A Decision Tree-based Approach to Dynamic Pointcut Evaluation

Robert Dyer
Department of Computer Science
Iowa State University
rdyer@cs.iastate.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
hridesh@cs.astate.edu

Abstract

Constructs of dynamic nature, e.g., history-based pointcuts and control-flow based pointcuts, have received significant attention in recent aspect-oriented literature. A variety of compelling use cases are presented that motivate the need for efficiently supporting such constructs in language implementations. The key challenge in implementing dynamic constructs is to efficiently support runtime adaptation of the set of intercepted join points at a fine-grained level. This translates to two high-level requirements. First, since the set of intercepted join points may change, such implementations must provide an efficient method to determine this set membership, i.e., whether the currently executing join point needs to be intercepted. Second, the frequency with which such set membership needs to be determined must be minimized. In previous work, Dyer and Rajan proposed a dedicated caching mechanism to address the second requirement. In this work, we propose a mechanism to address the first requirement. This requirement translates to efficiently evaluating whether a join point is intercepted by a set of pointcut expressions. In the worst case, at every join point there may be the need to determine whether it is intercepted. Therefore, even modest savings in such mechanisms is likely to translate to significant savings in the long run.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features — Control structures; Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors — Code generation, Run-time environments

General Terms Algorithms, Design, Languages

Keywords Pointcut evaluation, decision tree, optimization

1. Introduction

In aspect-oriented (AO) languages [9, 15], join points are implicitly-defined by the language as certain kinds of standard actions (such as method calls) in a program's execution. Pointcut designators (PCDs) are used to declaratively select a subset of join points in the program. These selected join points are then composed with additional behavior based on a declarative specification. For this composition (often called weaving), it is necessary to evaluate the PCDs to determine the sub-set of join points that they select. In statically-compiled AO languages, the bulk of the PCD evaluation is done at compile-time and the remaining evaluation is deferred until run-time (often called dynamic residue) [12].

PCD evaluation needs to be deferred until run-time in two cases. First, when the necessary information for PCD evaluation is not known until run-time, e.g., in the case of **`if`** PCDs where the boolean condition needs to be evaluated at runtime, **`this`** PCDs where the exact type of the object may not be known statically, **`cflow`**-like PCDs where the exact control-flow graph may not be known statically, etc. Second, when a new PCD is added to the system, e.g., by dynamically loading a class containing new PCDs and thereby changing the system configuration at runtime [21], by creating new PCDs in more dynamic approaches that support first-class PCDs [30], etc.

A number of techniques have appeared that optimize PCD evaluation for the first case, i.e., when PCD evaluation requires run-time information. Among others, Aotani and Masuhara [2] optimize analysis-based pointcuts, Avgustinov et al. [3] optimize control-flow based pointcuts, Bodden et al. [7] optimize history-based pointcuts, Klose, Ostermann and Leuschel [16] use partial evaluation to reduce the PCD evaluation done at run-time, and most recently Sewe, Bockisch and Mezini [26] optimize evaluation of dynamic residues by eliminating common pointcut expression evaluation. The focus of this paper is optimizing the second case, where new PCDs are added.

Allowing new PCDs to be added to already executing systems is useful for a number of use cases, e.g., in run-time monitoring, run-time adaptation to fix bugs or add features to long running applications, run-time update of dynamic policy changes, etc. AO constructs of dynamic flavor that fit into these categories are beginning to appear [1, 4–6, 8, 11, 13, 14, 18–21, 23–25, 27–29]. For example, one may want to dynamically modify the behavior of a long-running application (such as a web-server) to start monitoring incoming requests, perhaps after sensing a denial-of-service attack, and then later remove such monitoring once the attack has been thwarted. To model **`cflow`**-like PCDs one may want to start monitoring the likely join points, once the execution reaches the desired entry point in the control-flow, and turn-off monitoring once it reaches the desired exit points [8, 11].

To support such use cases it is important to investigate efficient techniques for runtime PCD evaluation. To that end, this paper makes the following contributions.

- A precise formulation of the PCD evaluation problem and its two different classes that call for different solutions;
- the notion of predicate ordering – based on evaluation cost to optimize PCD evaluation; and
- a decision-tree based technique, corresponding algorithms and data structures for PCD evaluation.

The rest of the paper is organized as follows. In the next section, we formalize the PCD evaluation problem. Section 3 presents our algorithms for PCD evaluation independent of the predicates used in writing PCD expressions. Our method for partially evaluating type predicates is discussed in Section 4. Section 5 discusses related work and Section 6 concludes.

2. PCD Evaluation Problem

In this section, we model the PCD evaluation problem. This is inspired from the formalization of the event matching problem in publish/subscribe systems as described by Fabret et al. [10].

2.1 Terminology

We show the basic terminology used throughout this paper in Figure 1. The definition of *PCD* is fairly straightforward. A *PCD* is either a basic predicate *pred* or a logical conjunction/disjunction of a *pred* and a *PCD*. Note that we do not have negation of a *PCD*, as we assume this is easily emulated using conjunction/disjunction and the operators provided by the various predicates.

