Behavioral Annotations in Java

ANONYMOUS AUTHOR(S)

While the Java language allows the developer to introduce new annotations, the power of these annotations is limited. Supporting more powerful annotations(e.g., `Immutable, NonNull`, etc.) requires the developer to write new annotation-processors in addition to the code for the annotations. This approach makes the task of the addition of new annotations cumbersome as the annotation developer needs to worry about the annotation-processor as well. In this paper, we present an extension to Java to overcome this challenge.

We extend the Java language with a new type based annotation scheme called *behavioral annotations* to control the expected behavior of the objects in the program, during different key operations (such as dereferencing, writing, printing, and so on); the new extension is called `BAJA`. We first extend Featherweight Java to present a core calculus for Java with behavioral annotations. These behavioral annotations are enforced via a combination of static and run-time checking. We then discuss the translation rules for converting code from `BAJA` to Java, our design of the cross compiler to translate code from `BAJA` to Java, and a series of optimizations to improve the performance of the translated code (by eliminating many useless runtime type-casts). The optimizations are based on a proposed novel annotation flow analysis. Our preliminary evaluation shows that programs written in `BAJA` are concise, and incur nominal performance overhead (max 7.13%, geomean 0.95%) compared to the hand-optimized versions.

1 INTRODUCTION

Annotations can be used to enforce different semantic properties and auxiliary actions in a program. A developer who intends to add a new annotation in Java requires to write an *annotation processor* (AP) in addition to the code corresponding to the new annotation. The task of writing an AP is cumbersome and arguably error-prone, as it involves working with the internals of the underlying complex compiler. Even using a powerful tool like the checker framework [29], where one can give a specification of the AP, is challenging. For example, the NonNull annotation specification in the checker framework is around 4K lines. Further, if a developer wants to design and use many annotations as part of his application then (s)he needs to include all the corresponding APs as extensions to the Java compiler; this renders such a scheme impractical. In this paper, we introduce an infrastructure to add new annotations without any need of programmer-defined APs.

We will motivate behavioral annotations using a typical example, shown in Fig. 1(a). The function `bar` creates two objects and calls the function `foo` twice, passing these objects one at a time. Say, the first object (assigned to `x1`) should be immutable (if any field of the object is modified after initialization, then an exception should be thrown). And for some debugging purpose, we want to log each read of the field `f1` of the other object (assigned to `x2`). Thus, the second call to `foo` should throw an exception. The developer may enforce these properties by explicitly modifying the code at each possible expression and assignment statement; the resulting code would be quite ugly. Alternatively, the developer can implement such behaviors via specialized annotations `Imm` and `Log` and replace the vanilla version of the `new` expressions with the ones with annotations (Fig. 1(b)). For example, the APs can generate code to (i) maintain a lookup table to identify objects that are marked immutable (or logged), and (ii) check before each field-write (or field-read) if it is marked immutable (or logged) and throw an exception (or add a log entry). We argue that designing different complex APs that do annotation specific checks/code generation (e.g., [14, 30]) for each such application-specific requirement is not practical, and we need a way to allow the programmer to specify object-specific behaviors in a convenient manner.

In this paper, we extend the Java language with a powerful annotation scheme called *behavioral*

2020. XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
class C{int f1,f2;
 void fu(C x)
  {x.f2=x.f1+13;}
 void bar() {
  C x1=new C();
  C x2=new C();
  fu(x2);fu(x1);}}
```

(a) Motivating example

```
class C{int f1,f2;
 void fu(C x)
  {x.f2=x.f1+13;}
 void bar() {
  C x1=new @Imm() C();
  C x2=new @Log("f1") C();
  fu(x2); fu(x1); }}
```

(b) Code with annotations

```
class Imm extends AObj<Object>{
 void fWBar(Object x, String f){
  throw new ImmException(); }}
```

(c) Immutability annotation.

