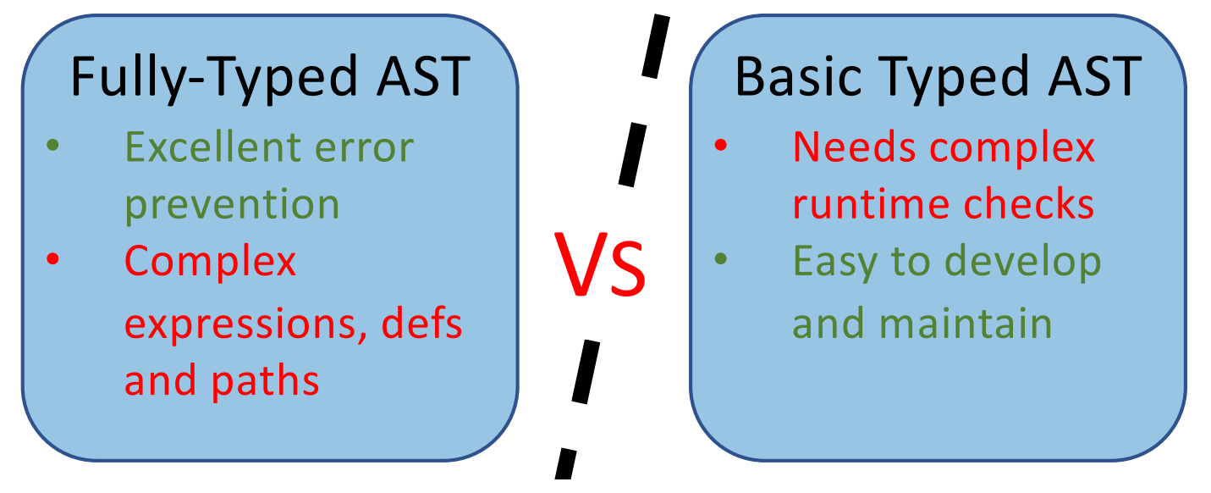
ElmSr: A Projectional Editor for Teaching

Narges Osmani[1], Emma Sabine Willson[1,2], Lucas Dutton[1], Sheida Emdadi[2], Christopher Kumar Anand[1,2]

[1] McMaster University, Computing and Software, [2] Fondation STaBL Foundation

Introduction

Goal: Developing a user-friendly, error-free projectional editor for teaching Elm programming language to children, leveraging the Zipper functional data structure to enhance the Abstract Syntax Tree (AST) manipulation experience.


Fully-Typed AST
Excellent error prevention
Complex expressions, defs and paths
VS
Basic Typed AST
Needs complex runtime checks
Easy to develop and maintain

Motivation:

- Our primary objective is to enhance computational thinking skills in children while minimizing their exposure to the complexities of programming syntax. By preventing errors and reducing frustration, we aim to create a more engaging and accessible learning experience.
- Elm's relatively simple type system allows us to create a fully modeled AST that is easy to understand, manipulate, and maintain.

Key Contributions: A projectional editor for Elm from specifications of the language:

- A fully-typed expression system that prevents [as much as possible] the programmer from entering the wrong code.
- A fast and easy-to-use web-based editor that can be run within a browser.
- A mechanism of defining program navigation and edit operations in terms of a fully keyboard-based set of AST operations.

