



Master's thesis

Implementation of a type-safe generalized syntax-directed editor

Sune Skaanning Engtorp

Advisor: Hans Hüttel

Submitted: 31/05/2024

Abstract

This thesis investigates the development and implementation of a type-safe, generalized syntax-directed editor. The goal is to create an editor capable of supporting any language, including but not limited to programming languages. The foundation of this work is a proposed generalized editor calculus, which has been encoded in an extended lambda calculus to theoretically establish the capability of building such an editor. This project realizes this calculus in practice by implementing it in the functional programming language Elm, which has already been proven capable of supporting a non-generic structure editor.

The report details the implementation process, encompassing the representation of abstract syntax, source code generation, and the handling of editor expressions. The implementation, written in Elm, features a language specification parser and a source code generator. Currently, the generated editor for a given language can perform basic edits on the abstract syntax tree, such as cursor movement and substitution.

Practical examples are provided for subsets of different languages, including C, SQL, and \LaTeX , demonstrating the editor's generality.

The thesis concludes with proposals for future work, including the implementation of missing editor expressions, handling of context-sensitive syntax and views of the program being edited.

The source code of this project is available at following public GitHub repository: <https://github.com/Sunese/generalized-editor>

Contents

1	Introduction	1
1.1	Related work	1
1.1.1	Structure editors	1
1.1.2	Editor generators	2
1.1.3	Partial evaluation	3
1.2	Problem statement	5
1.3	Report Structure	7
2	Background	8
2.1	Abstract Syntax	8
2.1.1	Abstract Syntax Trees	8
2.1.2	Abstract Binding Trees	9
2.2	Generalized editor calculus	10
2.2.1	Abstract syntax	10
2.2.2	Semantics	14
2.2.3	Encoding the generalized editor calculus in an extended λ -calculus	16
3	Implementation	25
3.1	Implementation	25
3.1.1	Representing Syntax	25
3.1.2	Code generation versus generic model	33
3.1.3	Generating source code	34
3.1.4	Editor expressions	38
4	Editor Examples	57
4.1	C	57
4.2	SQL	64
4.3	L ^A T _E X	70
5	Conclusion	76
5.1	Future work	76

Chapter 1

Introduction

1.1 Related work

1.1.1 Structure editors

Structure editors provide a way to manipulate the abstract syntax structure of programs directly, in contrast to writing and editing source code of a program in plain text, which also requires a parser to produce an abstract syntax tree. An early example of this is the Cornell Program Synthesizer by Reps and Teitelbaum[23] in 1981. By using a structure editor, the user can avoid syntax errors and might have a better overview of their source code. Moreover, unfinished blocks of code can be represented by syntactic holes, allowing the programmer to develop a mental model of their code, without getting distracted or blocked by syntax errors.

However structure editors like the Cornell Program Synthesizer[23] allow programmers to create syntactically ill-formed programs. This includes introducing use-before-declaration statements, as the editor cannot manage context-sensitive constraints within the syntax.

In 2017 Omar et al. introduced Hazel[17], a programming environment for a functional language with typed holes, which allows for evaluation to continue past holes which might be empty or ill-formed. The motivation for this work is to provide feedback about the dynamic behaviour of a program as it is being edited, in contrast to other programming languages and environments which usually only assign dynamic meaning to complete programs. In other words, the Hazel environment[17] provides feedback on programs, even if they are ill-formed or contain type errors. This is possible by surrounding static and dynamic inconsistencies with typed holes, allowing evaluation to proceed past them.

The Hazel environment is based on the Hazelnut structure editor calculus, de-

financed by operational semantics, that allow finite edit expressions and inserts holes automatically to guarantee that every editor is well-typed.

Hazel itself is not a structure editor, however the core calculus supports incomplete functional programs, also referred to as "holes", which are a central part of the work of Godiksen et. al [4]. They introduced a type-safe structure editor calculus which manipulates a simply-typed lambda calculus with the ability to evaluate programs partially with breakpoints and assign meaning to holes. It also ensures that if an edit action is well-typed, then the resulting program is also well-typed. The editor calculus and programming language have later been used to implement a type-safe structure editor in Elm[21].

1.1.2 Editor generators

A common property of the editor calculi and editors mentioned so far is that they are built to work with only one programming language. The calculi are strongly dependent on the language they can manipulate, and if the language were to change, it could require re-writing the complete editor calculus. A solution to this problem is *editor generators*.

A few years after presenting the Cornell Program Synthesizer, Reps and Teitelbaum presented The Synthesizer Generator[20] in 1984, which creates structure editors from language descriptions in the form of attribute grammar specifications.

Another example is the Centaur system[1], which takes formalism described in the Metal language[14], a collection of concrete syntax, abstract syntax, tree building functions and unparsing (a.k.a. pretty-printing) specifications. The abstract syntax is made of operators and *phyla*, where operators label the nodes of the abstract trees and are either fixed arity or list operators. Operators are defined as having *offsprings*, where fixed-arity operators can have offsprings of different kinds, whereas offsprings of list operators must be of the same kind. This concept is formalized by the concept of *phylum*, where *phyla* (plural of *phylum*) are sets of operators, describing what operators are allowed at every offspring of an operator. In other words, phyla constrain the allowed operators in every subtree of either a non-null fixed arity operator or a list-operator. Each operator-phylum relation is used to maintain syntactically correct trees. This definition of operators and phyla in Metal strongly relates to another definition of abstract syntax relevant to this project, which will be detailed in the Background section, namely Harper's definition of abstract syntax[6]. Harper defines abstract syntax using sorts, arity-indexed operators, and variables. In both definitions, operators represent nodes in an abstract syntax tree. The association of operators with phyla or being arity-indexed serves a similar purpose of constraining the possible children of each node, thereby maintaining syntactically correct trees.

The Centaur system[1] also supports specifying the semantics of languages in TY-

POL[2] and has logic-based enforcement of the semantic rules with a Prolog-based engine.

However, while Centaur provides a robust framework for syntax and semantics, it does not explicitly define a formal editor calculus. In contrast, a type-safe generalized editor calculus has been proposed by [16], which builds upon the work of Godiksen et al. [4]. This will be detailed in the Background section.

1.1.3 Partial evaluation

One of the goals of this project is to implement a generic syntax-directed editor based on the editor calculus proposed by [16]. Implementing such a generic editor relates to the question of how to balance between generality and modularity, brought up by Neil D. Jones[13] in the context of program specialization. If we are presented with a class of similar problems, such as instantiating a syntax-directed editor for different languages, one might consider two extremes: either write many small, but efficient, programs or write a single highly *parametrized* program which can solve any problem.