```
class Log extends AObj<Object>{
String fld;
public Log(String fld)
 { this.fld = fld; }
void fRBar(Object x, String currField){
 System.out.println("Read "+currField);}}
```

(d) Log annotation: logs reading of a specified field.

Fig. 1. Motivational example and sample annotations.

annotations to control the expected behavior of different objects during different *key* operations (such as object creation, field reading/writing, method calls, reading an object reference, passing the object as an argument, printing and so on); we name our extension `BAJA` (= Behavioral Annotations + Java). Each `BAJA` object is attached to a list of annotation objects. An annotation on an object can be specified at the time of its creation, and/or via the declarations of one or more variables that point to the object. The type of each annotation object is a special Java class that contains routines specifying the expected behavior of the object before/after different key operations. We refer to these routines as *barrier* routines.

For the `Imm` annotation used in Fig. 1(b), Fig. 1(c) shows the code required in `BAJA` to implement the immutability annotation – just three lines of Java code; compare this to the 13K lines of reference immutability AP code [30]. The special method `fWBar` is a barrier that is expected to be invoked before the write to any field. Similarly, the Log annotation (Fig. 1(d)) implements a simple code (in the special function `fRBar`) to be executed before each field read. The central idea here is that new annotations, incorporating object-specific behaviors, are implemented in `BAJA` with the help of annotation classes and fixed-named methods corresponding to the pre- and post-barrier methods for different key operations. It is the responsibility of the `BAJA` compiler to generate code to invoke the barrier methods at the appropriate places. `BAJA` does all of it without needing any specialized APs for different annotations. As another comparison, the implementation of NonNull annotation in `BAJA` takes only five lines of code (compared to the 4K lines of code required to implement the annotation in the checker framework [29]).

In contrast to the dedicated AP based approach, `BAJA` allows object-specific behaviors to be encoded by arbitrary Java code (including assertions), in the appropriate barrier routines. But considering the arbitrary nature of the admitted codes, these behaviors are not statically enforced (unlike an annotation processing scheme like Javarifier [30]). Instead, the explicit assertion code is evaluated at runtime. Overall, the `BAJA` annotation checking is spread across two phases: (i) compile-time type checking: ensures that the code has some basic type consistency; (ii) run-time checking: enforces the annotation related assertions.

There have been several studies on extending higher-level languages to capture the program semantics. For example, the JSR 308 proposal [8] is mainly concerned with reasoning about the program semantics by tracking the writes to the store. Similarly, the Applied Type System of Chen

L::=class C extends C{$\overline{\texttt{Z}}$ $\overline{\texttt{C}}$ $\overline{f}$;K $\overline{\texttt{M}}$}	Z::=@C($\overline{e}$)
M::=Z C m($\overline{\texttt{Z}}$ $\overline{\texttt{C}}$ $\overline{x}$) {return e;}	
K::=C($\overline{\texttt{Z}}$ $\overline{\texttt{C}}$ $\overline{f}$){super($\overline{f}$);this.$\overline{f}$=$\overline{f}$}	
e::=x | $e.f$ | $e.m(\overline{e})$ | new Z C($\overline{e_2}$) | (Z C)e | *error*	

Fig. 2. BAFJ syntax; admits class-declaration (L), method-declaration (M), constructor (K), and expressions (e). We use C, m, f, and x to denote a class names, method names, field names, and identifier names, respectively.

and Xi [3], RSP1 language of Westbrook et al [38], and Ωmega system of Sheard [33] explore ways to enrich the type system to prove assertions over the writes to the store. Besides the writes to individual objects, BAJA allows the programmer to specify barriers for other key operations as well. Further, we enrich the notion of annotations so that the programmer may associate arbitrary actions (not just assertions) with the key operations, thereby even extending the program semantics. These actions may even be associated only during some part of the object's lifetime (e.g., the scope of a function or only when a variable/field is pointing to it).

AOP [16] is another powerful mechanism to increase the modularity of software by admitting the separation of cross-cutting concerns. AspectJ [19] is a popular AOP extension for Java. Its concept of "before / after advices" and have similarities to our proposed behavioral annotations. However, compared to AOP where the advises are specified *per* class, our annotations can be object-specific, thereby allowing a fine grain control on the behavior of individual objects. For example, the Imm, and Log annotations in Fig. 1(b) are attached to the first and second object, respectively and not all the instantiated objects of a particular type. Further, unlike AspectJ, BAJA does not have issues related to obliviousness [6], modular compilation [4] or type safety [10].

We argue that BAJA makes it easy to write and attach multiple object-specific behaviors as annotations, which helps in improving the programmability in general and modularity in particular. Further, the annotation development for a program does not interfere with the 'main' program development.

Our contributions:

• We extend Featherweight Java (FJ) [15] to present a core calculus for Featherweight Java with behavioral annotations (we call the extension BAFJ) and sketch the proof of type safety. BAFJ allows annotations to be associated with different objects (not per class). These annotations can be used to enforce different runtime properties on the key operations of the individual object. We show a translation scheme to translate BAFJ programs to FJ in a type and semantics preserving way. The soundness proof guarantees that for each object, the set of properties guaranteed to hold at different program points (specified by the annotations statically) are guaranteed to hold at runtime.

• We incorporate behavioral annotations in Java;the new language is called BAJA. Our extension presents a modular approach to introduce barriers that a future language designer may use to support newer types of barriers. We have designed a compiler to translate BAJA code to Java.

• We propose a novel context insensitive, field sensitive, flow insensitive annotation flow analysis to aid the optimizations. We present a series of optimizations to generate efficient code.

• We have done a preliminary evaluation: we find that interesting annotations can be encoded concisely in BAJA. For a varied set of test programs, our optimizing-compiler generates code whose performance is comparable to that of hand-optimized code; maximum overhead 7.13%.

2 FJ EXTENDED WITH BEHAVIORAL ANNOTATIONS

We now present a core calculus for Java with behavioral annotations, as an extension to Featherweight Java (FJ) [15]. The syntax for this extension is presented in Fig. 2; we call the language *Behavioral Annotations* + FJ = BAFJ. The main extension in BAFJ is the introduction of annotations (Z); the original FJ grammar can be obtained by ignoring all the occurrences of the non-terminal Z and the *error* token. Unlike in FJ, where a type is specified by a class-name, in BAFJ an annotation is specified along with. We use $\overline{\mathtt{X}}$ to represent a sequence consisting of zero or more elements: $\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_n$, and '$\overline{\mathtt{Z}}\ \overline{\mathtt{C}}\ \overline{f}$' to represent a sequence of field declarations. Similarly '$\mathtt{this}.\overline{f}{=}\overline{f}$' represents a sequence of assignments to the fields in the current object this. We allow the mixing of notation, and use $\mathtt{X}_0\overline{\mathtt{X}}$ to denote the sequence $\overline{\mathtt{X}}$ prefixed with $\mathtt{X}_0$. Standard Java subtyping rules apply to assignments and casts. A BAFJ program is of the form $(\overline{\mathtt{C}}\ \ e)$, where $\overline{\mathtt{C}}$ is a sequence of class declarations, and e is a closed expression. For the ease of presentation, we assume that field names in the derived classes do not clash with those of the parent class.

An annotation in BAFJ is defined as a class and it can be instantiated with some parameters (to create an annotation-object), we call such a class as the annotation-class. This annotation-class may be written in FJ extended with a special *error* value (see Section 6, for an intuition behind this).

Each BAFJ object is associated with one or more annotations. BAFJ annotations help admit different desired-actions (e.g., predicates to establish different properties on the object under consideration) that are executed before/after a key operation. These desired-actions are implemented using different *barrier* functions defined in the annotation-classes associated with the object. If the barrier operations encounters an undesirable state, it will "fail" and return the special value *error*. In the BAFJ grammar (Fig. 2), the annotation specification is handled using the non-terminal Z. This specification includes the annotation-class name and the optional arguments required to invoke the constructor of the annotation-class. We will illustrate these concepts using three representative barriers: for creating an object (oWBar), reading the fields of an object (fRBar), and post conditions on method calls (mPostBar). Later in Section 3, we will also include fWBar (barrier for writing to the fields of an object) in our presentation. Our presented techniques can be intuitively extended to handle barriers on other key operations, as well.

2.1 BAFJ Type system

We extend the type rules of FJ to derive the type rules for BAFJ (see Fig. 3). An underlying guideline for designing the type system for BAFJ is that a well typed annotated-program type checks using FJ types rules [15] (with suitable extensions to handle *error*), if the type annotations are erased.

We first list some of the auxiliary routines of Igarshi et al [15]. The list of all the fields visible in a class C (including the fields of its super classes) is given by *fields*(C). If a method m is declared in a class C then its type given by *mtype*(m, C) and *mbody*(m, C) returns a pair $(\overline{x}, e)$ consisting of the sequence of parameters of m and the return expression. The types admitted by our type system can be generated by the following grammar:

$\mathtt{T} ::= [@\mathtt{S}, \mathtt{S}] \mid \mathtt{S} \rightarrow \mathtt{S} \mid \bot \qquad \mathtt{S} ::= \mathtt{T} \mid id$

The type of every object in BAFJ is given by a pair [@T1, T2]; we refer to T1 and T2 as the *annotation-type* and the *class-type*, respectively, of the object. The type of a method is of the form $[@\overline{\mathtt{R}}, \overline{\mathtt{T}}] \rightarrow [@\mathtt{R}', \mathtt{T}']$. The type of a constructor is given by the types of the arguments; we omit the return type. The type $\bot$ is the subtype of all the types and is used to typecheck the *error* expression (Rule 1). Object is the super-type of all the classes and AObj (a subtype of Object) is the super-type of all the annotation classes. We now describe the type rules, by mainly focussing on object creation and method typing. The rest of the rules are straightforward.

Error $error : \ \bot$ (1)

Variable lookup $\Gamma \vdash x : \ \Gamma(x)$ (2)

Field lookup
$$\frac{\Gamma \vdash e : [@A', C'] \quad fields(C') = [\overline{@A}, \overline{C}] \ \overline{f}}{\Gamma \vdash e.f_i : [@A_i, C_i]} \quad (3)$$

Method call
$$\frac{\Gamma \vdash e' : [@B, E] \quad mtype(m, E) = [\overline{@A''}, \overline{D}] \rightarrow [@A, C] \quad \Gamma \vdash \overline{e} : [\overline{@A'}, \overline{C'}] \quad [\overline{@A'}, \overline{C'}] \leq [\overline{@A''}, \overline{D}]}{\Gamma \vdash e'.m(\overline{e}) : [@A, C]} \quad (4)$$

Object creation
$$\frac{fields(A) = [\overline{@A'}, \overline{D}] \ \overline{f} \quad \Gamma \vdash \overline{e} : [\overline{@B}, \overline{E}] \quad fields(C) = [\overline{@A''}, \overline{D'}] \ \overline{f'} \quad \Gamma \vdash \overline{e'} : [\overline{@B'}, \overline{E'}] \quad [@AObj\overline{@B}, AObj\overline{E}] \leq [\overline{@A'}, \overline{D}] \quad [@AObj\overline{@B'}, A\overline{E'}] \leq [\overline{@A''}, \overline{D'}] \quad @A \ C : OK}{\Gamma \vdash \texttt{new} \ @A(\overline{e}) \ C(\overline{e'}) \ : \ [@A, C]} \quad (5)$$

Type casting
$$\frac{\Gamma \vdash e : \ [@A', D]}{\Gamma \vdash \ (@A \ C)e : \ [@A, C]} \quad (6)$$

Method typing
$$\frac{\overline{@A} \ \overline{D} : \overline{OK} \quad @A' \ C' : OK \quad \overline{x} : [\overline{@A}, \overline{D}], this : [@B, C] \vdash e : [@A'', E] \quad [@A'', E] \leq [@A', C']}{@A' \ C' \ m(\overline{@A} \ \overline{D} \ \overline{x}) \ \{\texttt{return} \ e;\} \ : \ OK \ IN \ C} \quad (7)$$

Class typing
$$\frac{\overline{@A'} \ \overline{D} : \overline{OK} \quad \overline{@A} \ \overline{C'} : \overline{OK} \quad K = C(\overline{@A'} \ \overline{D} \ \overline{g}, \ \overline{@A} \ \overline{C'} \ \overline{f}) \ \{ \texttt{super}(\overline{g}); \ this.\overline{f} = \overline{f}\} \quad fields(D) = \overline{@A'} \ \overline{D} \ \overline{g} \quad \overline{M} : \overline{OK} \ IN \ C}{\texttt{class} \ C \ \texttt{extends} \ D \ \{\overline{@A} \ \overline{C'} \ \overline{f}; \ K \ \overline{M} \ \} \ : \ OK} \quad (8)$$

Annotation typing
$$\frac{\begin{array}{l}\texttt{class A extends B } \{\overline{A'} \ \overline{C'} \ \overline{f}; K \ \overline{M} \ \}: \ OK \quad B \leq AObj \\ @A \ C \ oWBar \ (A \ C \ x) \ \{\texttt{return} \ e_3;\}: \ OK \ IN \ A \\ @A \ C \ fRBar \ (A \ C \ x, \ String \ f) \ \{\texttt{return} \ e_2;\}: OK \ IN \ A \\ @AObj \ Object \ mPostBar(A \ C \ x, \ String \ m, \\ \quad Object \ rv)\{\texttt{return} \ e_1;\}: \ OK \ IN \ A\end{array}}{@A \ C: \ OK} \quad (9)$$

Fig. 3. BAFJ Type rules

Variable lookup, field lookup, and method call: We use Γ to denote the type environment. Rules 2-4 are standard type rules, extended with behavioral annotations; we skip the details. We define the subtype relation ($\leq$) as follows: $[A, C] \leq [A', C']$, only if, $A \leq A'$ and $C \leq C'$. Like in FJ, the subtype relation ($\leq$) on the class names (such as, A and A′) is the reflexive and transitive closure of the immediate subclass relation established by the extends clauses in the class tree. Further, $\overline{X} \leq \overline{Y}$, iff $X_1 \leq Y_1, X_2 \leq Y_2, ..., X_n \leq Y_n$.

Object creation: The constructor of each object passes an implicit first argument that contains the object of the corresponding annotation. The object creation must ensure that the arguments to the constructors (including the implicit argument) of both the annotation and the class must match the respective fields. Further, the annotation and the class must match – @A C : OK; see the rule for 'annotation typing'.

Type casting: Our type cast rule is similar to FJ. For brevity, we skip the "stupid" cast warning [11] for classes and annotations.

Method typing: this rule is standard and we skip its discussion. Similar to FGJ [15] we allow covariant overriding on the method return type. This helps in admitting barrier methods in the derived classes (rules for overriding are similar to those in FGJ). For the ease of presentation, we avoid

function overloading. The type checking of constructors is not explicitly shown; it is done similar to other methods, by skipping the rules for the return type. Class typing rule is intuitive and is skipped.
Annotation typing: An annotation must be declared as a class which implements at least the three barrier functions `oWBar`, `fRBar`, and `mPostBar`. The annotation can be applied on a type `C` only if the barrier functions act as barriers on objects of type `C`; that is, the types of the barrier functions must be compatible with `C`. These barriers are invoked before constructing the object (`oWBar`), or before reading any field in the object (`fRBar`), or after the invocation of an instance method (`mPostBar`). The `oWBar` function takes as an argument the object (x) on which the annotation is attached; the return type of `oWBar` matches the type of x. Compared to `oWBar`, the function `fRBar` takes an additional string argument; this helps the barrier to be specialized for specific field. For this specialization to work, we assume that `FJ` is extended with a String type and String literals. Compared to `fRBar`, the function `mPostBar` takes a third argument (`rv`) that is used to specify the return value of the method under consideration. The return type of `mPostBar` should match that of `rv`. Though convenient for the purpose of illustration, in practice, we may avoid passing the string argument and instead specialize the barrier for each field / method name. The `AObj` class implements dummy barriers, wherein the functions `oWBar` and `fRBar` return the first argument and `mPostBar` returns the last argument.
Program typing: A given program is considered well typed if the "main" expression in the program is well typed.

2.2 BAFJ Operational Semantics

We now extend the operational semantics of `FJ` [15] to derive the operational semantics for `BAFJ`. We assume that the base `Object` class is modified to contain (i) a field `aLst` to store the attached annotation objects; (ii) routines `addAnn` and `remAnn` to add and remove annotations, respectively, from `aLst`; and (iii) wrapper methods, one each, corresponding to the barrier methods of `AObj` class. These wrapper methods call the corresponding barrier method of every attached annotation with appropriate arguments. A sketch of these additions is shown in Fig. 5.

Note that the type of the field `aLst` is `ArrayList`, which is not part of FJ. We can completely avoid such a modification and instead rely on a VM runtime data structure which can store the mapping from each object to its attached annotations.

Fig. 4 gives the operational semantics over the reflexive and transitive closure of the reduction relation $\rightarrow_V$. Each reduction step is of the form $e \rightarrow_V e'$. To avoid name conflict, we use `FJnew` to denote the `FJ` object allocation primitive. Besides `FJnew`, we use another intermediate object allocation operator (called `Tnew`) in our intermediate code; any user-defined field read of a `Tnew` object, or any user-defined method invocation of a `Tnew` object requires the invocation of the appropriate barrier (invoked via the corresponding wrapper function). We use v_i to denote non-error values, and define the admitted values in `BAFJ` as:
Values ::= `FJnew C`$(\overline{v})$ | `[(@A C)]Tnew C`$(\overline{v})$| *error*

In `BAFJ`, when we pass a value (*obj*) to a method with signature `[@A(`$\overline{t}$`), C]` $\rightarrow$ `[@A'(`$\overline{t'}$`), C']`, we attach an annotation object of type `A`$(\overline{t})$ to *obj* before the method invocation (represented by *obj*$\downarrow$`@A`$(\overline{t})$). Similarly, this annotation object is detached from *obj* when the method ends (represented by *obj*$\uparrow$`@A`$(\overline{t})$). Further, anytime an annotation is attached to an object, we invoke the corresponding `oWBar` to enforce the operations desired by the annotation; we call it the *annotation-type conformance*. We present an intuition on why such additional check is logical. Say, an annotation `@Div(`n`)` gives a guarantee on the divisibility by n. The expression '`new @Div(3) Integer(6)`' creates an `Integer` object with its value set to 6 and checks that it is divisible by 3. Now, say this value is to be passed to an formal argument whose declared type is `[@Div(4), Integer]`, this should

Computation

$$\frac{mtype(\texttt{K},\ \texttt{C}) = [\overline{\texttt{@A}'(\overline{t})}, \overline{\texttt{C}'}] \quad fields(\texttt{C}) = \overline{\texttt{@A}''(\overline{t'})}\ \overline{\texttt{C}''}\ \overline{f}}{\texttt{new @A}(\overline{u})\ \texttt{C}(\overline{v}) \rightarrow_V \texttt{Tnew C}(\texttt{FJnew A}(u, \overline{(\texttt{@A}'(\overline{t}))(v{\downarrow}\texttt{@A}''(\overline{t'}))})\texttt{.oWWrap}();} \quad (10)$$

$$\texttt{Tnew C}(\overline{v})\texttt{.}f_i \rightarrow_V \texttt{FJnew C}(\overline{v})\texttt{.fRWrap("}f_i\texttt{")}\texttt{.}f_i \quad (11)$$

$$\texttt{Tnew C}(\overline{v})\texttt{.m}(\overline{d}) \rightarrow_V \texttt{C}\ x\ \texttt{=}\ \texttt{FJnew C}(\overline{v})\texttt{;}\ x\texttt{.mPostWrap("m",}\ x\texttt{.m}(\overline{d})\texttt{)} \quad (12)$$

$$\frac{fields(\texttt{C}) = \overline{\texttt{@A}}\ \overline{\texttt{C}}\ \overline{f}}{\texttt{FJnew C}(\overline{v})\texttt{.}f_i \rightarrow_V v_i} \quad (13)$$

$$\frac{mtype(\texttt{m},\ \texttt{C}) = [\overline{\texttt{@A}(\overline{t})}, \overline{\texttt{C}'}] \rightarrow [\texttt{@A}'(\overline{t'}), \texttt{C}'']}{\texttt{FJnew C}(\overline{v})\texttt{.m}(\overline{d}) \rightarrow_V \texttt{((@A}'(\overline{t'}))(\overline{[d{\downarrow}\texttt{@A}(\overline{t})/\overline{x}]}\texttt{[FJnew C}(\overline{v})\texttt{/this]}e))\{\overline{d{\uparrow}\texttt{@A}(\overline{t})}\}} \quad (14)$$

$$\frac{\texttt{C} \leq \texttt{D} \quad (v_1\ \texttt{=}\ \texttt{FJnew A}'(\overline{w})\ \texttt{\&\&}\ \texttt{A}' \leq \texttt{A})}{\texttt{(@A}(\overline{u})\ \texttt{D)(FJnew C}(\overline{v})\texttt{)} \rightarrow_V \texttt{(@A}(\overline{u})\texttt{)((D)FJnew C}(\overline{v})\texttt{)}} \quad (15)$$

$$\texttt{(@A}(\overline{u})\texttt{)}e \rightarrow_V \texttt{FJnew A}(\overline{u})\texttt{.oWBar}(e) \quad (16)$$

Congruence

$$\frac{e \rightarrow_V e'}{e.f \rightarrow_V e'.f} \quad (17)$$

$$\frac{e \rightarrow_V e'}{e.m(\overline{e_1}) \rightarrow_V e'.m(\overline{e_1})} \quad (18)$$

$$\frac{e \rightarrow_V e'}{(\texttt{@A C})e \rightarrow_V (\texttt{@A C})e'} \quad (19)$$

$$\frac{e_1 \rightarrow_V e_1'}{\begin{array}{c}\texttt{v.m}(v_1, v_2, \cdots e_1, \cdots e_n) \rightarrow_V \\ \texttt{v.m}(v_1, v_2, \cdots e_1', \cdots e_n)\end{array}} \quad (20)$$

$$\frac{e_1 \rightarrow_V e_1'}{\begin{array}{c}\texttt{new @A}(v_1, v_2, \cdots e_1, \cdots e_n)\ \texttt{C}(\overline{e}) \rightarrow_V \\ \texttt{new @A}(v_1, v_2, \cdots e_1', \cdots e_n)\ \texttt{C}(\overline{e})\end{array}} \quad (21)$$

$$\frac{e_i \rightarrow_V e_i'}{\begin{array}{c}\texttt{new @A}(\overline{u})\ \texttt{C}(v_1, v_2, \cdots e_i, \cdots e_n) \rightarrow_V \\ \texttt{new @A}(\overline{u})\ \texttt{C}(v_1, v_2, \cdots e_i', \cdots e_n)\end{array}} \quad (22)$$

Error handling

$$error.f \rightarrow_V error \quad (23)$$

$$error.m(\overline{e}) \rightarrow_V error \quad (24)$$

$$\texttt{(@A C)}\ error \rightarrow_V error \quad (25)$$

$$v.m(v_1, v_2, \cdots v_j, error, e_k, \cdots e_n) \rightarrow_V error \quad (26)$$

$$\texttt{new @A}(v_1, v_2, \cdots, v_j, error, e_k, \cdots e_n)\ \texttt{C}(e) \rightarrow_V error \quad (27)$$

$$\texttt{new @A}(\overline{u})\ \texttt{C}(v_1, v_2, \cdots, v_j, error, e_k, \cdots e_n) \rightarrow_V error \quad (28)$$

Fig. 4. BAFJ: Reduction rules

evaluate to *error*. However, applying only the type rules (to enforce subtyping checks) does not disallow such an argument passing, which is against the intended semantics.

Object allocation rule (Rule 10): To create an object (of class-type C, with constructor K) we pass an annotation object (of type A) as the first argument; this annotation object is stored in the aLst field of Object. Since the annotation class is written in FJ, its constructor is not modified. The created object invokes the oWWrap method which in turn invokes the oWBar method of the annotation. The actual arguments' annotation-type conformance is enforced by Rule 16.

The user-defined field read and user-defined method invocation rules on a Tnew object invoke the fRWrap and mPostWrap functions, respectively, to execute the corresponding barriers (Rules 11 and 12). Note: $x = e$ is a syntactic sugar for new Holder(e).f, where Holder is a class with a field f and whose constructor sets f to e. Similarly, $e_1; e_2$ is a syntactic sugar for new Seq(e_1,e_2).f, where Seq is a class (with a field f) whose constructor sets f to e_2.