A predicate is defined as a 3-tuple (a, o, v): an attribute, an operator and a value. Some example attributes are: modifier(s), return type(s), argument type(s), receiver type, receiver name, method name, control flow, join point kind, etc. The collection of attributes available distinguishes the pointcut expression language. Operators are defined as higher-order functions $o : \mathcal{A} \times \mathcal{V} \rightarrow (\mathcal{A} \times \mathcal{V} \rightarrow \{\texttt{true},\texttt{false}\})$. A predicate can also be thought of as a function ($pred : \mathcal{A} \times \mathcal{V} \rightarrow \{\texttt{true},\texttt{false}\}$) obtained by evaluating the expression $o(a, v)$.

$$\begin{array}{rcl}
\textit{pred} & ::= & (\ a\ ,\ o\ ,\ v\ ) \\
\textit{fact} & ::= & (\ a\ ,\ v\ ) \\
\textit{PCD} & ::= & \textit{pred} \\
 & | & (\ \textit{PCD}\ ) \\
 & | & \textit{pred}\ \&\&\ \textit{PCD} \\
 & | & \textit{pred}\ ||\ \textit{PCD} \\
j & ::= & \textit{fact} \\
 & | & \textit{fact}\ \&\&\ j \\
 & & \\
a & \in & \mathcal{A}\text{, the set of attributes} \\
o & \in & \mathcal{O}\text{, the set of operators} \\
v & \in & \mathcal{V}\text{, the set of values}
\end{array}$$

Figure 1. Basic Terminology

A join point is defined as either a basic *fact* or a logical conjunction of a *fact* and a join point. A *fact* is defined as a pair (a, v) meaning that at the join point the attribute a takes the value v. Evaluating whether a join point satisfies a predicate is equivalent to evaluating each *fact* in the join point w.r.t. the predicate. A *fact* (a', v') satisfies the predicate (a, o, v) if and only if $(o(a, v))(a', v')$ evaluates to **`true`**.

2.2 Example

The following example illustrates the terminology for *PCD* and join points in the context of a small PCD expression language. Let,

- $\mathcal{A} ::= \{modifier, type, name\}$,
- $\mathcal{V} ::= \{v : v$ is a modifier, type or name in the program$\}$, and
- $\mathcal{O} ::= \{$`==`, `!=`$\}$, where the operators have their usual meaning.

An example PCD expression in such a language would be:

```
(modifier, ==, public)                 &&
(type, !=, void) && (name, ==, "Set")
```

and an example join point would be:

```
(modifier, public) && (type, FElement) &&
(name, "Set")
```

2.3 PCD Evaluation

For a program, let $\mathcal{J}$ be the set of join points (possibly unknown statically) and $\mathcal{P}$ be the set of *PCD* expressions as defined previously. We present two alternative formulations of the PCD evaluation problem ($PCDEval$).

1. Given a join point and a set of pointcut expressions, we are interested in determining the subset of pointcut expressions that may select this join point. Formally, $PCDEval : \mathcal{J} \times 2^{\mathcal{P}} \rightarrow 2^{\mathcal{P}}$, where $2^{\mathcal{P}}$ is the power set of $\mathcal{P}$.
2. Given a pointcut and a set of join points, we are interested in determining the subset of join points that are selected

by such a pointcut. Formally, $PCDEval' : \mathcal{P} \times 2^{\mathcal{J}} \rightarrow 2^{\mathcal{J}}$, where $2^{\mathcal{J}}$ is the power set of $\mathcal{J}$.

These formulations are useful for compile-time, load-time, and runtime pointcut evaluation. For example, most AO compilers today use the first formulation for weaving. The rationale is that often the total number of join points is larger compared to the total number of pointcuts and an efficient solution to the first formulation may reduce the number of iterations through the set of join points. This formulation also fits better with aspect-oriented systems that allow load-time deployment with a closed-world assumption for aspects, such as the AspectJ's load-time weaver. Here, by closed-world assumption for the set of aspects we mean that all aspects are loaded before any class is loaded. During class loading, each join point shadow in the class is matched with the set of pointcut expressions. Furthermore, aspect-oriented systems that allow run-time deployment, such as the *Nu* virtual machine [8], can also utilize this strategy. In *Nu* for example, a lazy strategy is used for run-time weaving, where a join point is not matched until it executes at least once.

On the other hand, for incremental compilation of AO programs, in cases where the increments introduce new *PCD* expressions, it would perhaps be more appropriate to use efficient solutions to the second formulation ($PCDEval'$). Load-time weavers with an open world assumption for aspects would also benefit from the second formulation, if they employ an eager strategy for matching. Such an eager strategy would match the PCDs just loaded with join point shadows in all classes already loaded, perhaps to avoid matching overhead during execution. Similarly, runtime weaving systems can also utilize $PCDEval'$ for cases where deployed aspects affect the "hot" segments and they are unlikely to be un-deployed. Other formulations can also be conceived that perhaps take a hybrid approach, but for the purpose of this paper we will not consider them.

