Direct manipulation of abstract syntax trees

Theodore S. Norvell
Dept. Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University
theo@mun.ca

***Abstract*— Programming languages and document description languages, such as HTML or LaTeX, require the user to enter and manipulate abstract syntax trees. Usually this is done through editing text files using tools that are generally oblivious to the syntax of the language. This can be awkward, frustrating, and unproductive. As a part of developing an editor for the PLAAY programming language, I have developed general techniques and a robust library for directly manipulating abstract syntax trees via a combination of keyboard and mouse actions. This ensures that the tree being edited is at all times valid or can be extended to a valid tree. One pleasant surprise was that entering the tree in the first place can take about as many keystrokes as entering the tree using a text editor.**

***Index Terms*— Trees, Graphical user interfaces, Structured Text, Programming languages.**

I. Introduction

An *abstract syntax* can be thought of as a set of node-labelled ordered trees. For example, in HTML, the nodes are either text nodes or elements.[1] Each text node is labelled with a sequence of characters; each element node is labelled with a tag (which is a string) and a sequence of attibute/value pairs. Element nodes may have a sequence of child nodes, so the entire document is arranged as an ordered tree. HTML has numerous rules that restrict the tree: the root of the tree must be an element tagged with 'html'; this node must have exactly 2 children and they much be elements tagged with 'head' and 'body' in that order; 'ul' tagged elements may contain 'li' tagged elements, whereas 'body' tagged elements may not; text nodes must have no children; and so on. In HTML and XML terminology, trees that respect all these constraints are termed *valid* trees. In this paper I will borrow that term to refer to trees in the abstract syntax as opposed to trees that violate some rule of the abstract syntax.

In many applications, people need to create and revise documents that conform to a specific abstract syntax. Examples include computer programming languages, data formats such as JSON and YAML, markup languages such as LaTeX, HTML, numerous applications of XML such as DocBook, and the internal representations in word processors such as Microsoft Word or Scientific Workplace.

In this paper I present a methodology for creating interactive editors for such structured languages. These editors maintain the validity of the document being edited. Much of this methodology is captured in a library written in Typescript [1] and thus can be directly reused in other structure editors. I'll consider both the user experience and the software design aspects. Specific examples and experience will be drawn from the implementation of the PLAAY programming language — a visual programming language currently being implemented at Memorial University.

[1]This is a bit of an over simplification. For example, I am ignoring comment nodes.

II. Trees and representations of trees

A. Mathematical trees

Suppose we have a set L. A *labelled, finite, ordered tree* (henceforth *tree*) over L consists of a member of L called its label and a finite sequence of shorter trees called its children. The height of a tree is one if the tree has no children, and otherwise is one more than the height of its tallest child. For any tree t, we write $t.\mathrm{label}$ for its label and $t.\mathrm{children}$ for its sequence of children.

Suppose for each label ℓ there is a boolean function $\ell.\mathrm{isValid}$ that takes a sequence of trees. A tree t is *valid* iff all its children are valid and $t.\mathrm{label}.\mathrm{valid}(t.\mathrm{children})$. An *abstract syntax* can be defined as the set containing all valid trees whose label is a particular label called the *root label*.

Our aim is to ensure that document being edited is always valid. The editor starts with a valid tree. The basic UI cycle is as (a) The tree is rendered in some visual form. (In the PLAAY system we do this by translating it to HTML.) (b) The system waits for an event such a key press or mouse action. (c) The editor attempts to compute a new valid tree. (d) If the attempt is successful, the tree is updated with the new tree; back to (a). If the attempt is unsuccessful, back to (b).

B. Representation of trees

In software, we represent labels as objects that realize an generic interface called DLabel.

```
interface DLabel
    <L extends DLabel<L,T>,
     T extends DNode<L,T> > {
        isValid : (children:Array<T>)
                    => boolean ;
    ... }
```

The isValid method determines whether a valid tree can be built using this label and the given children.

