Draft of March 25, 1990

An Incremental Priority-Based Solution to the Multiple Asynchronous Edit Problem

Simon M. Kaplan*
University of Illinois at Urbana-Champaign

Gail E. Kaiser†
Columbia University

Abstract

We present a solution to the problem of incrementally updating attributes in the context of multiple asynchronous edits to an attributed syntax tree. Our solution is based on using priority numbers to capture topological order of dependency information among attributes, which allows us to efficiently merge the attribute propagations originating from the various edit points. Our algorithm has worst-case time complexity $O(k \cdot (|\ \mathit{Influenced}\ | \cdot \log(|\ \mathit{Influenced}\ |)))$, where k is the number of disjoint times at which overlapping edits occur and *Influenced* is the number of attribute instances reevaluated during the attribute propagation process. Best-case complexity is $O(|\ \mathit{Influenced}\ | \cdot \log(|\ \mathit{Influenced}\ |))$, which will arise when propagations from edits do not overlap, or all edits occur simultaneously. The log factor derives from priority queue maintenance; in practice, the size of the priority queue will often be small enough to make priority queue operations effectively unit-time, reducing the complexity for synchronous replacements to $O(|\ \mathit{Influenced}\ |)$.

1 Introduction

We are concerned with the problem of incremental attribute evaluation in the context of multiple asynchronous edits. There are multiple editing cursors corresponding to separate threads of control, but they share an attributed abstract syntax tree (AAST) representation of the program. At all times, the cursors must point to the roots of disjoint subtrees, that is, one cursor cannot point to the ancestor of another; cursor movements that attempt to violate this invariant are disallowed. Otherwise, each of the editing cursors operates independently, and may move from one node to another or carry out editing operations at any time. Each edit is a subtree replacement, either the *pruning* of a subtree from, or the *grafting* of a subtree onto, the AAST, so that the syntactic constraints on the tree structure implied by the attribute grammar are not violated.

*Supported in part by the National Science Foundation under grant CCR-8809479 and by AT&T through the Illinois Software Engineering Project.

†Supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

In the classic work on incremental attribute evaluation [11, 14], there is exactly one editing cursor, and editing operations and attribute evaluation proceed in turn and may not be interleaved with each other. Instead, after every editing operation, an attribute evaluator is invoked and attribute consistency is fully reestablished before another editing operation is allowed.

Reps, Marceau and Teitelbaum [12] give an algorithm for attribute reevaluation after a set of (effectively) *simultaneous* edits. There may be one or more editing cursors, but again editing operations and attribute evaluation may not be interleaved. Instead, a sequence of edits is performed, and then attribute consistency is fully reestablished before a new round of editing operations may begin. An unpublished paper by Geitz [3] extends the RMT algorithm to asynchronous edits, but at high bookkeeping costs.

An earlier attempt to solve this problem by the authors [8] supports multiple asynchronous edits as described above. However, this solution was not optimal because it suffered from what we call the *merge problem*: Suppose there is an attribute evaluation already running and an asynchronous edit occurs. How can the knowledge and activity of the two independently initiated attribute evaluations be merged in such as way as to minimize the overall costs of maintaining attribute consistency?

In the RMT algorithm the problem of multiple points of introduced inconsistency is overcome for simultaneous edits through the mechanism of "summary edges" that would act as "placeholder dependencies" from β to α. This assumes that all the roots of subtree replacements are known prior to initiating evaluation with respect to any of them, an assumption that does not hold in the case of asynchronous edits. Further, this requires the *a priori* computation of a very large set of "summary edge templates" between all possible attribute combinations. To obtain this information, RMT use an extended version of the Kastens Ordered Attribute Grammar (OAG) analyzer [9] that computes the ordered attribute information and then uses this to compute the summary information. The Geitz algorithm extends this approach to asynchronous edits by inserting "braces" on the fly.

One of the authors has previously developed a solution to the merging problem [10] that corrects our original algorithm to minimize attribute evaluations, but incurs high bookkeeping costs. In this paper we do better by following the approach of RMT and Geitz to restrict consideration to a subclass of the noncircular attribute grammars.

The central idea in the classic algorithm, RMT, Geitz, our original algorithm, and in fact most work on incremental attribute evaluation, is that there should exist a *model*: a graph that summarizes the direct and transitive dependencies among attributes that may be reevaluated. Treating the model as a scheduling graph is intended to prevent unnecessary evaluations of attributes. Following earlier work by one of the authors on attributed graph evaluation [7], which in turn is heavily influenced by the work of Alpern *et al.* [1], we propose here a different solution: a *priority queue*. Each attribute instance is assigned a *priority number*. Attribute instance α depends on β if it has a higher priority number. We show that this scheme:

- resolves the merge problem.
- is "optimal" in a sense defined below.
- for synchronous edits, is cheaper than the RMT algorithm both in run-time and in terms of space for precomputed dependencies.

In the body of the paper we give an overview of priority numbering and then present our new algorithms, which are illustrated with a running example.

```
(1)  R -> E                    (root)
     R.res = E.val
     E.st = <>

(2)  E -> L                    (let binding)
     E.val = L.val
     L.st  = E.st

(3)  E -> int                  (integer)
     E.val = int.intval

(4)  E -> id                   (identifier:  lookup returns 0 if no binding found)
     E.val = lookup(id.idval, E.st)

(5)  E -> P                    (plus)
     E.val = P.val
     P.st  = E.st

(6)  L -> id E1 E2
     L.val = E2.val
     E2.st = <id.idval,E1.val> + L.st
     E1.st = L.st

(7)  P -> E1 E2
     P.val = E1.val + E2.val
     E1.st = P.st
     E2.st = P.st

(8)  E -> bottom               (completing production)
     E.val = 0

(9)  P -> bottom               (completing production)
     P.val = 0

(10) L -> bottom               (completing production)
     L.val = 0
```

Figure 1: Example Grammar

2 Example: Overview

To illustrate our ideas through the paper, we will employ the example given in Figure 1. This example gives a grammar for a simple "desk calculator" with `let` expressions in which the attributes compute both the symbol table of bound variable names and the result of the computation. The grammar has the advantage of being familiar to many readers and easy to understand, while at the

```
D(1) ((E.val, R.res))

D(2) ((L.val, E.val) (E.st, L.st))

D(3) ((int.intval, E.val))

D(4) ((id.idval, E.val) (E.st, E.val))

D(5) ((P.val, E.val) (E.st, P.st))

D(6) ((E2.val, L.val) (id.idval, E2.st) (L.st, E2.st) (E1.val, E2.st)
      (L.st, E1.st))

D(7) ((E1.val, P.val) (E2.val, P.val) (P.st, E1.st) (P.st, E2.st))
```

