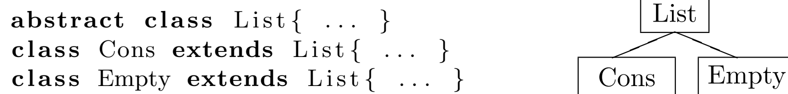
Algorithms in DemeterF

Bryan Chadwick

July 30, 2009

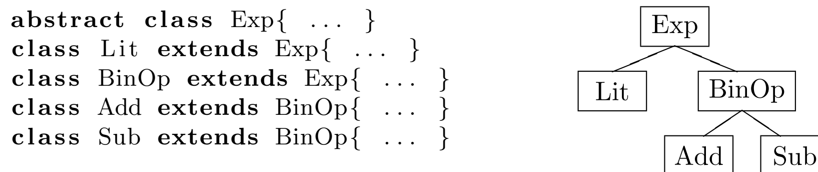
1 Background

In describing forms of static analysis for programming languages we often deal with meta-information representing *types*. In DemeterF we primarily deal with single inheritance, resulting in a *tree* (not a DAG) of types where the parent/child relationship represents both inheritance and subtyping. Some typical examples would be trees representing lisp-style cons lists and simple numerical expressions (Figures 1 and 2). We choose this visual (and mental) representation for class hierarchies, which fits nicely into various graph problems related to multi-method type checking.




```
abstract class List{ ... }
class Cons extends List{ ... }
class Empty extends List{ ... }
```

Figure 1: List classes and representative trees




```
abstract class Exp{ ... }
class Lit extends Exp{ ... }
class BinOp extends Exp{ ... }
class Add extends BinOp{ ... }
class Sub extends BinOp{ ... }
```

Figure 2: Expression classes and representative trees

Graph cartesian products (GCPs) in particular are useful when reasoning about method coverage for multiple dispatch in languages that support it. We define GCPs over trees of types, similar to the ones above, but for consistency we describe a tree as a graph, $T = (V, E)$, where every vertex except the root has in-degree of 1, and the leaves of T are vertices with out-degree of 0. Note that there is no restriction on out degree.

A GCP, G, is defined over a sequence of trees, $(T_1, \ldots, T_n)$. With each $T_i = (V_i, E_i)$, we define G as:

$$V = V_1 \times \cdots \times V_n$$
$$E = \{\, (\vec{u}, \vec{v}) \in V \times V \mid \exists i.(u_i, v_i) \in E_i \wedge \forall j \in [1..n]\,.\, i \neq j \Rightarrow u_j = v_j \,\}$$

Vertices here are tuples (or vectors) containing vertices from each of the corresponding trees. Above is actually just long-winded way of saying that there is an edge between vertices that differ by only one element position, and there exists an edge in that corresponding tree. For example, the graph cartesian product of

the two earlier trees is shown in Figure 3. Because both trees are of height greater than one, we see sharing at the leaves of the graph, meaning it can be characterized as a directed acyclic graph, or DAG.

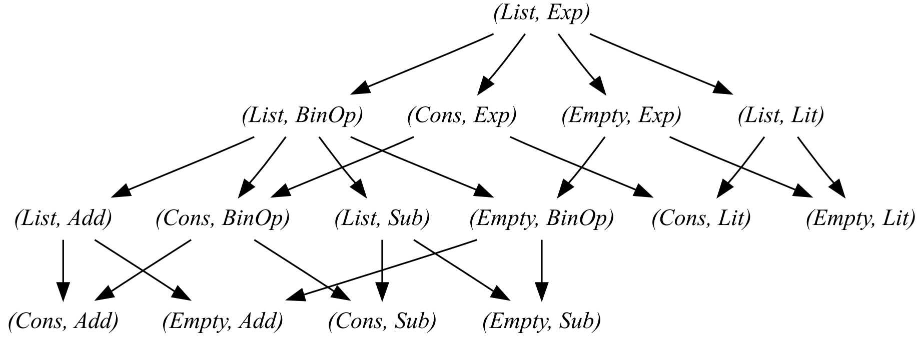


Figure 3: Graph cartesian product.

2 Leaf Covering

Multi-methods in multiple dispatch languages like CLOS and MultiJava rely on selecting the most specific method/function for the runtime types of its arguments. Interestingly, DemeterF (like CLOS) uses an *asymmetric* (think lexicographic) multiple dispatch strategy where the leftmost argument is given precedence in matching. MultiJava employs a *symmetric* strategy where all arguments are given equal weight, and method ambiguity is not allowed at runtime. In either style dispatch, there are several checking algorithms that can be reduced to various coverage problems on GCPs.

In DemeterF we want to be sure that for each sequence of argument types, all possible sequences of concrete types (leaves of the GCP) have an applicable method. For example, if the sequence $(List, Exp)$ is encountered for dispatch, we want to be sure that there is a method applicable for each of the concrete combinations:

$$(Cons, Lit),\ (Cons, Add),\ (Cons, Sub),$$
$$(Empty, Lit),\ (Empty, Add),\ (Empty, Sub)$$

.