Valid trees are represented by objects of class DNode.

```
abstract class DNode
    <L extends DLabel<L,T>,
     T extends DNode<L,T> > {

        protected _label:L;
        protected _children:Array<T>;
        ...
}
```

In the implementation of the PLAAY programming language we instantiate and extend these types with types PNode and PLabel.

```
interface PLabel
    extends DLabel<PLabel,PNode> { ... }
class PNode
    extends DNode<PLabel,PNode> { ... }
```

Henceforth, I'll speak of PNodes and PLabels, mostly to save space and to allow concrete examples.[2] Most of what I say about them will apply equally to any subtypes of DLabel and DNode.

Objects of type PNode and PLabel are immutable, meaning that the values of their fields can not change. Making these objects immutable helps make the software robust and also allows node and label objects to be shared. *To use immutable types is the first important design decision.*

The constructor for PNode checks validity and throws an exception if (a representation of a) a valid tree can not be built.

```
constructor( label:PLabel,
             children:Array<PNode>) {
  assert.check( label.isValid(children),
      "Attempted to...");
  this._label = label;
  this._children = children.slice(); }
```

Thus only valid trees can be built and, because of immutability, representations can never change to represent invalid trees. Validity is an invariant of class PNode. *To ensure validity on construction is the second important design decision.*

However, throwing exceptions is not the road to robust software. Programmers have a habit of ignoring that exceptions may be thrown, assuming the exception will be caught at a higher level, or assuming that exceptions have been caught as a lower level. Thus, when a tree needs to be made, we call the function

```
function tryMake( label:PLabel,
                  children:Array<PNode> )
    : Option<PNode> { ... }
```

This function returns an object of type Option<PNode>. There are two kinds of option objects: Some<PNode> objects wrap a PNode; None<PNode> objects contain no extra information. We can think of option objects as being lists of length 0 or 1. Of course the tryMake function returns a Some<PNode> object if the validity check is passed and a None<PNode> object otherwise. The use of tryMake forces the client programmer to either deal with the possibility of failure or to reflect that possibility in the signatures of their methods. Objects of type Option support the monad operations of map and bind [7]. Throughout our system, code that deals with options generally does so using these operations. *To use the option type consistently is the third important design decision.*

[2]In some cases this will lead me to tell white lies, for example the constructor code shown for PNode is in fact in DNode and is generic. The actual constructor for PNode simply calls the generic constructor of the super class.

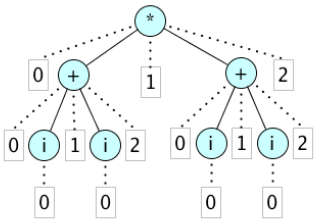

Fig. 1. Positions in a tree

Fig. 2. Display of selections

C. Selections

Because of sharing, several tree nodes may be represented by one PNode. To identify a tree node, we use a root PNode and a list of numbers: the empty list [] represents the root, [0] represents the first child of the root; the list [4, 2] represents the third child of the fifth child of the root; and so on. Each tree node with n children is associated with $n + 1$ *positions* numbered 0, 1, ... , n. The first n are to the left of the corresponding child; the last is to the right of the last child. In the case of a node with no children, its sole position is below it. Fig 1 shows a tree and its positions.

A *selection* consists of a PNode called its *root*, a *path* identifying a tree node, and two positions under that node, called the *anchor* and *focus*. When the anchor and focus are equal, the selection identifies a position in the tree and we say the selection is empty. When the anchor and focus are not equal, we say that the nodes between the two positions are the selected nodes.[3] Note that the root can not be selected. In the PLAAY system, empty selections are

[3]In most cases it doesn't matter whether the anchor or the focus is bigger. Usually the anchor is the smaller. But in some cases they may be the other way around.

presented to the user by displaying a small grey rectangle at the selected position; nonempty selections are shown by giving the selected nodes grey backgrounds. Fig. 2 shows how the PLAAY system displays selections of sizes 0, 1, and 2. Selections are represented in the software by immutable objects of class Selection.

A nice benefit of immutability is that undo and redo in the PLAAY editor is trivially implemented by having two stacks of selections in addition to the current selection.