Figure 2: Direct Dependency Graphs for Example Grammar

```
P(1) ((E.val, R.res) (E.st, E.val) (E.st, R.res))

P(2) ((L.val, E.val) (E.st, L.st)  (L.st, L.val) (E.st, E.val)
      (E.st, L.val)  (L.st, E.val))

P(3) ((int.intval, E.val))

P(4) ((id.idval, E.val) (E.st, E.val))

P(5) ((P.val, E.val) (E.st, P.st) (P.st, P.val) (E.st, E.val)
      (E.st, P.val)  (P.st, E.val))

P(6) ((E2.val, L.val)   (id.idval, E2.st) (L.st, E2.st)      (E1.val, E2.st)
      (id.idval, L.val) (E1.st, E1.val)   (E2.st, L.val)     (L.st, L.val)
      (L.st, E1.st)     (E2.st, E2.val)   (id.idval, E2.val) (L.st, E1.val)
      (E1.st, E2.st)    (E1.st, E2.val)   (E1.st, L.val))

P(7) ((E1.val, P.val) (E2.val, P.val) (P.st, E1.st)  (P.st, E2.st)
      (E1.st, E1.val) (E2.st, E2.val) (E1.st, P.val) (E2.st, P.val)
      (P.st, E1.val)  (P.st, E2.val))
```

Figure 3: Induced Dependency Graphs for Example Grammar

same time being rich enough to illustrate the points we wish to make.

The grammar is ordered; we show the direct dependency graphs in Figure 2 and induced dependencies in Figure 3.

3 Some Basic Definitions and Background Theorems

Definition 1 *A cluster is an internal node of an AAST and its direct descendants.*

Theorem 2 *A cluster is an instance of a production in an AAST.*

Proof: By contradiction: if this were not true, the underlying tree would not be an abstract syntax tree.
□

We call the production of which a cluster is an instance the production of the cluster.

Theorem 3 *Each node in the tree is a member of at most two clusters.*

Proof: Any internal node is both a member of the right-hand side of some production and the root of a new production. The goal symbol of the grammar and terminal symbols are the only exceptions, and will be members of exactly one cluster.
□

We now present some theorems and definitions from the literature[1] that form a basis for our work:

Theorem 4 *For an ordered attribute grammar, all possible dependencies among attributes of a production are captured in the induced dependency graph.*

Proof: see [9, 15].
□

Consider an AAST representation of a program constructed as follows:

- Construct the syntax tree T and attach uninitialized attribute instances where appropriate.
- For each cluster, insert all the dependencies in the induced dependency graph of the production of the cluster. When dependencies for all clusters are inserted, we will have constructed $G(T)$, the complete dependency graph of the tree.

Theorem 5 *If attribute instances of T are evaluated in an order that respects a topological sort of $G(T)$, then no attribute instance will be evaluated more than once.*

Proof: see [11]
□

In the classic work, the superior and subordinate characteristic graphs are used as templates for construction and extension of the model, which always holds enough actual and summary information to be faithful to a topological order evaluation of the attribute instances. We are going to replace the graphs by priority numbers, so we have to show that we can capture the same information in this new form.

[1] In some cases wording and notation have been altered slightly.

4 Priority Numbers and Attribute Instances

We summarize dependencies among attributes using a priority numbering scheme. For this to work we require that we can always insert a new priority number between any two existing numbers. Alpern [1] describes a scheme for the generation and maintenance of such numbers such that:

- Priority numbers are represented as rational numbers.
- A new number can always be generated that fits between two existing numbers.
- Rational number operations can be performed in constant time.

Each attribute instance is assigned a priority number such that the invariant that no attribute on which the attribute depends (directly or transitively) has a lower priority number is maintained in an amortized way for efficiency. Thus priority numbers capture and summarize all the dependencies among attributes.

If we were to want to perform attribute evaluation after a single edit, we would place all the attribute instances into a priority queue. The attribute instance with lowest priority number must, by definition, be ready for evaluation. We evaluate it, and if changed, insert its descendants into the queue. This process cycles until the priority queue is empty.

For this to work we assume that all attribute instances have the correct priority number. In theory this is achievable because the Kastens algorithm yields all possible dependencies among attributes, and this information can be used to assign priorities to attribute instances as the AAST expands. If the tree only expands (by expansion of nonterminals), it is easy enough to see that the invariant described above is maintainable: The induced dependency graph for a production gives an evaluation order for the attributes of a production. This information, together with the priority numbers of the attribute instances of the goal of the production, can be used to assign priority numbers to the new attribute instances introduced during expansion of the tree by the production so that the numbers of the attribute instances of the goal are unchanged and the numbers of the new instances satisfy the constraints of the induced dependency graph.

However, editing is not just expansion of the AAST: it is quite feasible that a subtree will be pruned out of the tree and then inserted elsewhere. It could be arbitrarily large. The attribute instances in the subtree will all already have priority numbers, but it is highly unlikely that these numbers will be correct. One solution to this problem is to walk the entire subtree and renumber every attribute instance: However, this will take time $O(n)$, where n is the number of expanded nonterminals in the tree[2]. Such a result would destroy the optimality of our evaluation algorithm.

Instead, we employ an approach analogous to the way that superior and subordinate characteristic graphs are maintained in the classic algorithm. We know that after an edit:

- Attribute instances are correctly numbered everywhere on the tree but in the newly inserted subtree.
- The only instances in the subtree about which we care are those that will participate in the attribute evaluation process.

[2]The evaluation time for each production is constant-bounded because the size of the largest production is known; priority number computation, we have argued above, is constant time.

- We will immediately know if these attribute instances have incorrect numbers, because there will be a dependency (identifiable in a dependency graph for a production) for which the priority number order is inverted.

We can therefore leave the renumbering up to the attribute propagation algorithm. As part of its expansion routine, it can identify where attribute instance priority numbers are wrong and replace them on the fly. Because the amount of work needed to do this is bounded by the size of the largest production in the grammar, we can consider this a constant factor in our computation (just as in the classic algorithm computation of superior and subordinate graphs is a constant factor).

To illustrate priority numbering, consider the expression in the example language:

```
let x=5 in let x=8 in let z=4 in x+z
```

We show the tree representation of this expression in Figure 4. Note that where there is usually a dependency graph overlaying the structure; in our case this is replaced by the priority numbers, which are shown in an oblique font. In order to save space, the attributes `val` and `st` are represented in the figure (and subsequent figures) by the greek symbols ϖ and σ respectively.

4.1 Priority Templates.

We now consider how we construct the priority information from the results of the Kastens algorithm.

