The Schutz Data Structure

Rodney Bates

1. Introduction

The primary data structure in Schutz, which represents code being edited, is called an EST, for *Extended Syntax Tree*. Imbedded within an EST is an AST, abstractly viewable as a standard *abstract syntax tree*, with some significant representational differences.

For each language, there is a single flat space of token codes, of several kinds. The lowest numbered are common to all languages and have special purposes. These are declared in LbdStd.i3. For each langauge, there are several categories of codes. *constant terminals*.

In a usual ast, a leaf node corresponds to a lexical token that, for parsing purposes, is a terminal in the grammar of the language, but has additional information attached, for uses beyond parsing. Identifiera and string literals are typical examples. In Schutz, these are called VarTerms, for *variable terminal* . These are represented by a node containing the token code and a pointer to atom of the character string.

In a conventional ast, each significant syntactic construct, e.g., an if-statement or a subrange-type, is represented by an interior node that contains an ast token code denoting which particular construct it represents, and child subtrees that represent the components it is constructed from. The children are, in general, subtrees of unbounded size and complexity. Most such children are constrained to be a subtree rooted by a particular subset of the ast tokens. Each ast token implies a possibly different list of children and their kinds, and the ast node is often statically typed with this list.

A Schutz est is similar. Each interior est node has an ast token, and some of its leaf nodes are VarTerms. But there are possibly other children, besides ests and VarTerms. These are called mods. for *modifier*. They represent nonsyntactic content such as comments, error messages, etc. Moreover, the children are stored in a runtime variable list.

Displaying content texually entails a variation on *paramodulation*. There is a language-dependent set of rewrite rules, one for each ast token. Each rule gives a list of what, in a paramodulation system, are called *variables* that match the ast children of the ast node. Interspersed among these are other items that are to be inserted into the textual equivant of the est node. Most commonly, ond of these is a terminal symbol of fixed spelling, called a constterm, for *constant terminal*. A constterm correspond to a delimiter token, such as "IF", or ";", that brackets and separates the textual representations of the subtrees.

In the theory, paramodulation means rewriting the tree by systematic application of such rules. In Schutz, the thus-rewritten tree is never physically constructed, but remains conceptual. Several est traverse algorithms alternate between traversing an est node and traversing its rewrite rule, performing appropriate actions on components of each.

In Schutz, the rewrite rules of a language are orginally specified textually, in ldl1, by the language definer. The rules are called fsrules, for *format syntax* rules. Ldl processing of these converts each to a linked tree data structure, which is loaded into RAM as part of the supporting data for a language. This is called an fstree, *format syntax* tree. The fstree for an ast node is located directly by the node's token code. Nodes of an fstree that call for inserting a constterm are called *insertion tokens* and are leaf nodes.

There are several additional complications possible in a format syntax rule/tree. First, along the leaf nodes, there may also be *line breaks*. These specify that a new line is to be begun where the line break occurs. The inserted new line is indented by a dynamically computed amount inherited from above, plus a static amount attached to the line break.

There are three conditions controlling whether a line break is actually taken. All line breaks in a range of textual material have the same condition. A group of *vertical* line breaks are always inserted. A group of *horizontal* line break is inserted only if the entire range of textual material they belong to fits on the current line. Otherwise, none of them is taken. a *fill* line break, independently of the others in its range, is taken if the material from it to the following (fill) line break or the end of the range fits on the line. Thus sections of material delimited by fill line breaks are treated the way words are filled by a work processor.

The relevant range of textual material, in the simple case, is the entire fsrule. Alternatively, the rule can be broken into subsections, either nested or disjoint, each of which has a different condition on its line breaks.

An fsrule/fstree can also have subsections that are conditionally used in their entirity. These are controlled by any of an extensive set of predicates on properies of both the children and the Est node. For example, this can be used to selectivly insert required parentheses around a subexpression of lower precedence than the est node itself.

The leaf nodes of an fsrule are numbered sequentially, left-to-right. These are called fsnos, for *format syntax numbers*. They are used to match fstree leaf nodes to est leaf nodes. Each est leaf also has a fsno stored it it. This will be equal to the fsno of the fsrule leaf that corresponds to it.

In addition to ast children, all of the modifier children of the est, have a fsno too. This attaches the modifier to an ast child. Each kind of modifier is statically either *leading* or *trailing*, the former being more common. Every leaf of an fstree matches a range, possibly empty, of est children all with the same fsno as the fstree leaf. This will include zero or more leading modifiers, an optional est child, and zero or more trailing modifiers.

