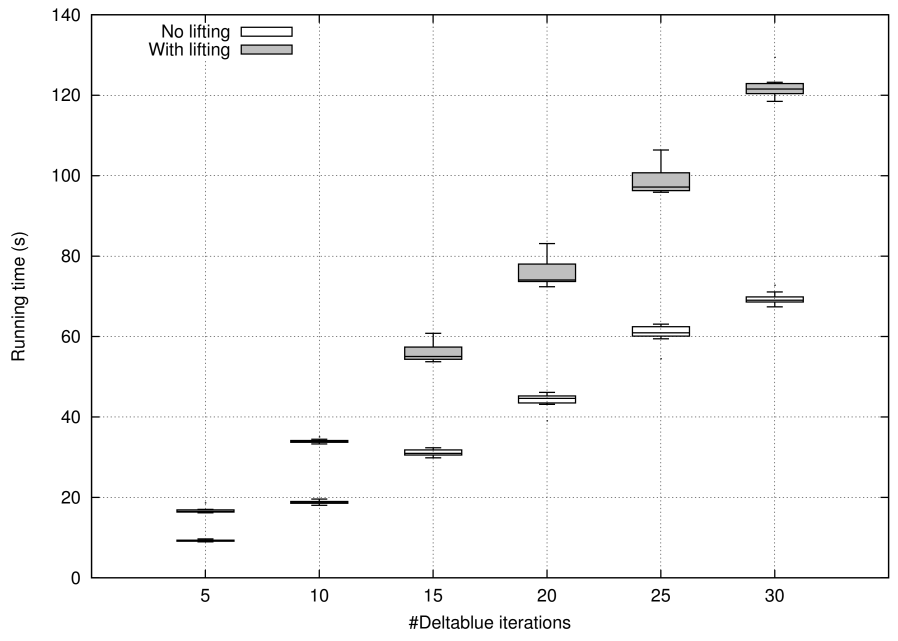

Figure 15: Runtime of the lifted interpreter compared to the baseline interpreter.

The benchmark results are shown in Figure 16. It is immediately clear from Figure 16 that lifting does incur significant overhead. The non-lifted interpreter is about twice as fast as the lifted versions, and gets even faster as soon as the JIT compiler kicks in (around 540,000). This is because the non-lifted interpreter involves one call per expression evaluation, whereas the lifted interpreters always involve two.

The "fast" lifting results show a small improvement over vanilla lifting. The optimization in fast lifting is based on moving the creation of new closures outside of dynamic expression evaluation, and using side-effects to bind the propagated parameter. As a result, all additional closures are created once and for all when the expression itself is created. For instance, the optimized lifting of the addition construct is defined as follows:

```
def add(l: EvE, r: EvE): EvE = {
  var env: Env = null
  val lw = () => l(env)
  val rw = () => r(env)
  val add = base.add(lw, rw);
  e => { env = e; add() }
}
```

Instead of creating the lowered argument closures `lw` and `rw` within the body of the returned closure, they are hoisted to the level of `add` itself. Similarly, the call to `base.add` is also hoisted, avoiding additional runtime overhead. To ensure that the environment is propagated correctly, the returned closure assigns the received environment `e` to the

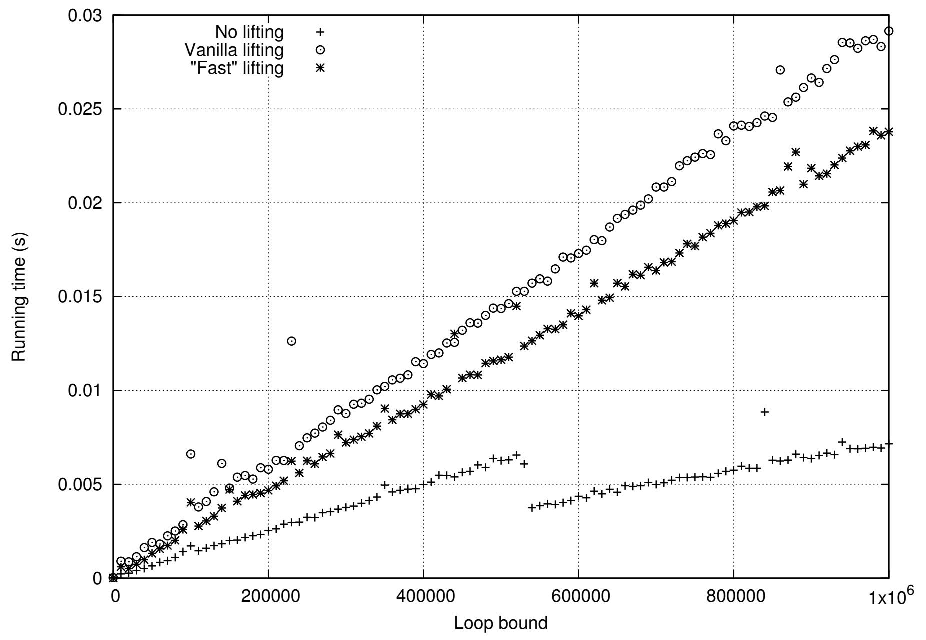

Figure 16: Executing sum(n, add(lit(1), lit(2))) without lifting, with vanilla lifting, and with "fast" lifting.

mutable variable env.