The first step is to associate with each production a *priority template*, constructed as follows:

Definition 6 *The* priority number template *of production p, denoted $N(p)$, is a list of attribute/number pairs such that:*

- *The attributes are the attributes of the symbols in p, and all attributes of all symbols in p are included.*
- *The numbers satisfy the definition of priority number given above.*
- *If the pairs are sorted by number the resulting order is faithful to the topological ordering of the induced dependency graph of the production.*

The priority templates for our example are shown in Figure 5.

4.2 Priority Numbering an AAST.

Second, we consider priority numbering the tree as a whole. Consider the AAST constructed as follows:

Definition 7

- *Construct the syntax tree T and attach uninitialized attribute instances where appropriate.*
- *For each cluster, insert all the dependencies in the direct dependency graph of the production of the cluster. When this process is complete, we will have constructed $D(T)$, the direct dependency graph of T.*

- *For each node, assign a priority number to each of the attribute instances associated with that node such that:*
  - *each attribute instance has a priority number greater than any of its ancestors in $D(T)$.*
  - *For each cluster in which the node is a member, a topological sort of all the attributes instances in the cluster will be faithful to the ordering specified by the priority template of the production of the cluster.*

We denote the priority number of an attribute instance α as $\mathcal{P}(\alpha)$

We can now prove the following theorem:

Theorem 8 *Given an AAST T, with complete dependency graph $G(T)$, and an AAST T' isomorphic to T with direct dependency graph $D(T')$, for which every attribute instance has been assigned a priority number as described above. A topological walk of $D(T)$ and an enumeration of attribute instances of T' in increasing priority number order will yield the same result, up to isomorphism.*[3]

Proof: By contradiction. We know that:

- For $G(T)$, all dependencies, direct and transitive, are in place on the tree.
- For T', the priority numbers are assigned based on the induced dependency graphs of the productions from which the tree was constructed, in such a way that the dependencies induced by the ordered grammar analysis are respected.

Suppose the walk (of the topological ordering) and the enumeration (of the priority numbers) did not yield the same result up to isomorphism. This would imply that either the topological ordering does not reflect the actual direct and transitive dependencies, which we know cannot be (by Theorem 4), or the priority numbering does not respect the induced dependency graph, which also cannot occur (by Definition 7). Therefore the results must be the same up to isomorphism. □

This leads naturally to the following corollaries:

Corollary 9 *Given two priority-numbered attribute instances α and β such that $\mathcal{P}(\alpha) < \mathcal{P}(\beta)$:*

- *α cannot depend on β*
- *β may depend on α*
- *Evaluating β after α cannot cause the value of α to become inconsistent.*

Corollary 10 *If all attribute instances of an AAST T are correctly prioritized by definition 7, and attribute instances are evaluated in increasing priority order, then no attribute instance will be evaluated more than once.*

[3] They may not be absolutely identical because the ordering is partial, thus the isomorphism requirement.

5 Reestablishing Priority Numbers after an Edit

The discussion above assumed a tree always has all attribute instances correctly priority-numbered. However, this need not be the case, and after an edit it may be necessary to reestablish correct priority numbering.

To see why this is the case, consider pruning out a subtree t of an AAST and grafting it in at some other point where it can legally be placed in the AAST. The attribute dependencies will all "connect up", but the priority numbers are almost certainly not going to be correct. However, we do have the correct numbers for the attribute instances of the parent of the root of the tree and all its siblings, which give the upper and lower bounds for the priority numbers of the entire tree t, and we have the priority templates for the productions which were used in constructing t and the parent of t. With these pieces of information, we can renumber t by applying first the algorithm `renumber-init` in Figure 8 and then the algorithm `renumber-all` in Figure 9 to t.

All this can be illustrated by the considering an edit to the example expression. Suppose we perform the following operations:

- Remove the subtree representing the subexpression `x+z`.
- Replace it with the expression `6+bottom`, where `bottom` means an unexpanded subtree.
- Expand the `bottom` by grafting back in the subexpression `x+z` pruned out above.

Snapshots of the tree are shown in Figure 6, which shows the tree after pruning out `x+z` and inserting the new `plus` subtree with the `6` already filled in and the tree correctly renumbered, and in Figure 7 which shows the tree after grafting back in the subtree for `x+z`. The most important thing to note is that the priority numbers for the `x+z` subtree are incorrect because the subtree has been grafted back in a different spot from that from which it was removed.

The actual operation of renumbering a single cluster can be considered to take constant time because:

- The boundary numbers within which priority numbers must be generated are known.
- Priority number operations can be seen as constant-time.
- The number of priority numbers to be generated is bounded by a constant factor, the size of the largest production in the grammar.

Thus, the running time of `renumber-all` is linear in the size of the tree to be renumbered, while the running time of `renumber-init` is constant-bounded.

Clearly, the `renumber-all` algorithm is not satisfactory for the following reasons:

- If we renumber the entire newly introduced subtree before attribute evaluation can commence, then this adds greatly to the cost of the algorithm. In the worst case the size of the tree to be renumbered can approach the size of the entire AAST.
- The amount of work we do is no longer proportional to the changes in attribute values; it it quite feasible that many if not all of the attributes renumbered will not change value during attribute evaluation; renumbering such attributes is unnecessary effort.

To get around this problem we note that

> An attribute instance's priority number is not relevant unless it needs to be evaluated during the course of attribute evaluation.

This is obviously true; if an attribute instance is not required during some particular evaluation run, then it does not matter what its priority number is. Conversely, however, if the attribute instance does not have the correct number then it is necessary for this to be corrected before it is evaluated.

We therefore outline a strategy for amortized incremental computation of priority numbers, which works as follows:

- Consider a tree T and a subtree t with root r that we are planning to graft into T.
- Attribute instances of r can be given the correct numbers in constant time by `renumber-init`.
- Everywhere else in t, attribute instances must be considered to have the wrong priority numbers.
- We mark r as "incorrect".
- Whenever we want to expand during attribute evaluation, if we try to expand onto a node marked "incorrect" we renumber the cluster rooted in that node and mark its children (if any) "incorrect'. (A node marked "incorrect" is therefore actually correctly numbered, but its children are incorrect.)

The algorithm for renumbering and marking is given in Figure 11. The cost both of renumbering a single cluster (see above) and of marking nodes is bounded by the size of the largest production in the grammar. Figure 10 shows our example after grafting in the `x+z` subtree, renumbering the root (`plus` node) and its children, and marking the children. Note that the root and children now have correct priority numbers but the grandchildren of the root of the subtree do not.