The first approach, involving multiple small programs, has the advantage of allowing the programmer to focus solely on performance optimization for each specific case. However, it has the significant drawback of being hard to maintain, as it involves managing multiple codebases. This approach can also lead to performance overhead due to the need to pass data between programs and convert various internal representations.

The second approach is general and has the advantage of being easier to document and maintain, due to it being a single program. However, it is arguably not well-performing, as some amount of time needs to be spent interpreting the parameters instead of the actual problem-solving computation.

Using a partial evaluator[12] to specialize a highly parametrized program into smaller, customized versions, is arguably a better solution than both approaches. This way, only one single program requires maintenance, while allowing it to be specialized, taking advantage of program speedup.

The notion of partial evaluation in terms of program specialization[12] includes a partial evaluator, which receives a general program and some static input, resulting in a specialized program that, if specialized in a meaningful way, performs better than the original program. An example provided by Jones is a program computing x^n (program p in listing 1.1), which can be specialized to having $n = 5$ (program p_5 in listing 1.2), unfolding the recursive calls and reducing $x * 1$ to x . Formally, the general program p and static input in_1 are passed to a partial evaluator mix , which outputs a specialized program p_{in_1} , which can take dynamic input in_2 and produce some output. For an illustration of this example, see figure 1.1.

```

1 f(n, x) = if n = 0 then 1
2   else if even(n) then f (n/2,x)^2
3   else x * f(n-1,x)

```

Listing 1.1: Two-input program p

```

1 f5(x) = x * ((x^2)^2)

```

Listing 1.2: Specialization of program p

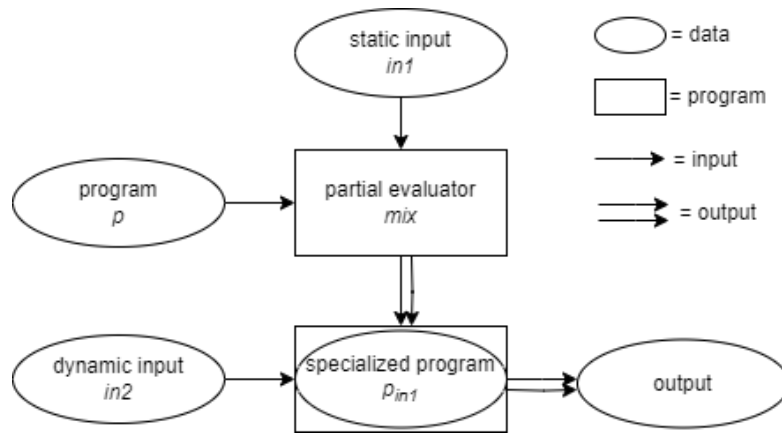


Figure 1.1: Visualisation of partial evaluation of a two-input program

This idea of specialization can also be applied to the implementation of this project, where the static input is the syntax of a language, which if partially evaluated with a general program, results in a specialized program. This program represents a syntax-directed editor instance for the static input's language, which can take dynamic input in the form of editor expressions, resulting in either cursor movement or updates to the target language's program. This can be seen as a curried function with following signature:

$$f : t_1 \rightarrow t_2 \rightarrow t_3$$

where t_1 is the type of language specifications, t_2 is the type of editor expressions and t_3 is the type of programs. Given a language specification, f produces a new function f' of type $t_2 \rightarrow t_3$, i.e., a new function that takes an editor expression and returns a program. Here, f' is an instance of the editor for a specific language.

In other words, the implementation can be specialized given some syntax, potentially skipping the process of parsing syntax and generating source code every time a user wishes to use the same editor instance for some language.

1.2 Problem statement

Structure editors provide a way to manipulate the abstract syntax structure of a program directly, in contrast to writing and editing the source code of a program in plain text, which also requires a parser to produce an abstract syntax tree. An early example of this is shown by Reps and Teitelbaum[20] in 1984. By using a structure editor, the user can avoid syntax errors and might have a better overview of their source code. Moreover, unfinished blocks of code can be represented by syntactic holes, allowing the programmer to develop a mental model of their code, without getting distracted or blocked by syntax errors.

This project will implement a generalized type-safe structure editor, based on a proposed calculus[16].

The criteria for a good solution are the following:

- The user can edit the abstract syntax from any program directly
- The editor is generic, in the sense that it can handle the syntax of any language, not limited to programming languages
- The editor can handle context-sensitive syntax, specifically binding mechanisms
- The editor supports multiple views of the code being edited, such as a tree-view or a pretty-printed program based on the language's concrete syntax
- The user should be able to specify the abstract and concrete syntax of any language in a non-challenging way

The implementation relies on a proposed syntax-directed editor calculus[16] which promises to be language-independent and able to handle binding mechanisms. Hence, a minimal solution should fulfil the first two criteria, where a user can edit an AST directly (i.e. it is a structure editor) and the editor is generic, while the ability to handle context-sensitive syntax is crucial, yet not indispensable.

Since this project will propose a generic editor, it is crucial to provide a meaningful set of concrete examples, demonstrating that the solution is indeed generic and can be used for any language. This project will attempt to provide a meaningful set of concrete examples in a non-biased manner, with languages (or subsets thereof) of different paradigms and purposes.

The set of examples should contain one or more popular general-purpose programming languages. All examples should have a clear syntactic distinction, to exclude dialects of the same language and otherwise very similar languages.

GitHub provides a top 30 ranking[9] by number of unique git pushes with a given

programming language on their platform, and this project will consider entries in the chart of popular character.

Usage of the editor is not limited to programming languages, hence the set of examples should also contain one or more domain-specific languages and markup languages suitable for structure editing, in the sense that it contains declarations and binders.

Based on prior criteria, the project will provide examples for subsets of the following languages:

- C
- SQL
- L^AT_EX

The C language[10] is a popular general-purpose programming language deemed as a good candidate because of it being an imperative language, which stands in contrast to recent works on structure editors[16][4][17] that primarily focus on functional languages. The C programming language also has a strict and symbol-rich concrete syntax, which can easily lead to ill-formed programs when editing a C program in plain text. Consider Fig. 1.2 for an example.