```
class Object{ArrayList aLst=new ArrayList();
 Object() { }
 Object(AObj ao) {addAnn(ao);}
 void addAnn(AObj a) {aLst.add(a);}
 void remAnn(AObj a) {aLst.remove(a);}
 Object oWWrap() {
  for(AObj a:aLst) a.oWBar(this);
  return this;}
 Object fRWrap(String f) {
  for(AObj a:aLst) a.fRBar(this,f);
  return this;}
 Object mPostWrap(String m, Object rv) {
  for(AObj a:aLst) a.mPostBar(this,m,rv);
  return rv;}}
```

Fig. 5. Sketch of the modifications to the `Object` class.

The field read rule on the `FJnew` objects is standard (Rule 13). Rule 14 enforces that the annotation type of the return expression conforms to the return type of the method; we use $[x/y]$ to indicate that all occurrences of y are replaced with x.

The type casting (Rule 15) not only ensures the subtyping relationship on both the class and its annotation, but also ensures annotation-type conformance. Note: the first element (v_1) of the argument list $\overline{v}$ holds the annotation object.

Unlike the operational semantics rules of `FJ` [15], where the order of evaluation is not completely fixed (for function arguments), we ensure a left to right evaluation (Rule 20), to keep it closer to Java. Our rules ensure that in a `BAFJ` object allocation operation, the arguments to the annotation constructor are evaluated before the arguments to the class constructor (Rules 21, 22). The reduction rules (23-25) involving error values are standard: any expression which has *error* as a component, evaluates to *error*.

Considering the intermediate nature of the `Tnew` operator, if the reduction terminates with a value containing one or more `Tnew` operators then we clean it up by replacing every such occurrence with `FJnew`. This results in a desirable property that a given `BAFJ` program reduces to an `FJ` object or *error*.

`BAFJ` Type soundness: The type system presented in section 2.1 is sound – a well typed program does not go wrong. We state the same using a type soundness theorem.

Definition 2.1. Available annotations of an object. The annotation added to an object x during its creation (using the new-statement) is considered available for x during its complete lifetime. If an object x is passed as a parameter to a function then the annotation on the corresponding formal parameter is considered available for x until the return from the function.

Definition 2.2. An expression e is stuck if either (i) e cannot take a reduction step and e is not a value, or (ii) e refers to an object x that does not include an annotation from the set of available annotations of x.

A program $(\overline{\mathrm{C}}\ \ e)$ goes wrong, if $e \rightarrow_V^* e'$ and e' is stuck.

A simple example of a stuck program due to condition (ii) can involve an expression having an object o that was created with an immutability annotation (like the `Imm` annotation shown in Fig. 1(c)), but not having the annotation at some point in the program. A consequence of such a lapse would permit the update of any field of o – against the expected behavior of an object annotated with such an annotation.

	⟦BAFJ code⟧ ⟶ FJ translation
1.	⟦`class C1 extends C2{`$\overline{\texttt{@A(e)}}$ $\overline{\texttt{C}}$ $\overline{f}$`;K` $\overline{\texttt{M}}$`}`⟧ ⟶ `class C1 extends C2{`$\overline{\texttt{C}}$ $\overline{f}$`;`⟦K⟧ $\overline{⟦\texttt{M}⟧}$`}`
2.	⟦`C(`$\overline{\texttt{@A}(\overline{e})}$ $\overline{\texttt{C}'}$ $\overline{f}$`) { super(`$\overline{f}$`); this.`$\overline{f}$ `=` $\overline{f}$`; }`⟧ ⟶ `C(AObj f0,` $\overline{\texttt{C}'}$ $\overline{f}$`){ super(f0,`$\overline{f}$`);` $\overline{⟦f\texttt{↓@A}(\overline{e})⟧}$`; this.`$\overline{f}$`=`$\overline{f}$`;}`
3.	⟦`@A'(`$\overline{t'}$`) C'` $m(\overline{\texttt{@A}(\overline{t})}\ \overline{\texttt{C}}\ \overline{x})$ `{ return e; }`⟧ ⟶ `C'` $m(\overline{\texttt{C}}\ \overline{x})$ `{` ⟦$\overline{x}$↓$\overline{\texttt{@A}(\overline{t})}$⟧`; C' t_ret=`⟦e⟧`;` ⟦$\overline{x}$↑$\overline{\texttt{@A}(\overline{t})}$⟧`; return new A'(`$\overline{t'}$`).oWBar(t_ret); }`
4.	⟦$e.f$⟧ ⟶ *// say, typeOf(e) =* `[@A, C]` ; `((C)`⟦e⟧`.fRWrap( "`f`")).`f
5.	⟦$e'.m(\overline{e})$⟧ ⟶ *// say, typeOf(e')=* `[@A''(`$\overline{t}$`), C]`, *//mtype(m,* `C`*)=* `[`$\overline{\texttt{@A}}$`,` $\overline{\texttt{D}}$`] → [@A', C']`; `C` x `=` ⟦e'⟧`;` `(C')`x`.mPostWrap("`m`",` x`.`m`(`$\overline{⟦e⟧}$`))`
6.	⟦`new @A(`$\overline{e}$`) C(`$\overline{e'}$`)`⟧ ⟶ `(C)(new C(new A(`$\overline{⟦e⟧}$`),` $\overline{⟦e'⟧}$`).oWWrap())`
7.	⟦`((@A(`$\overline{t}$`) C)`e⟧ ⟶ `new A(`$\overline{t}$`).oWBar((C)`⟦e⟧`)`
8.	⟦x`↓@A(`$\overline{e}$`)`⟧ ⟶ `A` t_x `=new A(`⟦e⟧`);` t_x`.oWBar(`x`);` x`.addAnn(`t_x`);`
9.	⟦x`↑@A(`$\overline{e}$`)`⟧ ⟶ x`.remAnn(`t_x`);`

Fig. 6. Rules to translate BAFJ programs to FJ

THEOREM 2.3. **Type Soundness**: *If a program* (C e) *typechecks then* e *does not go wrong.*

The proof follows the standard techniques [39] and is skipped for brevity. The soundness property ensures that an object expected to exhibit a certain behavior (owing to the associated annotations) at a certain program point will exhibit the expected behavior.

2.3 Translating BAFJ to FJ

We now present a scheme to translate BAFJ programs to FJ (extended with *error*). We present the cross-compilation rules to translate statements present in each of the syntactic categories of BAFJ in Fig. 6. Our translation ensures that if the input BAFJ program type checks, the generated code will also type check. Such a translation can form the basis of writing a cross compiler for BAFJ.

The translation uses an Object class as described in Fig. 5. The annotations on field declarations, formal arguments, and return types are erased. The signature of each constructor is modified to accept one additional argument for receiving the annotation object for the object under consideration; the annotation object is recursively passed and is used to initialize aLst in the constructor of the Object class. We use the auxiliary routine ftype to get the types of fields of a class. Corresponding to the annotation present in the declaration of each of the formal arguments (say, x), a method/constructor declaration is modified to create an annotation object (say, pointed to by a special temporary t_x), invoke the corresponding oWBar method, and attach the annotation object to the received argument. We refer to this annotation object as the *declared-annotation-object*. Additionally, in case of a method declaration, we further detach the annotation object after the function call, and invoke the oWBar (corresponding to the return annotation type) on the return value.

The field read, method call, and object creation operations invoke the appropriate barriers. These translation rules follow the corresponding rules from Fig. 4. Since the return type of the wrapper methods is Object, we insert an appropriate type cast. The typecast expressions are translated to ensure that the type is checked for both the annotation and the 'main' object in that order.