6 The Algorithm Itself

This algorithm works only if the invariant that all of T (other than t) is correctly numbered. This invariant can be maintained during cursor movement, analogous to Reps' maintenance of characteristic graphs. Whenever a cursor moves to a node that is "incorrect", it renumbers the cluster and marks the children as above during expand. The amortized cost is constant time. Because we restrict to ordered attribute grammars and distinct editing cursors must be at the roots of disjoint subtrees, it can never be the case that an will invalidate priority numbers anywhere except in the new subtree.

We can now give the actual incremental algorithm. This algorithm is split into three parts. The first, `start`, runs within the thread of control associated with the editing cursor, and the second and third, (`attribute-evaluate` and `expand`), execute within a separate evaluator thread. Remember that all these threads share a single address space that contains the AAST.

- `start` (Figure 12). Takes the root of a newly grafted subtree and renumbers the cluster rooted there, then places each attribute instance of the cluster into the priority queue. Priority queue operations are the means of synchronization between editing threads and the attribute evaluation thread, and are therefore locked and unlocked with the `lock` and `unlock` primitives.

- **attribute-evaluate** (Figure 13). Whenever the priority queue is not empty, take the lowest-numbered attribute off the queue and evaluate it. If it changes value, expand the priority queue. Repeat until the queue is empty.

- **expand** (Figure 14). Takes as argument an attribute instance. Places the direct descendants of the argument instance into the priority queue. If any such descendent is on a node marked "incorrect", it first renumbers the cluster rooted in that node.

  Note that this version of **expand** is different from the classical definition: while the classical version expands the model by cluster (the first time a cluster is "touched" by the attribute propagation, *all* the instances on that node go into the model), our expand just puts direct descendants into the priority queue. The classic algorithm had to expand by cluster in order to capture all transitive dependency information in the model. However, in our approach that information is already encoded in the priority numbers, so a smaller expansion routine can be employed. (Since model expansion is theoretically unit time in the classic algorithm, there is not a theoretical advantage gained here, but in practice a constant-bounded number of attributes will avoid reevaluation using the priority-based approach.)

6.1 Correctness and the Merge Problem.

We argue informally the correctness of the attribute evaluation process, and that it solves the merge problem.

Consider first the case of a single edit, made when the priority queue is empty and for which the resulting attribute evaluation is allowed to quiesce before any other edit is made to the AAST. Initially the priority queue contains all the attribute instances of the cluster rooted at the edit point, as all of these could have changed value. Whenever an attribute instance is found to have changed value, its descendants could also have changed, so they are added to the priority queue. Thus, all attributes that could possibly change are considered.

Consider next the case where several edits are made simultaneously. Since no attribute instance goes into the priority queue before being renumbered where necessary, the priority numbering scheme captures the dependency relationships between an attribute instance and every other instance in the AAST. The priority queue automatically orders the attribute instances so that the attribute evaluations will respect these dependencies. Thus, if there is a potential for interference between the attribute evaluations spreading out from one edit point and another, this will automatically be resolved by our scheme.

Last, consider the case where an attribute evaluation resulting from an edit is in progress and another edit occurs. In this case, the attributes introduced by the new edit will interleave correctly with the attributes already in the priority queue. This means that if the newly introduced attribute instances have priority lower than the instances already in the queue, the evaluation paths being followed will be suspended so that the lower-numbered attributes can be evaluated first. This has both positive and negative impacts:

- If there are instances in the priority queue with higher priority than newly introduced instances, this implies that there is a potential collision (a merge problem) between these newly introduced attributes and those already in the queue. Evaluation of the higher-numbered instances will be delayed, thereby preventing potential unnecessary recomputation.

- On the negative side, it is possible that there exist dependencies among instances newly introduced into the priority queue and instances which *were* in the priority queue and have already been evaluated. These will need reevaluation. There is no way to prevent this because it is not possible to compensate for or take into account events that have not yet occurred; on the other hand, any computation resulting from such evaluations will be suspended until the newly inserted instances can "catch up".

6.2 Time Complexity.

It is hard to know exactly how to compute the time complexity for our algorithm, in particular the asynchronous edits case. We have therefore produced three analyses. The first applies in the case of single edits (the classic case), the second in the case of multiple synchronous edits (the RMT case), and the third in the case of multiple asynchronous edits.

Single Subtree Replacement. The cost of the attribute evaluation process after a single edit is the number of affected attributes and the cost of maintaining the priority queue. If *Affected* is the set of changed attributes, the running cost of the algorithm ignoring the priority queue would be $O(\mid \mathit{Affected} \mid)$, i.e., no attribute is evaluated more than once. While it has been shown that under some circumstances priority queue maintenance can be reduced to $O(1)$ [2], it is more generally accepted that this takes logarithmic time, making our algorithm have worst-case behavior $O(\mid \mathit{Affected} \mid \cdot \log(\mid \mathit{Affected} \mid))$. (As with the classic algorithm, the bounded size of productions makes the bookkeeping costs part of the constant overhead.) Thus, for the single edit case our algorithm does worse than the classic result. In practice, however, the size of the priority queue is often quite small, reducing the effect of the log factor significantly.

Multiple Simultaneous Subtree Replacements. If there are multiple simultaneous edit points, then their attribute instances will be interleaved by the priority numbering scheme. An attribute instance will not be evaluated until all instances with lower priority number have been removed. Thus the multiple edit points are automatically "merged" by our approach. Again, we can allow *Affected* to be the set of attributes that change value (in analogy to the RMT algorithm, our priority numbers summarize all possible dependencies including those that in the RMT algorithm are treated through special "summary" edges). Thus our running time remains $O(\mid \mathit{Affected} \mid \cdot \log(\mid \mathit{Affected} \mid))$. This is in contrast to the running time of the RMT algorithm, which is $O((\mid \mathit{Affected} \mid + k) \cdot \log(n))$, where k is the number of edit points and n is the size of the AAST. In cases where *Affected* is much smaller than the size of the tree, our algorithm will achieve better theoretical performance. If *Affected* approaches coverage of the entire AAST, then we will still be no worse (k is usually likely to be small), while having significantly smaller space overhead because we do not need to generate and store "summary edge templates". Again, in practice the size of the priority queue seldom approaches the size of *Affected*, reducing the impact of the log factor.

Multiple Asynchronous Subtree Replacements. The time complexity of multiple asynchronous edits is much more difficult to measure. The problem does not occur if the areas affected by the edit are disjoint, or if all the instances placed in the priority queue after an edit have higher priority numbers than any attribute instance that has been evaluated since the last time

the evaluator was dormant (i.e., had an empty priority queue). In these cases, the analysis for the simultaneous edit case would apply.

