Closure Joinpoints: Block Joinpoints without Surprises

Eric Bodden
Software Technology Group, Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt (CASED)
bodden@acm.org

ABSTRACT

Block joinpoints allow programmers to explicitly mark regions of base code as "to be advised", thus avoiding the need to extract the block into a method just for the sake of creating a joinpoint. Block joinpoints appear simple to define and implement. After all, regular block statements in Java-like languages are constructs well-known to the programmer and have simple control-flow and data-flow semantics.

Our major insight is, however, that by exposing a block of code as a joinpoint, the code is no longer only called in its declaring static context but also from within aspect code. The block effectively becomes a closure, i.e., an anonymous function that may capture values from the enclosing lexical scope. We discuss research on closures that reveals several important design questions that any semantic definition of closures or block joinpoints must answer. In this paper we show that all existing proposals for block joinpoints answer these questions insufficiently, and hence exhibit a semantics either undefined or likely surprising to Java programmers.

As a solution, we propose a syntax, semantics, and implementation of Closure Joinpoints, block joinpoints based on closures. As we show, our design decisions yield a semantics that follows the principle of least surprise.

Categories and Subject Descriptors

D.3.3 [**D.3.3 Language Constructs and Features**]: Procedures, functions, and subroutines; F.3.3 [**Studies of Program Constructs**]: Control Primitives

General Terms

Design, Languages, Theory

Keywords

Language design, Joinpoints, Pointcuts, Lambda expressions, Closures, Static and dynamic scoping

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. INTRODUCTION

In this work, we introduce Closure Joinpoints for AspectJ [6], a mechanism that allows programmers to explicitly mark any Java expression or sequence of statement as "to be advised". Closure Joinpoints are explicit joinpoints that resemble labeled, instantly called closures, i.e., anonymous inner functions with access to their declaring lexical scope.

As an example, consider the code in Figure 1, adopted from Hoffman's work on explicit joinpoints [24]. The programmer marked the statements of lines 4–8, with a closure to be "exhibited" as a joinpoint `Transaction` so that these statements can be advised through aspects. Closure Joinpoints are no first-class objects, i.e., they cannot be assigned to function variables. Instead, Closure Joinpoints are always instantly called: in line 8, the programmer calls the closure with the argument `Level.L1`. This will cause the closure to execute with the formal parameter `l` (line 4) bound to the value of `Level.L1`. Aspects advising joinpoints of type `Transaction` will intercept this execution.

Explicit joinpoints such as the Closure Joinpoints that we propose can be useful whenever pointcuts are either not expressive enough, too awkward, or too concrete to conveniently describe exactly which part of the execution should be advised. For instance, Steimann et al. [37] recently showed that out of 484 pieces of advice in an aspect-oriented version of BerkelyDB [9], 218 applied to some statements in the middle of a method. Single statements may be hard to select without explicit joinpoints, yielding bloated and fragile pointcuts. Sequences of statements need to be extracted into a method so that they can be advised in standard AspectJ. Closure Joinpoints make this extraction superfluous because they instead allow programmers to define an anonymous inner method right in place.

```
1  class Agent {
2  final CardProcessor cc = makeCardProcessor();
3  void createTrip(final Person p, final Flight f, final Hotel h) {
4      exhibit Transaction(int l) {
5          f.reserveSeat(p);
6          h.reserveRoom(p);
7          cc.debit(p.getCC(),f.total(),h.total());
8      } (Level.L1);
9      suggestRentalCars(p,f);
10 }
11 ...
12 }
```

Figure 1: Transaction aspect with closure joinpoint

Explicit joinpoints have another advantage over the implicit joinpoints that pointcuts can intercept. Implicit joinpoints only expose a set of pre-defined values from the execution context, via the pointcuts `this`, `target` and `args` or as special parameters to `after-returning` or `-throwing` advice. In some cases, these arguments may be insufficient for the aspect to function. For instance, in Figure 1 the value `Level.L1` depends on the particular execution context: Closure Joinpoints of type `Transaction` that are declared in different locations may pass a different actual parameter. Closure Joinpoints can easily supply such context information through parameters.

Another use case for explicit joinpoints often raised by practitioners is the ability to use aspects to conditionally disable regions of base code. For example, Bruntink et al. [12] have shown that tracing is not always implementable as an aspect, as this concern can show high variability. But programmers have tried to at least use aspects to disable logging conditionally [30]. Without explicit joinpoints it is impossible to also bypass the code that computed that call's parameters, leading to unnecessarily slow code. With explicit joinpoints, on the other hand, one can bypass calls to the logging interface with a non-proceeding around advice.

The general idea of explicitly marking code as "to be advised" is not novel. Other researchers have previously proposed language extensions to AspectJ already that would implement some kind of "explicit joinpoints" or "block joinpoints" [4, 24, 37]. However, as we argue in this paper, most existing approaches to such block-based joinpoints come with an unclear or rather surprising semantics. For instance, consider the code example in Figure 2. The code is adapted from Steimann et al. [37], who proposed one approach to explicit joinpoints. The example uses the authors' syntax, where joinpoints are explicitly declared first-class objects that can expose context through fields. The class `ShoppingSession` defines a `buy` method that adds a certain `amount` of products of type `item` to a shopping cart and keeps track of the total number of items bought. We assume that later on, the company decided to introduce a "buy two, get one free" policy for books. The aspect `BonusProgram` implements this policy: if the item's category is `BOOK`, then the aspect automatically increments the number of bought items by 50% before proceeding with the execution of the original joinpoint. In lines 8–11, the base code explicitly "exhibits" the `Buying` joinpoint that the aspect advises, as a block joinpoint, further exposing the value `category`.

Clearly, the explicit joinpoint eases aspect-oriented programming in this example. Nevertheless, we argue that the particular language design yields a rather surprising semantics. By assigning a field a new value, as in line 19, the original joinpoint in lines 8–11 will execute with that assigned value. While this is intended in the example (the `add` method should add the increased amount to the shopping cart and also the variable `totalValue` should receive the increased value), we argue that this semantics is rather surprising on the base-code side: within the joinpoint, the variable `amount` has a value different from the one that it had when entering the joinpoint, although there is no visible assignment to `amount`. The explicit joinpoint therefore introduces a kind of call-by-reference semantics, which is rather surprising in a call-by-value language like Java. No less surprising, the value of `amount` is even reset to its original value once the control flow leaves the explicit joinpoint again. This second assignment is also hidden in the chosen syntax. Another potential problem is unstructured control flow: what if the block joinpoint contained a `break` or `continue` statement with a target outside the block? Exposing the block as a joinpoint means that the block may execute in a different context in which these targets are not even on the execution stack any more. Clearly, a reasonable semantics for block joinpoints should consider these constraints.

```
1  class ShoppingSession {
2      int totalAmount = 0;
3      ShoppingCart sc = new ShoppingCart();
4
5      void buy(Item item, int amount) {
6          Category category = Database.categoryOf(item);
7          //assume category==Item.BOOK and amount==2
8          exhibit new Buying(category, amount) {
9              sc.add(item, amount); //then here amount==3...
10             totalAmount += amount;
11         }; //... and here amount==2 again
12     }
13 }
14
15 aspect BonusProgram {
16     joinpointtype Buying{ Category category; int amount; }
17     void around(Buying jp) {
18         if(jp.category == Item.BOOK)
19             jp.amount += jp.amount / 2;
20         proceed(jp);
21     }
22 }
```

Figure 2: Shopping session with a bonus aspect [37]

The key insight of this paper is that a block of code, when being exhibited as a joinpoint, effectively becomes a closure. As we show, by providing a language design for closures that is tailored towards the particular programming language at hand and yields a semantics that follows the principle of least surprise, one can automatically obtain a language semantics for Closure Joinpoints that has the same favorable property. To obtain such a design, we discuss several current proposals for adding closures to the Java language. Based on this discussion, we propose a design for Closure Joinpoints that yields a syntax and semantics easy to understand for Java programmers. We further expose an implementation of Closure Joinpoints on top of the AspectBench Compiler [7].