III. Modifying Trees

A. Abstract Edits

An edit is a partial function with the same source and target type.

```
interface Edit<A> {
    applyEdit : (a:A) => Option<A> ;
    canApply : (a:A) => boolean ; }
```

We require canApply(a) iff not applyEdit(a).isEmpty(); that is, the canApply method simply indicates whether the edit will succeed. Edits can be composed sequentially — compose(x,y) creates an edit that first applies x and then y to the result— or alternatively — alt ([x,y]) applies the first edit that will succeed. This allows us to build complex edits from simple elements. We have a domain specific language (DSL) of edits that uses compose and alt as its compound operators. Alt and compose almost form a semiring except that alt is not commutative and we don't have right distributivity, i.e.,

$$\mathsf{compose}(\mathsf{alt}([x, y], z) = \mathsf{alt}([\mathsf{compose}(x, z), \mathsf{compose}(y, z)])$$

is **not** true in general. Interestingly the lack of these laws makes the DSL more expressive.

For the purposes of this paper, the only Edit objects that we use are Edit<Selection> objects.

B. Concrete Edits

We define several realizations of Edit<Selection>.

One of the most useful is the InsertChildrenEdit. This edit is constructed from an array of trees. The edit replaces the selected children (if any) with the sequence of trees in the array. These edits can be used to delete nodes (if the sequence is empty), insert nodes (if the selection is empty) or replace one or more nodes with one or more trees (if neither is empty). This edit will fail if the resulting tree would be invalid.

Consider the implementation of the delete key. In the PLAAY language deleting a node may lead to an invalid tree. For example, nodes representing 'while' expressions must have exactly 2 children; the first must be an expression node of some sort; the second must be an expression sequence node. If the user selects the first node and presses the delete key, the node can not simply be deleted. Instead we would want to replace it with an expression place-holder node, which is a node that indicates to the user an expression at that spot is required. The implementation of the delete key is thus implemented by creating a choice of InsertChildrenEdit edits, each created with a different array of trees to replace

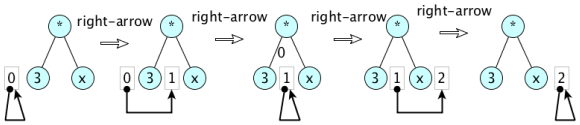

Fig. 3. Right arrow

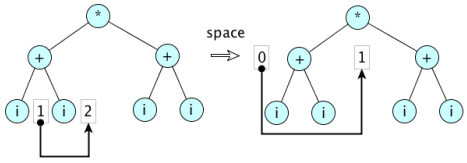

Fig. 4. Space bar

with. Given an array choices of arrays of trees, the require edit is computed by

```
alt( choices.map( (choice) =>
    new InsertChildrenEdit( choice ) ) )
```

Two closely related edits are the swap edit and the move edit. The swap edit exchanges the nodes between two selections. The move edit moves nodes from one selection to another, while replacing the source nodes with nodes from an array, similar to delete. Both these edits require that the two selections share the same root and create a new tree that differs from the old in two places. The code to do this is quite involved, but is shared by the swap and move edits. Note that we can not compose these edits out of two InsertChildrenEdits edits because the intermediate tree may be invalid. Swap and move are used together with the paste edit, which leaves the source selection alone, are used in implementing drag and drop operation between nodes of the tree. Such a drag and drop action could be interpreted as a paste, a move, or a swap. The PLAAY editor applies all three edits: if only one succeeds, that one is applied. If more than one succeeds, one is chosen, but a pop-up menu appears, giving the user a chance to pick a different interpretation. Immutability makes this sort of thing easy.

A number of edits only 'change' the path, anchor, and focus, not the rootFor example the various arrow keys are implemented by such edits. Fig. 3 shows the effect of the right arrow. The positions under the 3 and x nodes are skipped because no nodes can be inserted there. Two edits that will