```
class AObj<T extends Object> {
  public T oWBar(T a) { return a;}
  public void fRBar(T a, String f) {}
  public void mPostBar(T a, String m,
            Object rv, Object... args) {} }
```

Fig. 7. AObj (Parent class of each annotation)

3 BEHAVIORAL ANNOTATIONS FOR JAVA (BAJA)

We have designed a new extension to the Java language (called BAJA) that admits behavioral annotations. We have extended the translation rules discussed in Section 2.3 to write a BAJA compiler that translates BAJA code to traditional Java code. We now discuss some salient features of our implementation.

An interesting feature of our BAJA compiler implementation is that the translated code requires no modifications to the Java compiler or runtime.

The extension to the Java grammar is minimal (just around 20 lines of changes to the publicly available Java grammar present in JavaCC format). Compared to BAFJ, we allow the barrier code to include not just predicates, but any arbitrary code. Further, we use a `throw` statement to simulate the behavior of the *error* expression discussion in Section 2. Though we limit our discussion here to one annotation per BAJA type, extending it to multiple annotations is fairly straightforward.

The AObj class (Fig. 7) is the parent class of all the annotation classes. It provides a default implementation for the barrier methods, which can be overridden by any of the derived classes, thereby admitting custom barrier-routines. The signature of mPostBar is modified to pass the arguments of the methods on which it is called. A restriction that is imposed on the AObj and all its derived classes is that the constructors and the barrier methods must be defined in `public` scope. To enforce good programming practice, we recommend that the arguments to the annotation be immutable.

Considering our design goal of not modifying the existing library classes (including the Object class) we create a new class NObject (stands for 'New Object' class) and make it to be the super-type of all the user defined classes. The NObject class has some minor differences compared to the Object class defined in section 2.2. The addAnn/remAnn method adds/removes the annotation object aObj to/from aLst, only if aObj $\neq$ null. Similar to the modification done to mPostBar in AObj (Fig. 7), the signature of mPostWrap is modified to accept arguments (code not shown).

The BAJA to Java translation closely follows the rules explained in Fig. 6, but for the differences arising due to the extended syntax of Java (compared to FJ). For brevity, in Fig. 8, we discuss only those differences. Since Java allows explicit field write operations, we now include the associated barrier fWBar and its corresponding wrapper method fWWrap in our translation.

Class Declarations: For each class X, we use Rule 1 or Rule 2, depending on whether it is used as an annotation or not. If class X is declared as `class X extends Y` and X is used as an annotation, then Y is assumed to be used as an annotation as well.

Field and Variable Declaration: Rule 3 gives the translation for a field (or variable) declaration if the field (or variable) has a declared annotation. For each such field (or variable) x, a new declared-annotation-object is created and attached to a temporary t_x, unique to the field (or variable) x.

Constructor Declaration: During the translation of a constructor (Rule 4), new temporary variables are created to point to the declared-annotations-objects of the formal arguments. We also invoke the corresponding oWBar method on the received arguments. Note that the notation $x{\downarrow}\texttt{@A}(\overline{e'})$ (and $x{\uparrow}\texttt{@A}(\overline{e'})$) is similar to that of Fig. 6, except that it adds (removes) the annotation to (from) the

	⟦BAJA code⟧ ⟶ Java translation
1	/* *if* X *is used as an annotation* */ ⟦`class X`⟧ ⟶ `class X⟨T⟩ extends AObj⟨T⟩`
2	/* *if* X *is not used as an annotation* */ ⟦`class X`⟧ ⟶ `class X extends NObject`
3	⟦`@A(`$\overline{e}$`) C x;`⟧ ⟶ `C x;` `A t_x = new A(`$\overline{e}$`);`
4	⟦K`(`$\overline{\texttt{@A(}\overline{e}\texttt{)}\ \texttt{C}\ x}$`) {`$e'$`}`⟧ ⟶ `K(AObj f0,` $\overline{\texttt{C}\ x}$`) {` ⟦$\overline{x}$`↓@A(`$\overline{e}$`)`⟧`;` ⟦e'⟧`;` ⟦$\overline{x}$`↑@A(`$\overline{e}$`)`⟧`;}`
5	⟦$x = e$⟧ ⟶ `// Say typeOf(`x`) =[@A(`$\overline{t}$`), C]` `if(`x` != null)` x`.remAnn(`t_x`);` x `=` ⟦e⟧`;` t_x`.oWBar(`x`);` `if(`x` != null)` x`.addAnn(`t_x`);`
6	⟦$x.f = e$⟧ ⟶ `// Say typeOf(`x`)=[@A(`$\overline{t}$`), C] and typeOf(`f`,C)=[@A′(`$\overline{t'}$`), C′]` x`.fWWrap("`f`");` `if(`$x.f$`!=null)`$x.f$`.remAnn(`$x.t\_f$`);` $x.f$ `=` ⟦e⟧`;` $x.t\_f$`.oWBar(`$x.f$`);` `if(`$x.f$`!=null)`$x.f$`.addAnn(`$x.t\_f$`);` `if(`$x \neq$ `this) { `x`.oWWrap(); }`
7	⟦$y.m(\overline{e})$⟧ ⟶ `// Say typeOf(`y`)=[@B′, B] and mtype(B,`m`)=[`$\overline{\texttt{@A(}\overline{t}\texttt{)}}$`,` $\overline{\texttt{X}}$`] ⟶ [@A′, Y]` $\overline{\texttt{X}}$ $\overline{t\_arg}$ `=` ⟦$\overline{e}$⟧`;` `Y t_ret =` y`.`m`(`$\overline{t\_arg}$`);` y`.mPostWrap("m", t_ret,` $\overline{t\_arg}$`);` } // *if (returnType(m) != void)* y`.`m`(`$\overline{t\_arg}$`);` y`.mPostWrap("`m`", null,` $\overline{t\_arg}$`);` } // *if (returnType(m) == void)* `if (`$y \neq$ `this)` y`.oWWrap();`

Fig. 8. BAJA to Java translation rules