To summarize, this paper presents the following original contributions:

- the idea to consider block joinpoints as special cases of method-execution joinpoints over closures,
- a careful analysis of the semantic tradeoffs that surround control-flow and data-flow of closures,
- a detailed assessment of the design decisions taken in related work on block joinpoints and on closures,
- a syntax, semantics and an implementation for Closure Joinpoints, close to the Java programming language.

We structure the remainder of this paper as follows. In Section 2, we explain why a language design for explicit joinpoints should follow the same arguments about design decisions as for closures. In Section 3 we give background information on lambda expressions and closures and summarize different proposals for including closures in the Java

language, plus the tradeoffs involved. We propose our solution, Closure Joinpoints, in Section 4, discuss further related work in Section 5 and conclude in Section 6.

2. WHY EXPLICIT JOINPOINTS SHOULD BE CLOSURES

All related work defines explicit joinpoints as explicitly named block statements that allow programmers to mark the block as "to be advised". At first, such "block joinpoints" may appear simple to define and implement. After all, regular block statements in Java-like languages are constructs well-known to the programmer and have simple control-flow and data-flow semantics. In Java, wrapping a sequence of statements s into a block statement, yielding `{`s`}`, has only scoping implications: if s contains local-variable declarations then the scope of these declarations will now be constrained to the block. Introducing a block, however, does not in any way alter the execution semantics of s: the control flow and data flow into, within, and out of s remains unchanged. This includes unstructured control-flow through `break`, `continue` or `return` statements or exceptions, or assignments within s to variables declared outside of s. This simple semantics makes blocks so convenient to use that many programmers even use blocks simply to enhance readability, for instance in if-then-else constructs.

2.1 Blocks are confined to their lexical scope

However, one key insight of this paper is that the execution semantics of blocks is only so simple only because the execution of a block is, by definition, confined to the block's own lexical scope. The block executes when the program counter reaches the beginning of the block (and only then) and at this point in time the entire declaring execution context is known and accessible. But now consider the situation in which the programmer decides to "exhibit" the block as a joinpoint. This novel change *does* have wide-ranging semantic implications: the block of code can now execute in different contexts. It can still execute in its original declaring context, but in addition, the block can be called through the `proceed`-call of an around advice. In this situation, it is not clear how the block joinpoint would obtain access to values from its declaring scope. In an extreme case, the advice could `proceed`-call the joinpoint in a separate thread. For instance, Figure 3 shows an aspect that schedules advised joinpoints for execution by the AWT Event Thread. (This way, an aspect can implement thread-safe updates to Swing-based user interfaces [27].) In this case, either thread has no access to the other thread's execution stack and the local variables declared there. (As we will discuss later, some approaches avoid this problem by implicitly allocating local variables like on the heap. Accesses to these variables can then lead to subtle data races, though.)

```
1 void around() : methodsUsingSwing() {
2     Runnable worker = new Runnable() {
3         public void run() {
4             proceed(); //executes in another thread
5         }
6     };
7     java.awt.EventQueue.invokeLater(worker);
8 }
```

Figure 3: Synchronization aspect for Swing/AWT

But not only may data flow cause problems at the joinpoint boundary, the same holds for control flow, too. Assume, for instance, that the advised block contains a `return` statement. When executed in the context of the declaring scope, this statement would cause the program to return from the declaring method. If executed in the context of an aspect, however, the statement would cause the program to return from the joinpoint's block instead. While compiler tricks could be used [4] to return to "further up" the call chain in a single-threaded setting, this approach would fail in the situation mentioned above, where an advice "proceeds" in another thread. All these problems originate from the fact that blocks, unlike methods, are no modular unit of execution: their semantics is fundamentally tied to the scope in which they are declared.

One may argue that complications we described originate from the fact that we are discussing `around`-advice. After all, a `before` or `after`-advice cannot issue a `proceed`-call, and would therefore prevent the block joinpoint from executing anywhere but at the position at which is was declared. However, block joinpoints are really mostly[1] useful in combination with an `around`-advice in the first place. If only `before` or `after`-advice were to be used then one could use an "atomic" explicit joinpoint instead, i.e., a joinpoint that takes no block as argument and therefore resembles a point and not region in time [31]. In Section 5, we will discuss related work [24] that defines such more lightweight explicit joinpoints without blocks, avoiding many complications.

It is also worthwhile noting that the constraints that we described do not only apply to explicit block joinpoints but to any joinpoint that resembles a sequence of statements which can include an assignment, or a `break`, `continue` or `return` statement. The only exception are joinpoints that resemble the execution of an entire method. (Because, as mentioned above, a method is a modular unit of execution.) Most of AspectJ's built-in "kinded" pointcuts only match single statements that may be none of the above, or match an entire method body. There are some exceptions, however: handler pointcuts, for instance, match a `catch` block. Such catch blocks can, in principle, contain unstructured control-flow statements and also assignments to outside the `try/catch` block. And indeed, for this reason, AspectJ compilers do not support around advice for handler joinpoints. In Section 5 we will discuss other proposals for pointcuts that match sequences of statements, and show that they pose similar challenges.

2.2 From blocks over methods to closures

From the above findings we conclude that the syntax and semantics of traditional Java block statements are not the best starting point for defining block-like joinpoints for an aspect-oriented programming language. In this paper we hence propose to instead model explicit joinpoints as synchronous calls to closures, i.e., to anonymous methods with access to the enclosing lexical scope. Methods are modular units of execution that can execute in different contexts.

[1]Another possible use would be in combination with an `after`-advice. This combination would cause the advice to execute whenever control leaves the block joinpoint, no matter if the block is left after having executed its last statement, through an explicit `return` or by throwing an exception.

Their semantics in terms of control flow and data flow is well understood: input data is passed through parameters (incl. "`this`"), output data as a return value and unstructured control flow through `break`, `continue` and `return` statements is always confined to the execution of the method itself.

Because of the lack of block joinpoints in plain AspectJ, programmers frequently use the following workaround. They extract the block to be advised into a method m and then advise the execution of m. Considering our insights from above, this appears to be not a bad solution at all. However, also this approach yields several drawbacks:

1. The extracted method is callable from different contexts, not only from advising pieces of advice. This may be undesired from a maintenance perspective.
2. The extracted method has no implicit access to local variables defined in the lexical scope that enclosed the original block before extraction. All such variables must be explicitly assigned through parameters. This may be awkward in practice.