Figure 1.2: Ill-formed C program

Below is an example of a C program that at first glance looks well-formed, but will be rejected because of a missing semicolon at line 4:

```
1 int main() {  
2     int x = 0;  
3     for (int i; i = 0; i++) {  
4         x++  
5     }  
6     return 0;  
7 }  
8
```

The SQL language is a popular language deemed as a good candidate, because of it being a declarative language that also provides a contrast to both functional and imperative languages. There exist many dialects of SQL and for the sake of completeness, this project will refer to the PostgreSQL[5] variant in any SQL context. Different dialects, including PostgreSQL, have a strict syntax in common, which can also easily lead to ill-formed programs when editing in plain text. Consider Fig. 1.3 for an example.

Figure 1.3: Ill-formed SQL query

Below is an example of a PostgreSQL query which is ill-formed because of a missing comma in line 1 and wrong use of the equality operator in line 2:

```
1 SELECT col-a col-b FROM table
2 WHERE col-a = 'x';
```

\LaTeX [22] is a markup language suitable for structure editing, which provides a contrast to the already-picked examples since it is not considered a programming language. \LaTeX is suitable for structure editing as documents often involve complex structures with nested environments and commands, which can be cumbersome to manage in plain text. A structure editor can significantly simplify the process of creating and managing these complex documents by providing a more intuitive interface.

1.3 Report Structure

The report is structured as follows:

- Chapter 2 presents a summary of abstract syntax, as described by Harper[6], which is a central part of the editor calculus proposed by [16], both subjects inherently playing an important role in the implementation of the editor.
- Chapter 3 describes the implementation of the editor, from parsing abstract syntax to generating source code for an editor instance.
- Chapter 4 provides examples of the editor in use for different languages, demonstrating the editor’s generic nature.
- Chapter 5 concludes the report, summarizing the project and suggesting future work.

Chapter 2

Background

Following is a summary of the background material which provides the foundation for the implementation of the type-safe generalized syntax-directed editor.

2.1 Abstract Syntax

Following is an overview of abstract syntax trees and abstract binding trees, as described by Harper[6]. This has served as the foundation for the work presented in the generalized editor calculus project[16], which inherently also plays an important role in the implementation of this project.

2.1.1 Abstract Syntax Trees

An abstract syntax tree (ast for short) is an ordered tree describing the syntactical structure of a program.

Harper introduces the notion of sorts $s \in \mathcal{S}$, arity-indexed families \mathcal{O} of disjoint sets of operators \mathcal{O}_α of arity α and sort-indexed families \mathcal{X} of disjoint finite sets \mathcal{X}_s of variables x of sort s .

Sorts are syntactical categories which form a distinction between asts.

The internal nodes in an ast are operators o with arity $(s_1, \dots, s_n)s$, describing the sort of the operator itself and its arguments.

Leaves of an ast are variables x of sort s , which enforce that variables can only be substituted by asts of the same sort.

Formally, \mathcal{S} is a finite set of sorts. An arity has the form $(s_1, \dots, s_n)s$ specifying the sort $s \in \mathcal{S}$ of an operator taking $n \geq 0$ arguments of sort $s_i \in \mathcal{S}$. Let $\mathcal{O} = \{\mathcal{O}_\alpha\}$ be

an arity-indexed family of disjoint sets of operators \mathcal{O}_α of arity α . When o is an operator with arity $(s_1, \dots, s_n)s$, we say o is an operator of sort s with n arguments, each of sort s_1, \dots, s_n .

Example 1: Operators

Let us define a set of sorts $\mathcal{S} = \{exp\}$ and an operator $plus \in \mathcal{O}_\alpha$ with arity $\alpha = (exp_1, exp_2)exp$. Then we say that the operator $plus$ is an operator of sort exp with 2 arguments of sort exp .

Having a fixed set of sorts \mathcal{S} and an arity-indexed family \mathcal{O} of sets of operators of each arity, $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ is defined as a sort-indexed family of disjoint finite sets \mathcal{X}_s of variables x of sort s .

The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ of asts of sort s is the smallest family satisfying following conditions:

1. A variable of sort s is an ast of sort s : if $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$
2. Operators combine asts: if o is an operator of arity $(s_1, \dots, s_n)s$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \dots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \dots; a_n) \in \mathcal{A}[\mathcal{X}]_s$

2.1.2 Abstract Binding Trees

An abstract binding tree (abt for short) is an enriched ast with bindings, allowing identifiers to be bound within a scope.

Operators in an abt can bind any finite number (possibly zero) of variables in each argument with form $x_1, \dots, x_k.a$ where variables x_1, \dots, x_k are bound within the abt a . A finite sequence of bound variables x_1, \dots, x_k is represented as \vec{x} for short.

Operators are assigned a generalized arity of the form $(v_1, \dots, v_n)s$ where a valence v has the form $s_1, \dots, s_k.s$, or $\vec{s}.s$ for short.

Example 2: Operators with binders

Let us define a set of sorts $\mathcal{S} = \{exp, stmt\}$ and an operator $let \in \mathcal{O}_\alpha$ with arity $\alpha = (exp_1, exp_2.stmt)stmt$. Then we say that the operator let is an operator of sort $stmt$ with 2 arguments of sort exp , where the second binds a variable of sort $stmt$ within the scope of exp_2 .

If \mathcal{X} is clear from context, a variable x is of sort s if $x \in \mathcal{X}_s$, and x is fresh for \mathcal{X} if $x \notin \mathcal{X}_s$ for any sort s . If x is fresh for \mathcal{X} and s is a sort, then \mathcal{X}, x represents the notion of adding x to \mathcal{X}_s .

A fresh renaming of a finite sequence of variables \vec{x} is a bijection $\rho : \vec{x} \leftrightarrow \vec{x}'$, where \vec{x}' is fresh for \mathcal{X} . The result of replacing all occurrences of x_i in a by its fresh

counterpart $\rho(x_i)$, is written as $\hat{\rho}(a)$.

Having a fixed set of sorts \mathcal{S} and a family \mathcal{O} of disjoint sets of operators indexed by their generalized arities and given a family of disjoint sets of variables \mathcal{X} , the family of abts $\mathcal{B}[\mathcal{X}]$ is the smallest family satisfying following:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$
2. For each operator o of arity $(\vec{s}_1.s_1, \dots, \vec{s}_n.s_n)s$, if for each $1 \leq i \leq n$ and for each fresh renaming $\rho_i : \vec{x}_i \leftrightarrow \vec{x}'_i$, we have $\hat{\rho}_i(a_i) \in \mathcal{B}[\mathcal{X}, \vec{x}'_i]$, then $o(\vec{x}_1.a_1; \dots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$

For short, arity-indexed families \mathcal{O} of disjoint sets of operators \mathcal{O}_α and sort-indexed families \mathcal{X} of disjoint finite sets \mathcal{X}_s might be referred to as sets. For example, $o \in \mathcal{O}$ is a shorthand for operator o being part of one of the disjoint sets in the family \mathcal{O} .

