Recaf: Java Dialects as Libraries

Aggelos Biboudis
CWI, The Netherlands
University of Athens, Greece
biboudis@di.uoa.gr

Pablo Inostroza
CWI, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
CWI & University of Groningen
The Netherlands
storm@cwi.nl

Abstract

Mainstream programming languages like Java have limited support for language extensibility. Without mechanisms for syntactic abstraction, new programming styles can only be embedded in the form of libraries, limiting expressiveness.

In this paper, we present Recaf, a lightweight tool for creating Java dialects; effectively extending Java with new language constructs and user defined semantics. The Recaf compiler generically transforms designated method bodies to code that is parameterized by a semantic factory (Object Algebra), defined in plain Java. The implementation of such a factory defines the desired runtime semantics.

We applied our design to produce several examples from a diverse set of programming styles and two case studies: we define i) extensions for generators, asynchronous computations and asynchronous streams and ii) a Domain-Specific Language (DSL) for Parsing Expression Grammars (PEGs), in a few lines of code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages

Keywords language virtualization, object algebras, transformation, extensible languages

1. Introduction

Programming languages are as expressive as their mechanisms for abstraction. Most mainstream languages support functional and object-based abstractions, but it is well-known that some programming patterns or idioms are hard to encapsulate using these standard mechanisms. Examples include complex control-flow features, asynchronous programming, or embedded DSLs. It is possible to approximate such features using library-based encodings, but this often leads to code that is verbose and tedious to maintain.

Consider the example of a simple extension for automatically closing a resource in Java[1]:

```
using (File f: IO.open(path)) { ... }
```

Such a language feature abstracts away the boilerplate needed to correctly close an IO resource, by automatically closing the `f` resource whenever the execution falls out of the scope of the block.

Unfortunately, in languages like Java, defining such constructs as a library is limited by inflexible statement-oriented syntax, and semantic aspects of (non-local) control-flow. For instance, one could try to simulate `using` by a method receiving a closure, like `void using(Closeable r, Consumer<Closeable> block)`. However, the programmer can now no longer use non-local control-flow statements (e.g., `break`) within the closure block, and all variables will become effectively final as per the Java 8 closure semantics. Furthermore, encodings like this disrupt the conventional flow of Java syntax, and lead to an atypical, inverted code structure. More sophisticated idioms lead to even more disruption of the code (case in point is "call-back hell" for programming asynchronous code).

In this work we present Recaf[2], a lightweight tool[3] to extend Java with custom dialects. Extension writers do not have to alter Java's parser or write any transformation rules. The Recaf compiler generically transforms an extended version of Java, into code that builds up the desired semantics. Hence, Recaf is lightweight: the programmer can define a dialect without stepping outside the host language.
Recaf is based on two key ingredients:

- Syntax extension: Java's surface syntax is liberated to allow the definition of new language constructs, as long as they follow the pattern of existing control-flow or declaration constructs. (For instance, the `using` construct follows the pattern of Java's `for`-each.) A pattern that is

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...
http://dx.doi.org/10.1145/2993236.2993239

[1] similar to C#'s `using` construct, or Java's try-with-resources

[2] As in *recaffeinating coffee* which describes the process of enhancing its caffeine content [7].

[3] The code is available at `https://github.com/cwi-swat/recaf`.

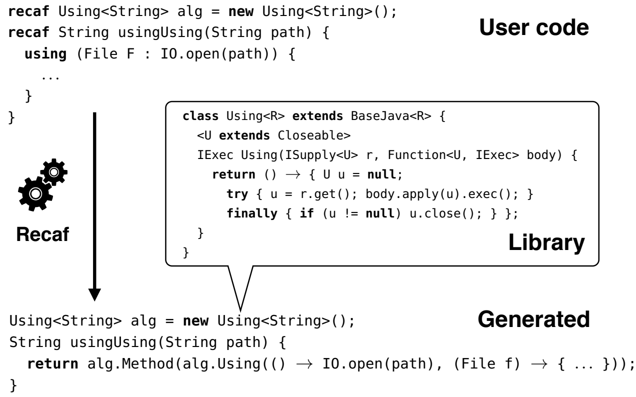

Figure 1. High level overview of Recaf

used, drives a corresponding transformation that Recaf performs.

- Semantics extension: a single, syntax-directed transformation maps method bodies (with or without language extensions) to method calls on polymorphic factories which construct objects encapsulating the user-defined or customized semantics.

The factories are developed within Java as Object Algebras [20], which promote reusability and modular, type-safe extension of semantic definitions.

The combination of the two aforementioned key points enables a range of application scenarios. For instance, Recaf can be used to: extend Java with new syntactic constructs (like `using`), modify or instrument the semantics of Java (e.g., to implement aspects like tracing, memoization), replace the standard Java semantics with a completely different semantics (e.g., translate Java methods to Javascript source code), embed DSLs into Java (e.g., grammars, or GUI construction), define semantics-parametric methods which support multiple interpretations, and combine any of the above in a modular fashion (e.g., combine `using` with a tracing aspect). Developers can define the semantics of new and existing constructs and create DSLs for their daily programming needs. Language designers can experiment with new features, by quickly translating their denotations in plain Java. In other words, Recaf brings the vision of "languages as libraries" to mainstream, object-oriented programming languages.

The contributions of this paper are summarized as follows:

- We present a transformation of Java statement Abstract Syntax Trees (ASTs) with extended syntax to virtualize their semantics, and we show how the semantics can be defined as Object Algebras (Section 3).
- We generalize the transformation to virtualize Java expressions, widening the scope of user defined semantics (Section 4).
- We describe the implementation of Recaf, and how it deals with certain intricacies of the Java language (Section 5).
- We evaluate the expressiveness of Recaf with two case studies: i) providing language support for generators and asynchronous computations and ii) creating a DSL for parser combinators (Section 6).

The results of the case studies and directions for future work are discussed in Section 7.