In this paper, we therefore propose Closure Joinpoints, an approach to block joinpoints based on lambda expressions and closures. In the context of Java, lambda expressions resemble anonymous methods. Because the methods are anonymous, they can only be referenced through a first-class reference. We will define lambda expressions in such a way that no such references can be created; instead the expressions are always called at the site of their definition, and only there. This solves Problem 1. Our expressions are also closures: they can implicitly reference values from their enclosing lexical scope, which solves Problem 2.

In the following Section we discuss lambda expressions and closures in detail and also discuss multiple approaches to introducing these language features into Java. In Section 4 we will then return to the problem of explicit joinpoints and explain our design of Closure Joinpoints.

3. DESIGN TRADEOFFS FOR CLOSURES IN JAVA

In the following, we first give background information on lambda expressions and closures and then discuss several language designs that researchers and practitioners have proposed for introducing lambda expressions and closures into the Java programming language. Some of these proposals allow closures to be very powerful programming constructs, however they come at the significant cost of a complex semantics that can sometimes be surprising, especially to Java programmers. Nevertheless, a thorough discussion of the tradeoffs involved builds a strong motivation for our design of Closure Joinpoints.

3.1 Lambda expressions vs. Closures

Lambda expressions originate from Church's 1932 work on the lambda calculus [16]. Lambda expressions define anonymous functions where variables prefixed with a λ are bound in the subsequent expression. Such variables are effectively parameters to the anonymous function. For instance, the expression "$\lambda x\ .\ x \cdot x$" is a lambda expression denoting the square function. This expression is said to be "closed", as all its variables are bound through lambdas. Conversely, "open" lambda expressions have free variables. For instance, the expression "$\lambda x\ .\ x \cdot y$" is the function that will multiply its parameter x with the free variable y. Note that one cannot reduce such an open lambda expression to a ground value. To allow reduction, one needs to close the open lambda expression, i.e., to bind its free variables. One way to close open lambda expressions is, as the name suggests, a closure.

Closures [28] combine a function with free variables with an environment that assigns values to these variables. In the following, we will say that the closure "captures" the variable in the environment. The choice of environment can vary but the most widespread [32,38] use of the term closure refers to the case where free variables are bound by the lexical scope in which the closure was declared.[2]

According to Landin [28], only *open* lambda expressions, paired with an environment, are actually closures, closed lambda expressions are just simple anonymous functions. Conversely, closures do not necessarily have to be anonymous. For example, one can regard Java's inner classes as an awkward notation for closures; they can capture fields and (`final`) local variables defined in their enclosing lexical scope. Figure 4 defines an anonymous inner class capturing the local variable `result`.

3.2 The design space for closures in Java

As Figure 4 shows, inner classes (even anonymous ones) can be quite heavyweight: instead of being able to just define a lambda-expression, one has to define an anonymous class, which itself has to adhere to some pre-defined interface (here `Runnable`) and therefore must define a named method (here `run`), as specified by that interface. If that interface is unsuitable for the usage context at hand, further complications arise. In our example, `Runnable` defines a `run` method with return type `void`, preventing the call to `syncExec` from passing values back to the calling context. Further, inner classes in Java may only access those local variables from the lexical scope that are declared as `final`. This, paired with Java's pass-by-value semantics, prevents programmers from passing back the return value through a simple variable assignment. The only way for the programmer to "return" a value in this situation is therefore to construct a box object, or an array as in the example.[3]

To avoid such awkward syntax in the future, several parties have issued proposals to add a syntax for closures to

[2]Other possible semantics include instead using the scope in which the closure is invoked. This yields dynamic-scoping semantics [19].

[3]As we will see later, though, there is a good reason for the restriction to `final` variables. Local variables are allocated on the stack, which precludes access to values of non-`final` local variables from other threads. The user-define array is allocated on the heap, which both the environment declaring the inner class and the one executing it can access.

```
final boolean[] result = new boolean[] { false };
display.syncExec(new Runnable() {
    public void run() {
        result [0] =
          MessageDialog.openQuestion(shell, title, message);
    }
});
return result [0];
```

Figure 4: Use of anonymous inner class within class `ProgressTask` of the Android SDK [1]

version 8 of the Java language. All of these proposals appear to restrict themselves to introducing syntactic enhancements that can be compiled to Java bytecode that predates Java 8 and therefore require no changes to the virtual-machine specification. Nevertheless, full closures provide features that can have wide-ranging semantic implications. Next, we will discuss some of these features and their implications here. In the Java context, most proposals talk of closures although they actually define lambda expressions that may be closures when capturing variables from their enclosing lexical scope. To ease our presentation, in the following we will mostly use the term closure, too.

We discuss three proposals to closures in Java, widely known by the names of BGGA, FCM and CICE:

BGGA is named after the proposal authors Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé [11]. Until Version 0.5 of their proposal, BGGA presented full closures, supporting full access to variables from the lexical scope and non-local transfer of control (see below). Recently, this proposal was split into two, separating a restricted form of closures targeting end-users (Version 0.6a) from a form of full closures targeting language experts (0.6b).

FCM stands for First Class Methods, an approach proposed by Stephen Colebourne and Stefan Schulz [18]. FCM extends the idea of BGGA by introducing function types and pointers as a general concept to Java. Both can be used for named functions as well as lambda expressions. Unlike in BGGA, the closures that Colebourne and Schulz propose restrict variable capture and do not allow for non-local transfer of control.

CICE is an acronym for Concise Instance Creation Expression and is put forward by Bob Lee, Doug Lea, and Joshua Bloch [29]. CICE provides syntactic sugar for creating anonymous inner classes based on interface types that only define a single method. This makes the approach restricted but very lightweight.

Kreft and Langer provide a very helpful comparison [26].

3.2.1 Non-local transfer of control

BGGA first introduced the idea of closures mostly to facilitate an abstraction of commonly used control flow. As an example, consider Figure 5, which defines a `foreach` loop using BGGA closures. First, lines 1–5 define the method `foreach`, which takes an `Iterable` object and a block of code as argument. The block has the declared type {T => **void**}. The body of `foreach` then invokes the body for every argument in the `Iterable`. The `main` method assigns a lambda expression to the function variable `print`, and then passes this variable to the `foreach` method. The program from Figure 5 will therefore print the values 1 to 5.

```
1  public static <T> void
2    foreach( Iterable<T> seq, {T => void } fct) {
3      for (T elm : seq)
4          fct.invoke(elm);
5  }
6
7  public static void main(String[] args) {
8      {Integer => void } print =
9        { Integer arg ==>
10         //if (arg == 3) return;
11         System.out.println(arg);
12       };
13     foreach(new int[]{1,2,3,4,5}, print);
14 }
```

Figure 5: Implementing a `foreach` loop through BGGA closures, after [26]

Next, consider the case, though, in which line 10 in this example is not commented out. The crucial question is, which scope the execution of the `return` statement will return from; will it return from the body of the closure or from the enclosing lexical scope? According to the semantics of "full" BGGA closures, `break`, `continue` and `return` statements are bound to their lexical scope: the modified program would print the values 1 and 2 and then exit, because the `return` statement at line 10 will return from `main`, not the closure. While this semantics may appear surprising to many Java programmers, BGGA argue that this semantics aids the refactoring of existing code into closures [11, v0.5]:

> "One purpose for closure literals is to allow a programmer to refactor common code into a shared utility, with the difference between the use sites being abstracted into a closure literal by the client. The code to be abstracted sometimes contains a break, continue, or return statement. This need not be an obstacle to the transformation. One implication of the specification is that a break or continue statement appearing within a closure literal's body may transfer to a matching enclosing statement. A return statement always returns from the nearest enclosing method or constructor. A function may outlive the target of control transfers appearing within it."