```
class C extends NObject{ int f1, f2;
  C(AObject a) {super(a);}
  void foo(C x) {
    x.fRWrap("f1"); x.fWWrap("f2");
    x.f2 = x.f1 + 13;
    if (x != this) x.oWWrap(); } ...
  bar() {
    C x1=new C(new Imm()); x1.oWWrap();
    C x2=new C(new Log("f1")); x2.oWWrap();
    foo(x2); foo(x1); } ...
```

Fig. 9. Translated code for the input code in Fig. 1(b).

`aLst` only if x is non-null. Since Java requires that the call to the constructor of a parent class (using `super`) can only be present as the first statement in the constructor, we introduce the above mentioned code immediately after the `super` call. The translation of the `super` call is similar to that discussed in Section 2.3. For all the constructors that have no explicit `super` call, we introduce '`super(f0);`' as the first statement in the constructor.

Before returning from the constructor or method, if the object pointed to by a formal argument x

is not `null` then the declared-annotation-object (pointed to by the temporary t_x) is detached by invoking x.`remAnn`(t_x). Additionally, in a similar manner, the declared-annotation-objects on the local variables are removed from the objects pointed to by the local variables. This is not explicitly shown in Fig. 8.

Variable and Field Assignment: If a variable (or field) has a declared annotation, Rule 5 (Rule 6) of Fig. 8 gives the translation for the variable (field) assignment. The translation for field assignment starts by emitting code to invoke the `fWWrap` to execute the field-write related barriers. Before assigning the expression (say e) to a field (variable), if the object pointed to by the field (variable) is not `null` the annotation object corresponding to the declared annotation is removed. After the assignment, `oWBar` is invoked on the declared-annotation-object of the field (variable). This is done to check if the object pointed to by the field/variable conforms to the desired annotation property of a declared annotation. If the expression e doesn't evaluate to `null` then the above mentioned declared-annotation-object is added to the object pointed to by the field (variable) after the assignment.

Any update to the fields of an object needs to reaffirm that the resulting object conforms to the behavior defined in the annotation of the object. Hence, in Rule 6, we also invoke the `oWWrap` on the *enclosing object* of the field. We say an object $o1$ to be an enclosing object of object $o2$, if one of the fields of $o1$ points to $o2$. Considering the aborting semantics we use, we do not get a chance to revert the field update if the corresponding `oWBar` throws an *error*. Once we allow exceptions to be thrown, such rollback code can be put in a catch block. In practice, an internal method may be called to update multiple fields of an object and the object may conform to the annotation only after the completion of all the updates. Hence, we do not invoke the `oWWrap` function when we are updating a field in the `this` object.

Method calls: The generated code differs slightly, based on whether the method returns a value or not. Similar to field update, we do not invoke the `oWWrap` function after a method call when the receiver of the callee and the caller match.

Object allocations in the absence of annotations: Unlike `BAFJ`, the annotations during the object allocations are optional in `BAJA`. In such cases, `null` is passed as a first argument to the constructor.

Example: Fig. 9, shows the translated code, for the input `BAJA` code shown in Fig. 1(b). Two salient points to note are: i) a new constructor has been added, ii) the `fRWrap` and `fWWrap` functions are invoked before the statement (to avoid a useless type-cast expression, if it is present inline in the expression), and (iii) the `oWWrap` is invoked to ensure that none the annotations are violated because of the write. Note: since the fields `f1` and `f2` are scalar fields and have no annotations on them, no calls are added to add/remove annotations or invoke the `oWBar` function (shown in Rule 5, Fig. 8).

3.1 Extensions

Our proposed translation scheme poses interesting challenges in the context of behavior annotations on objects of standard library classes (such as on Vector, or Integer), primitive types (such as int) and arrays, which are discussed below.

3.1.1 Handling library classes. Our treatment of behavioral annotations involves modifying the declaration of each class. However, this is not feasible in the context of library classes, or classes extending library classes. Further, we also want to preserve the features of *auto-boxing* and *auto-unboxing* (admitted on classes such as `Integer`, `Double`, `Boolean`, and so on). We handle the challenge of library classes by using a two phase pre-pass, before invoking the `BAJA` to Java translator.

Phase I: If there exists a type expression of the form `@A(`$\bar{t}$`) C`, where `C` is a library class, then we replace all the occurrences of `C` in the program with a special *boxed* representation that can

```
class BBox<T extends Object> extends NObject{
 private T obj;
 public BBox(AObj ann,T x){super(ann);obj=x;}
 public final T get(){ return obj; }
 public final void oWWrap(){
  for(AObj ann:aMap.keySet())ann.oWBar(obj);}
 ...
 public String toString(){return obj.toString();}}
```

Fig. 10. Declaration of the class `BBox`

```
public class Odd extends AObject<Integer> {
   public Integer oWBar(Integer x) {
      assert (x % 2) == 1; return x; } }
```

(a) Annotation class Odd.

```
@Odd int i = 3;
int x = i + 2;
```

(b) Input Snippet.

```
@Odd Integer i=new Integer(3);
Integer x=new Integer(i +
                new Integer(2));
```

(c) Code after handling primitive types.

```
@Odd BBox<Integer> i = new BBox<Integer>(3);
BBox<Integer> x=new BBox<Integer> (i.get() +
                  (new BBox<Integer>(2)).get());
```

(d) Code after Phase I and II.

```
Odd t_i=new Odd();
BBox<Integer> i=new BBox<Integer>(null, 3);
t_i.oWBar(i); if (i != null) i.addAnn(t_i);
BBox<Integer> x=new BBox<Integer>(null, i.get()+
            (new BBox<Integer>(null, 2)).get());
```

(e) Code after `BAJA` to Java translation.

Fig. 11. Handling of primitive and library classes: (a) Annotation code, (b) code with annotation on primitive data, (c) translated to code with annotation on library class objects, (d) final Java code.

support invocation of different barriers. For example, `Integer` is replaced by `BBox<Integer>`. A snippet of the `BBox` class is shown in Fig. 10. For brevity, we show the definition of only one of the wrappers. Since the barrier methods are defined on the underlying boxed object (`obj`), the wrapper methods invoke the corresponding barrier methods by passing `obj` as an argument.

Phase II: For each dereference of an object, say returned by an expression e_i, whose type was modified in the Phase I, we replace it by $e_i.get()$.

3.1.2 Handling primitive types and literals. The `BAJA` translator requires that the annotations are present only on non-primitive data. This poses a challenge to associate annotations on values of primitive types and literals. We follow a two step process to address this issue: Step I: If an

```
public class DeepAObject<T> extends AObj<T>{
  public DeepAObject() {}
  public DeepAObject(DeepAObject a){ super(a);}}
```

Fig. 12. `DeepAObject` (Parent class of each deep-annotation)

annotation is found on a primitive data type (such as `int, double, boolean`, and so on), then the translator replaces the primitive data type with its corresponding Boxed variant (such as `Integer, Double, Boolean`, and so on). Fig. 11(b) shows an example `BAJA` program that uses literals and has annotations on primitive data. After the step I, all the occurrences of `int` are replaced with `Integer`. Step II: If the type of a primitive typed expression e_i is t, and t was replaced with its boxed version T in Step I, then we replace e_i with '`new` T`(`e_i`)`'. For the example shown in Fig. 11(b), we find three expressions whose type is `int`: `3`, `2`, and `i + 2`. Each of these expressions are replaced in this step to result in the code shown in Fig. 11(c). Java's auto boxing/unboxing feature guarantees that it is type-safe to make these changes. After this phase, we invoke the previous two-phase pre-pass to process `BAJA` code involving library classes. For the above discussed example, Fig. 11(d) shows the code generated after the pre-pass, and Fig. 11(e) shows the final code after `BAJA` to Java translation.

3.1.3 Handling Arrays. The `BAJA` translator handles the annotations on arrays in a similar manner as the annotations on library types. The barrier methods `fRBar` and `fWBar` of class `AObj` are overloaded, as the user may want to enforce properties based on the array index:

```
public void fWBar(T obj, int index){}
public void fRBar(T obj, int index){}
```

3.1.4 Deep Annotations. The annotations we have seen so far are attached to an object and are not propagated to the objects that are reachable by one or more levels of dereference. In practice, we also see the need of annotations that have to be attached with an object and all the reachable objects thereof. Let us consider a *deep* variation `dImm` of the immutable annotation discussed in Section 1, such that when a complex graph object annotated with `dImm`, it should not permit the modification of any part of the graph (not just the immediate fields of the object). One way to realize such a feature is by writing an annotation class that disallows any writes to the fields of the class, attaching this annotation to the desired object *obj* and then ensuring that this annotation is attached to all the reachable objects of *obj*. In `BAJA`, if a programmer wants to write such a deep annotation then the corresponding annotation class should be implemented as a subclass of `DeepAObject` class (Fig. 12).

To admit deep annotations we modify two existing methods (`addAnn` and `remAnn`) and add two other (`deepAddAnn` and `deepRemAnn`) in the `NObject` class. The modified method `addAnn` (`remAnn`) first adds (removes) the argument annotation object to (from) `aLst`. Then it calls the method `deepAddAnn` (`deepRemAnn`) if the annotation object is a subtype of a `DeepAObject` class. The method `deepAddAnn` (`deepRemAnn`) recursively adds (removes) the annotation object to (from) the `aLsts` of all the objects reachable via the fields of the current object; we use reflection to access all the fields of the current object. For brevity, we avoid presenting the actual code for `deepAddAnn` and `deepRemAnn`. One limitation of the deep annotations is that they can only be associated with user classes and not with library classes; the fields of the library method do not extend `NObject` and hence do not admit annotations.

Example: Fig. 13 demonstrates the use of `dImm` (defined in Fig. 13(a)). An object of type `C`, created at Line#5 of Fig. 13(b), is annotated with `dImm`. An object created at Line#3, but reachable from the root object, is modified at Line#6. As the root object is immutable the program should throw

```
class dImm extends DeepAObject<Object> {
  Object fWBar(Object obj, String f) {
    throw new ImmException(); } }
```

(a) Imm class (subclass of `DeepAObject`).

```
1  class D{ int x; }
2  class C{ D f;
3    public C(){ f = new D();}
4    main(){
5      C o1=new @dImm C(); D o2=o1.f;
6      o2.x=13;}}
```

(b) Sample program using annotation `dImm`

Fig. 13. Example to illustrate a need of `DeepAObject`.

```
public String sWrap() {        // In NObject
  for(AObj a:aLst)
    if (a overrides the sBar)
      return a.sBar(this);
  return this.toString(); }

public String sBar(T a) {      // In AObj
   return a.toString(); }
```

Fig. 14. Additions to `NObject` and `AObj` classes.

an exception at Line#6. Since `dImm` is declared as a subclass of `DeepAObject`, the annotation object gets attached to all the objects reachable from `o1`, and will throw the expected exception at Line#6.

3.1.5 Stringizing barrier - a practical extension. From our experience, we have seen the need of a new barrier called the *stringizing* barrier, which is essentially a special case of `mPreBar` for the `toString()` method. Such a 'barrier' controls the behavior of the object when it is "printed". Similar to the procedure we followed in introducing the three barriers discussed in Section 3, we introduce this new barrier. This gives the reader some more understanding on the ease with which new barriers can be added by the language designer.

As an example for stringizing barrier, say we want to write an annotation that will facilitate the printing of binary strings for integers. For example, if the variable contains the value '9' then printing the variable would output 1001. The rest of the operations on binary integers are same as that of normal integers.

To admit the stringizing barrier, we can extend `NObject` and `AObj` to include a wrapper function (`sWrap`), and a dummy barrier (`sBar`), respectively. A sketch of the sample extension is shown in figure 14. The programmer can now write his own extensions by overriding the `sBar` function in the annotation class. Since an object can be attached with multiple annotations, we have to choose which `sBar` method's return value is returned by `sWrap`. We break this tie among the attached annotations by returning the result of the first one that has overridden `sBar`. Instead of checking this using reflection (which is expensive), the `BAJA` compiler emits code to maintain a special bit in the annotation class to indicate if that the class has overridden `sBar` and check this bit in the `sWrap` method (code not shown, for brevity).

We show a possible implementation of the binary annotation in Fig. 15(a), a use thereof in

```
class binary extends AObj<Integer> {
   String sBar(Integer k) {
      return k.toBinaryString(); } }
```

(a) Annotation class `binary`.