2. Overview

2.1 Recaffeinating Java with Recaf

Figure 1 gives a bird's eye overview of Recaf. It shows how the `using` extension is used and implemented using Recaf. The top shows a snippet of code illustrating how the programmer would use a Recaf extension, in this case consisting of the `using` construct. The programmer writes an ordinary method, decorated with the `recaf` modifier to trigger the source-to-source transformation. To provide the custom semantics, the user also declares a `recaf` field, in scope of the `recaf` method. In this case, the field `alg` is initialized to be a `Using` object, defined over the concrete type `String`.

The downward arrow indicates Recaf's source-to-source transformation which virtualizes the semantics of statements by transforming the Recaf code fragment to the plain Java code at the bottom. Each statement in the user code is transformed into calls on the `alg` object. The `using` construct itself is mapped to the `Using` method. The `Using` class shown in the call-out, defines the semantics for `using`. It extends a class (`BaseJava`) capturing the ordinary semantics of Java, and defines a single method, also called `Using`. This particular `Using` method defines the semantics of the `using` construct as an interpreter of type `IExec`.

In addition to using a `recaf` field to specify the semantics of a `recaf` method, it is also possible to decorate a formal parameter of a method with the `recaf` modifier. This allows binding of the semantics at the call site of the method itself. Thus, Recaf supports three different binding times for the semantics of a method: static (using a static field), at object construction time (using an instance field), and late binding (method parameter). Recaf further makes the distinction between statement-only virtualization and full virtualization. In the latter case, expressions are virtualized too. This mode is enabled by using the `recaff` keyword, instead of `recaf`. Section 4 provides all the details regarding the difference.

2.2 Object Algebras

The encoding used for the `Using` class in Figure 1 follows the design pattern of Object Algebras [20] which has already been applied to numerous cases in the literature [1, 12, 21]. Object Algebras can be seen as an object-oriented encoding of tagless interpreters [4]. Instead of defining a language's abstract syntax using concrete data structures, it is defined using generic factories: a generic interface declares generic methods for each syntactic construct. Implementations of such interfaces define a specific semantics by creating se-

```
interface MuJavaMethod<R, S> { R Method(S s); }

interface MuJava<R, S> {
   S Exp(Supplier<Void> e);
   S If(Supplier<Boolean> c, S s1, S s2);
   <T> S For(Supplier<Iterable<T>> e, Function<T, S> s);
   <T> S Decl(Supplier<T> e, Function<T, S> s);
   S Seq(S s1, S s2);
   S Return(Supplier<R> e);
   S Empty();
}
```

Figure 2. Object Algebra interfaces defining the abstract syntax of μJava method bodies and statements.

mantic objects representing operations like pretty printing, evaluation, and so on.

Object Algebras are a simple solution to the *expression problem* [31]. As such they provide type-safe, modular extensibility along two axes: adding new data variants and adding new operations over them without changing existing code. For instance, the `Using` algebra extends the base Java semantics with a new syntactic construct. On the other hand, the generic interface representing the abstract syntax of Java can also be implemented *again*, to obtain a different semantics. In the remainder of this paper we define the algebras as Java 8 interfaces with default methods to promote additional flexibility for modularly composing semantic modules.

3. Statement Virtualization

In this section we describe the first level of semantic and syntactic polymorphism offered by Recaf, which restricts virtualization and syntax extension to statement-like constructs.

3.1 μJava

μJava is a simplified variant of Java used for exposition in this paper. In μJava all variables are assumed to be final, there is no support for primitive types nor `void` methods, and all variable declarations have initializers. Figure 2, shows the abstract syntax of μJava statements and method bodies in the form of Object Algebra interfaces.

Both interfaces are parametric in two generic types, `R` and `S`. `R` represents the return type of the method, and `S` the semantic type of statements. The method `Method` in `MuJavaMethod` mediates between the denotation of statements (`S`) and the return type `R` of the virtualized method. The programmer of a Recaf method needs to ensure that `R` returned by `Method` corresponds to the actual return type declared in the method. Note that `R` does not have to be bound to the same concrete type in both `MuJavaMethod` and `MuJava`. This means that the return type of a virtualized method can be different than the type of expressions expected by `Return`.

The `MuJava` interface assumes that expressions are represented using the standard Java `Supplier` type, which represents thunks. Java expressions may perform arbitrary side-effects; the thunks ensure that evaluation is delayed until after the semantic object are created.

$$
\begin{array}{rcl}
\mathscr{M}_a\llbracket S \rrbracket & = & \texttt{return}\ a\texttt{.Method(}\mathscr{S}_a\llbracket S \rrbracket\texttt{);} \\
\mathscr{S}_a\llbracket e\texttt{;} \rrbracket & = & a\texttt{.Exp(() } \rightarrow \texttt{ \{}e\texttt{; return null;\})} \\
\mathscr{S}_a\llbracket \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2 \rrbracket & = & a\texttt{.If(() } \rightarrow e\texttt{, } \mathscr{S}_a\llbracket S_1 \rrbracket\texttt{,} \mathscr{S}_a\llbracket S_2 \rrbracket\texttt{)} \\
\mathscr{S}_a\llbracket \texttt{for(}T\ x\texttt{: }e\texttt{) } S \rrbracket & = & a\texttt{.For(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathscr{S}_a\llbracket S \rrbracket\texttt{)} \\
\mathscr{S}_a\llbracket T\ x \texttt{ = } e\texttt{; } S \rrbracket & = & a\texttt{.Decl(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathscr{S}_a\llbracket S \rrbracket\texttt{)} \\
\mathscr{S}_a\llbracket S_1\texttt{; } S_2 \rrbracket & = & a\texttt{.Seq(}\mathscr{S}_a\llbracket S_1 \rrbracket\texttt{, } \mathscr{S}_a\llbracket S_2 \rrbracket\texttt{)} \\
\mathscr{S}_a\llbracket \texttt{return } e\texttt{;} \rrbracket & = & a\texttt{.Return(() } \rightarrow e\texttt{)} \\
\mathscr{S}_a\llbracket \texttt{;} \rrbracket & = & a\texttt{.Empty()} \\
\mathscr{S}_a\llbracket \texttt{\{ } S \texttt{ \}} \rrbracket & = & \mathscr{S}_a\llbracket S \rrbracket
\end{array}
$$

Figure 3. Virtualizing method statements into statement algebras.

```
for (Integer x: l)
  if (x % 2 == 0)
    return x;
  else ;
return null;
```

```
return a.Method(
 a.Seq(
  a.For(() → l, (Integer x) →
    a.If(() → x % 2 == 0,
        a.Return(() → x),
        a.Empty())),
  a.Return(() → null)));
```

Figure 4. Example method body (left) and its transformation into algebra `a` (right).