For control abstraction, non-local transfer of control can sometimes be very useful. For instance consider the code from Figure 6. Readers familiar with aspect-oriented software development will easily identify that the use of explicit locks is scattered through the implementation of the stack class, and tangled with the code that implements the stack's

```
1  class Stack {
2      ...
3      void push(int elm) {
4          lock.lock();
5          try {
6              arr[cnt++] = elm;
7          } finally {
8              lock.unlock();
9          }
10     }
11
12     int pop() {
13         lock.lock();
14         try {
15             return arr[--cnt];
16         } finally {
17             lock.unlock();
18         }
19     }
20 }
```

Figure 6: Use of explicit locks in a Java program

```
1  <T> T withLock(Lock lock, { => T} block) {
2      lock.lock();
3      try {
4          return block.invoke();
5      } finally {
6          lock.unlock();
7      }
8  }
9
10 void push(int elm) {
11     withLock(lock) {
12         arr[cnt++] = elm;
13     }
14 }
15
16 int pop() {
17     withLock(lock) {
18         return arr[--cnt]; //return from pop!
19     }
20 }
```

Figure 7: Encapsulating locking with BGGA closures, after [26]

actual functionality. While control abstraction cannot avoid scattering or tangling, it can reduce the *amount* of scattered code. Consider the code in Figure 7. This code uses BGGA closures to encapsulate the locking-related code within the method `withLock`. The methods `push` and `pop` can call this method with a block of code as argument, yielding a rather declarative syntax. Importantly, note that the `return` statement in line 18 causes the code to not only return from the closure but also from the `pop` method; the `return` binds to the enclosing lexical scope, not the closure.

Although non-local transfer of control can be quite powerful, it also introduces several pitfalls. If no proper care is taken, control flow may become hard to reason about. Especially, multi-threaded execution of closures raises questions. Similar to the aspect from Figure 3, imagine a function `asyncExec` that passes a body of code for execution by another thread. When invoking `asyncExec` with a closure c declared within a method m, then it may happen that m returns long before c. But which version of m should an explicit return from c then jump to? BGGA call this the case of an "unmatched" non-local transfer of control. For lack of a meaningful semantics of such a situation, the authors propose to throw an exception in case the situation arises.

Note that the same problem can arise even in a single-threaded setting: if a closure can be stored in a field or returned from its declaring scope then it may be invoked after the program returned from this scope. The problem is known as the "upward FUNARG problem" [32].

To maintain simplicity, the FCM and CICE approaches do not allow for non-local transfer of control; here `break` and `continue` statements are only allowed within a closure if their control-flow target is part of the closure as well. Likewise, `return` statements return from the closure, not the enclosing lexical scope. Despite compelling examples such as the locking examples above, the complexity of non-local transfer of control has provoked much criticism from the Java community. Hence, Gafter and von der Ahé have recently split their proposal into two parts: normal closures are now "restricted", which means that they follow the control-transfer semantics of FCM and CICE; programmers can, however, use a specialized syntax defined in the second part to declare unrestricted closure with non-local transfer of control for the purpose of control abstractions.

3.2.2 Variable capture

A problem closely related to non-local transfer of control is the problem of non-local transfer of data. As we explained above, closures "close" open expressions by assigning free variables values from the enclosing lexical scope: the variables are captured in this scope. In many situations, this is unproblematic. For instance, in our inner-class example from Figure 4, even if the Java Language Specification [21] had not required the variable `result` to be `final`, the assignment to `result` would still have had a well-defined semantics, due to the fact that the `run` method executes synchronously right in its lexical scope. In fact, loosening the restriction that the `result` be `final` would have provided for a simpler syntax in this case: instead of having to declare a single-value heap-allocated array, the programmer could simply have used a primitive boolean variable instead. After all, as long as the closure is guaranteed to execute in the control flow of its declaring scope, the captured variables do not necessarily need to be heap-allocated because they are guaranteed to be on the stack at every access, although possibly a few frames further up.

As we can see, unrestricted access to variables from the lexical scope can be quite useful. However, how about situations where closures outlive the execution of their enclosing lexical scope? In general, this situation cannot be avoided, and when the situation arises, what semantics should accesses to this scope follow? Most closure proposals by now adopt a semantics where variables accessed through a closure must either be explicitly declared as `final`, or implicitly get `final`-semantics. (The latter approach is taken in C#.) This means that the closure receives copies of the variables from the lexical scope at the time the closure is captured, not at the time at which it is invoked. Because the variables are `final`, the closure cannot assign them. This trades flexibility for a rather straightforward semantics. Unrestricted BGGA closures use a semantics in which closures can read from and write to captured variables at will. Write accesses generate a warning unless the variable written to is annotated with @Shared. However, version 0.6a of the BGGA proposal now also adopts the more restricted semantics that only `final` variables may be accessed.

We will discuss more approaches to closures in our Related Work section, Section 5. As our discussion showed, the design space for closures strongly aligns with the design space for block joinpoints. It hence follows naturally to consider defining and implementing block joinpoints as closures.

4. CLOSURE JOINPOINTS

As we saw, the existing proposals for lambda expressions and closures in Java mainly differ in the amount kinds of control- and data transfer that they allow. In this section, we present our proposal for Closure Joinpoints. We first discuss the execution semantics of Closure Joinpoints with respect to data flow and control flow using our `ShoppingSession` example. We then describe in full generality our proposed syntax of Closure Joinpoints (which is a matter of taste and hence open for debate), how to type check programs in our language extension, and an implementation based on the AspectBench Compiler [7].

4.1 Closure Joinpoints by example

In our proposal, Closure Joinpoints are expressions of the following syntax:

exhibit ID([FormalParamList]) Block ([ActualParamList])

The block effectively defines a lambda expression, while the formal-parameter list defines its λ-bound variables. Opposed to regular lambda expressions, however, a Closure Joinpoint must be followed by an actual-parameter list: the closure is always immediately called, and cannot be assigned to function variables.

Figure 8 shows the shopping-session example from Figure 2 adopted to the proposed syntax. The aspect in lines 17–24 first declares a type signature for an explicit joinpoint type called `Buying`. Note that, unlike a named pointcut declaration, because a joinpoint declaration defines an expression type, it comprises a return type (`int`). The advice that follows declares to advise joinpoints of this type. Note that, opposed to the approach taken by Steimann et al., which we discussed in the introduction, the syntax for advice bodies remains completely the same as in AspectJ. In particular, our joinpoints do not carry context information as re-assignable fields but rather expose context to pieces of advice by binding advice parameters, just as in regular AspectJ. This makes it immediately clear that the values of `amt` and `cat` are visible in this particular piece of advice only, and that advice parameters are allocated on the stack, not the heap, and are therefore not subject to data races.