```
Integer i=4;

@binary Integer j=9;




print(i.toString());
print(j.toString());
```

(b) Example program

```
BBox<Integer> i =
  new BBox(null, 4);
binary t_j=new binary();
BBox<Integer> j =
       new BBox(null,9);
if(j!=null) j.addAnn(t_j);
t_j.oWBar(j);
print(i.sWrap());/*prints 4*/
print(j.sWrap());/* 1001*/}
```

(c) Generated code

Fig. 15. 'Stringizing' barrier example with `binary` annotation.

Fig. 15(b), and the code generated after translation in Fig. 15(c). As shown in the figure, one advantage of such an annotation is that we can use a common routine to process all types of integers (such as, binary, normal, padded, and so on), without knowing the details of the particular annotation. Also, the annotations can be developed in a modular way.

4 BAJA OPTIMIZATION FRAMEWORK

Looking at the code generated by our translation (see Fig. 9) it can be seen that the translated code may incur a fair amount of overhead owing to the number of indirections (due to the wrappers and BBox related operations), and the use of collection classes (`aLst`). Further, since the wrapper methods invoke barrier methods on all the attached annotations, many times, this may result in the invocation of the default barrier methods of the `AObj` class, which can be avoided. In this section, we propose a series of optimizations to address these performance related issues, that have been implemented as part of the `BAJA` compiler. Many of these optimizations are based on a novel context insensitive, field sensitive, flow insensitive *annotation flow analysis*. Our proposed annotation flow analysis tracks the objects, along with their attached annotations, that flow into different expressions.

4.1 Annotation Flow Analysis

The annotation flow analysis is an extension of standard flow analysis [28] and is done in two stages: constraint generation and constraint solving. We generate two types of subset constraints: unconditional and conditional. An unconditional constraint is of the form $x \in X$ or $X \subseteq Y$. A conditional constraint such as $p \Rightarrow q$ states that q will hold if p holds.

4.1.1 Constraint Generation. We assume that every expression in the program has a unique label. We use l, l', k and k' to denote the labels of expressions. Similar to previous sections, `A` and `A'` are used to denote the annotation class names, and `C`, `C'`, `D` and `D'` for generic class names. Thus, $\texttt{C}^l$ denotes an abstract object (of type `C`) created at label l. We use *Labels* to denote the set of labels, *Ao* to denote the set of abstract objects, and *Fields* to denote the set of all fields. We use ρ to denote the abstract environment: *Labels* $\rightarrow$ *Ao*. For an expression e at label l (denoted as e^l), if $\texttt{C}^k \in \rho(l)$,

- for each $\texttt{new}^l$ $\texttt{@A}(\overline{e'})$ $\texttt{C}(\overline{e})$, if $\texttt{closure}(\texttt{K},\texttt{C}) = [l', \overline{x}, \phi]$

$$\texttt{C}^l \in \rho(l) \quad (29a) \qquad \rho(l) \subseteq \rho(l') \quad (29b)$$

$$\rho(\texttt{lab}(e_i)) \subseteq \rho(\texttt{lab}(x_i)) \quad (29c) \qquad \texttt{A}^l \in \sigma(l, \texttt{aset}) \quad (29d)$$

$$\texttt{D}^k \in \rho(\texttt{lab}(e_i)) \Rightarrow \texttt{vAtyp}(x_i, \texttt{K}, \texttt{C}) \subseteq \sigma(k, \texttt{aset}) \quad (29e)$$

- for every $a^{l'} = e^l$ in method $\texttt{m}$ of class $\texttt{C}$

$$\rho(l) \subseteq \rho(l') \quad (30a)$$

$$\texttt{D}^k \in \rho(l) \Rightarrow \texttt{vAtyp}(a, m, \texttt{C}) \subseteq \sigma(k, \texttt{aset}) \quad (30b)$$

- for every $e'.f = e^l$ in method $\texttt{m}$ of class $\texttt{C}'$

$$\texttt{C}^{l'} \in \rho(\texttt{lab}(e')) \Rightarrow \begin{cases} \rho(l) \subseteq \sigma(l', f) & (31a) \\ \texttt{D}^k \in \rho(l) \Rightarrow \texttt{fAtyp}(f, \texttt{C}) \subseteq \sigma(k, \texttt{aset}) & (31b) \end{cases}$$

- for every $e.f^l$ in method m of class $\texttt{C}$

$$\texttt{D}^{l'} \in \rho(\texttt{lab}(e)) \Rightarrow \sigma(l', f) \subseteq \rho(l) \quad (32)$$

- for every $\texttt{return}\ e^l\texttt{;}$ in method m of class $\texttt{C}$, where $\texttt{closure}(\texttt{m},\texttt{C}) = [l', \overline{e'}, r]$

$$\rho(l) \subseteq \rho(r) \quad (33)$$

- for every $e.m(\overline{e'})^l$ in method m' of class $\texttt{C}'$

$$\texttt{C}^{l'} \in \rho(\texttt{lab}(e)) \Rightarrow \begin{cases} \texttt{let closure}(m, \texttt{C}) = [k', \overline{x}, r] & \\ \texttt{C}^{l'} \in \rho(k') & (34a) \\ \texttt{D}^k \in \rho(\texttt{lab}(e_i')) \Rightarrow & \\ \quad \texttt{vAtyp}(x_i, m, \texttt{C}) \subseteq \sigma(k, \texttt{aset}) & (34b) \\ \rho(\texttt{lab}(e_i')) \subseteq \rho(\texttt{lab}(x_i)) & (34c) \\ \rho(r) \subseteq \rho(l) & (34d) \end{cases}$$

- for every $\texttt{A}$ that extends $\texttt{DeepAObject}$

$$\left.\begin{array}{l} \forall l' \in \mathit{Labels}, \forall f \in \mathit{Fields} - \{\texttt{aset}\} \\ \texttt{A}^l \in \sigma(l', \texttt{aset}), C^k \in \sigma(l', f) \end{array}\right\} \Rightarrow \texttt{A}^l \in \sigma(k, \texttt{aset}) \quad (35)$$

Fig. 16. Annotation flow analysis.

then it indicates that the abstract object $\texttt{C}^k$ may flow into e. We use σ to denote the our abstract heap: $\mathit{Labels} \times \mathit{Fields} \rightarrow \mathit{Ao}$. The flow information of a field f of an object created at label l is given by $\sigma(l, f)$. The annotation flow information of any object is stored in a special field $\texttt{aset}$. Thus, $\sigma(l, \texttt{aset})$ returns the set of abstract annotation objects that may get attached to the object created at label l.

We now briefly describe the auxiliary routines used here: We use the routine $\texttt{lab}(e)$ to get the label of the expression e. As a special case, for any variable x, $\texttt{lab}(x)$ returns the label of its declaration. Further, in a method m of class $\texttt{C}$, $\texttt{lab}(\texttt{this})$ returns the label of m in $\texttt{C}$. For a method m of a

class C, closure(m, C) returns a 3-tuple $[l, \overline{x}, r]$, where l is the label of the method m, $\overline{x}$ is the list of formal arguments and r is the label of the return expression. Consider a class C, a field f and a method m in C, and a variable x in m. We use vAtyp(x, m, C) and fAtyp(f, C) to return the singleton sets containing the abstract objects corresponding to the declared annotation of x and f, respectively. These routines return the empty set if the variable/field has no declared annotation. Fig. 16 shows the constraint generation scheme.

Object allocation: Eq. 29a, 29b and 29c are standard. The abstract annotation object, $\mathtt{A}^l$ is attached to the object created, by storing in aset (Eq. 29d). For every formal argument of a constructor, if it has a declared annotation, then we attach the corresponding abstract annotation object to the actual argument as well (Eq. 29e).

Variable/Field Assignment: The Eq. 30a and 31a are standard. The Eq. 30b and 31b indicate that the abstract objects corresponding to the declared annotations of the variable and field may flow into aset of the RHS abstract objects.

Field read and Return Expression: The rules for field read (Eq. 32) and return expression (Eq. 33) are standard.

Method call: For every element $\mathtt{C}^{l'}$ of the flow set of the receiver: if closure$(m, \mathtt{C}) = [k', \overline{x}, r]$, then (i) $\mathtt{C}^{l'}$ is added to the flow set of the label of method m in class C (Eq. 34a). (ii) for every formal argument of m in class C, if it has a declared annotation, then we attach the corresponding abstract annotation object to the actual argument as well (Eq. 34b). Eq. 34c and 34d are standard.

Propagating Deep Annotations: Finally, we also need to generate constraints to propagate the flow of deep annotations across the fields of each abstract object. This is done by adding further conditional constraints as shown in Eq. 35.

4.1.2 Constraint Solving. The algorithm used to solve the flow constraints is standard [28] and is skipped, for brevity.

4.2 BAJA Optimizations

We now discuss a series of ten optimizations implemented as a part of the BAJA compiler. These optimizations are divided into 2 categories: ones that need the annotation flow analysis (first seven), and ones that can be directly used during the code generation, without needing the flow analysis (the last three).

4.2.1 Reducing memory overhead. As a part of BAJA to Java translation, every user defined class is made a subclass of NObject (Rule 2, Fig. 8). If every object of a user defined class is neither annotated, nor any annotation flows (using annotation flow analysis) into it, then there is no need to make that class a subclass of NObject.

4.2.2 Objects with unique annotations. The NObject class discussed in Sections 3, and 4.2.8 uses collection classes to hold the annotation objects. For a given object, if it can be established (by the annotation flow analysis) that at most one annotation object will be attached to the object, we avoid the collection-class and instead use a simple class name (applicable to the annotations used in Fig. 9, 11(e), and 15(c)).

4.2.3 oWBar *after a side-effect free function.* Our translation scheme introduces a call to the oWBar after each method call (Rule 7, Fig. 8). If the method can be identified as a side-effect free method (one that does not modify any fields or globals), and that the oWBar method has no other functionality but to enforce some consistency related properties, then we can elide the call to the oWBar method.

4.2.4 Removing calls to dummy wrappers/barriers. It can be easily seen that all the barrier methods may not be overridden in an annotation class. In such a case, invoking wrapper methods, which in turn call the dummy barrier methods (barriers of the class `AObj`), is a wasteful exercise. We use the annotation flow analysis, to infer the set of annotation objects flowing into the objects that a variable/field may be pointed to, and which in turn helps us infer the wrapper methods that call only the dummy barrier methods; we completely elide such calls.

Further, for field and method related barriers, we analyze the barrier code and identify for each method and field dereference if the barrier call is dummy (by propagating the method name related string constant). e.g., Say the `mPostBar(T a, String m, Object rv, Object...args)` function of an annotation executes some specialized code, only if `m = "m1"`; else it just returns. Our analysis would conclude that this barrier is dummy for method calls other than `m1`.

4.2.5 Avoiding null checks. The translation rules (e.g., Rule 5, Fig. 8) introduce many conditional statements with explicit `null` checks. We use our annotation flow analysis (that includes the normal flow analysis) to identify and eliminate many of these useless conditional checks (e.g., the null check in Fig. 11(e)). Note that we cannot depend on the JVMs builtin optimization for these type null check elimination, as that builtin optimization is applicable only for the implicit null-checks resulting from the object dereference.

4.2.6 Avoiding boxed *representations.* We have seen in Section 3.1 that for every variable of library/array type, we create a corresponding generic *boxed* representation. This approach can lead to significant overheads, especially because at runtime many objects may not be associated with any annotation. We use the annotation flow analysis to identify if any annotation may flow into the objects pointed to by field/variable. Otherwise, the declared type of the field (or variable) is not replaced with the boxed variation. (e.g, In Figure 11(e), the code is optimized so that `x` can be declared as of `Integer` type).

4.2.7 `BBox` *removal for Java standard boxed types.* Note that the objects of standard boxed types (e.g., `Integer`, `Double`, and so on) are immutable; can't be updated once created. Our translation replaces all these types (say, T) with a `BBox`$\langle T \rangle$ type, so that we can store the annotation objects inside the `BBox` object. Consider a scenario where the only barrier method overridden by all the annotations to be attached to an object of such immutable type is `oWBar`. In such cases (identified by the annotation flow analysis), addition and removal of the annotations from the object, becomes unnecessary and instead we directly call the `oWBar` method on the object and avoid storing the annotation object, altogether. To facilitate this process further, the `AObj` class is updated to include additional `oWBar` methods that take individual standard boxed types as arguments. (e.g. `oWBar(Integer i)`). After applying the previous and current optimizations, the code shown in Fig. 11(e) gets translated to:

