Direct Manipulation of Abstract Syntax Trees

Presented at NECEC 2018, St. John's, NL

Theodore S. Norvell

Computer Engineering Research Labs, Dept. ECE, MUN

2018 November 13

This talk is about two things

- Keeping software *simple* and *robust*.
  - **Small details matter!!**
- Making the user experience *pleasant* and *efficient*.
  - **Small details matter!!**

There is nothing difficult or advanced here.
And that's the point.
A few key decisions make the bulk of the work straightforward.
And the software robust and reusable.

Trees are everywhere.

I think that I shall never see,
a structure lovely as a tree.
With root aloft and leaves aground,
for nature's trees are upside-down.

- Programs are trees
- File systems are trees
- Documents are trees
  - HTML
  - XML

How do we make a good user experience for entering and editing trees?

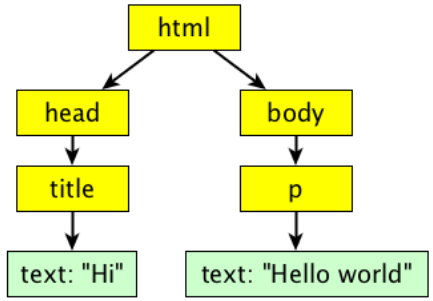
The context

- I've been working on a visual programming language: PLAAY.
  - *Low threshold:* Should be easy for beginners to use.
  - *High ceiling:* Should be useful to professionals.
- In PLAAY syntax errors can not be made because:
  - The user edits the program by directly manipulating the abstract syntax tree.
  - The editor only permits valid abstract syntax trees.
  - This is good for beginners *and* professionals.

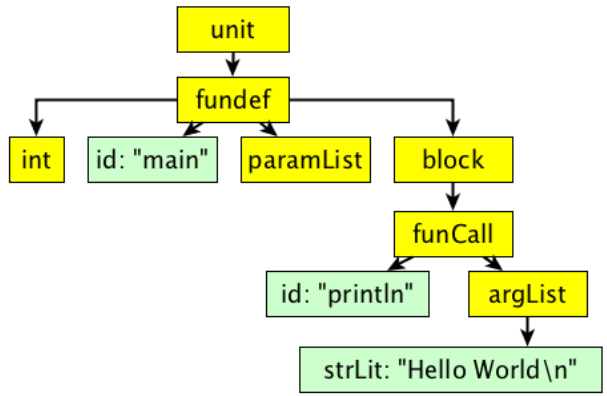
Trees

- A tree is consists of
  - A label.
    - Typically consists of a tag or kind possibly some other information such as a string or a boolean
  - A sequence of 0 or more shorter trees.

```
<html><head><title>Hi</title></head><
body><p>Hello world</p>
```



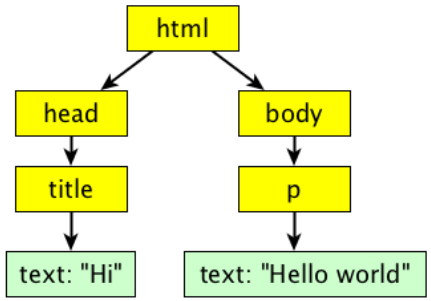

```
int main() { println("Hello world\n") ; }
```

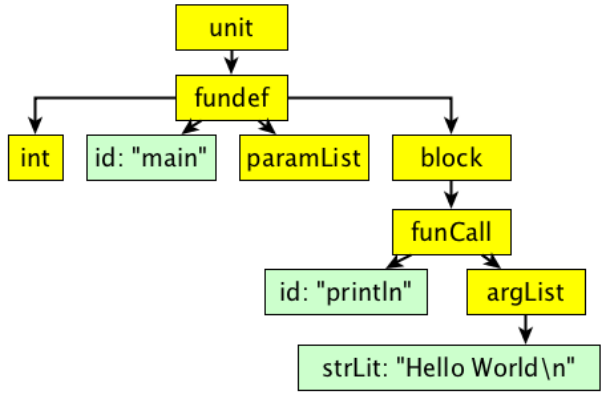


Valid Trees

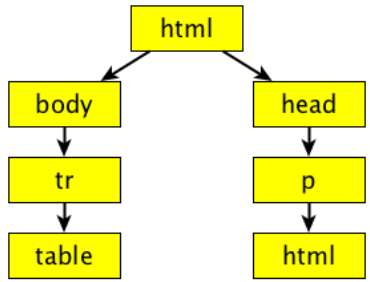
Not all trees are valid.