3. PCD Evaluation Algorithm

This section describes our approach for PCD evaluation. The minimum requirement is fairly straightforward: the worst-case time complexity of our technique should not exceed the time complexity of current PCD evaluation methods. An additional requirement that we impose on our technique is that its amortized complexity should be independent of the number of PCDs in the system. Note that in this paper we are only considering the first formulation $PCDEval : \mathcal{J} \times 2^{\mathcal{P}} \rightarrow 2^{\mathcal{P}}$.

3.1 Predicate Ordering

The first step in our PCD evaluation technique is to order the evaluation of predicates in the PCD. Note that a PCD consists of conjunction and/or disjunction of one or more predicates. To determine whether a join point is selected by a PCD, it is necessary to determine whether there exists a satisfiable assignment of the predicate(s) at the join point for which the PCD evaluates to true.

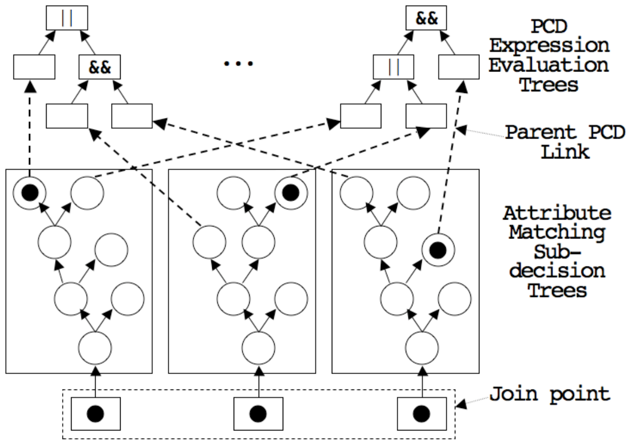

||
&&
&&
||
...
PCD Expression Evaluation Trees
Parent PCD Link
Attribute Matching Sub-decision Trees
Join point

Figure 2. An Overview of our Decision Tree-based Approach for PCD Evaluation

Let $\mathcal{C}$ be an amortized cost function such that $\mathcal{C}(a_i)$ is the amortized cost of evaluating the attribute a_i over operators defined for a and the set of values for a_i. The first step in our technique is to order the evaluation of each $a_i, a_j \in \mathcal{A}$ such that a_i is evaluated prior to a_j, if $\mathcal{C}(a_i) < \mathcal{C}(a_j)$. If $\mathcal{C}(a_i) = \mathcal{C}(a_j)$ ordering may be determined heuristically.

The rationale for adopting this policy for predicate evaluation ordering is to decrease the amortized cost of PCD evaluation by eliminating as many PCDs as possible at a lower cost. This strategy goes back to the efficient ordering of boolean predicate evaluation in SAT solvers. It has also recently been applied by Sewe, Bockisch, and Mezini [26] for optimizing evaluation of dynamic residues.

3.2 Data Structures for PCD Evaluation

An overview of the data structures maintained for our PCD evaluation algorithm is shown in Figure 2. From the top to bottom, the figure shows the set of PCD expression evaluation trees, sub-trees for matching attributes in the pointcut expression language, and a join point (a logical conjunction of facts) being matched. The number of sub-decision trees depend on the types of attributes available in the pointcut expression language. For example, a pointcut expression language that only allows matching based on types would just have one such decision tree.

For languages that provide different kinds of join points, e.g. **`execution`**, **`handler`** and **`set`**, in an AspectJ-like language, it would be sensible to maintain this data structure separately for each join point kind, as these would be disjoint. Furthermore, for each join point kind it would be appropriate to customize the set of attribute sub-decision trees, e.g., decision trees for name and type for **`set`** and **`get`** join points.

The PCD expressions are organized into a forest of PCD expression evaluation trees. These trees may have cross-links. These cross-links are created for common-subexpression elimination. Addition of a new PCD to this forest proceeds as follows:

(a) add the component predicates of the PCD to their respective attribute decision tree,
(b) add the tree representation of the PCD to the forest, and
(c) create a parent PCD link between attribute nodes and the leaf node of the PCD expression evaluation tree (shown as bold dotted arrows in the figure).

Removing a PCD is the reverse of this process, except that optimizations due to common subexpression elimination must be taken into account. For this purpose, a simple reference count is maintained that reflects the number of PCDs that contain this attribute node or a PCD expression sub-tree as a parent.

3.3 Optimizations of PCD Expression Tree

We assume that the PCD expressions are locally optimized before being added to the PCD expression evaluation forest. For example, parts of a PCD that will never match are eliminated, common local subexpressions are eliminated, etc. Sewe, Bockisch, and Mezini [26] discuss some of these techniques.

We also optimize PCD expressions by reorganizing the PCD expression trees. An example reorganization is shown in Figure 3, where the OR operator is successively propagated downward.