However, if an edit introduces an attribute instance that has a priority number *less than* the lowest number in the queue, then the question of duplication of work arises. An attribute instance α can be evaluated and then be reevaluated later because of another edit. In the worst case it can be returned to its first value. This could cause long chains of attributes having their values first switched around one way, and then just as the evaluation quiesces, another starts up that switches them all back (consider a subtree delete followed by undo).

This renders the use of the size of *Affected* insufficient as a measure of the time complexity of the algorithm. Instead, we use *Influenced*, the set of reevaluated attribute instances. An attribute in *Influenced* could change value as many as k times if there are k overlapping sets of edits (each set being a group of simultaneous edits; we know from the analysis above that simultaneous edits introduce no additional overhead, so in the simultaneous edit case *Affected* and *Influenced* are the same set). The time complexity would then be $O(k \cdot (\mid \mathit{Influenced} \mid \cdot \log(\mid \mathit{Influenced} \mid)))$. This is however a worst-case scenario. The best-case time complexity is identical to the synchronous case and would occur when there is just one set of edits. Again, the effect of the log factor in practice is likely to be less than implied here as the size of the priority queue seldom approaches the entire size of *Influenced*, and will often be small enough that the log factor approaches 1.

We argue that this is an "optimal" result because no attribute instance is ever evaluated more than once unless a collusion of the temporal ordering of edits and the speed of the attribute evaluator cause an attribute to be evaluated *before* an edit that would have delayed the evaluation. For the result to be truly optimal (without quotes) we would need to prove that the log factor occasioned by the priority queue is necessary; this is an open question. However, we do know that the priority queue maintenance will often be a very small factor, reducing the algorithm to linear-time for synchronous edits, with the multiplicative k factor for asynchronous groups of edits.

6.3 Example Update

To illustrate how the algorithm works, consider the following scenario:

- Make the edit as shown in the figures.
- Start attribute propagation.
- While attribute propagation is proceeding, replaced the nested `x` by a `y` in the expression, which will change the entire result of the expression.

When the edit is made, the priority queue will be:

$$< P.st, 70.1 >< E.st, 70.11 >< E.val, 70.12 >< E.st, 70.2 >< E.val, 70.4 >< P.val, 71.9 >$$

Now suppose we evaluate *P.st* and the right-hand *E.st* (number 70.11). Before we proceed any further, the `x` bound to 8 is replaced by a `y`. This will cause the attribute instances at the root of the change to be entered into the priority queue, which will then be:

$$< id.idval, 12 >< E.val, 70.12 >< E.st, 70.2 >< E.val, 70.4 >< P.val, 71.9 >$$

The id will be reevaluated, causing the symbol-table propagated into the `let` expression to be altered, causing the *P.st* and *E.st* attributes to be eventually recomputed. But all other computation for the insertion of the `x+z` subtree will be halted until the symbol table propagation catches up, because all inserted attribute instances until that time will have lower priority number. In other words, we have successfully merged the attribute updates from the two edits.

7 Applications

Our results are important to support the collaborative effort of teams of programmers. Consider a scenario where each programmer is using an editor that checks syntax and static semantics incrementally by applying attribute evaluation after every editing operation (e.g., Synthesizer Generator [13]). When one programmer changes some code, many other programmers may potentially be affected (say, if the first programmer changes the interface to a module that is imported by other modules). Programmers work at their own pace, which means the program representations in the editors are being modified asynchronously. In order for changes made by one programmer to affect the results of incremental analysis for another programmer, the editors and program representations must be coordinated in some fashion. In the bulk of this paper, we have assumed a single editor with multiple editing threads of control, one evaluation thread, and one shared AAST.

To make this paradigm practical, it is necessary to segment the AAST into multiple subtrees, where a particular programmer can edit only within one such segment and there is only one editing cursor per segment. Each programmer has his own process, with his own editing thread, his own evaluation thread, and a private address space containing his portion of the global AAST. An attribute propagation layer would coordinate the separate user processes to ensure global attribute consistency. Such as system, called Mercury [6], has been implemented using our original algorithms. Mercury also includes special algorithms [5] for handling the reliability and availability problems where segments may reside on different machines, and machines and network links may fail and return to availability.

Another application of incremental attribute evaluation for multiple asynchronous edits is implementation of the "triggers" in database systems. The gist is that certain derived data depends on other base data, and must be rederived whenever the base data changes; there may be a chain of transitive dependencies among derived data items. A "trigger" notices an update to a data item and rederives any data that depends on it. In a multi-user database system, there are necessarily multiple asynchronous modifications to base data. If a database is modeled by an AAST, database updates by subtree replacements and dependencies among data items by an attribute grammar, the incremental attribute evaluation algorithms are directly applicable. In practice, attributed graph grammars (which generate graphs) would replace attribute grammars (which generate trees), and edits, consequent attribute evaluations and the goal of merging to avoid unnecessary evaluations would have to be reconciled with transactions. Some of these ideas have been explored in the Cactis system [4].

References

[1] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and Kenneth Zadeck. *Incremental Evaluation of Attributed Graphs.* Technical Report CS-87-29, Brown University, Department

of Computer Science, December 1987.

[2] P. F. Deitz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth ACM Symposium on Theory of Computing*, pages 365–372, May 1987.

[3] Bob Geitz. Asynchronous subtree replacement for language-based editors. 1987. Oberlin College and Cornell University.

[4] Scott E. Hudson and Roger King. Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[5] Gail E. Kaiser and Simon M. Kaplan. Reliability in distributed programming environments. In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45–55, Kingsmill–Williamsburg VA, March 1987.

[6] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 58–67, November 1987.

[7] Simon M. Kaplan and Steven K. Goering. Priority controlled incremental attribute evaluation in attributed graph grammars. In *Proceedings TAPSOFT '89*, Barcelona, Spain, March 1989.

[8] Simon M. Kaplan and Gail E. Kaiser. Incremental attribute evaluation in distributed language-based environments. In *Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 121–130, Calgary Alberta, Canada, August 1986.

[9] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[10] Josephine Micallef and Gail E. Kaiser. *Support Algorithms for Incremental Attribute Evaluation of Multiple Asynchronous Subtree Replacements (Extended Abstract).* Technical Report CUCS-450-89, Columbia University Department of Computer Science, November 1989. Submitted for publication.

[11] Thomas Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, MA, 1984.

[12] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, St Petersburg FL, January 1986.

[13] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 41–48, Pittsburgh PA, April 1984. Special issue of *SIGPlan Notices*, 19(5), May 1984.