The constructs `For` and `Decl` employ higher-order abstract syntax (HOAS [23]) to introduce local variables. As a result, the bodies of declarations (i.e., the statements following it, within the same scope) and `for`-each loops are represented as functions from some generic type `T` to the denotation `S`.

3.2 Transforming Statements

The transformation for μJava is shown in Figure 3, and consists of two transformation functions $\mathscr{M}$ and $\mathscr{S}$, respectively transforming method bodies, and statements. The transformation folds over the syntactic structure of μJava, compositionally mapping each construct to its virtualized representation. Both functions are subscripted by the expression a, which represents the actual algebra that is used to construct the semantics. The value of a is determined by the `recaf` modifier on a field or formal parameter.

As an example consider the code shown on the left of Figure 4. The equivalent code after the Recaf transformation is shown on the right. The semantics of the code is now virtualized via the algebra object `a`. The algebra `a` may implement the same semantics as ordinary Java, but it can also customize or completely redefine it.

3.3 Statement Syntax

Statement syntax is based on generalizing the existing control-flow statement syntax of Java. Informally speaking, wherever Java requires a keyword (e.g., `for`, `while` etc.), Recaf allows

$$\begin{aligned}
\mathscr{S}_a[\![x!\ e;]\!] &= a.x(\texttt{()} \rightarrow e) \\
\mathscr{S}_a[\![x\ (T\ y\texttt{:}\ e)\ S]\!] &= a.x(\texttt{()} \rightarrow e,\ (T\ y) \rightarrow \mathscr{S}_a[\![S]\!]) \\
\mathscr{S}_a[\![x\ T\ y = e;\ S]\!] &= a.x(\texttt{()} \rightarrow e,\ (T\ y) \rightarrow \mathscr{S}_a[\![S]\!]) \\
\mathscr{S}_a[\![x\ (e)\ S]\!] &= a.x(\texttt{()} \rightarrow e,\ \mathscr{S}_a[\![S]\!]) \\
\mathscr{S}_a[\![x\ \{\ S\ \}]\!] &= a.x(\mathscr{S}_a[\![S]\!])
\end{aligned}$$

Figure 5. Transforming syntax extensions to algebra method calls.

the use of an identifier. This identifier will then, by convention, correspond to a particular method with the same name in the semantic algebra.

The following grammar describes the syntax extensions of statements (S) for μJava:

S	::=	$x!\ e\,;$	Return-like
	|	$x\ (T\ x\text{:}\ e)\ S$	For-each like
	|	$x\ (e)\ \{S\}$	While-like
	|	$x\ \{S\}$	Try-like
	|	$x\ T\ x = e;$	Declaration-like

This grammar defines a potentially infinite family of new language constructs, by using identifiers (x) instead of keywords. Each production is a generalization of existing syntax. For instance, the first production, follows syntax of `return` e, with the difference that an exclamation mark is needed after the identifier x to avoid ambiguity. The second production is like `for`–each, the third like `while`, and the fourth follows the pattern of `if` without `else`. Finally, the last production supports custom declarations, where the first identifier x represents the keyword.

Transforming the extension into an algebra simply uses the keyword identifier `x` as a method name, but follows the same transformation rules as for the corresponding, non-extended constructs. The transformation rules are shown in Figure 5.

3.4 Direct Style Semantics

The direct style interpreter for μJava is defined as the interface `MuJavaBase`, implementing it using default methods and is declared as follows:

```
interface MuJavaBase<R> extends MuJava<R, IExec> { ... }
```

The type parameter `R` represents the return type of the method. `S` is bound to the type `IExec`, which represents thunks (closures):

```
interface IExec { void exec(); }
```

The algebra `MuJavaBase` thus maps μJava statement constructs to semantic objects of type `IExec`. Most of the statements in μJava have a straightforward implementation. Non-local control-flow (i.e., `return`), however, is implemented using exception handling.

The method `Method` ties it all together and mediates between the evaluation of the semantic objects, returned by the algebra, to the actual return type of the method:

```
default R Method(IExec s) {
  try { s.exec(); }
   catch (Return r) { return (R)r.value; }
   catch (Throwable e) { throw new RuntimeException(e); }
  return null;
}
```

Since the mapping between the statement denotation and the actual return type of a method is configurable it is not part of `MuJavaBase` itself. This way, `MuJavaBase` can be reused with different `Method` implementations.

Example: Maybe As a simple example, similar to the `using` extension introduced in Section 1, consider a `maybe` construct, to unwrap an optional value (of type `java.util.Optional`). In a sense, `maybe` literally overrides the semicolon, similar to the bind operator of Haskell. Syntactically, the `maybe` operator follows the declaration-like syntax. It is defined as follows:

```
interface Maybe<R> extends MuJavaBase<R> {
  default <T> IExec Maybe(Supplier<Optional<T>> x,
                          Function<T, IExec> s) {
    return () → {
       Optional<T> opt = x.get();
       if (opt.isPresent()) s.apply(opt.get()).exec(); };
  }
}
```

The `Maybe` method returns an `IExec` closure that evaluates the expression (of type `Optional`), and if the optional is not empty, executes the body of `maybe`.

3.5 Continuation-Passing Style Semantics

The direct style base interpreter can be used for many extensions like `using` or `maybe`. However, language constructs that require non-local control-flow semantics require a continuation-passing style (CPS) interpreter. This base interpreter can be used instead of the direct style interpreter for extensions like coroutines, backtracking, call/cc etc. It also shows how Object Algebras enables the definition of two different semantics for the same syntactic interface.