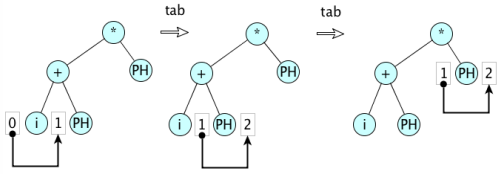

Fig. 5. Tabbing

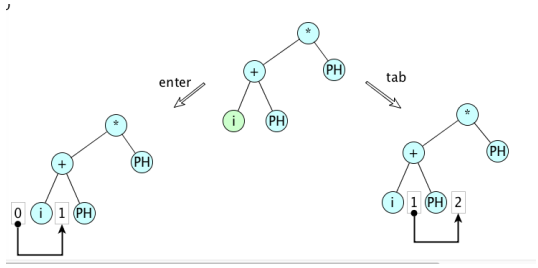

Fig. 6. Closing a label

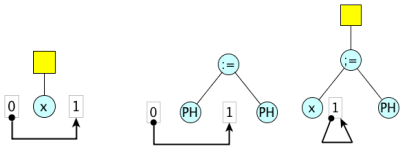

Fig. 7. Engulf

Keystrokes	Edits	Display
+	replace	
*	replace	
a	replace	
tab	change label, tab	
x	replace	
tab	change label, tab	
tab	tab	
*,b,tab,y,tab		

TABLE I

PREFIX ENTRY:

be important in the next section are the OutEdit and the TabForwardEdit. The first selects the parent of the selection if possible; in PLAAY this edit is bound to the space bar; see Fig. 4. The second searches to the right checking selections of size one or zero until either a place-holder node is selected or a position where a node can be inserted is selected; positions that are next to place holders are skipped. Fig. 5 shows a sequence of TabForwardEdits being applied. If the second child of the + node were not a place-holder, the first tab would advance the anchor to 1 and the second would move advance both the anchor and focus to 2; this is because + nodes can have more than 2 children.

Some edits only affect the labels of the selected nodes. For example there are edits to change the string associated with a label. Since labels are immutable, this means a new label is created and the tree is modified to use the new label. Some labels are capable of being in an open state. In PLAAY this true of labels representing identifiers and numbers. Open labels are displayed as text fields that can be edited. When the user has finished editing the field (indicated by an enter key or a tab key) the label is changed to a closed label with a possibly new string. Fig. 6 shows what happens when a label is closed by an enter versus a tab key in the PLAAY editor.

A very useful edit is called an *engulf* edit. An engulf edit is based on a selection called a template t. Applied to a selection s, the selection edit works in four steps. First, the selected nodes of s are used to replace the selected nodes in t to form a selection t'. Next, the selected nodes of s are replaced with the root of t'. Finally, the path, anchor and focus are adjusted so that the selection is empty and corresponds to the position to the right of the nodes that moved. Fig. 7 shows an example. From left to right we have the orginal selection, the template and the result. Engulf is usually followed by an implicit tab if need to make a sensible selection. In the example in the figure, since assignment (:=) nodes can only have two children, the selection would be moved forward to select the place-holder.

Engulf is often alternated with a *replace* edit, which is simply an InsertChildrenEdit using the template as the source of new nodes composed with a TabForwardEdit edit. A replace-or-engulf edit gives a choice of either an engulf or a replace. The replace is preferred when the selection is a place-holder. Otherwise engulf is preferred. All these edits can be extended to use a choice of templates. Replace-or-engulf edits are used in the PLAAY editor mostly for two purposes: Key press events are bound to replace-or-engulf edits. For example, the ':' key is bound to a replace-or-engulf edit that uses the template shown in the middle of Fig. 7. Secondly, palette items are bound to replace-or-engulf edits. Palette items may be dragged and dropped on the tree or clicked on, which means they are applied to the current selection.

IV. EFFICIENT ENTRY OF TREES

In this section, I'll focus on the entry of trees. We will see that entering code in PLAAY takes about as many keystrokes as entering equivalent code using a programmer's text editor.

The binding of actions to keys and palette items is designed to facilitate either prefix or infix entry of expressions. Prefix entry means that the tree is entered top down. Infix entry means that the first child of a node is entered before the node. Infix entry is only possible when the parent and the first child share the same syntactic role, e.g., if both are expressions or both are types.

Table I shows prefix entry of a simple expression, assuming the selection is initially empty. This takes 12 keystrokes as opposed to 7 in C. Most of the overhead comes from the need to explicitly terminate identifiers. With longer identifiers, the ratio would be closer to one. It should be noted that in PLAAY, the expression corresponding to $(a + x) * (b + y)$ would also require 12 keystrokes, since the abstract syntax of PLAAY does not include parentheses.