[14] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[15] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

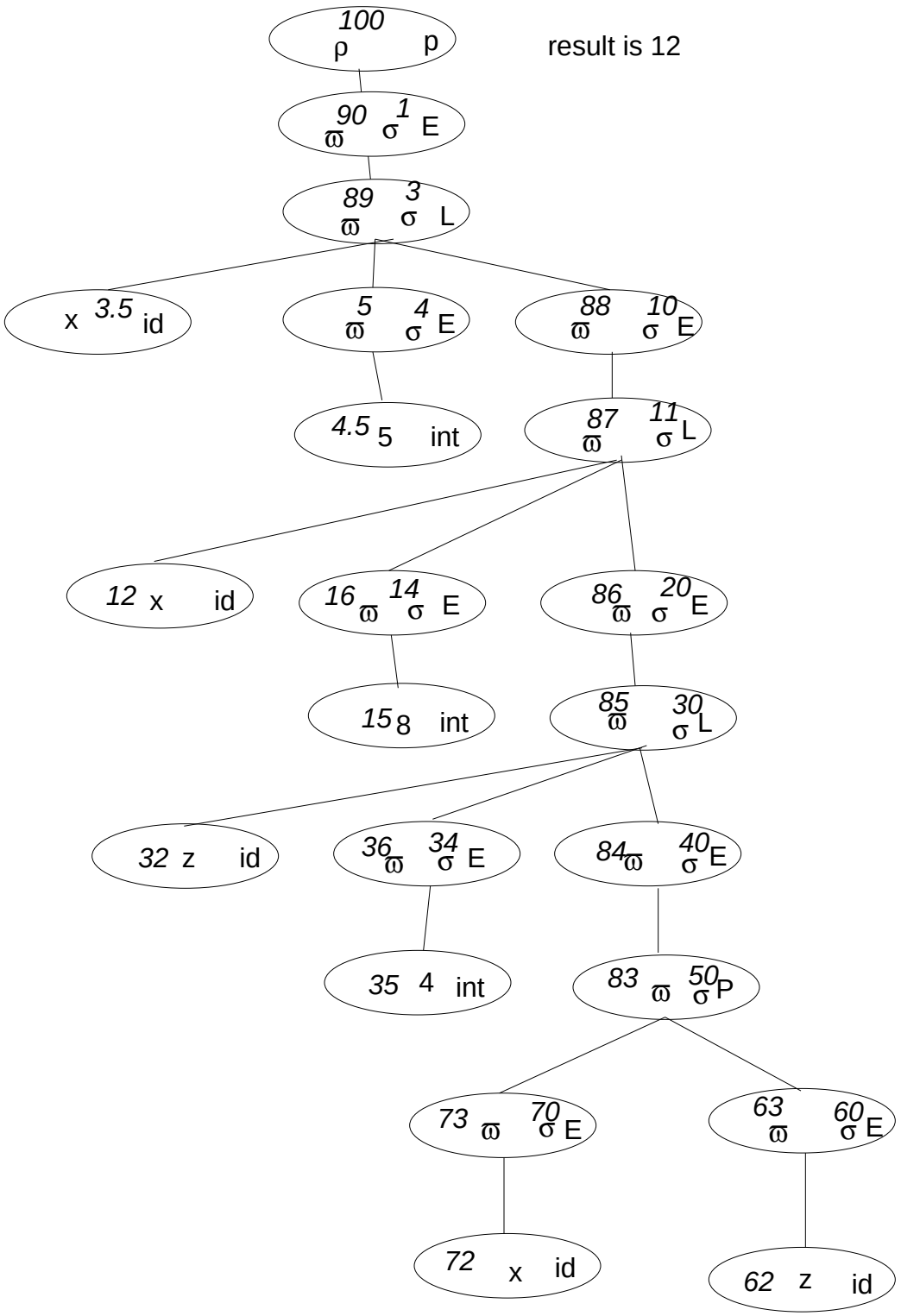

Figure 4: Priority-Numbered View of Example Expression

(1) (E.st, 10) (E.val, 20) (P.res, 30)

(2) (E.st, 10) (L.st, 20) (E.val, 30) (L.val, 40)

(3) (int.intval, 10) (E.val, 20)

(4) (E.st, 10) (id.idval, 20) (E.val, 30)

(5) (E.st, 10) (P.st, 20) (P.val, 30) (E.val, 40)

(6) (id.idval, 10) (L.st, 15) (E_1.st, 18) (E_1.val, 30) (E_2.st, 35)
(E_2.val, 40) (L.val, 80)

(7) (P.st, 10) (E_2.st, 20) (E_1.st, 30) (E_1.val, 40) (E_2.val, 60)
(P.val, 60)