The explicit joinpoint in the base code (lines 9–13) was adapted to the syntax of Closure Joinpoints. As in Figure 2, the base code declares to exhibit a joinpoint of type `Buying`. However, the header of the `exhibit`-clause now contains a formal-parameter list. The actual-parameter list from Figure 2 follows the joinpoint body. Figure 9 shows the general syntax as a syntactic extension to AspectJ (shown in gray). Note that Closure Joinpoints can appear as *StmtExpr*, in which case their return value (if any) is discarded.

```
1  import static BonusProgram.Buying;
2  class ShoppingSession {
3      int totalAmount = 0;
4      ShoppingCart sc = new ShoppingCart();
5
6      void buy(final Item item, int amount) {
7          Category category = Database.categoryOf(item);
8          //changes start here
9          totalAmount = exhibit
10             Buying(Category c, int amount) {
11                 sc.add(item, amount);
12                 return totalAmount + amount;
13             } (category,amount);
14     }
15 }
16
17 aspect BonusProgram {
18     joinpoint int Buying(Category cat, int amount);
19     int around Buying(Category cat, int amt) {
20         if(cat==Item.BOOK)
21             amt += amt / 2;
22         return proceed(cat, amt);
23     }
24 }
```

Figure 8: Example from Figure 2 with Closure Joinpoints

Expr ::= ... | *ClosureJoinpoint*.

StmtExpr ::= ... | *ClosureJoinpoint*.

ClosureJoinpoint ::=
  "`exhibit`" *ID* "`(`" [*ParamList*] "`)`" *Block*
    "`(`" [*ArgList*] "`)`" |

  "`exhibit`" *ID Block*.

AspectMember ::= ... | *JoinpointDecl*.

JoinpointDecl ::=
  "`joinpoint`" *Type ID* "`(`" [*ParamList*] "`)`" [*ThrowsList*].

AdviceDecl ::= ... | *CJPAdviceDecl*.

CJPAdviceDecl ::=
  [*Modifiers*] *CJPAdviceSpec* [*ThrowsList*] *Block*.

CJPAdviceSpec ::=
  Type "`before`" *ID* "`(`" [*ParamList*] "`)`" |
  Type "`after`" *ID* "`(`" [*ParamList*] "`)`" |
  Type "`after`" *ID* "`(`" [*ParamList*] "`)`"
    "`returning`" ["`(`" [*Param*] "`)`"] |
  Type "`after`" *ID* "`(`" [*ParamList*] "`)`"
    "`throwing`" ["`(`" [*Param*] "`)`"] |
  Type "`around`" *ID* "`(`" [*ParamList*] "`)`".

Figure 9: Syntax for Closure Joinpoints, as a syntactic extension to AspectJ (shown in gray)

Certainly, our syntax is more verbose than it would have to be. However, due to the verbosity, it is close to the syntax of Java, which is similarly verbose in itself. A different syntax may be advisable in connection with other languages. Note, though, that we do provide "**exhibit** ID Block" as a shorthand for "**exhibit** ID() Block ()". With this shorthand syntax, a closure joinpoint appears just as a block joinpoint, but with a quite restrictive semantics: as we explain below, the joinpoint has only access to fields and `final` local variables and may not have any non-local transfer of control.

4.2 Data-flow semantics

The formal-parameter list of the Closure Joinpoint declares local names for the variables that this joinpoint exposes. Through this syntax, it is immediately clear to Java programmers that variables from this formal-parameter list shadow variable definitions from the enclosing scope. For instance, within the Closure Joinpoint of Figure 8, the variable references to `amount` (lines 11 and 12) bind to the formal parameter of the Closure Joinpoint, not to the second parameter of `buy`. This, in turn, indicates that the values of the variables of the formal-parameters list may change, should the Closure Joinpoint be advised; programmers are used to parameters being assigned different values in different execution contexts. We conclude that the introduction of a formal-parameters provides a syntax and semantics that should appear more natural to Java programmers than the approach taken e.g. by Steimann et al.: the data flow is more obvious and helps to avoid unwanted surprises.

Not all Closure Joinpoints are closed lambda expressions. As the name suggest, and our example shows, they can be closures: in the example, line 12 captures the field `totalAmount`. After our initial discussion, the reader may want to know whether we allow programmers to capture non-final

local variables. Although there are compelling use cases for this feature, supporting it would require implicit heap allocation, and we argue that implicit heap allocation is dangerous. Particularly, it is not uncommon to execute joinpoints by "proceeding" in other threads: both Laddad [27] and Kienzle [25] independently advocate this mechanism for implementing transactions with aspects, and we ourselves used such asynchronous execution of joinpoints for N-Version Programming [10]. Multi-threaded execution easily causes races on unprotected heap-allocated objects. We hence feel that programmers should be required to heap-allocate objects explicitly, if desired, to make them aware of the risk of races. We therefore disallow Closure Joinpoints to write to captured local variables. Nevertheless, as shown in the example, we do allow access to fields of any kind; fields are heap-allocated anyway and Java programmers therefore know that accessing them may require synchronization.

Because we force Closure Joinpoints to be called right at the place of their declaration, we can give them the semantics of proper expressions, allowing them to return a value. In our example, the joinpoint returns the properly updated value of `totalAmount`, which the enclosing scope then assigns to the appropriate variable. In case an `around` advice decides to defer execution of the joinpoint by scheduling it for execution in another thread, the body of the `around` advice will have to return a value nevertheless. Consider Figure 10: because the aspect "proceeds" in a separate thread, it cannot return the return value of this `proceed`-call but must return some made-up return value instead. In our example, this does not really make sense: it is impossible to come up with a sensible return value before the `proceed`-call returns; deferred execution is just no option for this particular joinpoint. But at least, our syntax and semantics make the programmer aware of this fact. Compare this to the approach taken by Steimann et al.: there, block joinpoints are block statements and not expressions, and therefore do not have an explicit return value. Instead, their block joinpoints pass values to the declaring scope through assignments to heap-allocated, captured variables. But in our example, these assignments would be deferred with the `proceed`-call, and therefore yield a subtle data race.

4.3 Control-flow semantics

As we saw in Section 3.2.1, non-local transfer of control can be useful in certain situations, for instance for "hiding" explicit locks behind a `withLock` closure which to the programmer appears just like a `synchronized` block (Figure 7). Aspects, to a certain extent, also fulfil the purpose of encapsulating control structures into a separate module. In fact, both practitioners and researchers have used aspects to address concerns like multi-threaded dispatch or transactions [25, 27]. Therefore, the prospect of allowing explicit joinpoints non-local transfer of control may seem appealing.

```
1 aspect Scheduler {
2     int around BonusProgram.Buying(int amt, Category cat) {
3         new Thread() {
4             public void run() { proceed(amt,cat); }
5         }.start ();
6         return 0; //always returned
7     }
8 }
```

Figure 10: Aspect deferring execution of a Closure Joinpoint

However, already at the beginning of the paper we argued for a semantics that follows the principle of least surprise: many programmers already perceive regular AspectJ as rather complex, and adding closures to Java (or AspectJ for that matter) will increase complexity even more. We therefore argue for a rather simplistic approach, even though this may result in a somewhat more verbose syntax. For Closure Joinpoints, we therefore disallow non-local transfer of control. This means that `break` and `continue` statements are only allowed within a joinpoint's block if they bind to targets within the same block. Further, `return` statements return from the joinpoint, not the enclosing lexical scope.