```
int i = 3; new Odd().oWBar(i); int x = i + 2;
```

4.2.8 Efficient storage for annotation. One performance issue with the representation of `NObject`, discussed in Section 3, is that it may end up storing many copies of an annotation object (especially, in the context of recursive procedures, where formal arguments have declared annotations). Say, we are guaranteed that (i) no object is attached to two annotation-objects of the same annotation class, with varying values of their instance data, and (ii) all of the barrier routines are side effect free; we call it the *single-annotation-object* scenario. In such cases, we can improve the efficiency of the generated code, if we replace the annotation object storage data structure, in the `NObject` class, from an `ArrayList<Object>` to an efficient `LinkedHashMap<AObj,Integer>`, which instead of storing an annotation object multiple times, stores a single instance of each annotation object (in the order in which it gets attached) and the number of times it is attached. The methods

`addAnn` and `remAnn`, and all the wrapper methods are modified to update the frequencies in the hash-map. This optimization is applicable to all the annotations used in Fig. 9, 11(e), and 15(c).

4.2.9 Singleton annotations. The annotation which stores neither the state information of the annotation object nor the state information of the object to which it is attached to, is called as a *singleton* annotation. A single instance of a singleton annotation can safely be created and shared between all the objects of the program to which it is attached to. The `Odd` and `binary` annotations (Fig. 11(a) and 15(a)) are examples of *singleton* annotations. We use the *singleton design pattern* to implement *singleton* annotations: A `private` field `single` of annotation type is added and initialized (e.g., `private static Odd single = new Odd()`), in the annotation class. A `public static` method `getInstance` is added to the annotation class, which returns the annotation object stored in the field `single`. If an annotation is of type *singleton* then every object allocation expression of it is replaced by a call to the `getInstance` method.

4.2.10 Inlining accesses to the object enclosed in `BBox`. As discussed in Section 3.1, we create a `BBox` to admit annotations on objects of library/array type. As a result, each access to the boxed value of these objects go through a `get` method of the `BBox` object. This increases the execution time significantly. To overcome this challenge, we inline the `get` method of the BBoxed class at every point. To further improve the code (that reduces the number of dereferences), we maintain an additional local temporary variable that points to the boxed value, and each access to the boxed value is now done via the temporary variable.

5 EVALUATION

Our implementation of the `BAJA` to Java compiler uses JTB [27] to generate the annotated syntax tree visitors and JavaCC to generate the parser generator. The compilation is divided into three passes. In the first pass, it collects the `BAJA` type information, The second pass does the annotation flow analysis and the third pass generates Java code, optimized using the techniques discussed in Section 4. The second pass and the optimizations invoked in the third pass are optional (invoked using a compiler switch). In the absence of this compiler switch, the `BAJA` compiler emits the unoptimized code.

To argue about the versatility of the behavioral annotations, ease of use and the effectiveness of our proposed optimizations, we first identify a set of interesting annotations, covering various types barriers, discussed in this paper. Fig. 17 lists a series of annotations: (1) `NegCycle` enforces that there are no negative edge cycles in the graph. (2) `dImm` implements the deep immutable annotation (Fig. 13). (3) `NonNull` annotation on a variable/field ensures that it never points to `null`. (4) `Balanced` ensures that after the insertion of any new node, the tree remains balanced. (5) `Sorted` ensures that the argument array to the binary-search method is sorted. (6) `ReadWriteCount` keeps a log of the number of reads and writes of array elements. (7) `NonNegEdge` enforces that the graph has no edges with negative weight. (8) `ImmGraph` disallows adding or removing of edges in the graph. (9) `GraphAffinity` enforces user specified upper limits on the in and out degree of any graph node. (10) `PositiveNumber` ensures that the argument to Fibonacci function is a positive number. As it can be seen (in column 2), these annotations can be implemented with very little coding (3-30 effective lines of code, excluding the standard import statements). The fourth column in Fig. 17 lists the different barriers overridden by each of these annotations.

Fig. 17 also lists a set of synthetic test programs, taken from some well known texts [7, 12], used for evaluation: BF (Bellman Ford BFS), BS (Binary Search), SSSP (Single Source Shortest Path algorithm of Dijkstra), SCC (Strongly Connected Component algorithm), AT (AVL tree and Binary search tree implementation), QS (Quick Sort), GR (Operations on Graphs represented as adjacency lists), FIB (Fibonacci function). We use two versions of these programs: (i) written in `BAJA`, and (ii) hand optimized Java code implementing similar behavior. For the annotations listed in Fig. 17, the

Seq	Annotation	ELOC	Barrier(s) overridden	Test Programs
1.	NegCycle	10	oWBar	BF
2.	dImm	3	fWBar	BS
3.	NonNull	5	oWBar	BS,SSSP,SCC
4.	Balanced	9	mPostBar	AT
5.	Sorted	16	oWBar	BS
6.	RWCount	16	fRBar, fWBar, sBar	QS
7.	NonNegEdge	7	oWBar	SSSP
8.	ImmGraph	4	mPostBar	SCC
9.	GraphAffinity	29	oWBar, mPostBar	GR
10.	PositiveNum	5	oWBar	FIB

Fig. 17. Annotations used for comparisons.

Test programs	Number of lines of code		Diff	% Diff
	Hand Optimized	BAJA		
BS	120	97	23	19.1
BF	185	129	56	30.3
QS	108	96	12	11.1
SSSP	242	162	80	33.1
SCC	220	164	56	25.5
GR	136	129	7	5.2
AT	297	265	32	10.8
FIB	30	32	-2	-6.6

Fig. 18. Comparison of number of lines of code.

last column gives the names of the BAJA programs that use it. The codes for all these annotations and testcases can be found on GitHub [2].

Comparison of number of lines of code (LOC): Fig. 18 shows a comparison between the LOC of the test programs written in BAJA (including the code for the annotation classes) and hand-optimized Java programs. It can be noted that programs written in BAJA are typically smaller compared to their Java counterparts (around 5 to 33%). In case of FIB, the hand optimized code has a simple inlined code to check that the argument is a positive number. In contrast, BAJA version has additional code, to declare the annotation class and the barrier method, which results in minor increase in the code size.

Overhead due to BAJA:

To show the effectiveness of translation and the practicality of our proposed BAJA language, in Fig. 19, we compare the execution times of hand optimized versions of the test programs against our compiler optimized ones. When the overhead was less than 0.1%, we capped it to 0.1%. It shows

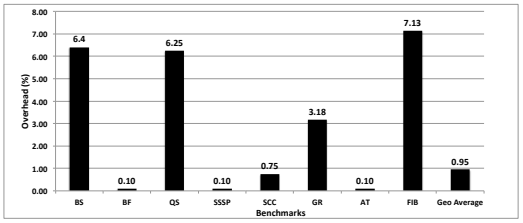

Fig. 19. Hand-Optimized code (h) Vs Generated code (c). Overhead = $100 * (c - h)/h$

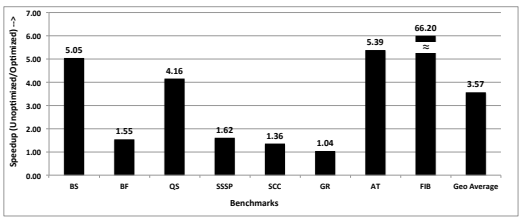

Fig. 20. Comparison of Optimized Vs Unoptimized code.

that compared to the hand optimized codes, the compiler optimized codes incur nearly no overhead ($< 1\%$) for four of the eight test programs (BF, SSSP, SCC and AT), negligible overhead ($\approx 3\%$) for GR, and marginal overhead ($\approx$6-7%) for the rest. Overall the overhead varied between 0.1 to 7.13%; geomean=0.95%.

Among the programs that incur the marginal overhead, BS and QS use annotations on array data type, and FIB uses annotations on primitive data. Both of which involves accesses via the additional wrapper (`BBox`) structures. Further, in case of BS, the array object is attached three different annotations, making it quite challenging to reduce the overhead further. For the other test programs, where the annotations were on objects of user defined classes, we see that the overhead is actually negligible.

Considering the advantages of `BAJA` with respect to enforcing behavioral annotations (in terms of programmability), we believe that the minor overhead is acceptable. We have also measured the `BAJA` compilation/optimization time and did not notice any significant overheads.

Impact of the proposed optimizations: Fig. 20 shows a comparison of the execution times of the test programs generated by the unoptimized compiler to that of the optimized one. And it shows that on average (geometric) our optimizations result in 3.57 × savings. This shows the significant impact of our proposed optimizations have in reducing the overhead. We have observed that optimizations related to objects with unique annotations (Section 4.2.2) and removing calls to dummy barriers (Section 4.2.4) for AT; inlining accesses to the contained objects of `BBox` (Section 4.2.10) for BS; inlining accesses to the contained objects of `BBox` (Section 4.2.10), and objects with unique annotations (Section 4.2.2) for QS; removing calls to dummy barriers (Section 4.2.4), `BBox` removal (Section 4.2.7), and singleton annotation (Section 4.2.9) for FIB have been most effective.

6 DISCUSSION

We now discuss some of the pertinent design issues of `BAJA`.

Annotation class must be written in Java. If we allow an annotation class to be written in `BAJA`, then the program may never terminate. Consider the code shown in Fig. 21. Say an object has an attached annotation of type `A1`. The `fRBar` method of `A1` is invoked upon the dereferencing of any field of the object. At line number 4, `fRBar` will be invoked again by the object pointed to by `c1`, which leads to an infinite recursion. To avoid such issues, we require that the annotations must be written in Java and not `BAJA`.

Inheritance and annotations. If an annotation class `A1` extends `A0` then it inherits all the barriers defined in `A0` and is at liberty to add further barriers and override the existing ones (i.e., specify new behavior). A natural question to ask is "Does an object attached to an annotation of type `A1` additionally inherit the behavior specified in the overridden barriers of `A0`"? In our current design, we answer it in negative and expect the annotation writer to explicitly call the barrier method of the parent annotation class, if so desired.