Figure 5: Priority Templates for Induced Dependency Graphs

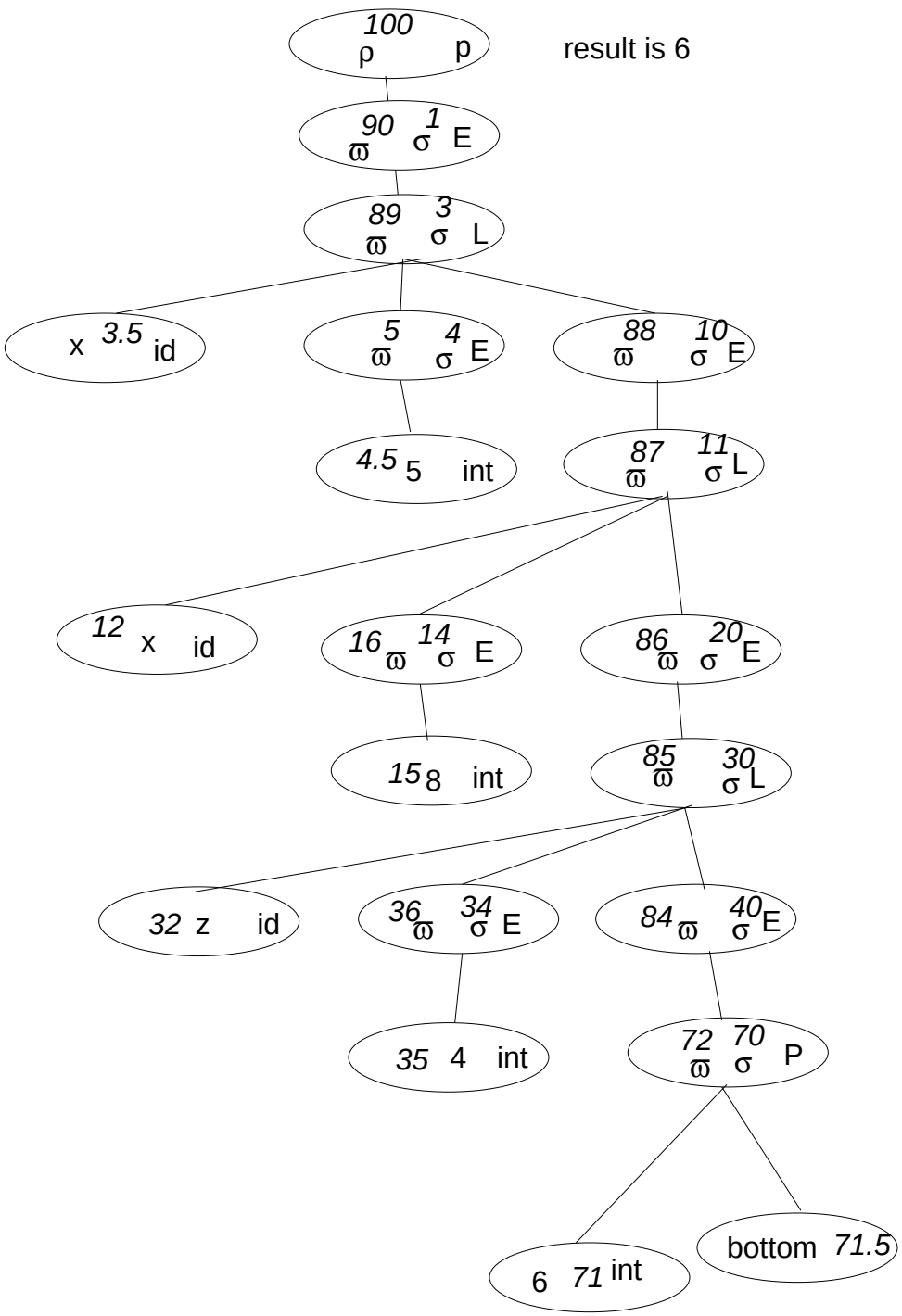

Figure 6: Snapshot of Tree Edit: 1

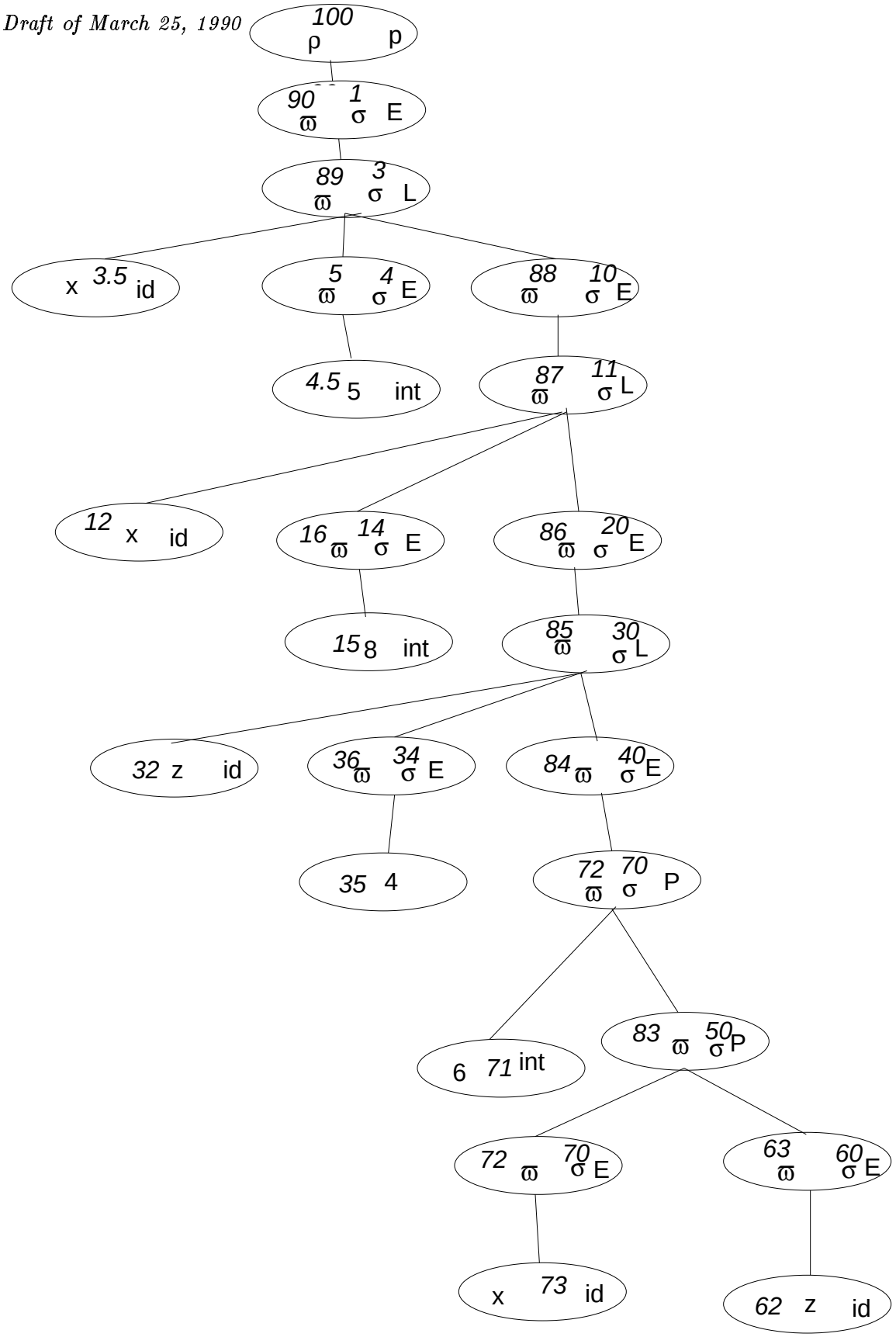

Figure 7: Snapshot of Tree Edit: 2

```
renumber-init(t)
Precondition:  true
Postcondition: all attribute instances of t are correctly numbered.
let
   t      = a node that is the root of a tree
   c(n)   = cluster rooted at node n
   p(c)   = production used to construct cluster c
   P(p)   = priority template for production p
   aic(c) = attribute instances of cluster c
   ain(n) = attribute instances of node n
   par(n) = parent of node n
in
   renumber each element of ain(t)  such that
     (a) the priority numbers of ain(par(t)) and all children of par(t)
         except t are unchanged and
     (b) the priority numbers of each element of ain(t) respect the
         ordering implied by P(p(c(par(t))))
   renumber;
ni
```