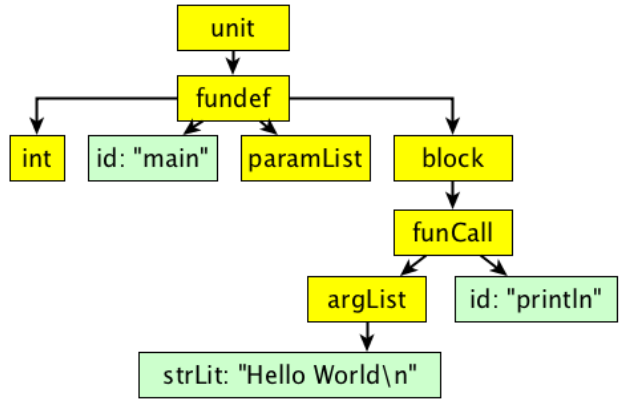
- Valid trees





- Invalid trees





Trees that are valid for a given language we say are within the *abstract syntax* of the language.

Editing Valid Trees

- User interface cycle:
  1. t := some valid tree
  2. display t to the user
  3. wait for a UI event
  4. attempt to make a new valid tree
  5. if successful: t := the new tree ; go to 2
  6. else: go to 3
- *Validity is a loop invariant!*

Representing trees

Three key decisions

- Make it impossible to build an invalid tree.
- Make it impossible for tree to become invalid.
- *Validity is a data invariant!*
- Use Options to deal with failure

Representing trees

The classes

All labels must have a boolean method that determines whether or not it can be used with a given list of children.

```
interface PLabel {
    public isValid(children : Array<PNode>) : boolean;
    ...  }
```

An example `PLabel` class:

```
class WhileLabel implements PLabel {
  public isValid(children : Array<PNode>) : boolean {
    return children.length === 2 &&
      children[0].isExprNode() &&
      children[1].isExprSeqNode() ; } ... }
```

Representing trees

The classes

There is one class of nodes

```
class PNode {
  protected _label: PLabel ;
  protected _children:Array<PNode>;
  ... }
```

Representing trees

The truth

- Actually the `PLabel` and `PNode` classes are specializations of generic classes called `DLabel<L,N>` and `DNode<L,N>`.
- This allows us to reuse much of the code for other applications:
  - HTML editors
  - LaTeX editors
  - Generic XML editors
- For the sake of this presentation, I'll ignore the genericity. See the paper for details.

Representing trees

First key decision

It is impossible to make an invalid tree.

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

The first line of the constructor will throw an exception if we try.

Representing trees

Second key decision

It is impossible for an existing tree to become invalid.

- This is enforced by making the PNode class *immutable*.
- And all classes that implement PLabel must be *immutable*.

Consequences

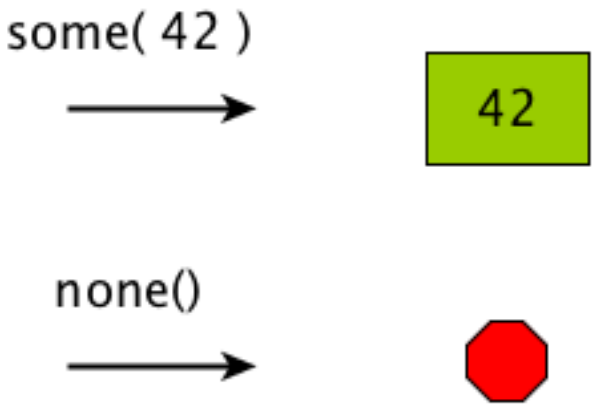
- Immutability permits sharing. Trees are represented by DAGs.
- Changes are functions from trees to trees
- In short we use functional programming w.r.t. trees.

Digression on the Option type

Type `Option<number>` represents numbers that might be absent. There are two kinds of option objects.