Exceptions. Exceptions are another possible way to introduce unstructured control-flow into a Java program. None of the closures proposals that we mentioned in Section 3 treats exceptions in a special way. If a closure raises an exception then this exception will propagate up the call stack, i.e., into the calling scope, not the declaring lexical scope. Handling exceptions is therefore up to the environment that calls the closure, not to the one that declares it. The same semantics apply to Closure Joinpoints, too. For checked exceptions, we demand that they be declared in the definition of the joinpoint type (note the optional *ThrowsList* for *JoinpointDecl* in Figure 9). Once declared, we check that all contexts that may call Closure Joinpoints of this type do in fact handle or explicitly propagate exceptions of this type.

4.4 Type checking Closure Joinpoints

Due to the fact that Closure Joinpoints rely on the well-known notion of closures, and closures are special methods, type checking Closure Joinpoints is quite similar to type checking method definitions and calls. However, since we use a joinpoint declaration both for type checking closures and calls to those closures, unlike for method calls we cannot allow covariant return types nor contravariant argument types. This problem is well known for around advice and can be alleviated by resorting to more elaborate typing annotations, such as proposed in StrongAspectJ [20].

For an AspectJ program using Closure Joinpoints to be well-typed, it must obey the following rules:

- If a piece of advice a declares to advise a Closure Joinpoint j then j must be declared and a and j must have the same return type and parameter types.
- A Closure Joinpoint c of declared type j may only return a value if j's return type r is not `void`. c must then return a value v on all paths, with the type of v a sub-type of r. The return type of c is the declared return type r. j must have the same formal-parameter types as c. The actual parameters passed to c must be of sub-types of the respective formal parameters.
- A Closure Joinpoint may access its formal parameters and `final` local variables and fields from the enclosing lexical scope.
- A Closure Joinpoint declared in a static method may only access members from the enclosing scope that are also static. It has no access to "`this`".

- A Closure Joinpoint must not contain `break` or `continue` statements whose targets are not part of the same joinpoint.

- If an advice a advises Closure Joinpoints of type j which declares to throw checked exceptions of type e then a must either catch exceptions of this type or declare to propagate them. Similarly, all call sites of Closure Joinpoints of declared type j must handle or propagate these exceptions.[4]

4.5 Argument binding and reflection

Since pieces of advice reference Closure Joinpoints explicitly and not through a pointcut, there are no `this`, `target` or `args` pointcuts. Similarly, reflective access via `thisJoinPoint` returns `null` for these bindings. If a Closure Joinpoint wishes to expose "`this`" to an aspect, it can do so via an explicit joinpoint parameter. An `after returning` advice can capture the return value of the Closure Joinpoint and an `after throwing` advice can capture exceptions escaping the closure's execution respectively.

4.6 Implementation

We implemented Closure Joinpoints as an extension to the AspectBench Compiler (abc) [7]. First, we type-checking each Closure Joinpoint j for conformity with its joinpoint declaration. Here we also check whether all context explicitly handle all declared checked exceptions. We then re-use an existing rewrite [36] in the JastAdd front-end to add for every j a private method m (to j's declaring class) whose signature comprises j's declared return type, its declared checked exceptions, its declared formal parameters and additional formal parameters for referenced local variables. The rewrite automatically replaces j by a call to m with the appropriate arguments. We tag this method call with a special internal annotation telling the compiler that this method call originated from a Closure Joinpoint. The tag also carries the joinpoints fully qualified name. Pieces of advice that reference Closure Joinpoints are implicitly assigned a special pointcut. This pointcut only matches method calls that carry a tag with the fully qualified name of the correct joinpoint type. The actual advice weaver remains unchanged. Our implementation, along with many test cases, is available as open source at: `http://bodden.de/research/cjp/`

5. RELATED WORK

We first discuss other approaches to explicit joinpoints. Further, we discuss different approaches to expressive pointcuts that can match statement sequences, context-oriented programming and closures in Scala, C# and Java.

5.1 Explicit joinpoints

Steimann et al. propose explicit, block-based joinpoints in combination with a sub-typing relationship for joinpoints in general, to increase modularity when using implicit invocation with implicit announcement (IIIA) [37]. Allowing for sub-typing of joinpoints appears highly useful, as it allows pieces of advice that were designed for one joinpoint type to be re-used for other joinpoint types. In Figure 8, the piece of advice is strongly coupled with the joinpoint type `Buying`. Sub-typing would allow the same piece of advice to be reused for all subtypes of `Buying` as well. While we strongly encourage the use of such a joinpoint type hierarchy, and will investigate such a mechanism in future work, Steimann et al. used a semantics for their block joinpoints that we find a sub-optimal fit for Java. As we already showed in Section 1, the author's joinpoints introduce a surprising semantics with respect to variable bindings where variables are implicitly assigned values through around advice, but only for the duration of the block's execution. Further, pieces of advice receive joinpoints as objects that hold context information as fields. While users can declare some of these fields as final, pieces of advice can re-assign the other fields. To lower the barrier of adoption, we instead opted to leave the syntax for context access and proceed-calls unchanged. The only drawback of this syntax is that our explicit joinpoints can only return a single value to their lexical scope, whereas the explicit joinpoints from IIIA could "return" multiple values by assigning multiple fields. But this design saves us from having to allocate captured variables on the heap. As we discussed, heap allocation can cause subtle data races. Further, the authors do not discuss the semantics of unstructured control flow through `break` and `continue` statements and exceptions and the semantics that such control flow should have. As we showed in Section 2, the question of control flow is non-trivial and the semantics of control flow should be well defined. To summarize, we believe that a combination of the joinpoint types from Steimann et al. with our Closure Joinpoints would yield a powerful aspect-oriented language. Sub-typing for joinpoints would increase re-use, while Closure Joinpoints would likely lower the barrier of entry for Java programmers.

The idea of allowing base code to expose joinpoints explicitly pre-dates the work by Steimann et al. To the best of our knowledge, Hoffmann and Eugster were the first to propose explicit joinpoints as a language extension to AspectJ [24]. Interestingly, the authors propose two different kinds of joinpoints in their approach. Their regular explicit joinpoints resemble atomic events, syntactically similar to calls to static methods. Such joinpoints have no duration, as they do not enclose any base-program statements. Instead, they merely mark a location in the code. Such joinpoints presumably only make sense in combination with `before` or `after` advice, as there is no execution to replace using an `around` advice. Hoffmann and Eugster hence additionally introduce "scoped joinpoints" as their version of explicit block joinpoints. Similar to Steimann et al. also Hoffmann and Eugster do not discuss the problem of unstructured control flow. Regarding variable capture, the authors write:

> "Scoped EJPs [...] can access and modify local variables in all visible outer scopes, method signatures can remain unchanged and concerns can be added or removed simply by wrapping or unwrapping a block of code within a scope."