2.2 Generalized editor calculus

The generalized editor calculus[16] is a generalization of the type-safe structure editor calculus proposed by Godiksen et al.[4], where the editor calculus is able to account for any language given its abstract syntax, in contrast to the calculus in Godiksen[4] et al. which is tailored towards an applied λ -calculus.

2.2.1 Abstract syntax

It is assumed that the abstract syntax is given by a set of sorts \mathcal{S} , an arity-indexed family of operators \mathcal{O} and a sort-indexed family of variables \mathcal{X} , as described by Harper[6].

Cursors and holes are important concepts in syntax-directed editors, where the cursor represents the current selection in the syntax tree, and holes are missing or empty subtrees.

The calculus proposed by Godiksen et al. [4] has a single term for the cursor and hole in the abstract syntax. For a generalized calculus, it is necessary to add a hole and cursor for every sort in \mathcal{S} .

Definition 1: Abstract syntax of a language

The abstract syntax of a language is given by the following:

1. A set of sorts \mathcal{S}
2. An arity-indexed family of operators \mathcal{O}
3. A sort-indexed family of variables \mathcal{X}

Then, for every sort $s \in \mathcal{S}$, the following operators are added to \mathcal{O}

1. A *hole*_s operator with arity $()s$
2. A *cursor*_s operator with arity $(s)s$

Example 3: Abstract syntax of a simple language

Below is a simple language consisting of arithmetic expressions and local declarations:

Sort	Term	Operator	Arity
$s ::=$	$\text{let } x = e \text{ in } s$	<i>let</i>	$(e, e.s)s$
	$ \quad e$	<i>exp</i>	$(e)s$
$e ::=$	$e_1 + e_2$	<i>plus</i>	$(e, e)e$
	$ \quad n$	<i>num</i> [n]	$()e$
	$ \quad x$	<i>var</i> [x]	$()e$

In other words, we have sorts:

$$\mathcal{S} = \{s, e\}$$

and we have operators:

$$\mathcal{O} = \{\mathcal{O}_{(e, e.s)s}, \mathcal{O}_{(e)s}, \mathcal{O}_{(e, e)e}, \mathcal{O}_{()e}\}$$

where

$$\mathcal{O}_{(e, e.s)s} = \{\text{let}\}$$

$$\mathcal{O}_{(e)s} = \{\text{exp}\}$$

$$\mathcal{O}_{(e, e)e} = \{\text{plus}\}$$

$$\mathcal{O}_{()e} = \{\text{num}[n], \text{var}[x]\}$$

Example 4: Introduction of cursors and holes

Given the abstract syntax of a simple language consisting of arithmetic expressions and local declarations (Example 3), it would be extended with following cursor and hole operators:

Sort	Term	Operator	Arity
$s ::=$	$[s]$	$cursor_s$	$(s)s$
	$ \quad \llbracket \rrbracket_s$	$hole_s$	$()s$
$e ::=$	$[e]$	$cursor_e$	$(e)e$
	$ \quad \llbracket \rrbracket_e$	$hole_e$	$()e$

Editor calculus

The abstract syntax of the general editor calculus (Fig. 2.1) resembles the one from Godiksen et al., the only difference being the lack of an *eval* construct, since the generalized editor only considers the static and structural properties of abstract syntax. The key components of the calculus include:

- Atomic prefix commands (π): These commands facilitate the traversal and modification of the abstract syntax tree. They include operations for moving the cursor to a child or parent and operator substitution.
- Conditions (ϕ): These are logical expressions used to control the flow of the editor actions based on the structure and properties of the syntax tree. Conditions can involve logical negation, conjunction, disjunction, and modal operators that answer questions about the presence of operators in the syntax tree.
- Editor expressions (E): These are the core constructs of the calculus, representing the actions that can be performed on the syntax tree, including conditional execution and sequence of actions.

Figure 2.1: Abstract syntax of general editor calculus

$E ::=$	$\pi.E \mid \phi \Rightarrow E_1 E_2 \mid E_1 \ggg E_2 \mid rec\ x.E \mid x \mid nil$
$\pi ::=$	$child\ n \mid parent \mid \{o\}$
$\phi ::=$	$\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid @o \mid \Diamond o \mid \Box o$

Cursorless trees

The notion of cursorless trees is introduced to support a single cursor only being able to encapsulate a single node, hence the rest of the tree being considered

cursorless.

The abstract syntax of cursorless trees formalizes this notion by introducing new sorts and operators that represent the tree without an active cursor. These sorts and operators accommodate the requirement that the cursor can only be at one location at a time.

Definition 2: Abstract syntax of cursorless trees

The abstract syntax of cursorless trees is given by:

1. The sorts $\hat{\mathcal{S}} = \{\hat{s}\}_{s \in \mathcal{S}}$
2. The family of cursorless operators $\hat{\mathcal{O}}$ is made by adding the operator \hat{o} of arity $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ for every $o \in \mathcal{O}$ of arity $(\vec{s}_1.s_1, \dots, \vec{s}_n.s_n)s$
3. The family of variables $\hat{\mathcal{X}}$

Cursor Context

A cursor context C holds information about the current tree, up until a context hole, which is filled out by a well-formed tree including the cursor. In other words, the actual cursor is located somewhere in the well-formed tree but is not part of the cursor context. The cursor context acts as a template that outlines the structure of the tree around the cursor, which is inspired by the zipper data structure by Huet[8], as a way to locate the cursor in a tree.

Definition 3: Abstract syntax of cursor contexts

The abstract syntax of cursor contexts is given by:

1. The sorts $\mathcal{S}^C = \hat{\mathcal{S}} \cup \{C\}$
2. The family of operators $\mathcal{O}^C = \hat{\mathcal{O}}$ extended with the $[\cdot]$ operator with arity $()C$
3. For every operator $\hat{o} \in \hat{\mathcal{O}}$ of arity $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ the operator o_i^C of arity $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_i.C, \dots, \vec{\hat{s}}_n.\hat{s}_n)C$ to \mathcal{O}^C
4. The family of variables $\mathcal{X}^C = \hat{\mathcal{X}}$

Well-formed trees

Well-formed trees are characterized by having a single cursor, which is either at the root or one of the immediate children, with the rest of the tree being cursorless. This structure ensures the tree is considered well-formed because it contains only one cursor. The concept of well-formed trees is fundamental to the generalized

editor calculus. Edit actions are restricted to well-formed trees, which guarantees that the cursor is in a valid position and that there is only one cursor present in the tree.