The est child can occur only for a VarTerm, but even then is optional. It could be optional in the abstract grammer of the language, e.g. an initial value expression of a variable declaration. It can also be removed in various cases involving textual editing, syntax error repair suggestions, etc.

Insertion tokens are backwards in the sense that they are absent from the est, but implicitly part of its textual material. This saves a lot of memory, perticularly since many of them will be a single character. So when one needs to be removed, it is done, perversely, by inserting a *delete modifier* This is a leading modifier containing the fsno of the insertion token. It specifies that the token is not to be inserted. It can also apply to a line break. Unlike any other kind of modifier, it also gives an ending fmtno, so one modifier can delete multiple fsnos.

In Schutz terminlogy, a *newline* causes textual display to start on the next line. But unlike a carriage return, multiple newlines, concatentated with nothing visible in between, only cause moving to the next line once. One might say concatenation of newlines in idempotent. A newline can be inserted by a line break, or by certain modifiers.

A *blank line mod* is a leading modifier that specifies insertion of two or more new lines, but there is implicitly one or more lines of white but nonempty space between then, so they are not concatenated in the way described above. It is the only way to get blank lines to display.

There are four kinds of comment modifiers, a two-by-two cartesian product. They can be fixed, which are not moved horizontally, or movable, which are, tracking motion of syntactic material that results from editing, parsing, relayout, etc. They can be leading or trailing. Trailing comment modifiers are the only trailing modifiers at all. These are comments that occur to the right of all syntactic material on the line. In general, a comment modifier can have a newline at its beginning and/or its end.

When the user types new or altered text, which has yet to be reparsed, I *text modifier* gets inserted. All text altered or touched by a contiguous group of edits is removed from the est in the ways described above. Then any alterted text is inserted as a text modifier. It resembles a comment modifier, but will be rescanned and reparsed upon request from the user. It can have a newline at its beginning and/or its end.

A *token modifier* is used so as not to lose user-typed text that the parser suggests deleting, as part of its syntax repair. The token can have a large complement of leading and trailing modifiers as well as its own text, if a VarTerm. A token modifier is actually a mini-est with one root and children like a single fsno would have.

There are additional modifier kinds for a few kinds of messages and text not recognized by the scanner.

The children of an est node are kept in an array of *leaf elements*, which have additional fields beside the obvious pointer to the child node. One is the fsno of the child. This is a property that is meaningful only in the context of the parent node, so belongs here. Another is a cumulative count of nodes in the child subtree and all child subtrees to its right. The subscripts of the children increase right-to-left, and the cumulative node counts also reflect this. This ordering complements reductions during LR parsing. The meaning of this depends partly on the parent node too. Finally, there is a set of property bits, used to locate desired nodes in O(logN) time.

Most of the manipulation of ests operates at a level of abstraction where in interior node appears to have a flat list with a variable number of children. However, there are many places this would require linear-time search through the children, which in some cases, will be longish. Instead, at a lower level, a single est node and its children are actually represented by a *K tree*. After the initial development of the K tree data structure, it was pointed out that the term K tree was in use with another meaning, so they were renamed *Sequence trees* in later literature. But "KTree" was already in extensive use in the the code of Schutz, so has been retained there.

A K tree is a multi-level tree of nodes with variable child count, ranging up to a small static maximum, here eight. All paths to leaves are the same length. It represents a sequence of anything, with no search keys. Elements are identified only by their position in the sequence. It allows cutting out a subsequence or slice and concatenation of sequences in logarighmic time in the sequence length. It works functionally, i.e., with immutable and shared nodes, thus requiring a garbage collector.

In pure form, slicing, concatenating, and a few other operations are implemented as independent operations. However, in Schutz, there is a set of composite operatoins for building a K tree by concatenating, right-to-left, multiple slices of other K trees. Moreover, during such a build, a portion of the partially constructed tree is maintained in temporary, mutable data structures. This avoids a potentially large amount of allocation, copying, and garbage production in achieving the desired result.

The physical data structure integrates est trees and K trees. There is a cartesian product of nodes that are leaf/nonleaf K tree nodes, and est parent/est children.

Since a K tree node has a limited out degree, its children can be searched efficiently. Even then, a binary search can be performed on its children. Thus a search through an entire est can be done in logarithmic time.