The cps style interpreter is defined as the interface `MuJavaCPS`, similarly to `MuJavaBase`:

```
interface MuJavaCPS<R> extends MuJava<R, SD<R>> { ... }
```

The `MuJavaCPS` algebra maps μJava abstract syntax to CPS denotations (`SD`), defined as follows:

```
interface SD<R> { void accept(K<R> r, K0 s); }
```

`SD<R>` is a functional interface that takes as parameters a return and a success continuation. The return continuation `r` is of type `K<R>` (a consumer of `R`) and contains the callback in the case a statement is the return statement. The success continuation is of type `K0` (a thunk) and contains the callback in the case the execution falls off without returning.

To illustrate the CPS interpreter, consider the following code that defines the semantics of the `if-else` statement:

```
interface Backtrack<R>
extends MuJavaCPS<R>, MuJavaMethod<List<R>, SD<R>> {

   default List<R> Method(SD<R> body) {
      List<R> result = new ArrayList<>();
      body.accept(ret → { result.add(ret); }, () → {});
      return result;
   }

    default <T> SD<R> Choose(Supplier<Iterable<T>> e,
                             Function<T, SD<R>> s) {
      return (r, s0) → {
         for (T t: e.get()) s.apply(t).accept(r, s0);};
   }
}
```

Figure 6. Backtracking Extension

```
default SD<R> If(Supplier<Boolean> c, SD<R> s1, SD<R> s2) {
  return (r, s) → { if (c.get()) s1.accept(r, s);
                   else s2.accept(r, s);}; }
```

Based on the truth-value of the condition, either the then branch or the else branch is executed, with the same continuations as received by the if-then-else statement.

Example: Backtracking The CPS interpreter serves as a base implementation for language extensions, requiring complex control flow. We demonstrate the backtracking extension that uses Wadler's list of successes technique [30] and introduces the `choose` keyword. As an example, consider the following method which finds all combinations of integers out of two lists that sum to 8.

```
List<Pair> solve(recaf Backtrack<Pair> alg) {
  choose Integer x = asList(1, 2, 3);
  choose Integer y = asList(4, 5, 6);
  if (x + y == 8) {
    return new Pair(x, y);
  }
}
```

In Figure 6 we present the extension for μJava. Note that `Method` has a generic parameter type in the `MuJavaMethod` interface. In this case we change the return type of method to `List<T>` instead of just `T`. The result is the list of successes, so the return continuation should add the calculated item in the return list, instead of just passing it to the continuation. `Choose` simply invokes its success continuation for every different value, effectively replaying the execution for every element of the set of values.

4. Full Virtualization

The previous section discussed virtualization of declaration and control-flow statements. In this section we widen the scope of Recaf for virtualizing expressions as well.

```
interface MuStmJava<S, E> {
   S Exp(E x);
   <T> S Decl(E x,Function<T,S> s);
   <T> S For(E x,Function<T,S> s);
   S If(E c, S s1, S s2);
   S Return(E x);
   S Seq(S s1, S s2);
   S Empty();
}
```

```
interface MuExpJava<E> {
   E Lit(Object x);
   E This(Object x);
   E Field(E x, String f);
   E New(Class<?> c, E...es);
   E Invoke(E x, String m, E...es);
   E Lambda(Object f);
   E Var(String x, Object it);
}
```

Figure 7. Generic interfaces for the full abstract syntax of μJava.

4.1 Expression Virtualization

Until know we have dissected the `MuJava` interface of Figure 2. That interface still does not support virtualized expressions since it requires the concrete type `Supplier` in expression positions. To enable full virtualization, we have to use the algebraic interfaces shown in Figure 7, where expressions are represented by the generic type `E`. `MuExpJava` specifies the semantic objects for μJava expressions. The interface `MuStmJava` is similar to `MuJava`, but changes the concrete `Supplier` arguments to the generic type `E`.

Compared to full Java, the expression sub language of μJava makes some additional simplifying assumptions: there are no assignment expressions, no super calls, no array creation, no static fields or methods, no package qualified names, and field access and method invocation require an explicit receiver. For brevity, we have omitted infix and prefix expressions.

To support expression virtualization, the transformation of statements is modified according to the rules of Figure 8. The function $\mathscr{E}$ folds over the expression structure and creates the corresponding method calls on the algebra a. Consider, for example, how full virtualization desugars the μJava code fragment `for (Integer x: y) println(x + 1);`:

```
a.For(a.Var("y", y), Integer x →
   a.Exp(a.Invoke(a.This(this), "println",
                 a.Add(a.Var("x", x), a.Lit(1)))));
```

Note how the HOAS representation of binders carries over to expression virtualization, through the `Var` constructor. The additional `String` argument to `Var` is not essential, but provides additional meta data to the algebra.

Recaf does not currently support new syntactic constructs for user defined expression constructs. We assume that in most cases ordinary method abstraction is sufficient[4]. The examples below thus focus on instrumenting or replacing the semantics of ordinary μJava expressions.

4.2 An Interpreter for μJava Expressions

Just like statements, the base semantics of μJava expressions is represented by an interpreter, this time conforming to the

[4] The only situation where a new kind of expression would be useful is when arguments of the expression need to evaluated lazily, as in short-circuiting operators.