Definition 4: Abstract syntax of well-formed trees

The abstract syntax of well-formed trees is given by:

1. The sorts $\dot{\mathcal{S}} = \hat{\mathcal{S}} \cup \{\dot{s}\}_{s \in \mathcal{S}}$
2. The family of operators $\dot{\mathcal{O}} = \hat{\mathcal{O}}$ extended with an operator of arity $(\hat{s})\dot{s}$ for every $\hat{s} \in \hat{\mathcal{S}}$
3. For every operator $\hat{o} \in \hat{\mathcal{O}}$ of arity $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ the operator \dot{o}_i of arity $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_i.\hat{s}_i, \dots, \vec{\hat{s}}_n.\hat{s}_n)\dot{s}$ is added to $\dot{\mathcal{O}}$
4. The family of variables $\dot{\mathcal{X}} = \hat{\mathcal{X}}$

Given this, a well-formed abt $a \in \mathcal{B}[\mathcal{X}]$ is any abt that can be interpreted as $C[\dot{a}]$, where C is a cursor context and \dot{a} is a well-formed tree.

2.2.2 Semantics

The semantics of the editor calculus are defined through reduction rules that describe how editor expressions are evaluated and how the cursor is manipulated within the syntax tree.

Reduction rules for editor expressions are presented in Fig. 2.2, substitution in Fig. 2.3 and cursor movement in Fig. 2.4.

Figure 2.2: Reduction rules for editor expressions

$$\begin{aligned}
(\text{Cond-1}) \quad & \frac{a \models \phi}{\langle \phi \Rightarrow E_1 | E_2, C[a] \rangle \xrightarrow{\epsilon} \langle E_1, C[a] \rangle} \\
(\text{Cond-2}) \quad & \frac{a \not\models \phi}{\langle \phi \Rightarrow E_1 | E_2, C[a] \rangle \xrightarrow{\epsilon} \langle E_2, C[a] \rangle} \\
(\text{Seq}) \quad & \frac{\langle E_1, a \rangle \xrightarrow{\alpha} \langle E'_1, a' \rangle}{\langle E_1 \gg E_2, a \rangle \xrightarrow{\alpha} \langle E'_1 \gg E_2, a' \rangle} \\
(\text{Seq-Trivial}) \quad & \frac{}{\langle \text{nil} \gg E_2, a \rangle \xrightarrow{\epsilon} \langle E_2, a \rangle} \\
(\text{Recursion}) \quad & \frac{}{\langle \text{rec } x. E, a \rangle \xrightarrow{\epsilon} \langle E[x := \text{rec } x. E], a \rangle} \\
(\text{Context}) \quad & \frac{a \xrightarrow{\pi} a'}{\langle \pi.E, C[a] \rangle \xrightarrow{\pi} \langle E, C[a'] \rangle}
\end{aligned}$$

Figure 2.3: Reduction rules for substitution

$$(\text{Insert-op}) \quad \frac{}{[\hat{a}] \xrightarrow{\{o\}} [o(\vec{x}_1.\hat{\downarrow}_{s_1}; \dots; \vec{x}_n.\hat{\downarrow}_{s_n})]} \hat{a} \in \mathcal{B}[\mathcal{X}]_s \text{ where } s \text{ is the sort of } o$$

Figure 2.4: Reduction rules for cursor movement

$$\begin{aligned}
(\text{Child-i}) \quad & \frac{}{[\hat{o}(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_n.\hat{a}_n)] \xrightarrow{\text{child } i} o(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_i.[\hat{a}_i]; \dots; \vec{x}_n.\hat{a}_n)} \\
(\text{Parent}) \quad & \frac{}{o(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_i.[\hat{a}_i]; \dots; \vec{x}_n.\hat{a}_n) \xrightarrow{\text{parent}} [\hat{o}(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_n.\hat{a}_n)]}
\end{aligned}$$

Satisfaction rules for the propositional connectives and modalities define the conditions under which logical expressions hold true for a given cursorless tree encapsulated by the cursor (i.e. a well-formed tree).

The satisfaction rules for propositional connectives specify how logical negation, conjunction, and disjunction are evaluated, while the satisfaction rules for modalities describe how modal operators, which express following:

- $@o$ holds if the operator o is at the cursor
- $\Diamond o$ holds if the operator o is in one of the subtrees of the operator at the cursor

- $\Box o$ holds if the operator o is in all subtrees of the operator at the cursor

The rules are given in Fig. 2.5 and Fig. 2.6.

Figure 2.5: Satisfaction rules for propositional connectives

$$\begin{aligned}
& \text{(Negation)} \quad \frac{[\hat{a}] \not\models \phi}{[\hat{a}] \models \neg \phi} \\
& \text{(Conjunction)} \quad \frac{[\hat{a}] \models \phi_1 \quad [\hat{a}] \models \phi_2}{[\hat{a}] \models \phi_1 \wedge \phi_2} \\
& \text{(Disjunction-1)} \quad \frac{[\hat{a}] \models \phi_1}{[\hat{a}] \models \phi_1 \vee \phi_2} \\
& \text{(Disjunction-2)} \quad \frac{[\hat{a}] \models \phi_2}{[\hat{a}] \models \phi_1 \vee \phi_2}
\end{aligned}$$

Figure 2.6: Satisfaction rules for modalities

$$\begin{aligned}
& \text{(At-op)} \quad \frac{}{[o(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_n.\hat{a}_n)] \models @o} \\
& \text{(Necessity)} \quad \frac{[\hat{a}_1] \models \Diamond \dots [\hat{a}_n] \models \Diamond o}{[o(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_n.\hat{a}_n)] \models \Box o} \\
& \text{(Possibly-i)} \quad \frac{[\hat{a}_i] \models \Diamond o}{[o(\vec{x}_1.\hat{a}_1; \dots; \vec{x}_i.\hat{a}_i; \dots; \vec{x}_n.\hat{a}_n)] \models \Diamond o} \\
& \text{(Possibly-trivial)} \quad \frac{[\hat{a}] \models @o}{[\hat{a}] \models \Diamond o}
\end{aligned}$$

2.2.3 Encoding the generalized editor calculus in an extended λ -calculus

The following sections will show how to encode the generalized editor calculus in a simply typed λ -calculus, extended with pairs, pattern matching and recursion. The notation $\llbracket a \rrbracket$ will be used to describe the encoding of an abt a .

The simply typed λ -calculus is first extended with term constants o for every $o \in \mathcal{O}$ excluding cursors and base type s for every sort $s \in \mathcal{S}$. The abstract syntax of this can be seen in Fig. 2.7.

Figure 2.7: Abstract syntax of extended λ -calculus

$$\begin{array}{lcl}
 & \text{Terms} & \\
 M & ::= \lambda x : \tau. M & (\text{abstraction}) \\
 & | M_1 M_2 & (\text{application}) \\
 & | x & (\text{variable}) \\
 & | o & (\text{operator}) \\
 & \text{Types} & \\
 \tau & ::= \tau_1 \rightarrow \tau_2 & (\text{function}) \\
 & | s & (\text{sort})
 \end{array}$$

Abstract binding trees

The types of operators in the λ -calculus can be inferred by their arity. The typing rules for this is provided in Fig. 2.8.

Figure 2.8: Typing rules for λ -calculus operators

$$(\text{T-Operator}) \frac{o \in \mathcal{O} \text{ and has arity } (\vec{s}_1.s_1, \dots, \vec{s}_n.s_n)s}{\Gamma \vdash o : (\vec{s}_1 \rightarrow s_1) \rightarrow \dots (\vec{s}_n \rightarrow s_n) \rightarrow s}$$

Then, any abt can be encoded in the extended simply typed λ -calculus by the encoding in Fig. 2.9.

Figure 2.9: Encoding of abts

$$\llbracket o(\vec{x}_1.a_1, \dots, \vec{x}_n.a_n) \rrbracket = o(\lambda \vec{x}_1 : \vec{s}_1. \llbracket a_1 \rrbracket) \dots (\lambda \vec{x}_n : \vec{s}_n. \llbracket a_n \rrbracket)$$

Cursor Contexts

Cursor contexts will be represented in the λ -calculus as pairs, hence it is extended to support this notion. The abstract syntax is given in Fig. 2.10.

Figure 2.10: Abstract syntax of extended λ -calculus

$$\begin{array}{lcl}
 & \text{Terms} & \\
 M & ::= (M_1, M_2) & (\text{pair}) \\
 & | M.1 & (\text{first projection}) \\
 & | M.2 & (\text{second projection}) \\
 & \text{Types} & \\
 \tau & ::= \tau_1 \times \tau_2 & (\text{product type})
 \end{array}$$

Reduction rules for projecting the first and second value in a pair are given in Fig. 2.11.

Figure 2.11: Reduction rules for pairs

$$(E\text{-Proj1}) \frac{}{(M_1, M_2).1 \rightarrow M_1}$$

$$(E\text{-Proj2}) \frac{}{(M_1, M_2).2 \rightarrow M_2}$$

Typing rules for pairs in the λ -calculus are given in Fig. 2.12.

Figure 2.12: Typing rules for pairs

$$(T\text{-Proj1}) \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash M.1 : \tau_1}$$

$$(T\text{-Proj2}) \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash M.2 : \tau_2}$$

$$(T\text{-Pair}) \frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

The actual encoding of cursor contexts can be seen in Fig. 2.13. For short, in the following encodings, the cursor context is also given a type alias $Ctx = s \times s$.

Figure 2.13: Encoding of cursor contexts

$$\llbracket C[a] \rrbracket = (\llbracket a \rrbracket, \llbracket C \rrbracket)$$

$$\llbracket [\cdot] \rrbracket = [\cdot]$$

Atomic Prefix Commands

To encode the atomic prefix commands $\pi \in Apc$, it is necessary to extend the λ -calculus with pattern matching. The abstract syntax of this extension is given in Fig. 2.14.

Figure 2.14: Abstract syntax of extended λ -calculus

M, N	$::= \text{match } M \xrightarrow{\quad} \vec{N}$	(match construct)
p	$::= x$	(variable)
	$ _$	(wildcard)
	$ o \vec{p}$	(operator)
	$ (p_1, p_2)$	(pair)
	$.p$	(binding)

The reduction rules for the match construct are given in Fig. 2.15, which uses an auxiliary function *binds* which takes a term M and a pattern p and returns a function of variable bindings, e.g. $[x \rightarrow M]$ if a term M can be bound to pattern x , or *fail* if the term cannot be bound to the pattern.

Figure 2.15: Reduction rules for pattern matching

$$(E - \text{Match}) \frac{\sigma_i = \text{binds}(M, p_i) \neq \text{fail} \quad \forall j < i. \text{binds}(M, p_j) = \text{fail}}{\text{match } M \xrightarrow{\quad} \vec{N} \rightarrow N_i \sigma_i}$$

$$\text{binds} : M \times p \rightarrow (\text{Var} \rightarrow M) \cup \{\text{fail}\}$$

$$\begin{aligned} \text{binds}(M, x) &= [x \rightarrow M] \\ \text{binds}(M, _) &= [] \\ \text{binds}(o \ a_1 \ \dots \ a_n, \ o \ p_1 \ \dots \ p_n) &= \text{binds}(a_1, p_1) \circ \dots \circ \text{binds}(a_n, p_n) \\ \text{binds}((M, N), (p_1, p_2)) &= \text{binds}(M, p_1) \circ \text{binds}(N, p_2) \\ \text{binds}(M, .p) &= \text{binds}(M, p) \\ \text{binds}(\lambda x. M, .p) &= \text{binds}(M, .p) \end{aligned}$$

for remaining values in the domain of *binds* the result is defined as *fail*.
fail is defined as the function that always returns *fail*.

The typing rules for the match construct are given in Fig. 2.16.

Figure 2.16: Typing rules for pattern matching

$$(T\text{-Match}) \frac{\text{for all } i \text{ satisfying } \sigma_i = \text{binds}(M, p_i) \neq \text{fail} \quad \Gamma \vdash N_i \sigma_i : T}{\Gamma \vdash \text{match } M \xrightarrow{\quad} \vec{N} : T}$$

With the *match* construct, auxiliary functions for cursor movement (Fig. 2.17) can be defined in terms of matching an abt x against a pattern with some cursor, resulting in a new term. The shorthand $[a]$ is used to represent an abt a being encapsulated with a cursor.

Figure 2.17: Auxiliary functions for cursor movement

$$\begin{aligned}
\text{down} &\stackrel{\text{def}}{=} \lambda x : s.\text{match } x \\
&\quad [o(.a_1) \dots (.a_n)] \rightarrow o(. [a_1]) \dots (.a_n) \\
\text{right} &\stackrel{\text{def}}{=} \lambda x : s.\text{match } x \\
&\quad o(.a_1) \dots (. [a_i]) \dots (.a_n) \rightarrow o(.a_1) \dots ([a_{i+1}]) \dots (.a_n) \\
\text{up} &\stackrel{\text{def}}{=} \lambda x : s.\text{match } x \\
&\quad o(.a_1) \dots (. [a_i]) \dots (.a_n) \rightarrow [o(.a_1) \dots (.a_n)] \\
\text{set} &\stackrel{\text{def}}{=} \lambda a : s. \lambda x : s.\text{match } x \\
&\quad [a'] \rightarrow [a]
\end{aligned}$$

The encoding of atomic prefix commands $\pi \in \text{Apc}$ is done in terms of the cursor movement functions in Fig. 2.18. The *child* n is a recursive encoding, where the *right* function is applied on *child* $n - 1$, until n becomes 1, where it will be encoded as *down*.

Figure 2.18: Encoding of cursor movement

$$\begin{aligned}
\llbracket \text{child } 1 \rrbracket &= \text{down} \\
\llbracket \text{child } n \rrbracket &= \text{right } \llbracket \text{child } n - 1 \rrbracket \\
\llbracket \text{parent} \rrbracket &= \text{up} \\
\llbracket \text{insert } a \rrbracket &= \text{set } [a]
\end{aligned}$$

Editor expressions

To encode editor expressions $E \in \text{Edt}$, it is necessary to introduce recursion and a boolean base type to the extended λ -calculus.

A *fix* operator (Fig. 2.19) is introduced to support recursion, where E-FixBeta substitutes the term x in M with another *fix* operator, hence introducing a layer of recursion.

Figure 2.19: *fix* operator

$$\begin{array}{c}
\text{(E-FixBeta)} \quad \frac{}{fix(\lambda x : T.M) \rightarrow M[x := fix(\lambda x : T.M)]} \\
\\
\text{(E-Fix)} \quad \frac{M \rightarrow M'}{fix M \rightarrow fix M'}
\end{array}$$

The boolean term constants and base types are defined directly in the abstract syntax of the λ -calculus (Fig. 2.20). The patterns p terms have also been extended with the boolean constants, in order to support pattern matching for booleans. For this, the *binds* function has also been extended (Fig. 2.21).

Figure 2.20: Abstract syntax of extended λ -calculus

$$\begin{array}{lcl}
\text{Terms} & & \\
M & ::= & fix M \quad (\text{fixed point of } M) \\
& | & \top \quad (\text{true}) \\
& | & \perp \quad (\text{false}) \\
\\
p & ::= & \top \quad (\text{match true}) \\
& | & \perp \quad (\text{match false}) \\
\\
\text{Types} & & \\
\tau & ::= & Bool \quad (\text{boolean})
\end{array}$$

Figure 2.21: *binds* extended with boolean constructs

$$\begin{array}{l}
binds(\top, \top) = [] \\
binds(\perp, \perp) = []
\end{array}$$

Typing rules for *fix* and booleans can be seen in Fig. 2.22.

Figure 2.22: Typing rules for *fix* and booleans

$$\begin{array}{c}
\text{(T-Fix)} \quad \frac{\Gamma \vdash M : T \rightarrow T}{\Gamma \vdash fix M : T} \\
\\
\text{(T-False)} \quad \frac{}{\perp : Bool} \\
\\
\text{(T-True)} \quad \frac{}{\top : Bool}
\end{array}$$

With prior definitions, encoding of editor expressions and context configuration is

given in Fig. 2.23.

Figure 2.23: Encoding of editor expressions and context configuration

$$\llbracket \pi.E \rrbracket = \lambda C C : Ctx. \llbracket E \rrbracket ((\llbracket \pi \rrbracket C.1), C.2)$$

$$\llbracket nil \rrbracket = \lambda C : Ctx. C$$

$$\llbracket E_1 \gg E_2 \rrbracket = \lambda C : Ctx. \llbracket E_2 \rrbracket (\llbracket E_1 \rrbracket C)$$

$$\llbracket \text{Rec } x.E \rrbracket = \text{fix}(\lambda x : (Ctx \rightarrow Ctx). \llbracket E \rrbracket)$$

$$\begin{aligned} \llbracket \phi \Rightarrow E_1 | E_2 \rrbracket &= \lambda C : Ctx. \text{match}(\llbracket \phi \rrbracket C.1) \\ &\quad | \top \rightarrow \llbracket E_1 \rrbracket C \\ &\quad | \perp \rightarrow \llbracket E_2 \rrbracket C \end{aligned}$$

$$\llbracket \langle E, C[a'] \rangle \rrbracket = \llbracket E \rrbracket (\llbracket a \rrbracket, \llbracket C \rrbracket)$$

Conditional expressions

To encode conditional expressions $\phi \in \text{Eec}$, auxiliary functions *checkboth*, *checkone*, *or*, *and* and *neg* are defined in Fig. 2.24.

Figure 2.24: Auxiliary functions for conditionals

$$\begin{aligned}
 \text{checkboth} & \stackrel{\text{def}}{=} \lambda f : (Bool \rightarrow Bool \rightarrow Bool). \\
 & \quad \lambda g : (s \rightarrow Bool). \\
 & \quad \lambda h : (s \rightarrow Bool). \\
 & \quad \lambda a : s. \\
 & \quad f(ga)(ha) \\
 \\
 \text{checkone} & \stackrel{\text{def}}{=} \lambda f : (Bool \rightarrow Bool). \\
 & \quad \lambda g : (s \rightarrow Bool). \\
 & \quad \lambda a : s. \\
 & \quad f(ga) \\
 \\
 \text{or} & \stackrel{\text{def}}{=} \lambda b_1 : Bool. \lambda b_2 : Bool. \text{match } (b_1, b_2). \\
 & \quad (\perp, \perp) \rightarrow \perp \\
 & \quad (_, _) \rightarrow \top \\
 \\
 \text{and} & \stackrel{\text{def}}{=} \lambda b_1 : Bool. \lambda b_2 : Bool. \text{match } (b_1, b_2). \\
 & \quad (\top, \top) \rightarrow \top \\
 & \quad (_, _) \rightarrow \perp \\
 \\
 \text{neg} & \stackrel{\text{def}}{=} \lambda b : Bool. \text{match } b. \\
 & \quad \top \rightarrow \perp \\
 & \quad \perp \rightarrow \top
 \end{aligned}$$

With the auxiliary functions, the encoding of propositional connectives is given in Fig. 2.25 and the encoding of modal logic is given in Fig. 2.26, which also make use of the *fix* and *match* functions.