This suggests that the authors ignore (or are unaware of) the fact that, through this semantics, captured values need to be heap-allocated, and the problems that this may introduce. On the other hand, the authors do discuss an interesting feature that we did not consider: in their approach, aspects

[4]When defining this rule, we discovered a bug in the standard AspectJ compiler ajc: the compiler allows a `before` advice advising a `handler` joinpoint to throw checked exceptions even when the lexical scope of this joinpoint is not prepared to handle these exceptions. (see `https://bugs.eclipse.org/bugs/show_bug.cgi?id=326399` for details)

can "promise" to handle certain types of checked exceptions for certain types of scoped explicit joinpoints. If an aspect handles checked exceptions of type e for joinpoint type e then explicit joinpoints of type j may use code that can raise e without having to catch or forward the exception. In our eyes, such a feature appears very useful for certain aspects and would be a useful addition to the IIIA type system.

@Java [13] is a research project that aims at introducing Java 5 annotations in locations in which they are, according to the current Java Language Specification [21], not allowed. Specifically, programmers can add annotations to single statements, to independent blocks of code and to expressions. @AspectJ [15, 35] is a related project by the same authors that extends AspectJ with a mechanism such that the newly annotated syntactic elements can be advised. There is no semantic definition of @AspectJ and the work on @AspectJ does not mention any of the semantic complications that we discuss in this paper.

5.2 Pointcuts matching statement sequences

We next discuss approaches that introduce new pointcuts to the AspectJ language and match implicit joinpoints comprising multiple statement of a base program. In these approaches, the base code remains oblivious to being advised.

Akai et al. propose region pointcuts for AspectJ [4,5]. Region pointcuts consist a "region match pattern" over regular AspectJ pointcuts. This allows programmers to select regions, i.e., periods of time that start when matching one regular AspectJ pointcut and end when matching another. Region pointcuts are quite powerful in that they give aspect programmers very fine-grained control about which statements exactly constitute a joinpoint. On the other hand, region joinpoints may aggravate the problem of fragile pointcuts: because region pointcuts are very explicit about syntactic constructs of the base program, and even the order in which they occur, they may introduce tight coupling to the particular base code at hand. Explicit joinpoints circumvent this problem by assuming that the base-code programmer is aspect-aware and includes appropriate joinpoints in the base code. Although region pointcuts pick out implicit regions of the base code, these regions are nevertheless subjects to the very control-flow and data-flow constraints that we discussed. Akai et al. propose to handle assignments to captured variables through heap allocation. This results in assignment semantics similar to those of IIIA, with the same tradeoffs. Interestingly, the authors propose to handle unstructured control flow in a similar way [5]:

> "If there are jumps whose targets are outside of the region, [after weaving] their targets will be in another method. Since JVM dose not support such a inter-method jumps, these jumps will fail. To solve this problem, we assign identification (id) numbers to each jump to the outside. Next, each jump is replaced by the following instructions: 1. Save the id into a local variable (jump id variable) 2. Jump to the tail of the region. After the execution of the region, the jump id variable is checked, and then the thread jumps to the target specified by the jump id variable."

This solution may be possible when the around advice proceeds synchronously, i.e., executes in the control flow of its declaring context. However, the proposed solution will fail when the advice defers execution of the original joinpoint by passing a `proceed`-call to another thread (Figure 10). In such cases, jumps would have to raise an exception as was proposed in the original draft for BGGA closures (c.f. Section 3.2.1). The proposal for region pointcuts does not alter the syntax or semantics for pieces of advice.

Harbulot and Gurd propose an AspectJ language extension called LoopsAJ [23]. With LoopsAJ, the authors add a primitive pointcut `loop` that can match loops in the program and can expose values such as the minimal and maximal value of the loop-iteration counter and its stride. LoopsAJ's performs loop matching on a best-effort basis: because it attempts to recognize loops on the bytecode level, LoopsAJ can recognize reducible (i.e., well-structured) loops only [3]. Interestingly, though, such loops that are recognized automatically fulfill the data-flow and control-flow constraints that we mentioned. In particular, LoopsAJ will never match loops that break or continue to outside the loop or return abruptly. Similarly, LoopsAJ only matches loops that have a single assignment that is not loop-invariant. By definition, this assignment is the increment of the loop counter. If the loop assigns any other variable, e.g., a variable from the enclosing lexical scope, then the loop will simply not be matched. While such semantics seem reasonable for the special case of matching loops, they would certainly be unreasonable for explicit block joinpoints: an explicit joinpoint is, by definition, supposed to be advised. Hence, the decision of whether or not an advice will advise an explicit joinpoint should not depend on subtle control-flow and data-flow rules. Our type checks prevent programmers from violating such rules in the first place.

One particularly useful extension to AspectJ appears to be the notion of a joinpoint for synchronized blocks. As Xi et al. show [40, 41], joinpoints for synchronized blocks can allow programmers to implement different synchronization schemes in a truly modular fashion. Unfortunately, the authors do not address (nor even mention) the control-flow and data-flow constraints that we discuss in this paper. This causes unclear semantics, in particular for the multi-threaded programs that the authors consider.

Table 1 summarizes our comparison with other proposals for block-based joinpoints and pointcuts.

5.3 Joinpoint matching through Blueprints

Cazzola and Pini propose "The Blueprint Approach" for matching on joinpoints [14]. In this approach, the authors provide an AspectJ-like programming language with a high-level pattern-based join point model, where join points are described by join point blueprints, graphical representations of behavioral patterns describing where join points should be matched. Although the project's main goal is to make pointcut definitions more robust with respect to code evolution, the behavioral patterns that the blueprints also appear more expressive than regular AspectJ pointcuts. It therefore appears that blueprints could replace explicit joinpoint definitions at least in some situations where regular AspectJ pointcuts would fail.

5.4 Context-oriented programming

As Clarke et al. [17] define, Context-oriented programming treats execution context explicitly and provides means for context-dependent adaptation at runtime. In their paper, the authors address the semantic problem of closures

Approach	Joinpoints	Kind	Captured variables	Jump targets
IIIA [37]	explicit	stmt	locals & fields (heap-allocated)	?
Explicit Joinpoints [24]	explicit	stmt	locals & fields (heap-allocated)	?
Region Pointcuts [4, 5]	implicit	stmt	locals & fields (heap-allocated)	lexical scope
LoopsAJ [24]	implicit	stmt	restricted (data-flow analysis)	restricted (control-flow analysis)
Sync.-block Joinpoints [40, 41]	implicit	stmt	?	?
Closure Joinpoints (this paper)	explicit	expr	`final` locals & fields	local

Table 1: Comparison of related work with Closure Joinpoints

that escape the control flow of their enclosing lexical scope. As we mentioned, the semantics of executing such a closure not obvious. Clarke et al. discuss several different possible semantics for such an out-of-context closure execution, but leave the ultimate answer of the "best semantics" open. Indeed, it appears impossible to favor one semantics over another in all situations: a semantics close to dynamic scoping may appear natural in a dynamically scoped language like LISP, while programmers working with statically scoped languages like Scheme or Java would probably prefer the semantics that we propose. Costanza provides an excellent discussion of dynamic scoping in LISP, and even shows that one can implement dynamically scoped [39] advice in LISP by simply lifting dynamic scoping to function calls [19].