$$
\begin{aligned}
\mathscr{S}_a[\![e;]\!] &= a.\texttt{Exp}(\mathscr{E}_a[\![e]\!]) \\
\mathscr{S}_a[\![\textbf{if}\ (e)\ S_1\ \textbf{else}\ S_2]\!] &= a.\texttt{If}(\mathscr{E}_a[\![e]\!], \mathscr{S}_a[\![S_1]\!], \mathscr{S}_a[\![S_2]\!]) \\
\mathscr{S}_a[\![\textbf{for}(T\ x\texttt{:}\ e)\ S]\!] &= a.\texttt{For}(\mathscr{E}_a[\![e]\!], (T\ x) \to \mathscr{S}_a[\![S]\!]) \\
\mathscr{S}_a[\![T\ x = e;\ S]\!] &= a.\texttt{Decl}(\mathscr{E}_a[\![e]\!], (T\ x) \to \mathscr{S}_a[\![S]\!]) \\
\mathscr{S}_a[\![\textbf{return}\ e;]\!] &= a.\texttt{Return}(\mathscr{E}_a[\![e]\!]) \\
\\
\mathscr{E}_a[\![v]\!] &= a.\texttt{Lit}(v) \\
\mathscr{E}_a[\![x]\!] &= a.\texttt{Var}(\texttt{"}x\texttt{"}, x) \\
\mathscr{E}_a[\![\textbf{this}]\!] &= a.\texttt{This}(\textbf{this}) \\
\mathscr{E}_a[\![e.x]\!] &= a.\texttt{Field}(\mathscr{E}_a[\![e]\!], \texttt{"}x\texttt{"}) \\
\mathscr{E}_a[\![e.x(e_1,\ldots,e_n)]\!] &= a.\texttt{Invoke}(\mathscr{E}_a[\![e]\!], \texttt{"}x\texttt{"}, \mathscr{E}_a[\![e_1]\!],\ldots,\mathscr{E}_a[\![e_n]\!]) \\
\mathscr{E}_a[\![\textbf{new}\ T(e_1,\ldots,e_n)]\!] &= a.\texttt{New}(T.\textbf{class}, \mathscr{E}_a[\![e_1]\!],\ldots,\mathscr{E}_a[\![e_n]\!]) \\
\mathscr{E}_a[\![(x_1,\ldots,x_n) \to S]\!] &= a.\texttt{Lambda}((x_1,\ldots,x_n) \to \mathscr{S}_a[\![S]\!])
\end{aligned}
$$

Figure 8. Transforming statements (modified) and expressions.

interface `MuExpJava`, shown in Figure 7. This interpreter binds `E` to the closure type `IEval`:

```
interface IEval { Object eval(); }
```

As `IEval` is not polymorphic, we do not make any assumptions about the type of the returned object. The interpreter is fully dynamically typed, because Java's type system is not expressive enough to accurately represent the type of expression denotations, even if the Recaf transformation would have had access to the types of variables and method signatures.

The evaluation of expressions is straightforward. Field access, object creation (`new`), and method invocation, however are implemented using reflection. For instance, the following code defines the semantics for field lookup:

```
default IEval Field(IEval x, String f) {
   return () → {
      Object o = x.eval();
      Class<?> clazz = o.getClass();
      return clazz.getField(f).get(o);
   };
}
```

The class of the object whose field is requested, is discovered at runtime, and then, the reflective method `getField` is invoked in order to obtain the value of the field for the requested object.

Example: Aspects A useful use case for expression virtualization is defining aspect-like instrumentation of expression evaluation. The algebra methods of the base interpreter are overridden, they implement the additional behavior, and delegate to the parent's implementation with the `super` keyword. As an example, consider an aspect defining tracing of the values of variables during method execution. The algebra extends the base interpreter for Java expressions and overrides the `Var` definition, for variable access. Figure 9 shows how the overridden `Var` method uses the variable name and the actual received value passed to print out the tracing information, and then calls the `super` implementation.

```
default IEval Var(String x, Object v){
   return () → {
      System.err.println(x + " = " + v);
      return MuExpJavaBase.super.Var(x, v).eval();
   };
}
```

Figure 9. Intercepting field accesses for policy based access control.

Example: Library embedding In the following example we demonstrate library embedding of a simple constraint solving language, *Choco* [24], a Java library for constraint programming. Choco's programming model is heavily based on factories nearly for all of its components, from variable creation, constraint declaration over variables, to search strategies.

We have developed a Recaf embedding which translates a subset of Java expressions to the internal constraints of Choco, which can then be solved. The `Solve` algebra defines the `var` extension to declare constraint variables. The `solve!` statement posts constraints to Choco's solver. This embedding illustrates how the expression virtualization allows the extension developer to completely redefine (a subset of) Java's expression syntax.

```
recaf Solve alg = new Solve();
recaff Iterable<Map<String,Integer>> example() {
  var 0, 5, IntVar x;
  var 0, 5, IntVar y;
  solve! x + y < 5;
}
```

5. Implementation of Recaf

All Recaf syntactic support is provided by Rascal [14], a language for source code analysis and transformation. Rascal features built-in primitives for defining grammars, tree traversal and concrete syntax pattern matching. Furthermore, Rascal's language workbench features [9] allow language developers to define editor services, such as syntax highlighting or error marking, for their languages.

5.1 Generically Extensible Syntax for Java

Section 3.3 introduced generic syntax extensions in the context of μJava, illustrating how the base syntax could be augmented by adding arbitrary keywords, as long as they conform to a number of patterns, e.g. `while`- or declaration-like. We implemented these patterns and a few additional ones for full Java using Rascal's declarative syntax rules. These rules modularly extend the Java grammar, defined in Rascal's standard library using productions for each case that we identify as an extensibility point: return-like, declaration-like, for-like, switch-like, switch-like (as a for) and try-like.

5.2 Transforming Methods

Recaf source code transformation transforms any method that has the `recaf` or `recaff` modifier. If the modifier is attached to the method declaration, the algebra is expected to be declared as field in the enclosing scope (class or interface). If the modifier is attached to a method's formal parameter, that parameter itself is used instead. Furthermore, if the `recaff` modifier is used, expressions are transformed as well.

The transformation is defined as Rascal rewrite rules that match on a Java statement or expression using concrete-syntax pattern matching. This means that the matching occurs on the concrete syntax tree directly, having the advantage of preserving comments and indentation from the Recaf source file. As an example, the following rule defines the transformation of the `while`-statement:

```
Expr stm2alg((Stm)`while (<Expr c>) <Stm s>`,Id a,Names ns)
  = (Expr)`<Id a>.While(<Expr c2>, <Expr s2>)`
  when
    Expr c2 := injectExpr(c, a, ns),
    Expr s2 := stm2alg(s, a, ns);
```

This rewrite rule uses the actual syntax of the Java `while` statement as the matching pattern and returns an expression that calls the `While` method on the algebra `a`. The condition `c` and the body `s` are transformed in the `when`-clause (where `:=` indicates binding through matching). The function `injectExpr` either transforms the expression `c`, in the case of transformations annotated with the `recaff` keyword, or creates closures of type `Supplier` otherwise. The body `s` is transformed calling recursively `stm2alg`.