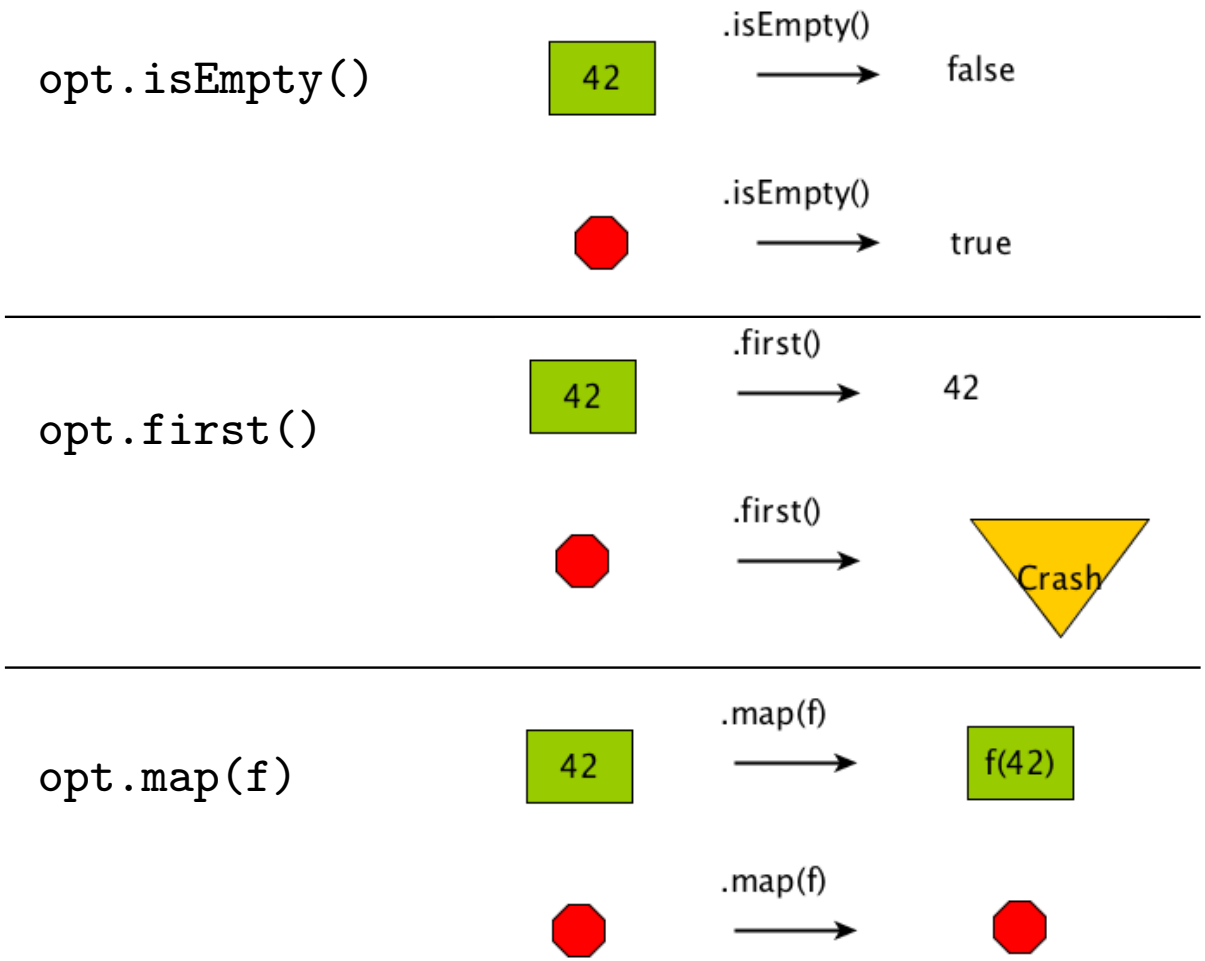
- `some<number>(42)` – The value 42, wrapped in a box
- `none<number>()` – Is a lump of coal. I.e. no number.

Think of an `Option<T>` object as a collection of 1 or 0 objects of type `T`.



Digression on the Option type

Operations on `Option<T>` objects.



Representing trees

Third key decision

Use Options

- Throwing exceptions from constructors does not promote robustness unless we either
  - *ensure they are always caught and handled* or
  - *ensure they are never thrown*
- We do the latter.

Rather than directly calling `PNode`'s constructor we call `tryMake`

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

which returns `none()` if the desired tree would be invalid.

The caller of `tryMake` must either deal with either outcome or must return an option.
For example:

```
const opt = tryMake(label, children);
return opt.map( newNode =>
                    new Selection( newNode, ... ) );
```

where `none().map(f) = none` and `some(x).map(f) = some( f(x))`. So this code returns an `Option<Selection>`. It passes the buck.

