Abstract—Denotational semantics is an approach for giving a mathematical meaning to programming languages and systems. It gives the language designers a tool for high level abstract definitions. In aspect oriented programming, advice is weaved in designated locations of an underlying program specified by a pointcut expression. It is the job of the language implementor to specify how weaved code gets to be inserted into the proper location. Current denotational semantics of languages do not have the necessary constructs for accepting this weaved code. In this paper, we present a denotational semantics formal description that embodies the representation of a construct to be woven by some aspect. It illustrates the use of advice, how and why advice can be woven at their joinpoints. The semantics presented are intended to be a general baseline for the use of any advice in any random joinpoint.

I. INTRODUCTION

Denotational semantics is a tool that offers a mapping from programming syntax into mathematical objects. These objects are a formal descriptions that provide semantics (i.e. meaning) of some program, or part of it. The mapping is a way to model several aspects in a programming language; such as memory, state of the program, etc. A mathematical object representing this meaning is called the denotation. The foundations of denotation semantic is more precise in defining the meaning compared to other types of formal semantics like operational or axiom semantics [2].

Aspect oriented programming (AOP) is an elegant framework for constructing and implementing program behavior that is orthogonal to the underlying program code base. This type of crosscutting concern historically meant that the code implementing the behavior would be scattered around in the program and not localized nor modularized. An AOP compiler not only converts the code into object format; it also has the duty of finding the proper location in the code and instrument that specific location with some injected (weaved) code segment that is orthogonal to the underlaying program.

This paper does not present the semantic of an aspect oriented programming language syntax, but rather of the semantic of a language that is capable of being weaved into (i.e. the underlying program of an aspect). For simplicity, we present a mini language showing the conceptual model of pre-woven code. The denotational semantics of this language accepts the before advice in the advice model. This means that the after advice and the around advice is left out of the formal desertion. Section II of this paper will present a background of AOP. Section III sheds insight about the approach and the methodology used. Section IV presents the analysis and the further work on this topic.

II. BACKGROUND

A. Aspect Oriented Programming

Aspect Oriented Programming (AOP) frameworks introduce the concept of modularizing crosscutting concerns. This scheme allows developers to focus on the program logic itself away from other features in the program that are essential but in some sense they cross the original logic of the program. An example on this could be found in a banking program. This program has in one of it modules a method that transfers a balance from one account to another. The business logic in this case is, to first ensure there is enough money to initialize the transfer, then add the funds to the destination account and subtract this amount from the source account. In actuality, the true implementation of this method has more than this simple logic. This implantation should first ensure that the user is securely authorized for this kind of transaction. Second it should ensure the transaction is complete after the new amount have been committed to the bank’s database. Third this method should append this transaction to the bank’s log. This presents a challenge for the developers given the fact these three crosscutting concern could be implemented in serval locations in code space.

AOP presents an elegant solution for these kind of issues. It presents a framework in which there crosscutting concerns are implemented in their own encapsulated modules called aspects. Then using some rules, these aspects are applied to underlaying program code. This gives developers a tool to factor out the crosscutting functionality in order to replicated as many times as need in the program source.

A basic AOP model defines some specific fundamental pointcut designators (PCD’s), which are features in the program execution where the advice of an aspect can be weaved

1Various literature refers to the basic built-in weaving-point designators of an aspect language as joinpoint types, primitive pointcuts, and pointcut designators. We use the latter term in this paper.
A composition language allows a pointcut expression to combine and constrain these to define a pointcut, which is a set of program joinpoints that satisfy the expression, and where the advice will be woven in.

In existing AOP frameworks, the fundamental pointcut designators are chosen somewhat pragmatically: they must be actually useful to an aspect programmer, but they must also be relatively practical to implement in the AOP system. Thus, in existing AOP systems, pointcut designators are typically points in the program where inserting instrumentation is "not too hard"; for example, method calls are very often used as one of the fundamental pointcut designators.

The most popular AOP system, AspectJ, implements AOP for Java programs. Its pointcut designators include method calls, method executions, object field accesses, exceptions, and a few others [5]. Other research [4] has extended to notion of where the location of pointcut designators could be defined and extended to be more detailed to include basic blocks of code, loops, and loop backedges. A typical example of an aspect is shown in figure ??.