The `ns` parameter represents the declared names that are in scope at this point in the code and is the result of a local name analysis needed to correctly handle mutable variables. Local variables introduced by declarations and for-loops are mutable variables in Java, unless they are explicitly declared as `final`. This poses a problem for the HOAS encoding we use for binders: the local variables become parameters of closures, but if these parameters are captured inside another closure, they have to be (effectively) final. To correctly deal with this situation, variables introduced by declarations or for-loops are wrapped in explicit reference objects, and the name is added to the `Names` set. Whenever such a variable is referenced in an expression it is unwrapped. For extensions that introduce variables it is unknown whether they should be mutable or not, so the transformation assumes they are final, as a default. In total, the complete Recaf transformation consists of 790 SLOC.

5.3 Recaf Runtime

The Recaf runtime library comes with two base interpreters of Java statements, similar to `MuJavaBase` and `MuJavaCPS`, and an interpreter for Java Expressions. In addition to `return`, the interpreters support the full non-local control-flow features of Java, including (labeled) `break`, `continue` and `throw`. The CPS interpreter represents each of those as explicit continuations in the statement denotation (`SD`), whereas the direct style interpreter uses exceptions.

The main difference between `MuExpBase` and the full expression interpreter is handling of assignments. We model mutable variables by the interface `IRef`, which defines a setter and getter to update the value of the single field that it contains. The `IRef` interface is implemented once for local variables, and once for fields. The latter uses reflection to update the actual object when the setter is called. In addition to the `Var(String,Object)` constructor, the full interpreter features the constructor `Ref(String,IRef<?>)` to model mutable variables. The expression transformation uses the local name analysis (see above) to determine whether to insert `Var` or `Ref` calls.

Since the Recaf transformation is syntax-driven, some Java expressions are not supported. For instance, since the expression interpreter uses reflection to call methods, statically overloaded methods are currently unsupported (because it is only possible to dispatch on the runtime type of arguments). Another limitation is that Recaf does not support static method calls, fields references or package qualified names. These three kinds of references all use the same dot-notation syntax as ordinary method calls and field references. However, the transformation cannot distinguish these different kinds, and interprets any dot-notation as field access or method invocation with an explicit receiver. We consider a type-driven transformation for Recaf as an important direction for future work.

6. Case Studies

6.1 Spicing up Java with Side-Effects

The Dart programming language recently introduced `sync*`, `async` and `async*` methods, to define generators, asynchronous computations and asynchronous streams [18] without the typical stateful boilerplate or inversion of control flow. Using Recaf, we have implemented these three language features for Java, based on the CPS interpreter, closely following the semantics presented in [18].

Generators. The extension for generators is defined in the `Iter` class. The `Iter` class defines `Method` to return a plain Java `Iterable<R>`. When the `iterator()` is requested, the statement denotations start executing. The actual implementation of the iterator is defined in the client code using two new constructs. The first is `yield!`, which produces a single value in the iterator. Its signature is `SD<R> Yield(ISupply<R>)`[5]. Internally, `yield!` throws a special `Yield` exception to communicate the yielded element to a main iterator effectively pausing the generator. The `Yield` exception contains both the element, as well as the current continuation, which is stored in the iterator. When the next value of the iterator is requested, the saved continuation is invoked to resume the generator process. The second construct is `yieldFrom!`

[5] `ISupply` is a thunk which has a `throws` clause.

which flattens another iterable into the current one. Its signature is `SD<R> YieldFrom(ISupply<Iterable<R>> x)` and it is implemented by calling `ForEach(x, e → Yield(() -> e))`. In the code snippet below, we present a recursive implementation of a `range` operator, using both `yield!` and `yieldFrom!`:

```
recaf Iterable<Integer> range(int s, int n) {
  if (n > 0) {
    yield! s;
    yieldFrom! range(s + 1, n - 1);
  }
}
```

Async. The implementation of `async` methods also defines `Method`, this time returning a `Future<R>` object. The only syntactic extension is the `await` statement. Its signature is `<T> Await(Supplier<CompletableFuture<T>>, Function<T, SD<R>>)`, following the syntactic template of foreach. The `await` statement blocks until the argument future completes. If the future completes normally, the argument block is executed with the value returned from the future. If there is an exception, the exception continuation is invoked instead. `Await` literally passes the success continuation to the future's `whenComplete` method. The Async extension supports programming with asynchronous computations without having to resort to call-backs. For instance, the following method computes the string length of a web page, asynchronously fetched from the web.

```
recaf Future<Integer> task(String url)
  await String html = fetchAsync(url);
  return html.length();
}
```

Async*. Asynchronous streams (or reactive streams) support a straightforward programming style on observables, as popularized by the Reactive Extensions [16] framework. The syntax extensions to support this style are similar to `yield!` and `yieldFrom!` constructs for defining generators. Unlike the `yield!` for generators, however, `yield!` now produces a new element asynchronously. Similarly, the `yieldFrom!` statement is used to splice on asynchronous stream into another. Its signature reflects this by accepting an `Observable` object (defined by the Java variant of Reactive Extensions, RxJava[6]): `SD<R> YieldFrom(ISupply<Observable<R>>)`. Reactive streams offer one more construct: `awaitFor!`, which is similar to the ordinary for-each loop, but "iterates" asynchronously over a stream of observable events. Hence, its signature is `<T> SD<R> AwaitFor(ISupply<Observable<T>>, Function<T, SD<R>>)`. Whenever, a new element becomes available on the stream, the body of the `awaitFor!` is executed again. An `async*` method will return an `Observable`.

Here is a simple method that prints out intermediate results arriving asynchronously on a stream. After the result is printed, the original value is yielded, in a fully reactive fashion.

[6] `https://github.com/ReactiveX/RxJava`

```
recaf <X> Observable<X> print(Observable<X> src) {
  awaitFor (X x: src) {
    System.out.println(x);
    yield! x;
  }
}
```

6.2 Parsing Expression Grammars (PEGs)

To demonstrate language embedding and aspect-oriented language customization we have defined a DSL for Parsing Expression Grammars (PEGs) [11]. The abstract syntax of this language is shown in Figure 10. The `lit!` construct parses an atomic string, and ignores the result. `let` is used to bind intermediate parsing results. For terminal symbols, the `regexp` construct can be used. The language overloads the standard sequencing and `return` constructs of Java to encode sequential composition, and the result of a parsing process. The constructs `choice`, `opt`, `star`, and `plus` correspond to the usual regular EBNF operators. The `choice` combinator accepts a list of alternatives (`alt`). The Kleene operators bind a variable x to the result of parsing the argument statement S, where the provided expression e represents the result if parsing of S fails.