Keystrokes	Edits	Display
a	replace	
enter	change-label	
*	engulf	
x	replace	
enter	change-label	
space	out	
+	engulf	
b, enter, *, y, tab		

TABLE II

INFIX ENTRY:

Table II shows infix entry of the same expression. Again 12 keystrokes are needed. The use of enter, rather than tab to terminate entry into the text field leaves the variable selected. The space-bar is used to select the enclosing expression.

Expressions can also be entered by dragging and dropping palette items or by clicking on them; this would be suitable for tablet computers lacking keyboards or for beginner programmers not familiar with the keyboard short cuts. The editor currently has two palettes: one for expressions including 19 items, and one for types, including 15 items. On a tablet, the example expression can be entered with 7 drag-and-drop actions and 7 entries on a (virtual) keyboard. Entering by clicking (or tapping) on the palette items is similar, but will require some extra taps to set the selection. With both the drag-and-drop and the click-on-palette-item methods, the tree can be built in either a prefix or infix order.

This particular expression has no repeated identifiers, but it should be noted that, once an identifier has been entered, we can use drag-and-drop to copy it to other places in the tree when it is needed.

As a longer example, the factorial function definition shown in Fig. 8 takes 44 keystrokes to enter. A roughly comparable function definition takes 45 keystrokes in C or Java with no unneeded whitespace:

```
1 int fact(int n){return n==0?1:n*fact(n-1);}
```

V. Related Work

Structure editors go back to the 1970s. Early work on generating structure editors based on abstract syntaxes include the MENTOR system [3], the GANDALF project, with its ALOE editor system [4] the Synthesizer Generator [5], and the Centaur system [2]. Visual languages have always demanded some form of structural editing. The Scratch language is one recent example [6].

In the area of document editing, structure editing is used in WYSIWYG (what you see is what you get) and WYSIWYM (what you see is what you mean) editors.

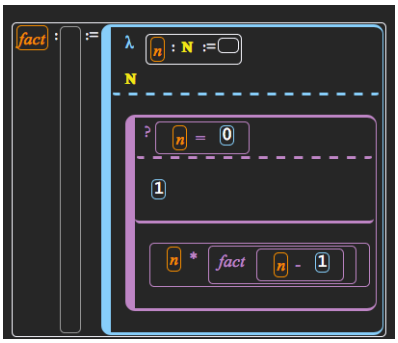

Fig. 8. Factorial function

VI. Conclusion

As can be seen from the previous section, structure editors are hardly new. The contributions of this paper, then are in the mechanics of programming such an editor, in some of the user interface considerations, and in how the two interact. More generally it serves as an example of achieving robust, flexible, and modular code through immutable structures and functional programming concepts such as the option monad and an embedded domain specific language.

Future work will include using the system described for other languages.

Acknowledgements

I'd like to thank the following students who have worked on the PLAAY system: Lawrence Bouzane, Dillon Butt, Jillian Hancock, Jessica Hillier, Jason Howell, Cem Kilic, Chris Martin, William Newhook, Chris Rodgers, Christopher Stanbridge, Ajay Vijayakumar, and Kamrule Hasan. Sunil Jaganathan deserves special mention for using the DNode system to implement another visual language.

References

[1] Gavin Bierman, Martí Abadi, and Masd Torgerson. Understanding typescript. In *ECOOP 2014—Object-Oriented Programming*, 2014.

[2] P. Borras, D. Clément, Th. Despeyroux, J Incerpi, G. Khan, B Lang, and V. Pascual. Centaur: The system. In *3rd Annual Symposium on Software Development Environments (SIGSOFT'88)*. ACM, 1988.

[3] Véronique Donzeau-Gouge, Gérard Huet, Bernard Lang, and Gilles Kahn. Programming environments based on structured editors: the mentor experience. Rapports de Recherche 26, INRIA, July 1980.

[4] A. Nico Habbermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[5] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer-Verlag, Berlin, Heidelberg, 1989.

[6] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, November 2009.

[7] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.