Another example:

```
const opt = paste( dragged, dropTarget ) ;
opt.map(sel => updateDisplay( sel ) );
```

Here the code does nothing if the option is empty. Otherwise it updates the display.
It is not impossible to screw up with options, for example I could have written

```
const opt = paste( dragged, dropTarget ) ;
update( sel.first() );
```

Recall `none().first()` crashes.
Failing to catch an exception is a *sin of omission*.
It is easy to make and hard to spot.
But using `first` instead of `map` is a *sin of commission*.
Any use of `first()` should set off alarm bells for the design reviewer.

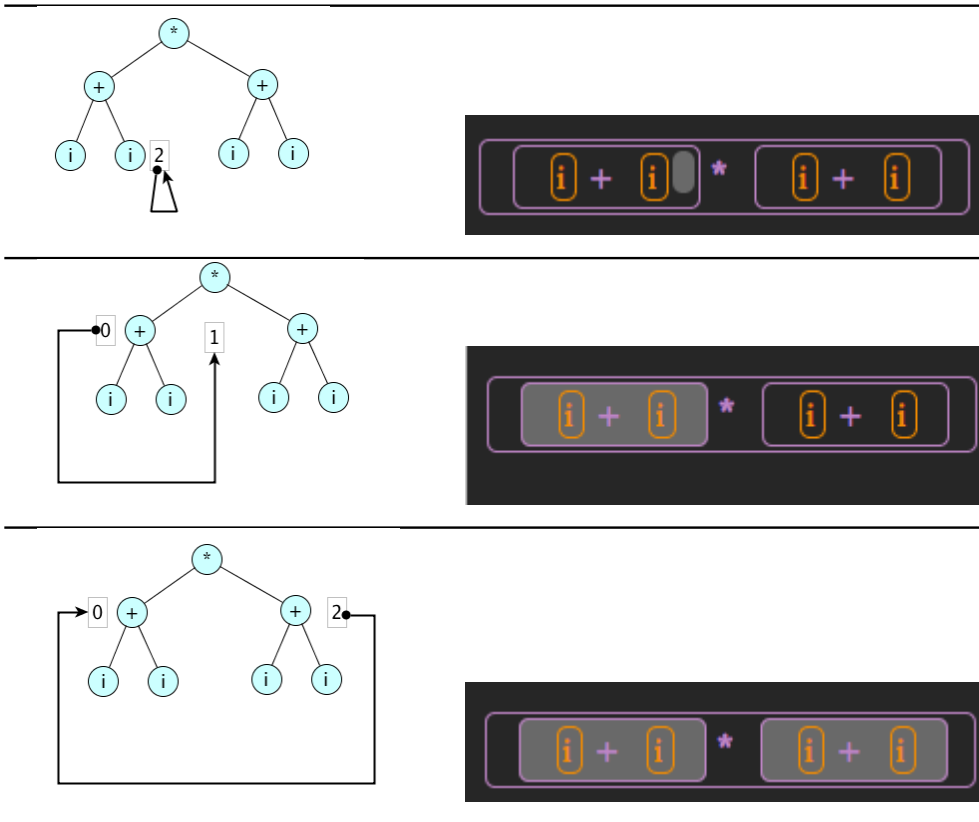
Selections

Fourth key decision: Use a simple selection model.

A selection consists of

- A PNode called the *root*
- A *path* in the tree
- A number called the *anchor*
- A number called the *focus*

Selections are *immutable*



Edits

Fifth key decision: Use a "little language" of edits.

An `Edit` maps each `Selection` to an `Option<Selection>`

- An `Edit` object represents a partial function

Composition of edits: If

$$h = f \circ g$$

then

$$\begin{aligned} h(x) &= f(x) \text{ if } f(x) \text{ is empty} \\ h(x) &= g(f(x).\texttt{first}()) \text{ otherwise} \end{aligned}$$

Biased choice of edits: If

$$h = f \oplus g$$

then

$$\begin{aligned} h(x) &= f(x) \text{ if } f(x) \text{ is not empty} \\ h(x) &= g(x) \text{ otherwise} \end{aligned}$$