The PEG language can be used by considering methods as nonterminals. A PEG method returns are parser object which returns a certain semantic value type. A simple example of parsing primary expression is shown in Figure 11. The method `primary` returns an object of type `Parser` which produces an expression `Exp`. Primaries have two alternatives: constant values and other expressions enclosed in parentheses. In the first branch of the `choice` operator, the `regexp` construct attempts to parse a numeric value, the result of which, if successful, is used in the `return` statement, returning an `Int` object representing the number. The second branch, first parses an open parenthesis, then binds the result of parsing an additive expression (implemented in a method called `addSub`) to `e`, and finally parses the closing parenthesis. When all three parses are successful, the `e` expression is returned. Note that the `return` statements return expressions, but the result of the method is a parser object.

Standard PEGs do not support left-recursive productions, so nested expressions are typically implemented using loops. For instance, additive expression could be defined as `addSub ::= mulDiv (("+"|"-") mulDiv)*`. Here's how the `addSub` method could define this grammar using the PEG embedding:

```
recaf Parser<Exp> addSub() {
  let Exp l = mulDiv();
  star Exp e = (l) {
    regexp String o = "[+\\-]";
    let Exp r = mulDiv();
    return new Bin(o, e, r);
  }
  return e;
}
```

The first statement parses a multiplicative expression. The `star` construct creates zero or more binary expressions, from

S	::=	`lit!` e;	Literals
	|	`let` $T\ x$ = e ;	Binding
	|	`regexp String` x = e ;	Terminals
	|	S ; S	Sequence
	|	`return` e ;	Result
	|	`choice{` $C+$ `}`	Alternative
	|	`opt` $T\ x$ = (e) S	Zero or one
	|	`star` $T\ x$ = (e) S	One or more
	|	`plus` $T\ x$ = (e) S	Zero or one
C	::=	`alt` l: $S+$	Alternative (l = label)

Figure 10. Abstract syntax of embedded PEGs.

```
recaf Parser<Exp> primary() {
  choice {
    alt "value":
      regexp String n = "[0-9]+";
      return new Int(n);
    alt "bracket":
      lit! "("; let Exp e = addSub(); lit! ")";
      return e;
  }
}
```

Figure 11. Parsing primaries using Recaf PEGs.

the operator (`o`), the left-hand side (`e`) and the right-hand side (`r`). If the body of the `star` fails to recognize a $+$ or $-$ sign, the `e` will be bound to the initial seed value `l`. The constructed binary expression will be fed back into the loop as `e` through every subsequent iteration.

The (partial) PEG for expressions shown above and in Figure 11 does support any kind of whitespace between elements of an expression. Changing the PEG definitions manually to parse intermediate layout, however, would be very tedious and error-prone. Exploiting the Object Algebra-based architecture, we add the layout handling as a modular aspect, by extending the PEG algebra and overriding the methods that construct the parsers.

For instance, to insert layout in between sequences, the PEG subclass for layout overrides the `Seq` as follows:

```
<T, U> Parser<U> Seq(Parser<T> p1, Parser<U> p2) {
   return PEG.super.Seq(p1, PEG.super.Seq(layout, p2));
}
```

Another concern with standard PEGs is exponential worst-case runtime performance. The solution is to implement PEGs as packrat parsers [10], which run in linear time by memoizing intermediate parsing results. Again, the base PEG language can be modularly instrumented to turn the returned parsers into memoizing parsers.

7. Discussion

Static Type Safety The Recaf source-to-source transformation assumes certain type signatures on the algebras that define the semantics. For instance, the transformation of binding constructs (declarations, for-each, etc.) expects `Function` types in certain positions of the factory methods. If a method of a certain signature is not present on the algebra, the developer of a Recaf method will get a static error at the compilation of the generated code.

The architecture based on Object Algebras provides type-safe, modular extensibility of algebras. Thus, the developer of semantics may enjoy full type-safety in the development of extensions. The method signatures of most of the examples and case-studies accurately describe the expected types and do not require any casts.

On the other hand, the statement evaluators represent expressions as opaque closures, which are typed in the expected result such as `Supplier<Boolean>` for the `if-else` statement. At the expression level, however, safety guarantees depend on the denotation types themselves. More general semantics, like the Java base expression interpreter, however, are defined in terms of closures returning `Object`. The reason is that Java's type system is not expressive enough to represent them otherwise (lacking features such as higher-kinded types and implicits). As a result, potentially malformed expressions are not detected at compile-time.

Another consequence of this limitation is that the `Supply`-based statement interpreters described in Section 3 cannot be combined out-of-the-box with expression interpreters in the context of Full Virtualization, as both interpreters must be defined in terms of generic expressions. Fortunately, the `Supply`-based statement interpreters can be reused by applying the Adapter pattern [29]. In the runtime library, we provide an adapter that maps a `Supplier`-based algebra to one that is generic in the expression type. As we have discussed earlier, this is unsafe by definition. Thus, although we can effectively integrate statement and expression interpreters, we lose static type guarantees for the expressions.

To conclude, Recaf programs are type-correct when using Statement Virtualization, as long as they generate type-correct Java code. However, in the context of Full Virtualization, compile-time guarantees are overridden as the expressions are fully generic, and therefore, no static assumptions on the expressions can be made.

Runtime Performance. The runtime performance depends on the implementation of the semantics. The base interpreters are admittedly naive, but for the purpose of this paper they illustrate the modularity and reusability enabled by Recaf for building language extensions on top of Java. The Dart-like extensions, reuse the CPS interpreter. As such they are too slow for production use (closure creation to represent the program increases heap allocations). But these examples illustrate the expressiveness of Recaf's embedding method: a very regular syntactic interface (the algebra), may be implemented by an

interpreter that completely redefines control flow evaluation. On the other hand, the constraint embedding case study only uses the restricted method syntax to build up constraint objects for a solver. Solving the constraints does not incur any additional overhead, the DSL is used merely for construction of the constraint objects.