Related Work

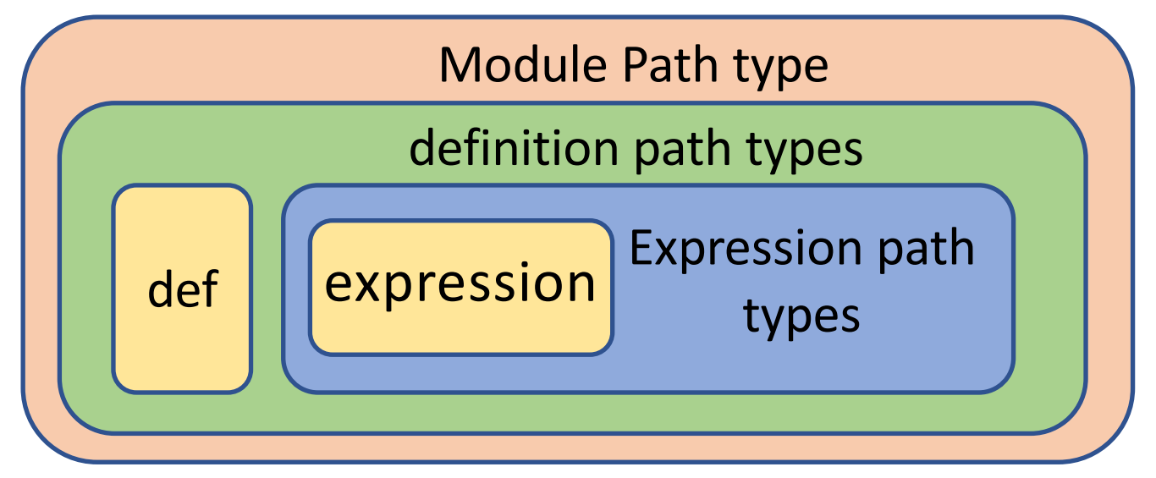
Structured editors like Mentor [1], focused on generating code through direct manipulation of program structure, ensuring syntax correctness and reducing errors. Projectional editors [2], offer a more flexible approach by editing code as projections, enabling language composition, and supporting custom notations. Text-based editors, such as Emacs, provide a familiar editing experience with features like syntax highlighting and autocompletion.

Our Approach

Zippers for trees of single node type: This type of zipper is good for homogeneous binary trees.

```
type Expression
     = IfThenElse Expression Expression Expression
     | ...

type Crumb
     = CondOf Expression Expression
     | ThenOf Expression Expression
     | ElseOf Expression Expression
     | ...

type Zipper
     = Zipper Expression (List Crumb)
```

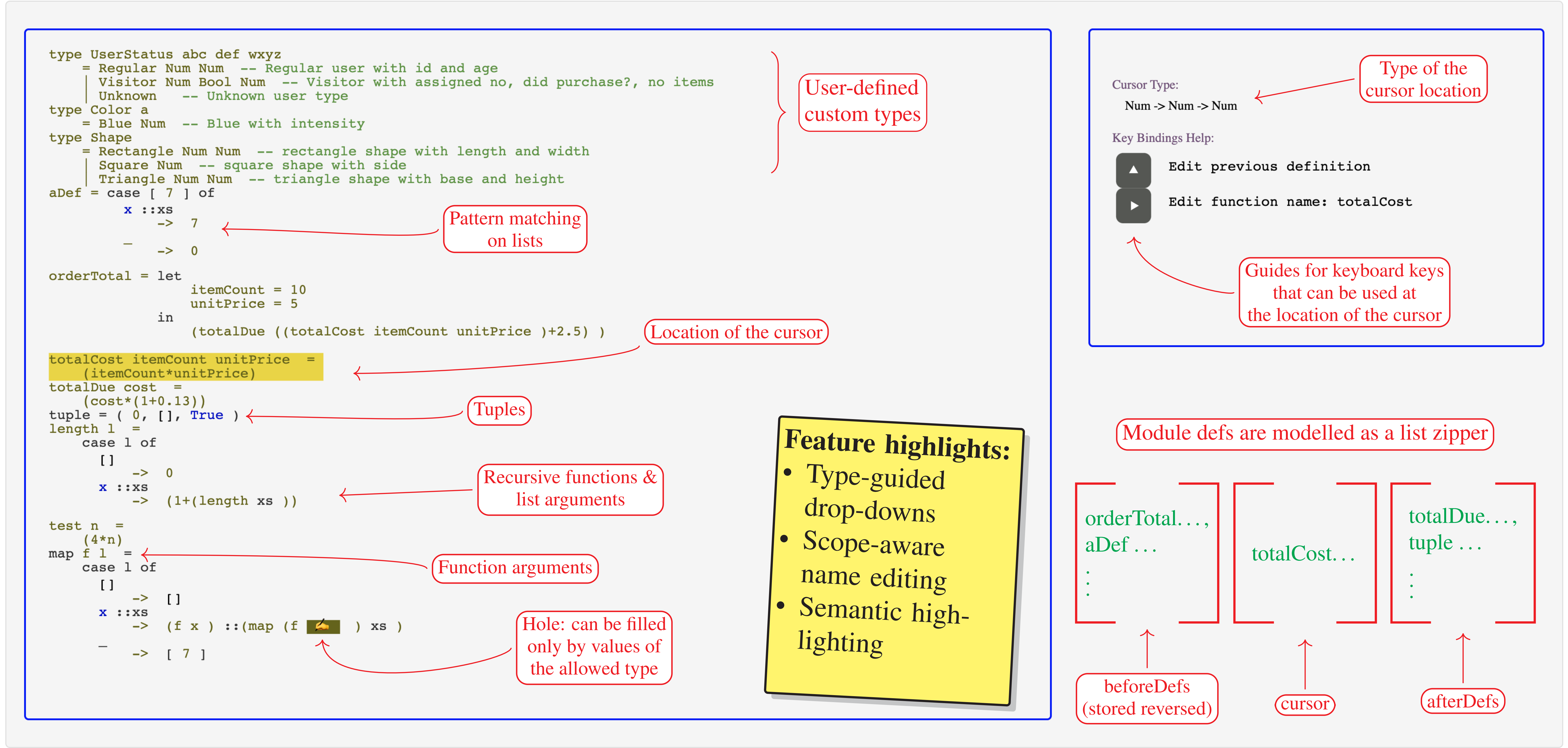
Projecting AST nodes: We render the cursor location, and then wrap it using recursive path types of location's outer scope.