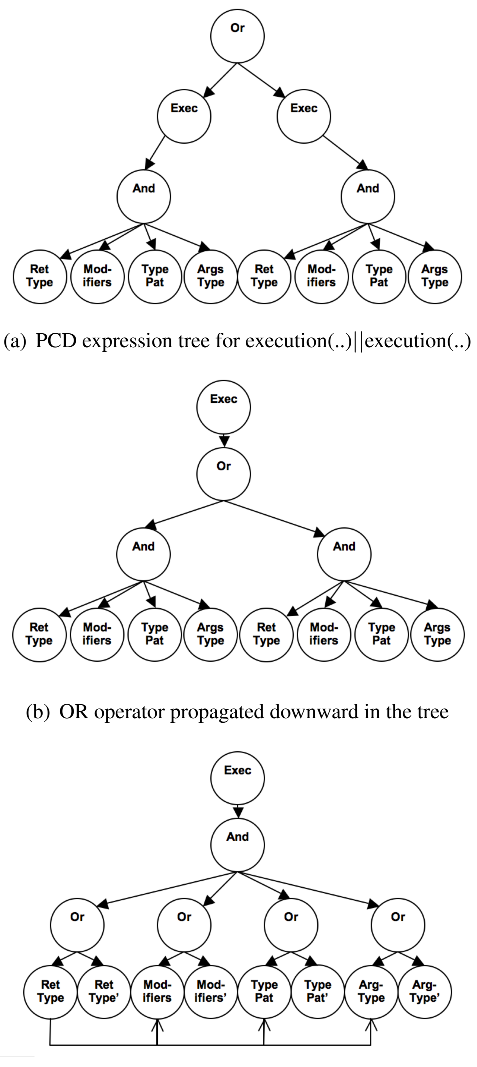
The reorganization is done using post-order tree-traversal technique and it terminates when a *classifier attribute* is the root node of every PCD expression tree. *Classifiers* are attributes that help pigeonhole PCD expression trees into a disjoint subset of join points. For example, the join point shadow kind is a type of classifier as it helps categorize the PCD expression tree into different classes based on which join point kind they match.

The reorganization of PCD expression trees has three benefits:

1. It helps reduce the depth of the PCD expression tree,
2. it helps classify PCD expression trees into disjoint sets for which separate PCD expression forests could be maintained, and
3. it enables elimination of certain PCD expressions by partial evaluation.

We will discuss more partial evaluation strategies in Section 4. These three benefits directly translate to decrease in the PCD evaluation overhead potentially reducing the runtime overhead of dynamic deployment of aspects.

There are no general techniques for maintaining the decision tree for each attribute. Instead it depends very much on the kind of the attribute. Efficient matching of modifiers, for example, requires completely different data structures and algorithms compared to matching of names and types. In this paper, we discuss algorithms and data structures for some of these, but we do not attempt to be exhaustive.



(a) PCD expression tree for execution(..)||execution(..)

(b) OR operator propagated downward in the tree

(c) OR operator propagated further down

Figure 3. Reorganizing pattern tree by propagating OR operator downward

3.4 Matching of Join Point Facts

A join point (or the conjunction of facts about a join point) is matched against this combination of PCD expression forest and attribute decision trees. The PCD evaluation starts with lowest cost attribute as discussed in Section 3.1. On traversing the attribute decision tree, at each node decisions are made about whether the current fact about the join point implies the predicate represented by that node.

If the predicate represented by the current node is implied, a *token* is sent to the leaf nodes of each parent PCD expression that contains that predicate as a component. The

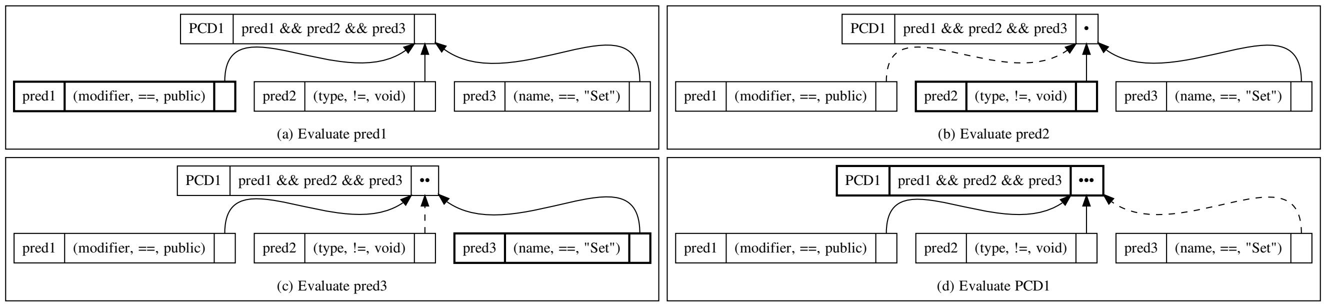


Figure 4. Example of matching the join point $(modifier, public)$ && $(type, FElement)$ && $(name, \text{“}Set\text{”})$. Dotted lines represent tokens being sent to a parent node. Bold boxes indicate the predicate being evaluated at that step.

leaf nodes in the PCD expression send tokens up the PCD expression forest depending on whether their parent node is a conjunction or a disjunction node.