Edits

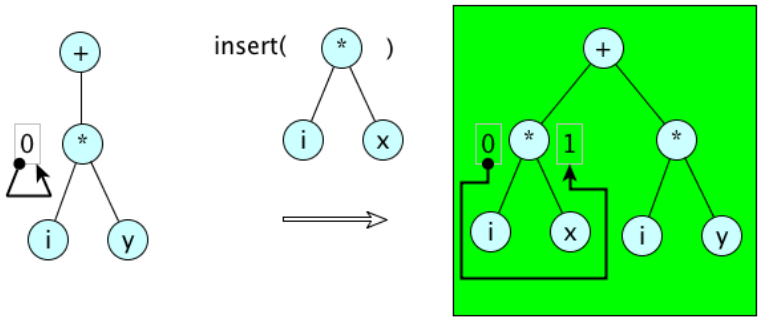
Assuming termination and no-side effects, edits form a simple algebra (two laws short of a semiring)

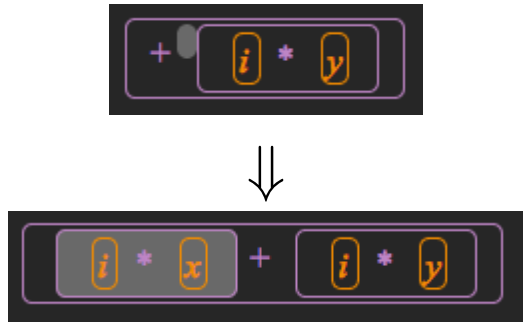
$$
\begin{aligned}
(f \oplus g) \oplus h &= f \oplus (g \oplus h) \\
0 \oplus f &= f \\
f \oplus 0 &= f \\
(f \circ g) \circ h &= f \circ (g \circ h) \\
1 \circ f &= f \\
f \circ 1 &= f \\
f \circ (g \oplus h) &= (f \circ g) \oplus (f \circ h) \\
0 \circ f &= 0 \\
f \circ 0 &= 0
\end{aligned}
$$

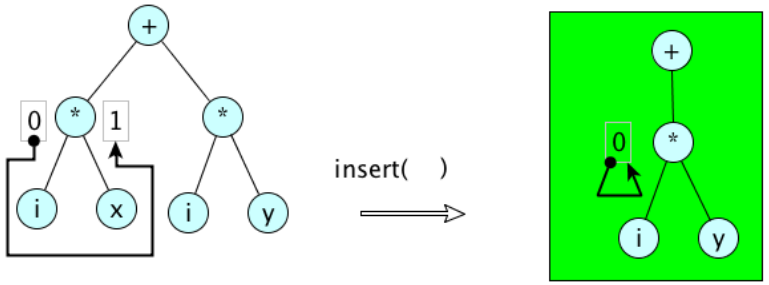
Where 1 is the identity edit and 0 is the edit that always fails.