The results of the micro-benchmark confirm the observed slowdown in the DeltaBlue benchmark. However, the primary cause seems to be additional call overhead, and not the allocation of closures, since that accounts for only a small part of the slowdown. Further research is needed to investigate ways of optimizing lifting interpreters.

9. Related Work and Discussion

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

9.1. Related Work

We discuss the related work in three categories: modular interpreters, component-based language development, and, finally, other techniques to implicitly propagate context.

9.1.1. Modular Interpreters

In this section, we present other approaches to modular interpreters. We first elaborate on the concept of monad transformers since this technique is the most well-known approach for defining modular interpreters. We then discuss some limitations of the original presentation of monad transformers and existing work that addresses them. Finally, we review other functional programming approaches to modular interpreters.

Monad transformers. 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. [31] consolidated much of the earlier work (e.g., [41, 10]) on how *monad transformers* (MT) can be used to solve this problem. The complete presentation of modular interpreters in their work exposes a monadic interpretation function whose signature is `interp :: Term -> InterpM Value`. This interpreter is extensible because all three components – the term type, the value type and the monad – can be composed from individual components. Both the term and the value types can be modularly defined and later extended/composed using extensible unions, while the monad `InterpM` can be defined using monad transformers.

To illustrate modular interpreters in monadic style, the left column of Table 5 shows the `Arith`, `Binding` and `Storage` languages introduced in Section 4 in Haskell. For reference, the Object Algebra implementations are shown in the right column. The monadic interpreters are defined as instances of the type class `InterpC`, where the `interp` operation is defined. The code fragments assume that extensible unions are used to combine the syntactic data types of each language module and the resulting `Value` data type used in `InterpM Value`. As a result, the algebraic data types in Table 5 recurse on the open type `Term`. For instance, the `Arith` and `Binding` modules can be combined using the **Either** type constructor and a **newtype** definition of `Term` to "tie the knot":

```
type ArithBinding = Either Arith (Either Binding ())
newtype Term = Term ArithBinding
```

The extensible `Value` is defined similarly, and requires auxiliary functions for injecting values into and projecting values out of the value domain. The functions `returnInj` and `bindPrj` are helper functions representing the monadic `return` and `bind` functions performing injection and projection. For instance, the `Arith` language injects integers into the value domain. Similarly, `Binding` and `Storage` require closures and locations to be part of the domain, respectively. Here, however, we focus on the modular effects part of the presentation.

The key observation about Table 5 is that the `interp` functions return monadic values in the type `InterpM`, but each module imposes different constraints on the actual defintion of this type. For instance, the `Arith` module does not make any assumptions on `InterpM` except that it is a monad. The `Binding` module, however, requires that the monad supports the functions `rdEnv` and `inEnv` in order to obtain the current environment and evaluate an expression in a particular environment, respectively. Finally, the `Storage` language requires allocating, inspecting and updating locations.

To support both environments and a store for side effects, `InterpM` should be defined as a stack of monad transformers, each of which allows lifting the operations from one kind of monad into another one. For example, the following definition of `InterpM` suffices for the composition of the `Arith`, `Binding` and `State` interpreters (notice that at the bottom of the stack we have the identity monad `Id`):

```
type InterpM = EnvT Env (StateT Store Id)
```

Given a composition of the syntactic data types representing `Arith`, `Binding` and `Storage` the interpreters can be composed where "effect oblivious" code propagates the

Monad transformers in Haskell	Implicit context propagation in Scala

data Arith
  = Lit **Int**
  | Add Term Term
 	
instance InterpC Arith **where**
  interp (Add l r) = interp l ‘bindPrj‘ \i
      −> interp r ‘bindPrj‘ \j
      −> returnInj ((i+j) : : **Int**)
  interp (Lit n) = returnInj n
	type Ev = () => Val
	
	trait EvArith **extends** Arith[Ev] {
	 def add(l : Ev, r : Ev) = () => IntVal(l() + r())
	
	 def lit(n : Int) = () => IntVal(n)
	}

data Binding
  = Lambda Name Term
  | Vari Name
  | Apply Term Term
 	
instance InterpC Binding **where**
  interp (Lambda x b) = rdEnv >>= \env
      −> returnInj $ Clos x b env
  interp (Vari x) = rdEnv >>= \env
      −> **case** lookupEnv x env **of**
        Just v −> v
 	