While traversing the attribute decision tree, the algorithm keeps track of whether any tokens have been sent to the PCD expression forest. When the traversal of the attribute decision tree is complete, i.e., a leaf node in the decision tree is reached, if no tokens are sent this far to the PCD expression forest, the PCD evaluation terminates. This helps ensure that the least costly attributes often help short-circuit PCD evaluation.

A simple example of PCD evaluation is given in Figure 4. In this example, there is one *PCD* in the system. The *PCD* is $(modifier, ==, public)$ && $(type, !=, void)$ && $(name, ==, \text{“}Set\text{”})$. The join point we are trying to match is $(modifier, public)$ && $(type, FElement)$ && $(name, \text{“}Set\text{”})$.

At each step of the example we are evaluating one *pred* or *PCD* (indicated with a bold box). Consider step (a), where we are evaluating the *pred* $(modifier, ==, public)$. Since the join point contains $(modifier, public)$ this evaluates to **`true`**. The *pred* then sends a token to each parent node (in this case, there is one parent node – PCD1). The action of a token moving to another node is shown with a dashed line. In steps (b) and (c), pred2 and pred3 evaluate to **`true`** and similarly each sends a token.

The final step is (d), where PCD1 is evaluated. Since PCD1 is a conjunction of three *pred*'s, in order to evaluate to **`true`** it must contain three tokens. In this example it does, so PCD1 would evaluate to **`true`** meaning that the join point being matched matches PCD1.

4. Partial Evaluation of Type Predicates

In this section, we describe our techniques for partial evaluation of type predicates. The key idea behind our partial evaluation technique is to utilize the implication relationships between types. These partially evaluated predicates are then organized as a decision tree that helps optimize runtime evaluation. In the rest of this section, we describe various aspects of our technique. First, the partial evaluation results for logical operators are defined. We then describe a simple data structure and efficient algorithms that utilize these partial evaluation results.

4.1 Semantics of Type Operators

Let us suppose A, B, C, ... be the types in the program and *op* be the type operator, where $op \in \{\lessdot, \gtrdot, \doteq, \neq\}$ such that:

- $\boldsymbol{A \lessdot B}$ means that A is a strict subtype of B, i.e., it excludes the case when $A \doteq B$ (see below).
- $\boldsymbol{A \gtrdot B}$ means that A is a strict super-type of B, i.e., it excludes the case when $A \doteq B$ (see below).
- $\boldsymbol{A \doteq B}$ means that A is exactly of the same type as B.
- $\boldsymbol{A \neq B}$ means that A is not of the same type as B. In addition, A is not a strict subtype of B, and B is not a strict subtype of A.

Note that the meaning of the type operators is slightly different from the standard definitions. The operators are defined in this manner to facilitate partitioning the type predicates into disjoint subsets. We have used slightly different symbols to remind readers of the difference.

4.2 Semantics of logical inverse on type operators

Our technique for partially evaluating type predicates relies on computing the inverse of a predicate. For simplicity we compute the inverse of a predicate by inverting the operator. For example, the inverse of $(A \lessdot B)$ is computed by inverting the $\lessdot$ operation. Below we define the inverse of type operators. Let us assume that ! is the inverse operator such that $!(A\ op\ B)$ is defined as:

- $!(A \lessdot B) \equiv A \gtrdot B \vee A \doteq B \vee A \neq B$
- $!(A \gtrdot B) \equiv A \lessdot B \vee A \doteq B \vee A \neq B$
- $!(A \doteq B) \equiv A \lessdot B \vee A \gtrdot B \vee A \neq B$
- $!(A \neq B) \equiv A \lessdot B \vee A \gtrdot B \vee A \doteq B$

4.3 Partial evaluation of logical conjunctions (and) on type predicates

Let $\wedge$ be the logical conjunction operator, let $op = op'$ be the equality operator where $op, op' \in \{\lessdot, \gtrdot, \doteq, \neq\}$, and let T and F be the Boolean truth values with standard meanings.

Let $\mathcal{J}_1$ = (type, A) be a **fact** in the system where A is a type. Let $\mathcal{P}_1$ = (type, op_1, B) and $\mathcal{P}_2$ = (type, op_2, C) be type predicates where B and C are types. To see if the **fact** matches both predicates, we thus want to evaluate $(A\ op_1\ B) \wedge (A\ op_2\ C)$.

In order to partially evaluate this expression, we use a given fact $B\ op\ C$ and derive implication rules among the types B and C. Figure 5 shows the results for all four operators.

As an example of how we derived these rules, consider the case $B \lessdot C$ when $op_1 == \lessdot$ and $op_2 == \lessdot$. Thus we are interested in evaluating $A \lessdot B \wedge A \lessdot C$. If we assume $A \lessdot B$, since we are given $B \lessdot C$ we can see this implies $A \lessdot C$.

Now consider the case $B \lessdot C$ when $op_1 == \lessdot$ and $op_2 == \gtrdot$. Thus we are interested in evaluating $A \lessdot B \wedge A \gtrdot C$. If we assume $A \lessdot B$, since we are given $B \lessdot C$ we already showed this implies $A \lessdot C$. This would contradict $A \gtrdot C$ and thus this reduces to **`false`**.