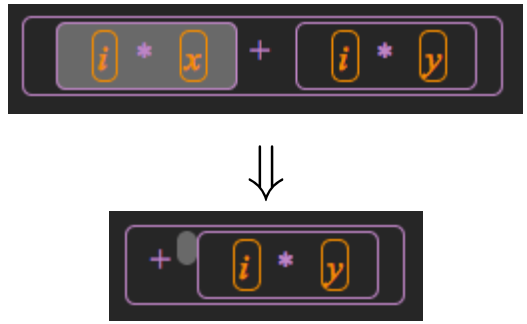
Insert edit

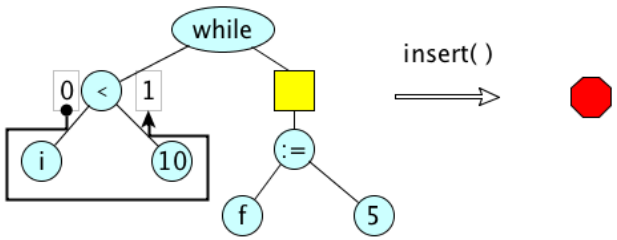
— replaces children with a new sequence

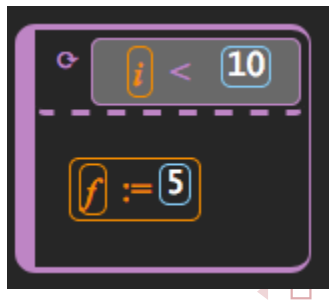








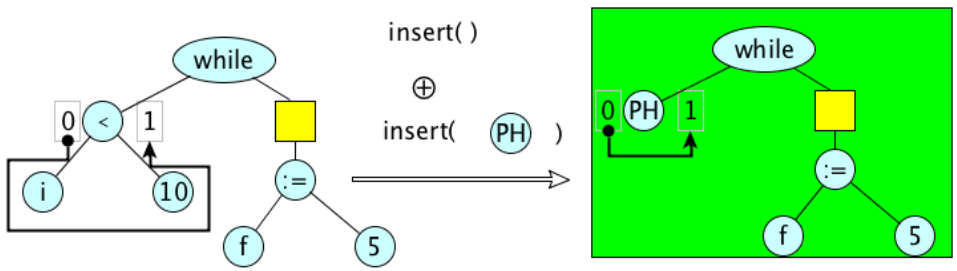


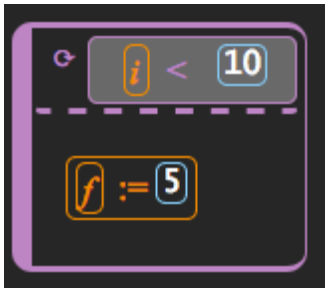


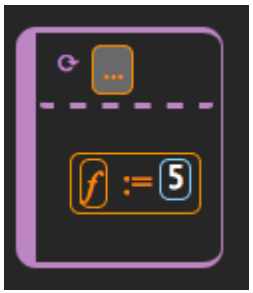
is unchanged.