Aside from systematic traversal of an entire est, there are two methods of locating a desired node in an est. One is for locating a node with some interesting property. Each est node a set of *kind bits*. This is a fixed set of bits, each denoting an interesting node property. Each est node holds bits for all properties found anywhere in its subtree. Thus finding the next/previous leaf node with a desired property, it is possible to descend thereto, by looking for a child containing the propery, in logarithmic time. Complete kind bit sets are maintained when building ests.

The other locating mechanism is through node numbering. Conceptially, all the nodes are numbered in a preorder, i.e., parent node first, then its child subtrees, in left-to-right order. Physically each interior est node contains only the count of the nodes in the subtree it roots, including itself. One one hand, this means there a node can be shared among mutiple ests, since it does not contain inherited information. On the other hand, it is possible to do a top-down traversal, developing full node numbers along the way.

With this any explicit node can be located by its global node number. Implicit items, i.e., the insertion tokens and line breaks, can be identified by number of a nearby node, plus the fsno of the implicit item. In the case of a new line given by a modifier that can contain one before and one after, an additional boolean will suffice.

The global node number mechanism is used to denote beginnings of text lines in an est. When there are multiple new lines concatenated, the rightmost is always the one identified in this way. For a blank line modifier that denotes more than one blank line, a line number withih the modifier is additionally used.

Interior points within a line, such as the typein cursor, selection limits, bookmarks, etc. are identified in the same way, except a character position within the item is also required.

During reparsing, another mechanism is required. Fragments of the est are being moved around independently, and global node numbers will not be meaningful. Prior to reparsing, a mark on an est leaf node is converted to a pointer directly to the node, node kind bits are inserted into the est that lead to each such pointed-to leaf node. These are called tempmarks, for *temporary marks*.

During rescanning, tree traversal, and reparsing there is a rather complex scheme of patching this kind of mark to follow the material it denotes. Finally, when the reparse is done, they are converted back to the the global node number kind of marks, using the kind bits to locate them. This is a fragile mechanism, but seems necessary to keep track of marked points during reparsing.

There is special treatment of est nodes that represent lists of varying numbers of syntactic children, all of the same class, e.g., a list of declarations or formal parameters. The usual way of representing these uses a left- or right-recursive production, which has the consequence that the list portion of the ast amounts to a linear linked list of the elements. In Schutz, this is undesirable for two reasons. First, it undermines the logarithmic time operations. This is particularly important, because it is mainly through ever longer lists of statements that edited programs become large.

Secondly, when doing incremental reparsing out in the middle of a list, this would require a lot of allocation and copying of otherwise unchanged nodes, just to alter their effective link pointer.

Instead, the est node for a list has all the children, i.e., the list elements, flattened into direct children of the est node. If the list is long, the K tree mechanism inside will balance the children and maintain logarithmic operations, regardless of which children are changed and which left alone. Moreover, it also allows the parsing grammer to use an ambiguous left-and-right recursive production, so the parser can leave untouched children alone, again regardless of their location in the list. Building the est child list during a reduction flattens the right had side of the production, so the ambiguity doesn’t matter.

To make the format syntax numbering and conditional formtting work in a list, the fsrule/fstree has exactly one leaf that is an est child, for the list element. This can occur somewhere in the middle of the fs children. The numbering starts with zero for this, continues righttward, wraps around to the left, then rightward to the list element. Traversing the fstree wraps around too, visiting all the fs children for each ast child.

Sometimes a list element needs to be omitted from the corresponding est but some insertion tokens and/or line breaks remain. Since the fsnos of est children keep going around, this can create ambiguities in the fsno matchup between the fstree an the est children. When this is possible, an explicit but dummy ast child is inserted into the est, to mark the place of the omitted child. It generates no syntactic or textual content.

Decisions about whether to take a horizontal line break are made without traversing, with the aid of *width info*, a collection of information about the space occupied by the text of an est node. This is complicated by the fact that, in addition to a simple character count, it can involve new lines at the beginning or end of the text, new lines properly inside, which are different, and the presence of material with a absolute start position, which some comments have. Width info distils needed information, independent of start position, and in a way that can be synthesized, bottom-up.

For leaf est nodes, it is computed when needed, directly from the node. For interior est nodes, it is computable from the width info of the children, and is stored in the node when it is created. This entails a concatenation operation on width info values, along with insertion of requires white space between children. A start position and a width info value can be used later to ascertain how far right node's text will extend.

For a fill line break, some traversal is unavoidable, but although it is a bit complex algorithmically, it need not traverse very far, so it is not too computationally demanding.