Most cases are easily derived in a similar fashion, and thus omitted for space. We will discuss one interesting case. Again we have $B \lessdot C$. When $op_1 = \gtrdot$ and $op_2 = \neq$ we are interested in evaluating $A \gtrdot B \wedge A \neq C$. This expression can not be reduced. $A \gtrdot B \wedge B \lessdot C$ implies that B is a subtype of both A and C. $A \neq C$ implies that neither A is a subtype of C, nor C is a subtype of A. Therefore, in the semantics of single inheritance languages this would evaluate to F; however, in the semantics of languages that support limited multiple inheritance such as through interfaces in Java this logical conjunction may not be reduced.

The rules for the $\gtrdot$ operator is the mirror image of the $\lessdot$ operator along the main diagonal line. The rules for the $\doteq$ operator are easily derived, since anything not on the diagonal is clearly a contradiction and evaluates to F. For everything else, we can simply choose either of the two **facts**, as they are actually identical. Derivation of the rules for the $\neq$ operator are omitted for space reasons.

4.4 Attribute Decision Tree for Types

In this section, we discuss the algorithms and data structures for maintaining a decision tree for types. A pointcut expression language can employ such decision tree for matching return type, receiver type and argument types of methods, constructors, etc as part of join points of kind **`execution`**, **`call`**, **`initialization`**, etc, for types of fields for join points of kind **`set`**, **`get`**, for types of exception for join points of kind **`handler`**, just to name a few. Thus, the data structures and algorithms for this attribute are likely to be helpful in the implementation of PCD evaluation for common PCD expression languages.

This attribute decision tree could also be useful for VM-based implementations of languages that match purely based on types, such as Ptolemy [22].

We first discuss a technique for adding a type predicate to the decision tree. This technique makes use of a partial-evaluation function to optimize the matching process. This partial evaluation function was described in the previous section. We then discuss a technique for matching type-related facts about the join point using this decision tree. As previously discussed, it would be sensible to maintain separate copies of this decision tree for each kind, e.g., return type, argument type, and receiver type.

4.4.1 Addition of Predicates to Type Decision Tree

Our algorithm for adding a type predicate to an existing decision tree is shown in Algorithm 1 and explained below.

Algorithm 1: Insert: Adds a Type Predicate to the Predicate Tree

```
Input: Predicate tree: Tree, Predicate: Pred
Current = Tree.Root;
if Pred == true then
    Current.Parents.Append(Pred);
    return
while Current.TrueBranch != NULL do
    Current = Current.TrueBranch;
    if Current == Pred then
        Current.Parents.Append(Pred);
        return
    else
        result = PartialEval (Current ∧ Pred);
        if result == Current then
            return
        if result == Pred then
            Swap (Current,Pred);
            return
        if result == false then
            if Current.FalseBranch == NULL then
                Current.FalseBranch = new
                Node(Pred);
                return
            else
                Current = Current.FalseBranch
end
Current.TrueBranch.Parents.Append(Pred);
```

The addition of a type predicate to the predicate tree is an incremental process that starts with the root node of the current tree and the predicate that is to be added. The type predicate evaluation tree is maintained as a binary tree with two branches labeled `TrueBranch` and `FalseBranch`

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot B$	F	F	F
$op_1 = \gtrdot$	$(A \gtrdot B) \wedge (A \lessdot C)$	$A \gtrdot C$	$A \doteq C$	F or $(A \gtrdot B) \wedge (A \neq C)$
$op_1 = \doteq$	$A \doteq B$	F	F	F
$op_1 = \neq$	$(A \neq B) \wedge (A \lessdot C)$	F	F	$(A \neq B) \wedge (A \neq C)$

(a) Case: $B \lessdot C$

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot C$	$(A \gtrdot C) \wedge (A \lessdot B)$	$A \doteq C$	$(A \neq C) \wedge (A \lessdot B)$
$op_1 = \gtrdot$	F	$A \gtrdot B$	F	F
$op_1 = \doteq$	F	$A \doteq B$	F	F
$op_1 = \neq$	F	F or $(A \gtrdot C) \wedge (A \neq B)$	F	$(A \neq C) \wedge (A \neq B)$

(b) Case: $B \gtrdot C$ - As expected, the partial evaluation matrix in this case is the mirror image of the matrix for $B \lessdot C$ along the main diagonal.

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot B$	F	F	F
$op_1 = \gtrdot$	F	$A \gtrdot B$	F	F
$op_1 = \doteq$	F	F	$A \doteq B$	F
$op_1 = \neq$	F	F	F	$A \neq B$

(c) Case: B $\doteq$ C

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$(A \lessdot B) \wedge (A \lessdot C)$	F	F	$(A \lessdot B) \wedge (A \neq C)$
$op_1 = \gtrdot$	F	$(A \gtrdot B) \wedge (A \gtrdot C)$	F	$(A \gtrdot B) \wedge (A \neq C)$
$op_1 = \doteq$	F	F	F	$A \doteq B$
$op_1 = \neq$	$(A \neq B) \wedge (A \lessdot C)$	$(A \neq B) \wedge (A \gtrdot C)$	$A \doteq C$	$(A \neq B) \wedge (A \neq C)$

(d) Case: B $\neq$ C

Figure 5. Partial Evaluation Rules for Type Predicates

(except for the root node as described below). Both these branches need to be present at all time.