To solve this problem, we reduce it to "leaf covering" for a GCP. As before, we view the class hierarchy as a set of (possibly connected) trees. The sequence of argument types becomes the *root* of our GCP that is generated from the corresponding type hierarchies. The type checker's task is to decide given a set of vertices, M, in the GCP, whether or not each leaf has some ancestor in M.

2.1 The Problem

The general algorithmic problem instance can be described as follows:

> Given a sequence of directed trees, $(T_1, \ldots, T_n)$, that implicitly define a GCP, $G = (V, E)$, and a set of vertices, $M \subseteq V$, we must decide whether or not each leaf of the GCP has some ancestor in M.

Leaves in the GCP can be defined as the vertices of V with *out-degree* of 0: $\{\, \vec{v} \in V \,|\, \forall\, \vec{u} \in V \,.\, (\vec{v}, \vec{u}) \notin E \,\}$.

2.2 Solution 1: Brute-Force

The brute-force solution is simply to enumerate all leaves of the implied GCP from the root, and check that for each leaf, $\vec{l}$, there exists a vertex $\vec{m} \in M$ such that each component of m is an ancestor of the

corresponding component of $\vec{l}$ in the corresponding tree. For a sequence of trees, $(T_1, \ldots, T_n)$, the running time of this algorithm is on the order of:

$$|M| * \prod_{n}^{i=1} |\, leaves(T_i)\,|$$

In this case, the total number of leaves of $(List, Exp)$ is 6, though in general this can be exponential in the number of trees. In usual cases we foresee n usually being smaller than 10 and $|M|$ being less than 3, usually 1 or 2.

2.3 Solution 2: Counting

A second, more involved solution deals with set intersection and counting. If we think about the graph in Figure 3 then we notice that there is leaf sharing amongst interior vertices. The number of overlapping leaves covered by a pair of vertices, say $\vec{u}$ and $\vec{v}$ is just:

$$overlap(\vec{u}, \vec{v}) = \prod_{n}^{i=1} |\, leaves(T_i, u_i) \cap leaves(T_i, v_i)\,|$$

For these two vertices, we can check if they cover all the leaves of the GCP by comparing the number of leaves to the number of non overlapping leaves:

$$\left(\prod_{n}^{i=1} |\, leaves(T_i)|\right) - overlap(\vec{u}, \vec{v})$$

If the difference of these two counts is 0, then the leaves of the GCP are covered by the two vertices, $\vec{u}$ and $\vec{v}$. This solution idea scales to any number of vertices, though in general calculating the intersection of n sets of leaves is exponential in n, using the set inclusion/exclusion principle. We believe that the leaf covering problem is *fixed parameter tractable*, though analyzing the constant factors involved will be interesting to determine which algorithm is better in practice. Given our probably values of n and $|M|$ counting may be our best strategy, though it doesn't provide an immediate witness that can be shared with the programmer.

2.4 Leaf Covering is coNP-Complete

We can show that leaf covering is *coNP-Complete* by reducing DNF validity (*tautology* checking) to leaf covering[1]. Consider a formula, F, in disjunctive normal form, where each $l_{i,j}$ is either a positive or negative assertion of a *variable*, *e.g.*, a or $\neg a$.

$$F = (l_{1,1} \wedge \cdots \wedge l_{1,n_1}) \vee \cdots \vee (l_{m,1} \wedge \cdots \wedge l_{m,n_m})$$

With an ordering on the variables used in F, say alphabetic, we create a sequence of trees with the variable names as roots, and the special symbols *true* and *false* as leaves. We then encode the clauses of our formula as elements of M, which include an encoding of each of the variables in order. We encode a positive literal as *true*, negative as *false*, and an unused variable as the root of its corresponding tree.

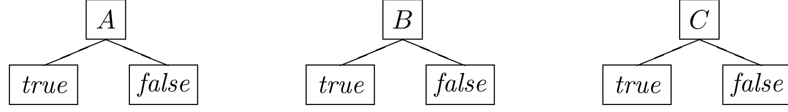
The leaves of the implied GCP contain all assignments of *true* and *false* to the variables of F. If the elements of M cover all the leaves, then all *concrete* assignments are covered by the formula, which means F is a tautology. If not, then one of the uncovered leaves represents an assignment that does not satisfy the formula.

As a complete example, consider the following formula:

$$F = (a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg b \wedge c) \vee (\neg a \wedge \neg c)$$

[1]Thanks to Yannis Smaragdakis for suggesting and detailing this reduction.

To convert the validity of this formula into a leaf covering problem, we order the variables as (a, b, c) and construct three corresponding trees:



The root of our GCP is the triple (A, B, C), and our set M encodes the clauses of F as triples:

$$M = \{(\mathit{true},\ \mathit{false},\ C),\ (\mathit{false},\ B,\ \mathit{true}),\ (A,\ \mathit{false},\ \mathit{true}),\ (\mathit{false},\ B,\ \mathit{false})\,\}$$

The leaves of the GCP include all triple permutations of *true* and *false*. Are all the leaves covered by the encodings of the selected clauses? The answer in this case is *no*; the leaves that are not covered, (*true*, *true*, *true*) and (*true*, *true*, *false*). The corresponding assignments to a, b, and c do not satisfy F: *i.e.*, F is not valid.

3 Dead Node Cover

The second DemeterF algorithmic problem deals with the execution of methods with signatures that overlap in some way. In general we want to be able to warn the programmer when the multi-methods that she writes will never be executed. Since our methods are called over a traversal, we know most of the types that will be encountered, and we can construct the corresponding leaf covering problem to match. But what if an element of M is shadows another? Then the most specific, closest to the leaf at runtime, should be chosen.

3.1 The Problem

Similar to leaf covering, the general algorithmic problem instance of dead node cover can be described as follows:

> Given a sequence of directed trees, (T_1, ..., T_n), that implicitly define a GCP, $G = (V, E)$, and a set of vertices, $M \subseteq V$, the selected vertex $\vec{m} \in M$ is considered `dead` if for each path from $\vec{m}$ to a leaf in the GCP, there exists another vertex in M.

3.2 Possible Solution

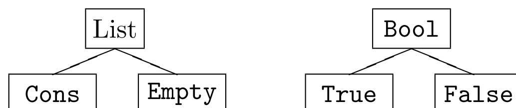
We must decide whether all the leaves that $\vec{m} \in M$ covers have another, more specific ancestor in M. If this is the case, then the more specific vertices (methods) will always be chosen, so this vertex is *dead*. Though we haven't yet tackled the implementation of this problem, the description leads us toward a (still exponential) bottom-up solution to both the dead node, and leaf covering problems.

Essentially, the idea is to recognize that if all the leaves below a vertex, v, of the GCP are covered by M, then it is the same as if v itself was part of M. If we begin this analysis at the leaves of the GCP, with an implied set of selected vertices, M', then each time the immediate neighbors of a vertex are in $M \cup M'$, that vertex can also be added to M'. If we code this algorithm recursively by traversing the GCP starting at the root, then we need only to find the first uncovered leaf.

After we finish, if the root of the GCP is in M', then *all* leaves of the GCP are covered by M. Otherwise, one of the vertices not in $M \cup M'$ can be given as a witness to the contrary. If during the course of running algorithm, we attempt to add an element $m \in M$ to M', then we know that all the leaves of m are already covered by other selected vertices so m will never be chosen, and can be considered dead code.

4 Method Residue

The method selection process in DemeterF is based on a deterministic algorithm that chooses the most specific signature based on the runtime types of its arguments. If there is only a single vertex selected then this decision can be made statically. If two methods overlap or are applicable to similar types then we must defer the selection to runtime. In DemeterF we want to generate traversal code statically by inlining method selections when possible, and writing efficient decision procedures that will be executed at runtime when the method cannot statically be determined.
For example, consider the following trees:



And selected vertices:

$$M = \{\, (\texttt{List}, \texttt{True}), (\texttt{Empty}, \texttt{False}), (\texttt{Cons}, \texttt{False}) \,\}$$

The first question is: does M cover the 4 leaves of the implied GCP? The answer is yes; the first tuple covers two of the leaves, and the second and third each cover one. The second question is: given an instance of the abstract tuple, (`List`, `Bool`), give a decision procedure that selects the most specific element of M, using the static information from the trees (subtype relationships), tuple selectors (`[1]`), and dynamic instance checks using `is`.

One possible efficient decision could be written as:

```
decide t =
  if t[2] is True
    then (List, True)
    else if t[1] is Cons
           then (Cons, False)
           else (Empty, False)
```

An example of a slightly less efficient, but correct, decision would be something like:

```
decide t =
  if t[1] is Cons
    if t[2] is False
      then (Cons, False)
      else (List, True)
  else
    if t[2] is False
      then (Empty, False)
      else (List, True)
```

Our task is to create a decision that minimizes the depth (both average, and maximum) of the `if` statements, while maintaining the correctness of dispatch. We currently have an algorithm for determining the decision, but it can generate inefficient code in some cases, and is rather ad-hoc. One key idea remains to be formalized: that of a *correct* dispatch decision. I believe this can be formulated by a specific ordering on the vertices of the GCP, but the details still need to be worked out.

5 Conclusion

That concludes a somewhat (in)formal description of the abstract algorithms involved in DemeterF. Many of them have been touched in the multi-method and predicate dispatch communities, but immediate application to our specific problems was not clear. We will continue working out the details, and look forward to discussing them further.