Pointcut designators could be any point in the program. This does not restrict the definition of a point to be point based (i.e. some location in code) it could be also expand this notion to be a point in time as well. Examples of this is to run an advise every T time units or with T time unit of the start of a program[10].

B. Denotational Semantics

In the late 1960s Christopher Strachey and and his research group with the help of Dana Scott [13] started on developing denotational semantics. The original purpose was for it to be a tool of analysis. Over the years, denotational semantics has proved to be a way for language design and abstract implementation as it makes a connection between the syntactic form and the semantics. The original idea was based on the recognition that programs and the objects they manipulate are symbolic realization of abstract mathematical objects.

The anatomy of a program attributes can broken down into three parts
1- The syntax: which is based on the grammar and the structure.
2- The semantics: in which some meaning is attached to part of the syntax.
3- The usability of the language or pragmatics: Where to use a particular language in the sense of effectiveness and efficiency.

The typical view of programming languages is in one of two paradigms; the declarative paradigm, and the imperative paradigm. Both paradigms have well known programming languages each with its own syntax, semantics, and pragmatics. The syntax is specified by the grammar and defined using the Backus-Naur form. The parser has the responsibility to check for syntax errors and abnormalities. Although the syntax sometimes is common in two more languages but the semantics and pragmatics could be different. A good example of this is the condition used in if-statements and loops in C, C++ and Java. The syntax of the if statement is:

```java
if (expression)
    statement(s);
else
    statement(s);
```

The traditional view an expression is to be a logical one; in which the expression evaluates to either true of false. The semantics behind this expression could be an opening of a file handler or some basic relational expression evaluating to some boolean value. Even though the syntax is the same for C and Java but differences are noted. Examine this segment of code which uses the same type of condition in the above if statement:

```java
int x=5;
while(x)
x--;
```

This code will compile in C and C++ without any issues. Although the variable x is of type integer but that does pose a problem; both C and C++ BNF forms indicate the expression to result in a final numeric form. Any value that does not yield to zero is semantically true; otherwise the result is zero and that means false. The domain of boolean was not part of C++ in its initial design. The introduction of boolean became part of C++ in C99 [12]. So the notion of boolean to be part of the BNF did not exist until then. However, this same segment of will produce a compile error in Java; the compiler will simply not accept to transform it to a boolean result. This is because, from the start Java was designed to have conditional expressions yield to a boolean result.

Even though the true meaning of the expression is the same in both C++ and JAVA. As we can see the BNF of both languages for defining the expression is different, and both compilers behave in a different way. This is because the compilers do not preserve the true meaning of the expression which should be independent of the implementation. The notion of using denotational semantics guarantees a consistent meaning regardless of the implementation.

There are three well-known methods for semantic specification. The first is operational semantics [11]. The meaning of some program is the evaluation history that the interpreter produces when it interprets the program. This interpretation has its limitation in configuration and complexity. The second method for semantic operation is called axiomatic semantics.
No explicit meaning of a program is given, rather properties of the language constructs are defined and used with symbolic logic. The third method is denotational semantics; it is more abstract than the other two method. It has a high level modular structure for designers. It also can be expanded with more ease than the other methods. There are three main components for denotational semantics [2], [3], these are.

- Syntax: The syntax is defined by the CFG
- Semantic Algebras: The semantic algebras is a tool to show the meaning of a program as mapping from one basic domain to another.
- Valuation Functions: A mapping from a languages abstract syntax structures to meanings drawn from semantic domains.

One could indicate there is some degree in equivalence in the above example concerning the boolean result. In fact, the operational semantic derivation method does not differentiate between the two because of the abstract meaning of "expression". Several research has been accomplished over the are semantics expanding the notion of abstraction and equivalence. A recent publication examines semantics with nominal Scott domains: "two program phrases have the same observable operational behaviour in all contexts if and only if they denote equal elements of the nominal Scott domain model" [7]. The domain of semantic usage extends also to other areas to include introducing a new calculus system for polymorphic programming languages used in game development [6].

III. Denotational Semantics of AOP

This section introduces the denotational semantics of an underlying languages that accepts advice to be woven into it. We assume the weaving could be static occurring at compile time. The weaving could also be dynamic. Meaning there is a place into where the woven advice could be inserted at runtime [1].