  interp (Apply e1 e2) = interp e1 ‘bindPrj‘ \f −>
      case f **of**
       Clos x e env −>
        inEnv
         (extendEnv (x, interp e2) env)
         (interp e)
	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))
	}

data Storage
  = Create
  | Update Term Term
  | Inspect Term
 	
instance InterpC Storage **where**
 interp (Create) = **do** loc <− allocLoc
      updateLoc (loc, returnInj (0 : : **Int**))
      returnInj loc
 interp (Update c v) = interp c ‘bindPrj‘ \loc
      −> interp v >>= \val
      −> updateLoc (loc, **return** val) >>
       return val
 interp (Inspect c) = interp c ‘bindPrj‘ lookupLoc
	trait EvMStorage **extends** Storage[EvS] {
	 def create()
	  = st => { **val** c = **new** Cell(st.size + 1)
	      st += c −> IntVal(0); c }
	
	 def update(c : EvS, v : EvS)
	  = st => { **val** c1 : Cell = c(st)
	      val v1 = v(st)
	      st += c1 −> v1; c1 }
	
	 def inspect(c : EvS)
	  = st => { **val** c1 : Cell = c(st); st(c1) }
	}

Table 5: Arithmetic, Binding and Storage language building blocks implemented in Haskell with monad transformers (on the left) and in Scala using Object Algebras (on the right).

	Monad transformers [31]	This work
Syntactic modularity	Extensible unions	Trait composition
Value extensibility	Extensible unions (with inj/prj)	Subtyping (with casts)
Context propagation	Monad transformers	Implicit propagation
Interpreter style	Monadic	Direct
Part of signature	Return type	Formal parameters
Purely functional	Yes	No
Type safe	Yes	Yes
Separate compilation	No	Yes
Effect interaction	Explicit	Implicit
Scope of interaction	Global, fixed	Locally overridable
Compositional propagation	No	Yes

Table 6: Comparing characteristics of modular interpreters using monad transformers vs. implicit context propagation

environment and store as defined in the monad transformers `EnvT` and `StateT`.

Note that the definition of `InterpM` is a global definition for all interpreter modules, which defines the set of effects and their composition once and for all. This means that the ordering of the effects and their interaction is defined and fixated at this point; it is not possible to change the interaction on a per module basis. Furthermore, any change to the `InterpM` type definition requires re-typechecking each module. As such, this formulation does not support separate compilation. It is also impossible to reuse a composition of interpreters as a black box component in further compositions.

Monad transformers are defined in a pair-wise fashion. This means that if there are feature interactions between two monads, they have to be explicitly resolved. With implicit context propagation, however, the interactions between effects are implicit, as they depend on the behavior of the context objects in the host language. Note however, that inadvertent feature interactions can always be explicitly resolved in our case by *overriding* lifted interpretations. An example of this is preventing capture of the dynamic environment as discussed in Section 4.2. Nevertheless, when the interactions are more complex (e.g., when the host language does not natively support the effects being modeled), the overriding code can be quite involved (cf. Section 4.4).

As a summary, Table 6 provides a qualitative appraisal of using monad transformers vs. implicit context propagation as presented in this paper, in terms of a number of meta level characteristics. The table shows that both approaches have different strengths and weaknesses. The most apparent feature of monad transformers is that they represent a purely functional approach to modular interpreters; no side-effects are needed. Implicit context propagation, on the other hand, emphasizes extensibility and simplicity, at the cost, perhaps, of sacrificing purity and explicitness. In particular, implicit context propagation supports compositional propagation: a lifted interpreter can be lifted yet again, to propagate additional context. The lifting operation is oblivious as to whether the interpreter to be lifted is a base interpreter or has been lifted earlier. This is a crucial aspect for incremental, modular development of languages.

Extensible syntax and multiple interpretations. Monad transformers in their original presentation do not feature separate compilation. Although we can define the language components in a modular fashion, at the moment of composition all the type synonyms (e.g. the type InterpM) who are referred by each module, must be resolved and therefore, the compilation is monolithic. Similarly, syntax definitions are not truly extensible either: adding data types to the extensible union requires recompilation of the existing interpreter code. To overcome these problems, Duponcheel [9] extended the work of [31] by representing the abstract syntax of a language as algebras, and interpreters as catamorphisms over such algebras to cater for extensible syntax. Additionally, this style support extensible operations as well, similar to the Object Algebra style employed in this paper.

Other approaches to modular interpreters. A different approach to extensible interpreters was pioneered by Cartwright and Felleisen [4]. 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 $Exp \rightarrow Env \rightarrow C$ 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. [27] generalized the approach of [4], allowed the administration functions to be modularized as well, and embedded the framework in Haskell using open unions for extensible syntax, and free monads for extensible interpreters. This approach to define modular interpreters excels at the definition of imperative embedded languages, where the monad sequencing operator is reused for the sequencing operators of the embedded language.