5.5 Closures in Scala, C# and Java

Lambda expression and closures are at the heart of the Scala programming language [33]. Scala satisfies some of the control-flow constraints that we mentioned by forbidding `break` and `continue` statements. The ambiguity of `return` statements is resolved as follows: by default, closures implicitly return the value of the last evaluated expression. Executing an explicit return statement, on the other hand, will cause the program to return from the enclosing lexical scope. Closures in Scala can read and write captured variables; they are heap-allocated. As we mentioned, this increases the risk for data races. In Scala, many programmers use actors, though, instead of explicit threads, which alleviates this problem again [22].

The C# programming language supports lambda expressions since version 3. Lambda expressions in C# follow a more conservative design than those in Scala. `return` statements return from the closure, not the enclosing lexical scope. `goto`, `break` or `continue` statements within closures are only allowed to bind to targets that reside in the closure itself. To ensure that read accesses to variables from the lexical scope can read a well-defined value, lambda expressions can only read such variables that are definitely assigned before the declaration of that expression [2, §7.4.14]. As in Scala, captured variables are heap-allocated.

In Section 3 we already discussed three quite prominent but informal approaches to adding lambda expressions and closures to Java. With JavaΩ [8], Bellia and Occhiuto propose a formal approach to enhancing Java with method types and pointers in general, and closures in particular. Control-flow semantics coincide with C#. With respect to data flow, users must explicitly choose: users can use `final` variables but can also annotate local variables with `shared`. This causes them to be heap-allocated, so that closures can write to them from any execution context. Closures cannot access variables that are neither final nor shared.

With Javaλ [34], Plümicke provides an approach to formalizing the existing proposals for adding lambda expressions to Java. Currently, though, the work formalizes the sub-typing relationship between lambda expressions only.

6. CONCLUSION AND FUTURE WORK

We have introduced Closure Joinpoints, a design for explicit block joinpoints that yields a syntax and semantics close to the Java programming language. We have discussed several important design questions for closures, and explained how these play into our design for Closure Joinpoints. Further, we have shown that most existing proposals for block-like joinpoints (both explicit or implicit) answer these questions insufficiently. Closure Joinpoints provide a comparatively clear and simple syntax and semantics.

In future work, we plan to combine Closure Joinpoints with sub-typing for joinpoints, as proposed by Steimann et al. Further, we will propose an integration of Closure Joinpoints into the Eclipse AspectJ project. To support programmers in writing Closure Joinpoints, we propose to implement an "extract joinpoint" refactoring, similar to the traditional "extract method" refactoring.

Acknowledgements.

We are very grateful to Max Schäfer, who contributed a great deal to our implementation by providing his refactoring package for JastAdd. We thank Lucas Satabin for helping us clarify the semantics of closures in Scala, and Friedrich Steimann for extensive discussions on the topic. Andreas Sewe and Jan Sinschek provided helpful feedback on a draft of this paper. We also thank the anonymous reviewers for their helpful comments. This work was supported by CASED (www.cased.de).

7. REFERENCES

[1] Android software development kit. `http://source.android.com/`.

[2] C# version 3.0 specification, September 2005.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Shumpei Akai and Shigeru Chiba. Extending AspectJ for separating regions. In *GPCE*, pages 45–54. ACM, 2009.

[5] Shumpei Akai, Shigeru Chiba, and Muga Nishizawa. Region pointcut for AspectJ. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 43–48. ACM, 2009.

[6] The AspectJ home page, 2003.

[7] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh

Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD*, pages 87–98. ACM Press, March 2005.

[8] Marco Bellia and M. Eugenia Occhiuto. JavaΩ: The structures and the implementation of a preprocessor for Java with m and mc parameters. *Fundamenta Informaticae*, 93(1-3):45–64, 2009.

[9] Oracle Berkeley DB, 2010. `http://www.oracle.com/technetwork/database/berkeleydb/`.

[10] Eric Bodden. AspectJ aspects for n-version programming, November 2007. `http://www.bodden.de/tools/`.

[11] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. BGGA closure proposal for Java, 2010. `http://www.javac.info/`.

[12] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *AOSD*, pages 199–211. ACM, 2007.

[13] Walter Cazzola, Emanuele Debenedett, Federico Pedemonte, Roberto Bentivogli, and Marco Poggi. @Java - a Java annotation extension. `http://homes.dico.unimi.it/~cazzola/atjava.html`.

[14] Walter Cazzola and Sonia Pini. On the footprints of join points: The blueprint approach. *Journal of Object Technology*, 6(7):167–192, August 2007. Aspect-Oriented Modeling.

[15] Walter Cazzola and Marco Poggi. @AspectJ - a fine-grained AspectJ extension. `http://homes.dico.unimi.it/~cazzola/ataspectj.html`.

[16] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2):pp. 346–366, 1932.

[17] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6. ACM, 2009.

[18] Stephen Colebourne and Stefan Schulz. First class methods for Java, 2007. `http://jroller.com/scolebourne/`.

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

[20] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In *AOSD*, pages 60–71. ACM, 2008.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification, 3rd edition*. Addison-Wesley Professional, 2005.

[22] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[23] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74. ACM, 2006.

[24] Kevin Hoffman and Patrick Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ*, pages 63–72. ACM, 2007.

[25] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the ACID properties for transactional objects. In *AOSD*, pages 202–213. ACM, 2006.

[26] Klaus Kreft and Angelika Langer. Understanding the closures debate, June 2008. `http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html`.

[27] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[28] P. J. Landin. The mechanical evaluation of expressions. *The Comp. Journal*, 6(4):308–320, 1964.

[29] Bob Lee, Doug Lea, and Josh Bloch. Concise instance creation expressions for Java, 2007. `https://docs.google.com/View?docid=k73_1ggr36h`.

[30] Philip Lee. pointcut to detect String concatenation at invocation of Log4j? aspectj-users mailing list, November 2004. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/`.

[31] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In *Programming Languages and Systems*, volume 4279 of *LNCS*, pages 131–147. Springer, 2006.

[32] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bulletin*, (15):13–27, 1970.

[33] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[34] Martin Plümicke. Formalization of the Javaλ type system. Technical Report tr_1010, Christian-Albrechts-Universiät zu Kiel, 2010. 27. Workshop der GI-Fachgruppe 2.1.4 "Programmiersprachen und Rechenkonzepte".

[35] Marco Poggi. @aspectj – an extension to the aspectj join point selection mechanism to support @java annotation meta-facility (in italian). Master's thesis, Università di Genova, 2009.

[36] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In *ECOOP*, pages 369–393, 2009.

[37] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1):1–43, 2010.

[38] Gerald J. Sussman and Guy L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[39] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In *AOSD*, pages 27–38. ACM, 2009.

[40] Chenchen Xi, Bruno Harbulot, and John R. Gurd. A synchronized block join point for AspectJ. In *FOAL*, pages 39–39. ACM, 2008.

[41] Chenchen Xi, Bruno Harbulot, and John R. Gurd. Aspect-oriented support for synchronization in parallel computing. In *LATE*, pages 1–5. ACM, 2009.