AOP contains three main types of advice; the Before advice, the After advice, and the Around advice. The before advice is woven before a particular pointcut expression. This could be a start a method invocation or simply the start of a loop. The After advice is woven after a specific pointcut expression. The Around advice wraps the execution of its target. It is considered very powerful because of the fact it can execute both before and after the target code. It could also prevent the target from executing in many AOP frameworks.

The similarities in the implementing the semantics is almost the same for the three types of advice. We provide the formal denotational semantics illustrating the use of the before advice in AOP. The syntax used here is the same used in Schmidt’s book in denotational semantics [2]. Below is the Backus-Naur form (BNF) describing the abstract syntax.

```
<program>::=<declaration><procedure><advice><block>
<declaration>::=var <identifier>
|<declaration>;<declaration>
|ε
```

The BNF describes a simple grammar used for illustrating the concept a the mini-language where weaved advice is applied. The language has the following constructs and properties; A program in this mini language begins in a typical fashion with a declarative part containing set of global declarations, followed by procedure declarations, followed by advice declarations; all of the above are presented as non-terminals, followed by the the terminal main procedure declared in the next block non-terminal. The language declares simple variables that use natural numbers only. The procedure definition is in the simplest manner. The procedure does not have a parameter list and does not take any arguments. The language handles only basic expressions. There are only two operations that can apply to an expression; these are the plus, and multiplication operations.

The statement structure defines a subset of what is found in a complete programming language. The languages does not contain any type of loops (while, for, nor do-while). Conditional structures like the if statement also do not exist. Finally there are no boolean expression or a boolean data type. We took the before advice and applied it to one pointcut designator (the execution designator).

This work assumes that matching is only based on qualifying procedure name, hence there is no access modifier type or access specifier. In the match also we are no passing any arguments to the procedure. Next, we present the semantic algebras for the language.

Identifiers

- Domain i ∈ Id=Identifier

Natural Numbers

- Domain n ∈ Nat= N

Operations

- zero, one, . . . , Nat
- plus : Nat Nat → Nat
- mul : Nat Nat → Nat
Program: `<declaration >→ store → store

Declaration[ var id]=λs.(isNat(s id) of isNat(a)→s
isProc(b)→s
isUndef(d)→(update s [id])

Declaration[D1:D2]=λs.
let S1=Declaration[D1]s
in (Declaration [D2]S1)

Block:<block >→ store → store
Block[D,St]=λs.

let S1=Declaration [D]js
in (Statement [St]S1)

Expression:<expression >→ store → ExpressableVal
Expression[n]=λs.inNat(Number[a])
Expression[id]=λs.cases(access s [id]) of
isNat(a)→inNat(a)s
isProc(b)→inProc(s)
isUndef(d)→inUndef()
Expression[e1+e2]=λs. cases(Expression[e1]s) of
isNat(a)→cases(Expression[e2]s) of
isNat(b)→inNat(plus a b)
isError(c)→inError()
isError(d)→inError()
Expression[e1*e2]=λs. cases(Expression[e1]s) of
isNat(a)→cases(Expression[e2]s) of
isNat(b)→inNat(mul a b)
isError(c)→inError()
isError(d)→inError()

An Expression is a mapping form a store to an ExpressableVal. There are four possibilities for an expression. The first is a natural number. The second and third is for it to be a addition or multiplication expression respectively. The fourth possibility is for it to be an identifier.

Statement:<statement >→ store → poststore

statement[id=e]=λs.cases(Expression[e]s) of
isNat(a)→cases(access s [id]) of
isNat(a)→inNat(Number[a])
isProc(c)→inProc(s)
isUndef(d)→inUndef()

Expression[st1;st2]=λs.let S1=Statement[st1]s
in (Statement [st2]S1)

statement[diveger]=λs.inError(s)

statement[call id]=λs.cases( access s [id]) of
isNat(a)→inNat(s)
isUndef(c)→inProc(s)
isProc(p)→let S1=Block p↓1 s
in (cases (S1) of isError(s)→inError(s)
isOk(s)→(Block p↓2 s3)

Advice:<advice >→ store → store

advice[id B]=λs.cases( access s [id]) of
isNat(a)→s
Undef(b)→s
isProc(p)→let S1=Block p↓1 s
in (cases (S1) of
isError(s)→inError(s)
isOk(s)→(Block p↓2 s3)