The current tree is traversed until an appropriate position for the current predicate is found. The root node of the tree represents the predicate **`true`** and it trivially matches any fact during the matching. As a special case all value types go to the false branch and all reference types go to the true branch of root.

All predicates `ret <: object` are trivially implied, if we are traversing the reference type branch. Therefore, if the predicate being added is that, it is simply appended to the root of the reference subtree. In particular, a parent PCD link is created from this node to the leaf node of PCD expression tree such that whenever this predicate evaluates to true parent PCDs can be notified. This step facilitates common-predicate elimination.

The algorithm terminates when the true branch of the root node does not exist. The new predicate is then added to the true branch of the root node.

If the true branch exists and the added predicate is not the trivially implied predicate `ret <: object` the main loop of the algorithm begins that continues until the current predicate being explored evaluates to **`null`**.

In this loop, implication relationships are used to determine the branch of the decision tree traversed. These relationships are computed using our partial evaluation rules for types as shown in Section 4. The function `PartialEval` facilitates that. If the result of this function is the current predicate `Current` or the predicate being added `Pred`, it means that true/false evaluation of one predicate implies true/false evaluation of the other. This works in general because implication is a transitive relation.

4.4.2 Simultaneous Evaluation of Predicates in the Type Decision Tree

Our algorithm for matching a type-related fact in an existing decision tree is shown in Algorithm 2 and explained below.

The evaluation of a type predicate tree against a fact starts at the root node of the tree. Note that the root node represents the predicate **`true`** and therefore any fact trivially matches this predicate. If there are complex predicates that contain true type predicates they are immediately notified.

If the fact implies the current predicate being evaluated, the current predicate and all predicates implied by it are automatically evaluated to be true and the matching algorithm terminates. If on the other hand the fact does not imply the current predicate being evaluated, we compute the relation-

Algorithm 2: Match - Evaluate a fact against a predicate tree, resulting in tokens at predicates that evaluate true for the fact

```
Input: Predicate tree: Tree, Fact: fact
1  Current = Tree.Root;
2  NotifyParents (Current);
3  while Current != NULL do
4      result = PartialEval (Current ∧ fact);
5      if result == Current then
6          NotifyParents (Current);
7          return
8      if result == Pred then
9          Current = Current.TrueBranch
10     else
11         Current = Current.FalseBranch
12     if fact == Current then
13         NotifyParents (Current);
14         return
15 end
```

ships between the type value in the fact `fact.Value` and the type value in current predicate `current.Value` to minimize matching. In particular, we evaluate the logical relationships that exist between these types (e.g., strict subtype of, strict supertype of, etc) as defined in Section 4.1.

The logical relationship between the type value in the fact and the type value in the current predicate is then used to lookup the partial evaluation results. We will describe these in details in later section. For understanding this algorithm, it is sufficient to know that the looking up statically computed partial evaluation results returns three different results. First, that suggests that the fact implies current predicate. Second, that suggests that the current predicate implies the fact. Third, that suggests that the fact may never imply the current predicate or vice-versa and fourth, that suggests that these values cannot be partially reduced.

The first case implies that the current predicate and all predicates implied by it will evaluate to **`true`** for the fact being matched. The second case implies that even though the current predicate will not evaluate to **`true`** for this fact, only predicates in its true subtree may evaluate to **`true`** (by construction), therefore only exploring the true subtree of the current predicate will be sufficient. The third and the fourth case imply that the current predicate and all predicates in its true branch will not evaluate to **`true`** for this fact, therefore only exploring the false subtree of the current predicate will be sufficient.

4.4.3 Implementations for Java

Very fast mechanisms exist for computing logical relationships between types such as the implementation of the **`instanceof`** construct in Java. The relationship information that we require can be computed with an **`instanceof`** and an `equals` comparison. Further discussion is beyond the scope of this paper, but it suffices to say that for a further reduced cost, an operator can also be implemented in the virtual machine that utilizes the information maintained for efficiently computing the **`instanceof`** relationships to compute these logical relationships at the cost of an **`instanceof`** operator.

There are two optimizations (not shown in Algorithm 2) implemented for languages such as Java, that support primitive value types and reference types and a top type `object`. The type predicate tree maintains a subtree for primitive value types and another subtree for reference types. If the fact is a value type, the evaluation proceeds with the value type subtree otherwise the reference type subtree is explored. Furthermore, all facts that proceed to match the reference type subtree, implicitly match the top type `object`.

5. Related Work

Recently, Sewe et al. described a method of using ordered binary decision diagrams (BDD) to eliminate redundant evaluations of dynamic residues [26]. Dynamic residues are the result of compilers statically performing partial evaluation on the pointcuts [17]. By converting the residues into an ordered BDD, they are able to evaluate the dynamic residues of all pointcuts for a given join point while only evaluating each atomic residue at most once.