Zippers in our editor: A fully-typed expression system that leverages Elm's compiler for error checking using recursive path definitions

```
type ElmType
     = ElmNum
     | ElmBool
     | ...

type ElmNum
     = NumIf ElmBool ElmNum ElmNum
     | ...

type ElmNumPath
     = ThenOfNumIf_ ElmBool ElmNum ElmNumPath
     | ElseOfNumIf_ ElmBool ElmNum ElmNumPath
     | ...

type ElmBoolPath
     | CondOfNumIf_ ElmNum ElmNum ElmNumPath
     | ...

type Location
     = LocInt ElmInt ElmIntPath
     | LocBool ElmBool ElmBoolPath
     | ...
```

Our Editor



```
type UserStatus abc def wxyz
    = Regular Num Num  -- Regular user with id and age
    | Visitor Num Bool Num  -- Visitor with assigned no, did purchase?, no items
    | Unknown   -- Unknown user type
type Color a
    = Blue Num  -- Blue with intensity
type Shape
    = Rectangle Num Num  -- rectangle shape with length and width
    | Square Num  -- square shape with side
    | Triangle Num Num  -- triangle shape with base and height
aDef = case [ 7 ] of
         x ::xs
             ->  7

         _
             ->  0

orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalDue ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalDue cost  =
    (cost*(1+0.13))
tuple = ( 0, [], True )
length l  =
    case l of
      []
          ->  0
      x ::xs
          ->  (1+(length xs ))

test n  =
    (4*n)
map f l  =
    case l of
      []
          ->  []
      x ::xs
          ->  (f x ) ::(map (f ✍  ) xs )

      _
          ->  [ 7 ]
```

User-defined custom types
Pattern matching on lists
Location of the cursor
Tuples
Recursive functions & list arguments
Function arguments
Hole: can be filled only by values of the allowed type
Cursor Type:
Num -> Num -> Num
Key Bindings Help:
▲ Edit previous definition
▶ Edit function name: totalCost
Type of the cursor location
Guides for keyboard keys that can be used at the location of the cursor
Module defs are modelled as a list zipper
orderTotal. . . , aDef . . .
totalCost. . .
totalDue. . . , tuple . . .
beforeDefs (stored reversed)
cursor
afterDefs