Further research is still needed, however, to remove interpretive overhead in order to make extensions of Java practical. One direction would be to investigate "compiler algebras", which generate byte code or (even native code) at runtime. Frameworks like ASM [3] and Javassist [6] could be used to dynamically generate bytecode, which could then be executed by the `Method` method.

8. Related Work

Syntactic and semantic extensibility of programming languages has received a lot of attention in literature, historically going back to Landin's, "Next 700 Programming Languages" [15]. In this section we focus on work that is related to Recaf from the perspective of semantic language virtualization, languages as libraries, and semantic language customization.

Language Virtualization. Language virtualization allows the programmer to redefine the meaning of existing constructs and define new ones for a programming language. Well-known examples include LINQ [17]) that offers query syntax over custom data types, Haskell's `do`-notation for defining custom monadic evaluation sequences, and Scala's `for`-comprehensions. Scala Virtualized [25] and Lightweight Modular Staging (LMS) [26] are frameworks to redefine the meaning of almost the complete Scala language. However, these frameworks rely on the advanced typing mechanisms of Scala (higher-kinded types and implicits) to implement concrete implementations of DSL embeddings. Additionally, compared to Scala, Java does not have support for delimited continuations so we rely on a CPS interpretation to mitigate that. Recaf scopes virtualization to methods, a choice motivated by the statement-oriented flavor of the Java syntax, and inspired by how the `async`, `sync*` and `async*` modifiers are scoped in Dart [18] and `async` in C# [2].

Another related approach is the work on F#'s computation expressions [22] which allow the embedding of various computational idioms via the definition of concrete computation builder objects, similar to our Object Algebras. The F# compiler desugars ordinary F# expressions to calls into the factory, in an analogous way to the transformation employed by Recaf. Note that the semantic virtualization capabilities offered by computation expressions are scoped to the expression level. Both, Recaf and F# support custom operators, however in F# they are not supported in control flow statements [27]. Carette et al. [4] construct CPS interpreters among others. In Recaf we use the same approach to enable control-flow manipulation extensions.

Languages as Libraries. Recaf is a framework for library-based language extension. The core idea of "languages as libraries" is that embedded languages or language extensions exist at the same level as ordinary user code. This is different, for instance, from extensible compilers (e.g., [19]) where language extensions are defined at the meta level.

The SugarJ system [8] supports language extension as a library, where new syntactic constructs are transformed to plain Java code by specifying custom rewrite rules. The Racket system supports a similar style of defining libary-based languages by transformation, leveraging a powerful macro facility and module system [28]. A significant difference to Recaf is that in both SugarJ and Racket, the extension developer writes the transformations herself, whereas in Recaf the transformation is generic and provided by the framework.

Language Customization. Language extension is only one of the use cases supported by Recaf. Recaf can also be used to instrument or modify the base semantics of Java. Consequently, Recaf can be seen as specific kind of meta object protocol [13], where the programmer can customize the dynamic semantics of a programming language, from within the host language itself. OpenC++ [5] introduced such a feature for C++, allowing the customization of member access, method invocation and object creation.

9. Conclusion

In this paper we have presented Recaf, a lightweight tool to extend both the syntax and the semantics of Java methods just by writing Java code. Recaf is based on two techniques. First, the Java syntax is generalized to allow custom language constructs that follow the pattern of the regular control-flow statements of Java. Second, a generic source-to-source transformation translates the source code of methods into calls to factory objects that represent the desired semantics. Furthermore, formulating these semantic factories as Object Algebras enables powerful patterns for composing semantic definitions and language extensions.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. We are grateful to Nick Palladinos, Jouke Stoel and Vasilis Karakostas for their valuable suggestions. Research carried out by the first author was supported by the CWI Internships program.

References

[1] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams à la carte: Extensible Pipelines with Object Algebras. In *Proc. of the 29th European Conference on Object-Oriented Programming*, Leibniz International Proceedings in Informatics, pages 591–613. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[2] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'N' Play: Formalizing Asynchronous C#. In *Proc. of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 233–257, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 30:19, 2002.

[4] J. Carette, O. Kiselyov, and C.-c. Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009.

[5] S. Chiba. A Metaobject Protocol for C++. In *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 285–299, New York, NY, USA, 1995. ACM.

[6] S. Chiba. Javassist–a reflection-based programming wizard for Java. In *Proc. of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, volume 174, 1998.

[7] R. Clarke and O. Vitzthum. *Coffee: Recent Developments*. John Wiley & Sons, 2008.

[8] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proc. of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 391–406, New York, NY, USA, 2011. ACM.

[9] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44, Part A:24 – 47, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[10] B. Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. ACM.

[11] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[12] P. Inostroza and T. v. d. Storm. Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras. In *Proc. of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 171–180, New York, NY, USA, 2015. ACM.

[13] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT press, 1991.

[14] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

[15] P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, Mar. 1966.

[16] E. Meijer. Reactive Extensions (Rx): Curing your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

[17] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[18] E. Meijer, K. Millikin, and G. Bracha. Spicing Up Dart with Side Effects. *Queue*, 13(3):40:40–40:59, Mar. 2015.

[19] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. of the 12th International Conference on Compiler Construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[20] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proc. of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 2–27, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature–Oriented Programming with Object Algebras. In *Proc. of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 27–51, Berlin, Heidelberg, 2013. Springer-Verlag.

[22] T. Petricek and D. Syme. The F# Computation Expression Zoo. In *Proc. of the 16th International Symposium on Practical Aspects of Declarative Languages*, PADL 2014, pages 33–48, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[23] F. Pfenning and C. Elliott. Higher-Order Abstract Syntax. In *Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[24] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015.

[25] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher Order Symbol. Comput.*, 25(1):165–207, Mar. 2012.

[26] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012.

[27] D. Syme. The F# 3.0 Language Specification, Sept. 2012.

[28] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages As Libraries. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 132–141. ACM, 2011.

[29] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[30] P. Wadler. How to Replace Failure by a List of Successes. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[31] P. Wadler. The Expression Problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, Dec. 1998.