Similar to our technique, they also order the evaluation of the atomic pointcuts using the cost of their evaluations for improved efficiency. Our matching technique however does not focus on the dynamic residues left over from previous partial evaluation of pointcuts by compilers. Instead, it focuses on dynamically evaluating the set of full pointcuts in the system against a join point. Both approaches do however make use of decision trees during the matching process.

Previous work by Klose et al. has shown that the use of partial evaluation techniques can reduce the amount of work needed to match a PCD at runtime [16]. Their specializer generates efficient checks for the program, however this is done offline. Similar to their approach, we use partial evaluation techniques to try and minimize the cost of matching a PCD, however our partial evaluation is performed online using dynamic information about the classes in the system. Thus, while we incur an overhead of performing the partial evaluation at runtime, we potentially have more information available for even more efficient PCD matching. This trade-off is most beneficial in systems where the set of PCDs changes often or systems that execute for a long period of time.

6. Conclusion and Future Work

The need for efficient support of dynamic aspect-oriented constructs dictates that more efficient techniques are provided in virtual machines for PCD evaluation. The use cases for dynamic aspect-oriented constructs are attractive and with the availability of more efficient implementations, more applications for such constructs can be explored, where concerns about overhead are an important factor in adoption. The decision-tree based technique for PCD evaluation that we present in this work seems promising in that regard, although a rigorous evaluation is needed to exactly characterize the benefits in terms of space and time complexity. In particular, it will be interesting to study the following parameters.

1. Total number of predicates in the system: Studying this parameter will show how the performance scales with the total number of predicates in the system.
2. Sub-classing (sub-typing) relationship: how many predicate values in the system are in the type hierarchy of other predicate values, i.e., strict super type of, strict subtype of, or equal.
3. Unrelated types: how many predicate values in the system are not related to other predicate values?
4. True predicates: studying this parameter will show how pointcuts that use a significant amount of wild-cards influence the performance of the type decision tree.
5. Percentage of successful matches: this parameter will show how successful matches contribute to the cost of performance evaluation. It will also help characterize the one time cost paid by the join points that match in a dynamic AO language model.
6. Percentage of unsuccessful matches: study of this parameter will show how unsuccessful matches contribute to the cost of performance evaluation. This will help determine the one time cost paid by the join points that do not match in a dynamic AO language model. In particular, the sooner the decision tree can determine that the join point is not going to match, the better.

Acknowledgments

This work is supported in part by Iowa State University's generous startup grant and the NSF grants CNS-06-27354 and CNS-07-09217. Thanks to the anonymous reviewers of the VMIL workshop for their comments.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th international conference on Object-Oriented Programs, Systems, Languages, and Applications*, New York, NY, USA, 2005. ACM Press.

[2] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-Oriented Software Development*, pages 161–172, New York, NY, USA, 2007. ACM Press.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[4] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 86–95, New York, NY, USA, 2002. ACM Press.

[5] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st international conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 109–124, New York, NY, USA, 2006. ACM Press.

[6] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[7] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 525–549. Springer-Verlag, 2007.

[8] Robert Dyer and Hridesh Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th international conference on Aspect-Oriented Software Development*, New York, NY, USA, 2008. ACM Press.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[10] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 international conference on Management of Data*, pages 115–126, New York, NY, USA, 2001. ACM.

[11] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 46–55, New York, NY, USA, 2004.

[12] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[13] Robert Hirschfeld. AspectS - aspect-oriented programming with Squeak. In *NODe '02: Revised Papers from the international conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[14] Robert Hirschfeld and Stefan Hanenberg. Open aspects.

Computer Languages, Systems & Structures, 32(2-3):87–108, 2006.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, Finland, June 1997. Springer-Verlag.

[16] Karl Klose, Klaus Ostermann, and Michael Leuschel. Partial Evaluation of Pointcuts. In *PADL '07: Proceedings of the 9th international symposium on Practical Aspects of Declarative Languages*, volume 4354, pages 320–334. Springer-Verlag, 2007.

[17] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC '03: Proceedings of the 12th conference on Compiler Construction*, pages 46–60. Springer-Verlag, 2003.

[18] Francisco Ortin and Juan Manuel Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71(3):229–243, 2004.

[19] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *REFLECTION '01: Proceedings of the 3rd international conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, London, UK, 2001. Springer-Verlag.

[20] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, New York, NY, USA, 2003. ACM Press.

[21] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[22] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*. Springer-Verlag, July 2008.

[23] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th international symposium on Foundations of Software Engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[24] Hridesh Rajan and Kevin J. Sullivan. Need for instance level aspect language with rich pointcut language. In *SPLAT '03: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[25] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[26] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Redundancy-free residual dispatch. In *FOAL '08: Foundations of Aspect-Oriented Languages workshop*, 2008.

[27] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *RV '05: 5th workshop on Runtime Verification*, 2005.

[28] Volker Stolz and Eric Bodden. Tracechecks: Defining semantic interfaces with temporal logic. *Software Composition*, pages 147–162, 2006.

[29] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[30] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167. ACM, 2003.