Feature highlights:

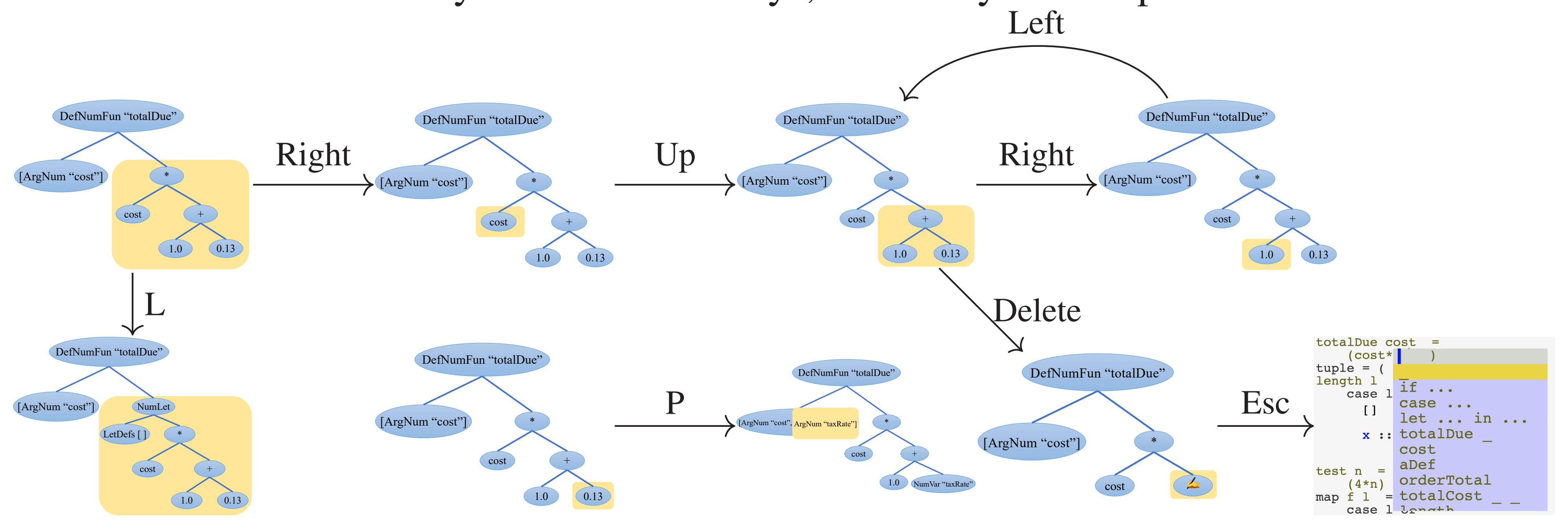
- Type-guided drop-downs
- Scope-aware name editing
- Semantic highlighting

Structural Navigation and Manipulation

Common navigation and editing operations are defined in terms of AST transformations.

AST Navigation: Done with arrow keys.

AST Edits: Done with keyboard shortcut keys, delete key and drop down menu.



Future Work

- Type inference for typed holes
- Better GUI design including mentor intervention, chat, and versioning
- Conducting empirical evaluations and user studies

References

[1] V. Donzeau-Gouge, et al. Practical Applications of a Syntax Directed Program Manipulation Environment. In *Proceedings of the 7th International Conference on Software Engineering*, pages 346–354, 1984.

[2] M. Voelter et al. Towards User-Friendly Projectional Editors. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, pages 41–61, 2014.

Conclusion

We have presented a research project focused on creating an error-free projectional editor for teaching Elm programming to children. The use of the Zipper data structure streamlines the manipulation of the AST, offering an efficient approach to understanding and maintaining code.

As a research project, there is potential for further refinement and enhancement before integrating it into educational environments, especially MacOutreach.Rocks. Future work will focus on improving the editor's features and usability, as well as testing its effectiveness in real-world teaching scenarios, ultimately fostering a new generation of skilled programmers and problem-solvers.

Supplementary Material: