A Proposal for FTL's New Scheduler

Eric Atkinson

July 21, 2014

1 Introduction

The current scheduler for FTL is based on an enumerate-and-check algorithm that searches through possible schedules somewhat haphazardly and stops once it finds a valid one. It has the advantage of being easy to implement given a verification algorithm, but it is not able to explore the space of schedules in a principled way (which may be useful for building extensions to the FTL system). Also, the verification conditions are tricky to get right in all cases. This proposal is for an alternative scheduler that operates more directly with the dependency graph, allowing it to explore the schedule space more thoroughly.

2 The Dependency Graph

Instead of a general algorithm for constructing the dependency graph from an attribute grammar, we will consider the motivating example shown in Figure 1. This grammar defines a set of trees whose nodes each have four attributes (except the root node, which has none). We would like to generate code that can compute all attributes for any tree in the set. However, the grammar defines some data dependencies that significantly impact this code's operation (e.g. we must compute a Leaf's c attribute before we can compute its d attribute).

We can begin constructing a dependency graph as follows: create a vertex for each (class, attribute) combination, adding extra attributes for a class's children. Also, create an edge whenever there is a *local* dependency between two attributes (these are dependencies we can "read off the grammar"). This is shown in Figure 2.

As first observed by Knuth, we can obtain the dependency graph for any concrete tree by "stitching" together these local dependency graphs. The way we compose them is defined by the structure of the tree, which is not known at scheduling time. We assume instead that any valid "stitching" is possible. This means that, for example, if a class reads an attribute of its child, we need to add dependencies from *any* class that child could possibly be. Figure 3 adds these dependencies on top of the local ones already present in Figure 2 to construct the complete dependency graph. It is also useful here to distinguish between *inherited* and *synthesized* dependencies. *Inherited* dependencies are inter-class

dependencies that flow from parents down to their children. *Synthesized* dependencies are the reverse, flowing from children up to their parents.

3 Scheduling

Let $G = (V, E)$ be the complete dependency graph, as described in the previous section (here V is the set of vertices and E is the set of edges). Our goal is to produce a suitable *traversal schedule* from this. A traversal T in this context is simply a subgraph $T = (V_T, E_T)$, where $V_T \subseteq V$ and $E_T = \{(v_1, v_2) | (v_1, v_2) \in E \wedge v_1, v_2 \in V_T\}$, that has some special properties. A *traversal schedule* is a partition of G such that each member of the partition is a traversal.

3.1 Traversal Properties

To be a traversal, a subgraph $T = (V_T, E_T)$ must satisfy two conditions:

- **Self Contained.** For any two verticies $v_1, v_2 \in V_T$, if $v_x \in V$ is on any path from v_1 to v_2, we must have $v_x \in V_T$. The reason we need this is to keep traversals "contiguous". If dependencies were allowed to wander out of traversals and back in again, we wouldn't be able to provide strong enough guarantees about when a vertex can be computed.
- **Traversable.** Each type of traversal makes assumptions about the order in which attributes are computed. The dependencies in T must match these assumptions for at least one type of traversal. While the specifics are different for each type, these conditions generally take the form "assume some subset of verticies T can be computed; the remaining verticies of T must be computable from these alone".

The self contained property provides much insight into the space of possible traversal schedules. Note that for any two different verticies $v_1, v_2 \in S$, where S is a strongly connected component of G, v_2 is on a path from v_1 to itself. This means that v_1 and v_2, or more generally all of S, must be on the same traversal.

This suggests a novel way of discovering all possible traversal schedules. First, find all of the strongly connected components and assign them to traversals. These components now must form a DAG that is a valid *traversal schedule*. We can construct other schedules by combining connected components together. While not always possible, in some cases we can continue this until the entire schedule is one big traversal. Hence the number of possible schedules is $O(2^{|E_C|})$, where E_C is the set of edges between the strongly connected components of G.

3.2 Traversal Types

We are interested in four different types of traversals, which each move around the tree in a different manner:

- **Trivial** The trivial traversal imposes no restrictions on which order nodes of the tree may be visited in. A valid implementation strategy for this traversal is to visit all nodes in parallel.
- **Top Down** In a top down traversal, we visit a parent before visiting its children.
- **Bottom Up** Bottom up traversals are exactly the reverse of top down ones. We visit all children of a node before visitng the node itself.
- **Inorder** On an inorder traversal, each child node is visited as part of some intermediate step of the parent visit. The parent can perform an arbitrary amount of computation before or after the visit, but each child is required to be visited exactly once.

To be traversable, we must be able to show that all nodes can be visited in one of these orders. This amounts to providing a proof that all vertices on the current traversal can be "computed". Usually, a vertex can be computed if all vertices it locally depends on can also be computed, but there are some special cases.

In general, we can assume that dependencies originating from outside the current traversal are always computable; we can presumably satisfy these by scheduling other traversals before the current one. For non-local dependencies within the current traversal, computability depends on which type of traversal we are considering. Below, we consider computability rules for two traversal types.

- **Bottom Up** Assume that any synthesized dependencies are satisfied. If all verticies are computable from this alone (i.e. they have all dependencies satisfied), the traversal is bottom up.
- **Inorder** For a given class introduce a *visit clause* for each one of its children. A *visit clause* depends on all inherited dependencies to that child, and any synthesized dependencies from the child are satisfied iff the child's *visit clause* is. Assuming all inherited dependencies are satisfied, if all vertices are computable, the traversal is inorder.

These conditions provide natural implementation strategies for each type of traversal. For example, with bottom-up traversals, once all of a node's children have been visited, any synthesized dependencies to the parent must have been satisfied, so we can immediately visit the parent and compute all of its attributes.

For inorder traversals, once we compute a child's inherited dependencies, we can visit that child and compute all of its attributes. This will satisfy any synthesized dependencies, allowing the parent to continue its visit.

Figure 4 lays out explicit logical rules for verifying a traversal is a particular type using only the dependency graph.

3.3 Merging Traversals

In some cases, two traversals can be merged into one. This is almost always neccessary after assigning traversals to strongly connected components because we wind up with the largest number of traversals possible, which is likely unsuitable for realistic implementations.

We define the merge of two traversals as follows: Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be traversals. Their merge $T' = T_1 + T_2$ is such that $V_{T'} = V_1 \cup V_2$ and $E_{T'} = E_1 \cup E_2 \cup E_C$, where E_C is the set of all edges across the cut between T_1 and T_2.

However, a merge is not always well defined. It only makes sense if the input traversals have specific types. Some of these type constraints are summarized below. These can be verified with the rules in Figure 4, though the proofs are ommitted here.

- If T_1 and T_2 are trivial, then T' is trivial iff E_C contains only local edges.
- If T_1 and T_2 are trivial, then T' is bottom up iff E_C contains no inherited edges.
- If T_1 and T_2 are trivial, then T' is top down iff E_C contains no synthesized edges.
- If T_1 and T_2 are top down, then T' is top down iff E_C contains no synthesized edges.
- If T_1 and T_2 are bottom up, then T' is bottom up iff E_C contains no inherited edges.
- If T_1 is top down and T_2 is inorder, then T' is inorder iff E_C contains no synthesized edges.
- If T_1 is inorder and T_2 is bottom up, then T' is inorder iff E_C contains no synthesized edges.

```
interface Top {}
interface Node {
    var a : int;
    var b : int;
    var c : int;
    var d : int;
}

class Root : Top {
    children {
        child : Node;
    }
    actions {
        child.a := child.b;
        child.c := child.a;
    }
}

class MidNode : Node {
    children {
        child1 : Node;
        child2 : Node;
    }
    actions {
        child1.a := a;
        child2.a := a;
        b := child1.b + child2.b;
        child1.c := c;
        child2.c := child1.d;
        d := child2.d;
    }
}

class Leaf : Node {
    actions {
        b := 0;
        d := c;
    }
}
```