Procedure:<procedure >→ store → store

procedure[id B]=λs.cases( access s [id]) of
isNat(a)→s

As mentioned earlier, the mini-language only deal with natural numbers. The two operations (addition and multiplication) resulting in a natural number as well. The store is a mapping of identifiers to store elements. The elements of the store are either variables, procedures, or undefined. The main idea behind this work is to modify the Proc for it to be prepared to have the probability of a "before advice" to be weaved before reaching the the actual procedure in the underlying program. This is why the Proc is part of the store and represents a tuple of two blocks denoted by proc=(Block,Block). The first block represents the block content of the advice. The second block represents the block content of procedure. All other identifiers that are neither variables, nor procedures will be undefined. Next the valuation functions are presented.
The semantics of the valuation functions for executing the advice is the following: a procedure is defined in such a way that it is contained within the type Proc. Proc has two parts; the first part stores the advice, and the second part stores the procedure. Each time an advice is declared the store is updated to find a procedure that matches the advice. We assume that each advice will have a matching procedure, but it is not necessary for a procedure to have an advice. If there is no advice for a procedure then a nil is stored in the first part. A procedure call first executes part1 of the block (the advice), and if there is no error, part2 (the actual procedure) is executed.

The above description solves the issue of weaving advice statically or at run-time; there is always a place holder for the advice procedure to be called. In case there is no such advice, then no harm is done, and the control flow will continue executing the normally for the underlying program. There are a couple of ways that this could be implemented though it is out of the scope of this paper. One way could to always invoke some advice method before every statement that has no action if no actual advice is found for that statement. This will prove very insufficient and a waste of CPU time. Another way is to place a no-op before every statement that is replaced either at runtime with a method call to the advice.

IV. ANALYSIS AND FURTHER WORK

Advice in AOP is viewed as a procedure that could be invoked and executed based on a pointcut expression associated with that advice. That pointcut expression could specify a code based (location) or a time based location. The advice itself could be weaved in statically at compile/ re-compile time. It could also be weaved in dynamically. In either case the denotational semantics presented in this paper is does not worry about the implementation and prepares the language to have the place holder for the advice to be weaved.

Figure 2 illustrates the abstract general representation of the underlying language to accept the before advice. The figure represents a general view of how could a language weave some advice at any given location. We assume that program segment in the figure could be an arbitrary location like a method call or a conditional structure or even more fine-grained such as an assignment statement and is not specific only to a procedure call.

Future work could even expand this concept to extend the denotational semantic definitions implementing an after advice and around advice description. There are several ways to implement this concept; one could be to expand the Proc part fit the store to preprint a 3-tuple of three blocks proc=(Block,Block,Block). The first block represents the before advice. The second is for the actual content of the program segment. The third represents the after advice. In this case applying the after advice is easy specification of having the first block as nil. The around advice is a two fold in this case

- Case 1: The around advice crosses to proceed with the program segment. In this case all the three blocks in Proc are executed.
- Case 2: the around advice can perform custom behavior before and after the program segment. In this case, the second block in the Proc will be nil

The new semantics can even prove to have more usefulness in other venues of research. For example, code coverage [8] requires detecting every single program basic block ² that has been reached in the control flow of program execution. The semantics described in this work can accomplish this task. In fact Figure 3 describes coverage. The advice is fined grained and logs entry and exit of every basic block in the execution flow.

²Not to be confused with the block in the denotational semantics. A basic block is a sequence of one or more consecutive instructions without any jumps.
aspect TraceBasicBlocks {
    before(int blockID): basicblock() && args(blockID)
    {
        System.err.println("Entering Block --> "+blockID+" at "+thisJoinPoint.getSourceLocation());
    }
    after(int blockID): basicblock() && args(blockID)
    {
        System.err.println("Exiting Block --> "+blockID);
    }
}

Fig. 3. Example aspect using the basic block pointcut.

V. CONCLUSION

One of the fundamental notions for AOP is the notion of the advice; a segment of code that is orthogonal to the underlying program code and executes at certain location(s) called joinpoints specified by some pointcut expression. This paper presented a formal description of how to define denotational semantics for a mini language to accept the before advice at any location in the code space of an underlying program. The after and around have somewhat similar ideas to formally describe, but we are leaving that for and extended version of this work.

Although this work focusses only on the semantics of how advice is placed. It is easy to see these formal semantics to be part of a domain specific language as well. We believe that these and other new ideas of specifying semantics of pre-woven code could be extended to include other aspects of AOP.

REFERENCES