9.1.2. 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) [21] 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 Assistants [22]. 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 [6, 7]. 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 [5]. Languages are defined by mapping abstract syntax to fundamental constructs (*funcons*), which in turn are defined using I-MSOS [33], 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 [15]. 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.

9.1.3. 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. [30], 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 [29]. 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 a powerful mechanism to extend or modify the behavior of existing code [20]. For instance, it can be used to implement aspects [8] or context-oriented programming [23].

Another area where implicit propagation has found application is in language engineering tools. For instance, [43] 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 [26].

Finally, the implicit propagation conventions applied in the context of I-MSOS [33], have been implemented in DynSem, a DSL for specifying dynamic semantics [42]. In both I-MSOS and DynSem, propagation is made explicit by transforming semantic specifications.

9.2. Discussion

Although most of the code in this paper, as well as the code of the case studies, is written in Scala, it is easy to port implicit context propagation to other languages. For instance, Java 8 introduces default methods in interfaces, which can be used for trait-like multiple inheritance. These interfaces were used in the discussion of using dynamic proxies for automating lifting in Section 5.2. 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.

As we could conclude from the analysis of existing work on modular interpreters, the main strength of implicit context propagation is its simplicity. 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 4.3). Other effects, such as error propagation, local backtracking (Section 4.4), non-local control flow (break, continue, return, etc.), and gotos and coroutines [2] 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. [18]).

A limitation of implicit context propagation is that certain complex feature interactions may occur when simulating propagation patterns that are not native to the host language (such as transaction-reversing choice). There is always the option to override semantic definitions to resolve the interaction manually, but this can be quite involved. Semantic feature interactions are more explicitly addressed in the work on monad transformers or effect handlers.

Another 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 have the actual implementation completely inferred. 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 exist, it would be an error. This is similar to how Scala implicit parameters are resolved [34]. We consider this as a possible direction for future work.

Another open challenge is the disambiguation of parameters with the same type when automating lifting. For instance, in the case of the Dynamic Scoping example (Section 4.2), one wants to explicitly indicate on the lifted carrier type (Env, Env) => Val which Env corresponds to the dynamic environment, and which one to the static one. In our current implementation, there is no way to specify this either with the macro approach or with dynamic proxies. When there are two parameters with the same type, the behavior of the @lift is currently undefined. In order to work around this, additional metadata (e.g., via annotations) should be provided in order to guide the disambiguation of same-type parameters and produce the right lifting.

10. 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, continuation-passing style, 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 or lifting can be generically performed at runtime using dynamic proxies. We have illustrated the usefulness of implicit context propagation in an extensible implementation of a simple DSL for state machines. 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.

Since lifting is based on creating intermediate closures, lifted interpreters can be significantly slower than directly implemented base interpreters. Running the well-known DeltaBlue benchmark on top of a modular interpreter for LambdaJS shows that lifting makes interpreters almost twice as slow. Further research is required to explore techniques to eliminate the additional call overhead during evaluation. One possible direction would be light-weight modular staging (LMS) [40] in Scala, although this approach would compromise separate compilation of base interpreters.

Other directions for further research include the integration of concrete syntax (cf. [15]), 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 [5].

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr, D. H. Bartley, R. Halstead, et al. Revised^5 report on the algorithmic language Scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

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

[3] 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.

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

[5] 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.

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

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

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

[9] L. Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. 1995.

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

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

[12] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[13] B. N. Freeman-Benson and J. Maloney. The DeltaBlue algorithm: An incremental constraint hierarchy solver. In *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, pages 538–542. IEEE, 1989.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

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

[16] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of Javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. LambdaJS code repository. Online, 2015. `https://github.com/brownplt/LambdaJS`.

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

[19] L. T. Hansen. Syntax for dynamic scoping, 2000. SRFI-15.

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

[21] 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.

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

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

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

[25] P. Inostroza and T. v. d. Storm. Modular interpreters for the masses: Implicit context propagation using Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 171–180. ACM, 2015.

[26] U. Kastens. The GAG-system: A tool for compiler construction. 1984.

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

[28] R. Lammel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, 2003.

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

[30] 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.

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

[32] K. J. Lieberherr. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996.

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

[34] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification v.211, 2014.

[35] 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.

[36] B. C. d. S. Oliveira, T. Van Der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with Object Algebras. In *ECOOP'13*, pages 27–51. Springer, 2013.

[37] Oracle. Dynamic proxy classes. Online, 2015. `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html`.

[38] Oracle. Java annotation processor. Online, 2015. `https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html`.

[39] J. C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[40] 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. ACM, 2010.

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

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

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

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