A biased choice of edits

Implementing the delete key





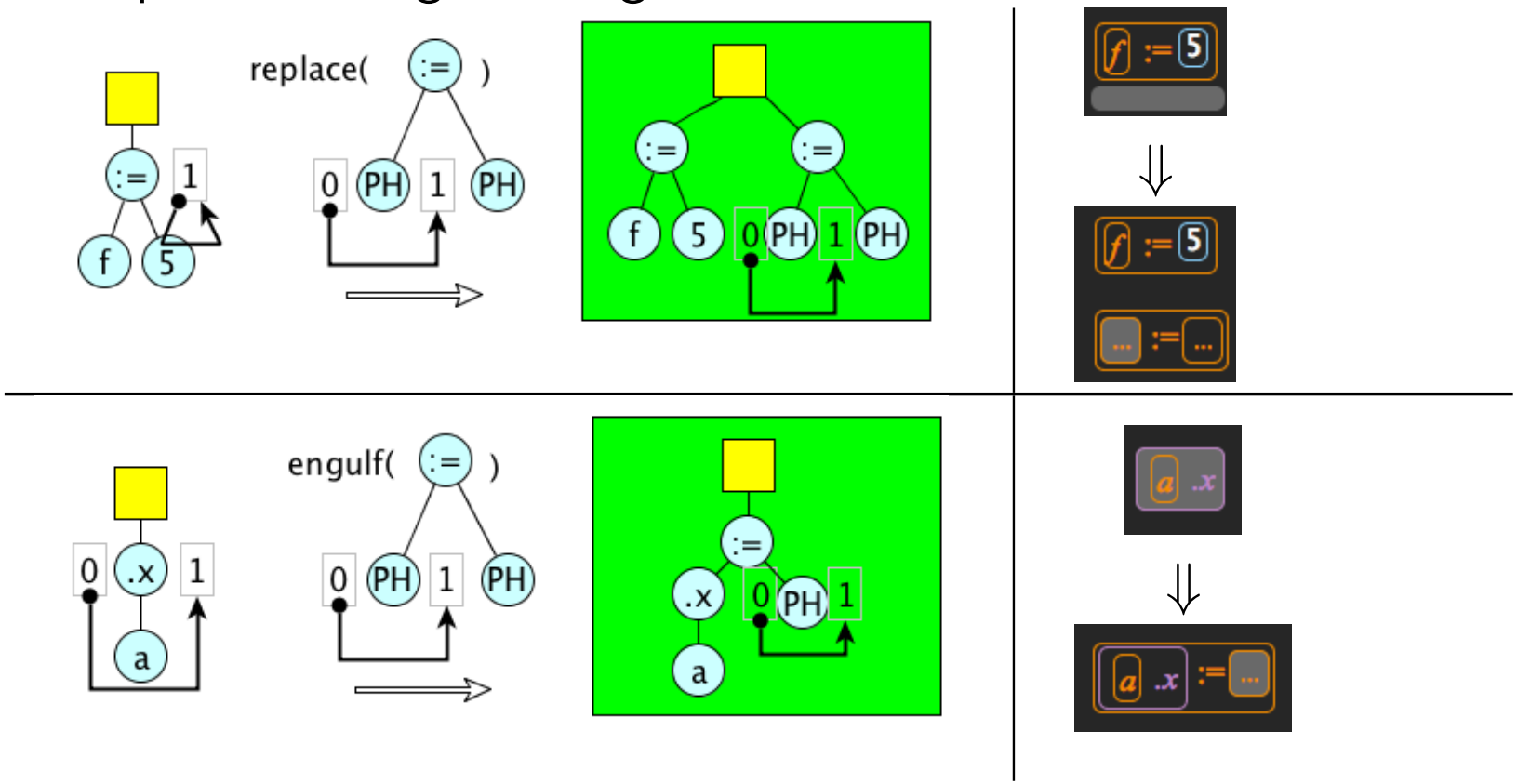


Biased choice allows context dependence.

Templates

Templates are selections. We use them to make edits.

Example: Inserting an assignment.



Many keys are bound to $\texttt{replaceOrEngulf}(t) = (\texttt{allPH} \circ \texttt{replace}(t)) \oplus \texttt{engulf}(t) \oplus \texttt{replace}(t)$ for some t.

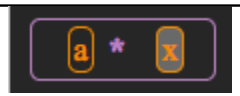
Making an efficient UI

Prefix entry of an expression

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

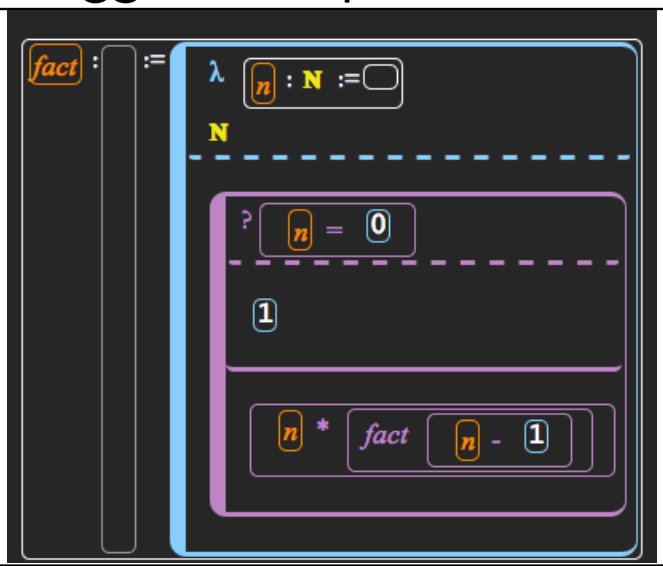
Making an efficient UI

Infix entry of an expression

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

Making an efficient UI

Both examples take 11 keystrokes.
The equivalent in C is at least 7: a*x+b*y
In general equivalent programs are roughly the same number of keystrokes.
A bigger example:




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

44 keystrokes	52 keystrokes

Reflection on the Software Engineering

Sometimes small decisions make significant differences.
In this project, we made several key small decisions:

- Checked constructors ensure invariants are established.
- Use immutable structures to ensure invariants are not broken.
- Validity is declarative and is checked by subclasses of `PLabel` that are application specific.
- Edits are not responsible for checking validity. This makes them largely application generic.
- Options mean that most errors are caught by the type checker. No "Oops, I forgot to check that ...".
- Complex edits are composed from simpler ones.

Reflection on the user experience engineering.

Sometimes small decisions make significant differences.
In this project, we made several key small decisions:

- Only allow valid trees.
- One UI action can mean several things.
  - We can pick the first edit to succeed.
  - Or we can leave the choice to the user.
- Make the current selection easy to choose and manipulate.
- Prefer easy of use to logical simplicity

Thanks to students who have worked with me on PLAAY

- Lawrence Bouzane
- Dillon Butt
- Jillian Hancock
- Kamrul Hasan
- Jessica Hillier
- Jason Howell
- Sunil Jaganathan
- Cem Kilic
- Chris Martin
- William Newhook
- Chris Rodgers
- Christopher Stanbridge
- Ajay Vijayakumar