Figure 1: Example grammar

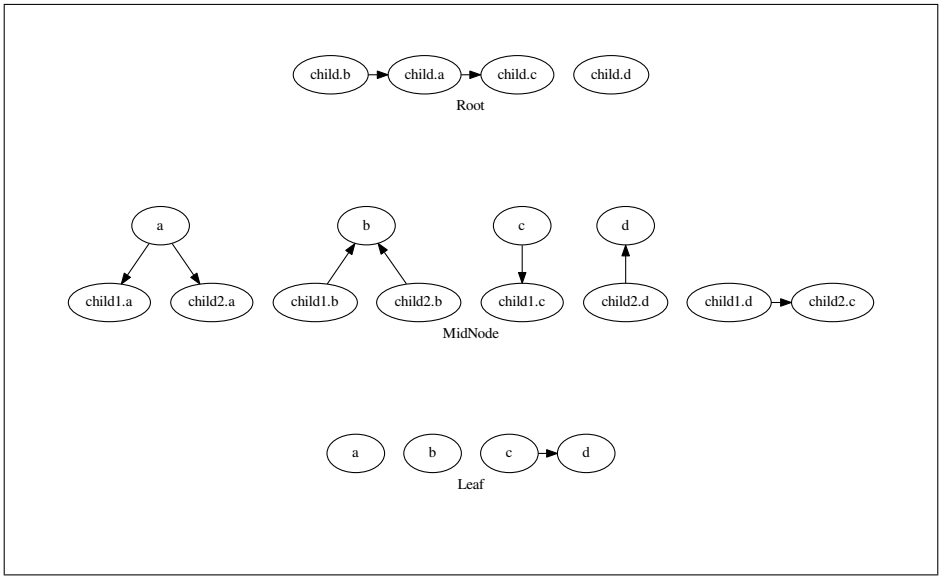

Figure 2: Local dependencies for the grammar in Figure 1

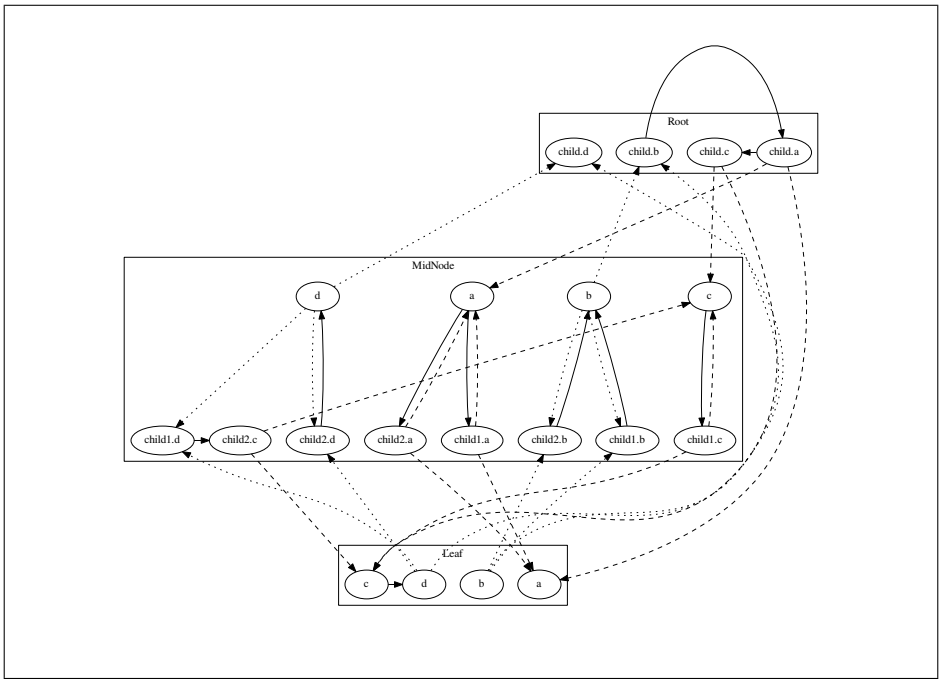

Figure 3: Complete dependency graph for the grammar in Figure 1

$$\frac{\forall e = (v_1, v_2) \in E, e \text{ is t-valid}}{v_2 \text{ is t-computable}}$$

$$\frac{e \notin E_T}{e \text{ is t-valid}}$$

$$\frac{e = (v_1, v_2) \in E_T \qquad e \text{ is local} \qquad v_1 \text{ is t-computable}}{e \text{ is t-valid}}$$

$$\frac{\forall v \in V_T, v \text{ is t-computable}}{T \text{ is trivial}}$$

$$\frac{\forall e = (v_1, v_2) \in E, e \text{ is td-valid}}{v_2 \text{ is td-computable}}$$

$$\frac{e \notin E_T}{e \text{ is td-valid}}$$

$$\frac{e \in E_T \qquad e \text{ is inherited}}{e \text{ is td-valid}}$$

$$\frac{e = (v_1, v_2) \in E_T \qquad e \text{ is local} \qquad v_1 \text{ is td-computable}}{e \text{ is td-valid}}$$

$$\frac{\forall v \in V_T, v \text{ is td-computable}}{T \text{ is top down}}$$

$$\frac{\forall e = (v_1, v_2) \in E, e \text{ is bu-valid}}{v_2 \text{ is bu-computable}}$$

$$\frac{e \notin E_T}{e \text{ is bu-valid}}$$

$$\frac{e \in E_T \qquad e \text{ is synthesized}}{e \text{ is bu-valid}}$$

$$\frac{e = (v_1, v_2) \in E_T \qquad e \text{ is local} \qquad v_1 \text{ is bu-computable}}{e \text{ is bu-valid}}$$

$$\frac{\forall v \in V_T, v \text{ is bu-computable}}{T \text{ is bottom up}}$$

$$\frac{e \in E \qquad e \notin E_T}{e \text{ is io-valid}}$$

$$\frac{e \in E_T \qquad e \text{ is inherited}}{e \text{ is io-valid}}$$

$$\frac{e = (v_1, v_2) \in E_T \qquad e \text{ is local} \qquad v_1 \text{ is io-computable}}{e \text{ is io-valid}}$$

$$\frac{\forall (v_1, v_2) \in \{e = (v_a, v_b) | v_a.class = c_1 \wedge v_b.class = c_2 \wedge e \text{ is inherited}\} \qquad v_1 \text{ is io-computable}}{visit(c_1, c_2)}$$

$$\frac{e = (v_1, v_2) \in E_T \qquad e \text{ is synthesized} \qquad visit(v_1.class, v_2.class)}{e \text{ is io-valid}}$$

$$\frac{\forall (v_1, v_2) \in E, (v_1, v_2) \text{ is io-valid}}{v_2 \text{ is io-computable}}$$

$$\frac{\forall v \in V_T, v \text{ is io-computable}}{T \text{ is inorder}}$$

Figure 4: Rules to determine traversal types from the dependency graph