Figure 8: Renumber Initialization

```
renumber-all (t)
Precondition:  attribute instances of t are correctly numbered.
Postcondition: all attribute instances of tree rooted in t
               are correctly numbered.
let
   t      = a node that is the root of a tree
   c(n)   = cluster rooted at node n
   p(c)   = production used to construct cluster c
   P(p)   = priority template for production p
   aic(c) = attribute instances of cluster c
   ain(n) = attribute instances of node n
   par(n) = parent of node n
in
   renumber each element of aic(c(t)) such that
     (a) the priority numbers of ain(r(t)) are unchanged and
     (b) the numbers of each attribute instance in aic(c(t))
         respect the ordering implied by P(p(c(t))).
   renumber;
   foreach n such that n is a child of r(t) and n is not a leaf
     do
      renumber-all (n)
     od
ni
```

Figure 9: Renumber Entire Tree

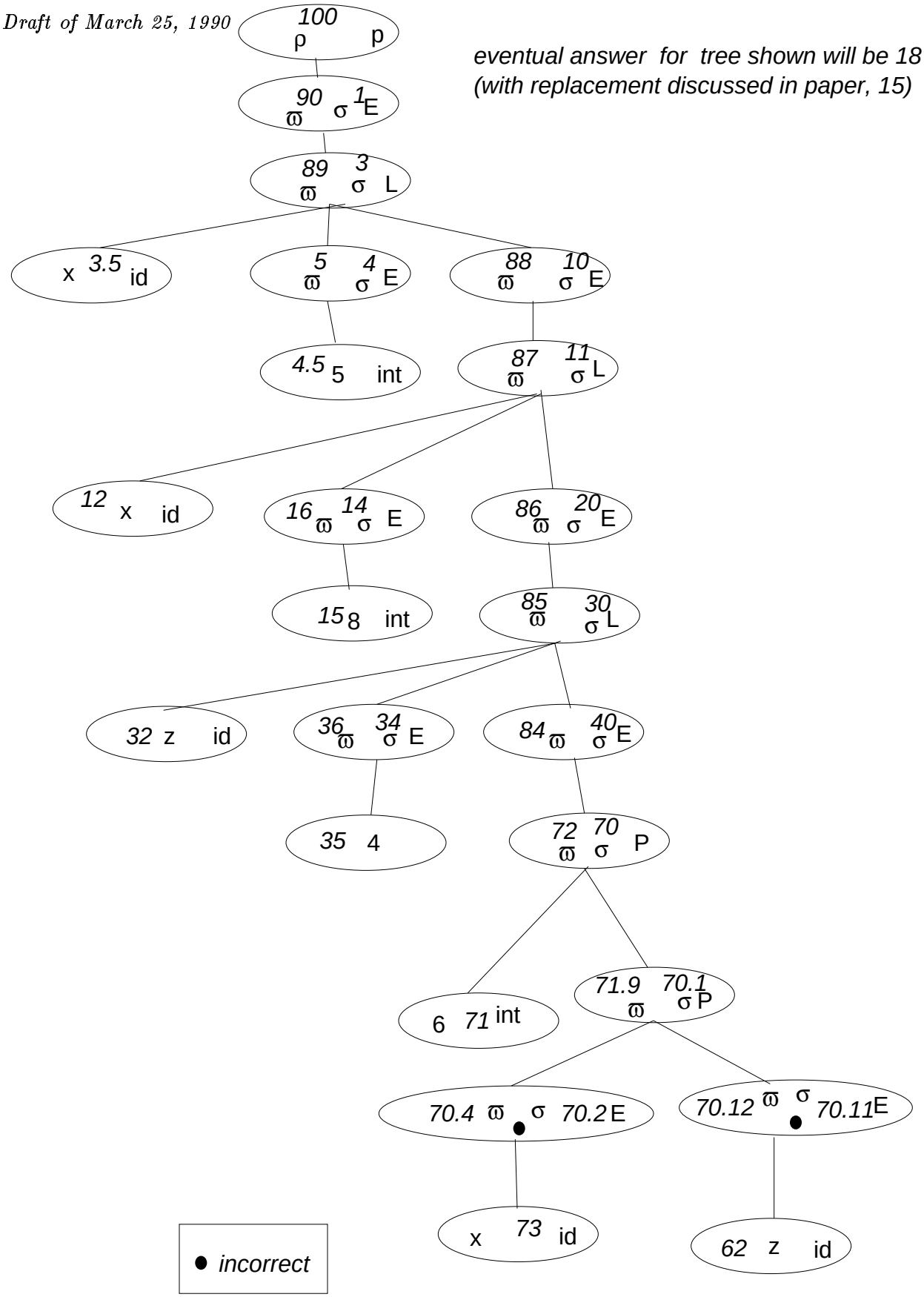

Figure 10: Example After Renumbering First Cluster

```
renumber-amortized(n)
let
  n      = a node
  m      = a node
  c(n)   = cluster rooted in n
  p(c)   = production from which cluster c was constructed
  P(p)   = priority template for production p
  d(n)   = children of node n
  aic(c) = attribute instances of cluster c
  ain(n) = attribute instances of node n
in
   renumber each element of aic(c(n)) such that
     (a) the priority numbers of ain(n) are unchanged and
     (b) the numbers of each attribute instance in aic(c(n))
         respect the ordering implied by P(p(c(n))).
   renumber;
   mark n ``correct'';
   for each m, m a child of n and m not a leaf node, do
     mark m ``incorrect''
   od
ni
```

Figure 11: Amortized Renumbering

```
start(t)
let
  t      = point of subtree replacement --- root of new subtree
  r(t)   = root of tree t
  c(n)   = cluster of node n
  a      = attribute instance
  aic(c) = attribute instances of cluster c
  priq   = priority queue, global variable
in
  renumber-init(r(t))
  renumber-amortized(r(t))
  lock(priq)
  foreach a in aic(c(r(t))) do
    insert-into-priq(a)
  od
  unlock(priq)
ni
```

Figure 12: Start from a Change Point

```
attribute-evaluate()
let
  priq    = priority queue, global variable
  a       = attribute instance
in
  forever do
   if priq-empty then sleep-until-priq-has-entries fi
   lock(priq)
   a = first(priq)
   priq = rest(priq)
   unlock(priq)
   evaluate a
   if changed(a)
     then
       expand(a)
   fi
  od
ni
```

Figure 13: Incremental Attribute Evaluator

```
expand(a)
let
 priq    = priority queue, global variable
 D(T)    = AAST
 a, b    = attribute instances
 node(x) = node associated with attribute instance x
in
  lock(priq)
  foreach b such that (a,b) in D(T) and b not already in priq do
    insert-into-priq(b)
    if marked(node(b))
      then
       renumber-amortized(node(b))
    fi
  od
  unlock(priq)
ni
```

Figure 14: Expand