```
1 class A1 extends AObj<Object>{
2 public void fRBar(Object obj, String fld){
3  C c1 = (C)obj;
4  int x = c1.f1; /*f1: field of C */ ...}}
```

Fig. 21. Annotations written in BAJA may not terminate.

Using `java.lang.reflect.Proxy` to design BAJA. An One could have imagined implemented the idea of annotations associated with objects using the powerful concept of reflection and proxies (provided by the `java.lang.reflect.Proxy` class). However, we did not take that path considering the well known performance related issue with reflection and proxies.

7 RELATED WORK

Dependent types: There has been great interest in bringing dependent types close to practice [8, 22, 23, 25, 32, 40]. The dependent types used in these works mainly focus on the constraints on the value written to an object. Similarly, our proposed write-barriers have similarities to subset types [21]. Stump And Wehrman [31] propose an extension to Java called *property types*, which have syntactic similarities to our annotations – the annotations are defined separately and can take immutable arguments. However, *property types* as well as *dependent information types* [18], are also restricted to only "write" barriers. Our behavioral annotations go beyond that and show how to extend the annotations with other barriers (for field read, method calls and print operations, etc).

Type classes [24, 35] allow dynamic modification of the software to introduce new functionality. Wehr and Thiemann [36] extend Java that among other things allows dynamic loading of interfaces and retroactive implementations of interfaces. Such a scheme brings in altogether new power to the traditional Java programs, where the behavior of all the classes (not that of individual objects, unlike BAJA) that extend this interface can change. In contrast, BAJA admits object specific annotations for multiple types of key operations.

Meta-programming is a powerful concept that allows the programmer to write program specifications that can be used to generate the actual program. Meta-programming can be used to modify the AST of a given program (for example, Stratego/XT [34], Groovy [1], BSJ [26]). Chlipala [5] presents a scheme to ensure that the code generated from a statically type checked meta-program is guaranteed to be type safe. Huang et al [13] use meta-programming to generate powerful aspect specifications that can be weaved into base classes. Compared to these works, behavioral annotations can be used to modify the behavior of (or enforce constraints on) individual objects, instead of the whole underlying class. Another similar natured idea is that of runtime meta-programming in Java by Westbrook et al [37] that generates code based on runtime parameters. Further, meta-programming can be used in an orthogonal way, alongside behavioral annotations.

Java annotations: There has been prior work on extending Java to provide specific popular annotations, such as reference immutability [30], possible nullability [9], non-null guarantee [14] (works on Java bytecode), and so on. They prove that annotated objects in a program adhere to the pre-described semantics. The checker framework [29] uses a specialized mechanism to specify new annotations which can be enforced at compile time. Compared to these schemes each of which require specialized APs, BAJA allows the programmer to specify the annotations along with its desired behavior, without needing any specialized APs. Further, the programmer in BAJA can use the full power of Java to provide the specification. One (unavoidable) disadvantage is that, we don't prove the correctness of these programmer specified barrier methods.

Contracts: Contracts are effective mechanism for debugging, testing and quality assurance. The languages like Eiffel [20], scala [23] and some variants of Java [17] support contracts. Our `mPostBar`

barriers can be thought as postcondition; `mPreBar` and `fRBar` as preconditions; and `oWBar` as an invariant. But unlike these languages where contracts are specified per class, `BAJA` allows barriers to be specified per object to gain fine grain control.

Flow analysis: The annotation flow analysis presented in this paper can be seen as a specialization of traditional flow analysis [28]. Our rules are specialized to take into consideration the semantics of behavioral annotations.

8 CONCLUSION AND FUTURE WORK

This paper presents `BAJA`, an extension of Java language with behavioral annotations, to control the expected behavior of objects during different key operations (such as dereferencing, writing, printing, and so on), without the need of writing any specialized annotation processing tool. We present a core calculus for Java with behavioral annotations. We design a cross compiler to translate `BAJA` programs to Java. We present a series of optimizations, based on a novel flow insensitive, context insensitive, field sensitive annotation flow analysis, to generate Java code that could execute without much overhead.

The current `BAJA` translator assumes that the whole program executes in a single thread. Presence of multiple threads in the program brings in new challenges to the `BAJA` translator (for instance, the impact of multiple threads executing the same barrier code may lead to unexpected behavior, unless these accesses are appropriately synchronized. We leave it as a future work to study the impact of multiple parallel threads in the program, on the `BAJA` translator (and the efficient resolution thereof). Similarly, extending BAJA to handle more advanced features like reflection and generics are left as important future works.

REFERENCES

[1] May, 2010. *Codehaus Foundation. Groovy - building AST guide*. http://groovy.codehaus.org/Building+AST+Guide

[2] Double Blind. 2020. BAJA testcases and Annotations. https://github.com/AnonyMous0777/BAJA.

[3] C. Chen and H. Xi. 2005. Combining programming with theorem proving. In *ICFP*. New York, NY, USA, 66–77.

[4] S. Chiba, A. Igarashi, and S. Zakirov. 2010. Mostly Modular Compilation of Crosscutting Concerns by Contextual Predicate Dispatch. In *OOPSLA*. 539–554.

[5] A. Chlipala. 2010. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In *PLDI*. 122–123.

[6] C. Clifton and G. T. Leavens. 2003. Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. TR03-15 (2003).

[7] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[8] M.D. Ernst. 2007. JSR - 308, Annotations on Java Types. (2007). http://types.cs.washington.edu/jsr308

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec 2007), 35–45.

[10] B. D. Fraine, M. Südholt, and V. Jonckers. 2008. StrongAspectJ: flexible and safe pointcut/advice bindings. In *AOSD*. 60–71.

[11] N. Glew and J. Palsberg. 2002. Type-Safe Method Inlining. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP âĂŹ02)*. Springer-Verlag, Berlin, Heidelberg, 525âĂŞ544.

[12] E. Horowitz, S. Sahni, and D. Mehta. 2006. *Fundamentals of Data Structures in C++*. Silicon Press, Summit, NJ, USA.

[13] S. S. Huang, D. Zook, and Y. Smaragdakis. 2008. Domain-specific languages and program generation with meta-AspectJ. *ACM Trans. Softw. Eng. Methodol.* 18, 2 (Nov. 2008), 6:1–6:32.

[14] L. Hubert. 2008. A non-null annotation inferencer for Java bytecode. In *PASTE*. 36–42.

[15] A. Igarashi, B. C. Pierce, and P. Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS* 23, 3 (2001), 396–450.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V Lopes, J.-M Loingtier, and J. Irwin. 1997. Aspect-Oriented Programming. In *ECOOP*. 220–242.

[17] N.M. Lê. 2012. *Contracts for Java: A Practical Framework for Contract Programming*. Technical Report. Google Switzerland GmbH.

[18] L. Lourenço and L. Caires. 2015. Dependent Information Flow Types. In *Proceedings of the 42Nd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15). ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2676726.2676994

[19] R. Miles. 2004. *AspectJ Cookbook*. O'Reilly Media.

[20] B. Myer. 1991. *Eiffel: The language*. Prentice Hall.

[21] B. Nordstrom and K. Petersson. 1983. Types and specifications. In *IFIP*. 915–920.

[22] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. 2008. Constrained Types for Object-oriented Languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 457–474. https://doi.org/10.1145/1449764.1449800

[23] M. Odersky, L. Spoon, and B. Venners. 2011. *The Scala Language Specification*. Artima Inc.

[24] B.C.d.S. Oliveira, A. Moors, and M. Odersky. 2010. Type Classes as Objects and Implicits. In *OOPSLA*. 341–360.

[25] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*. 437–450.

[26] Z. Palmer and S. F. Smith. 2011. Backstage Java - Making a Difference in Metaprogramming. In *OOPSLA*. 939–958.

[27] J. Palsberg. 2001. Java Tree Builder (JTB): A syntax tree builder. http://compilers.cs.ucla.edu/jtb/.

[28] J. Palsberg and M.I.Schwartzbach. 1991. Object-Oriented Type Inference. In *OOPSLA*. 146–161.

[29] M.M. Papi, M. Ali, T.L. Correa Jr., J.H. Perkins, and M.D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA*. 201–212.

[30] J. Quinonez, M. S. Tschantz, and M. D. Ernst. 2008. Inference of Reference Immutability. In *ECOOP*. Springer-Verlag, 616–641.

[31] A. Stump and I. Wehrman. 2006. Property Types: Semantic Programming for Java. In *FOOL*.

[32] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

[33] T.Sheard. 2004. Languages of the Future. In *OOPSLA*. 116–119.

[34] E. Visser. 2003. Program Transformation with Stratego/XT. In *Domain-Specific Program Generation*. 216–238.

[35] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL âĂŹ89)*. Association for Computing Machinery, New York, NY, USA, 60âĂŞ76. https://doi.org/10.1145/75277.75283

[36] S. Wehr and P. Thiemann. 2009. JavaGI in the Battlefield: Practical Experience with Generalized Interfaces. In *GPCE*. 65–74.

[37] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. 2010. Mint: Java multi-stage programming using weak separability. In *PLDI*. ACM, NY, USA, 400–411.

[38] E. Westbrook, A. Stump, and I. Wehrman. 2005. A language-based approach to functionally correct imperative programming. In *ICFP*. 268–279.

[39] A.K. Wright and M. Felleisen. 1994. A syntactic approach to type soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[40] H. Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (Mar 2007), 215–286.