Figure 2.25: Encoding of propositional connectives

$$\llbracket \phi_1 \wedge \phi_2 \rrbracket = \text{checkboth and } \llbracket \phi_1 \rrbracket \llbracket \phi_2 \rrbracket$$

$$\llbracket \phi_1 \vee \phi_2 \rrbracket = \text{checkboth or } \llbracket \phi_1 \rrbracket \llbracket \phi_2 \rrbracket$$

$$\llbracket \neg \phi \rrbracket = \text{checkone neg } \llbracket \phi \rrbracket$$

Figure 2.26: Encoding of modal logic

$$\begin{aligned} \llbracket @o \rrbracket &= \lambda x : s.\text{match } x \\ &\quad | [o _ \dots _] \rightarrow \top \\ &\quad | _ \rightarrow \perp \end{aligned}$$

$$\begin{aligned} \llbracket \Diamond o \rrbracket &= \text{fix}(\lambda f : (s \rightarrow \text{Bool}). \lambda x : s.\text{match } x \\ &\quad | [o _ \dots _] \rightarrow \top \\ &\quad | [_ (_ . a_1, \dots, _ . a_n)] \rightarrow \text{or}(\dots(\text{or}(f [a_1])(f [a_2])) \dots)(f [a_n]) \\ &\quad | _ \rightarrow \perp) \end{aligned}$$

$$\begin{aligned} \llbracket \Box o \rrbracket &= \lambda x : s.\text{match } x \\ &\quad | [_ (_ . a_1, \dots, _ . a_n)] \rightarrow \text{and}(\dots(\text{and}(\llbracket \Diamond o \rrbracket a_1)(\llbracket \Diamond o \rrbracket a_2)) \dots)(\llbracket \Diamond o \rrbracket a_n) \\ &\quad | _ \rightarrow \top \end{aligned}$$

The Aalborg project[16] also includes a proof of the soundness of the encoding, which is omitted here.

Given the encoding of the generalized editor calculus in the extended λ -calculus, the same encoding can be followed to encode the generalized editor calculus in a language like Elm or Haskell, since they are based on the principles of the λ -calculus and support pattern matching, pairs and recursion.

Chapter 3

Implementation

This chapter describes the implementation of the type-safe generalized syntax-directed editor, which serves as a proof of concept for the proposed editor calculus. The implementation aims to follow the calculus as closely as possible, maintaining the given guarantees.

3.1 Implementation

3.1.1 Representing Syntax

The generalized editor calculus[16] assumes that it is given an abstract syntax that is represented by a set of sorts \mathcal{S} , an arity-indexed family of operators \mathcal{O} , and a sort-indexed family of variables \mathcal{X} , as per Robert Harper’s notation [6].

A criterion for the good solution of this project is also the implementation of being able to pretty-print a program into a concrete syntax. For this, it is also necessary for the user to provide the concrete for a language they wish to edit.

It can be a challenge from the user’s perspective to provide a specification based on what the calculus assumes. Therefore, it is ideal for the implementation to provide other means of describing the syntax of a language.

In terms of early examples, Metal[14] has been used in the Mentor[3] and CENTAUR[1] systems. Metal compiles a specification containing concrete syntax, abstract syntax and tree building functions for a formalism F into a Virtual Tree Processor (VTP) formalism, a concrete syntax parser produced by YACC[11] and a tree generator which uses VTP primitives to construct abstract syntax trees.

Another example with the same purpose is Zephyr ASDL (Abstract Syntax Description Language)[24], where the authors have built a tool that converts an

ASDL specification into C, C++, Java and ML data-structure definitions. The authors consider ASDL a simpler alternative to other abstract syntax description languages, such as ASN.1[15].

However, both examples have lack of binding mechanisms in abstract syntax in common. This motivates another possibility of defining a specification language for the to-be-implemented generalized editor itself, which can assist the user in describing the syntax. This would also require a parser that can parse the necessary information assumed by the calculus (including binders). Picking this route allows the project to avoid spending time analyzing different tools and developing a workaround for binders.

Specification language

The specification language is chosen to expect some syntactic categories followed by concrete and abstract syntax in BNF notation. Every syntactic category is represented by one or more non-terminals with a term, arity and operator name. The term might refer to other syntactic categories or its own. Each term is the concrete syntax of an operator, while the arity, in combination with the operator name, is a concise representation of the abstract syntax of an operator. The abstract syntax only makes use of the defined syntactic categories and, inspired by Harper[6], binders can be specified in the arity description with a dot ('), e.g. $x.s$ specifies that variable x is bound within the scope of s .

Meta-variables can be represented in the concrete syntax (term) with a percentage sign (%) followed by a string, e.g. '%int' represents any integer. The operator names can also be supplied with type information, e.g. an operator named num can be followed by [Int], resulting in num[Int], to specify that it represents a number of type Int. This can later be used in source code generation to allow for passing concrete values of given type to the operator. But this only applies for operators of arity $()x$, where x is any sort.

From this specification language, it is possible to extract what is assumed by the generalized editor calculus[16]. The set of sorts \mathcal{S} is the set of syntactic categories. For example, a syntactic category $e \in Exp$ can get parsed into a sort s_e . The family of arity-indexed operators \mathcal{O} can be extracted from the BNF notation since each derivation rule represents an operator with a specified arity.

Example 5: Syntax of a small SQL language

Following is a specification of a subset of the PostgreSQL[5] language, per the specification language

$$\begin{aligned} q &\in \text{Query} \\ cmd &\in \text{Command} & id &\in \text{Id} \\ const &\in \text{Const} & clause &\in \text{Clause} \\ cond &\in \text{Condition} & exp &\in \text{Expression} \end{aligned}$$

Sort	Term	Arity	Operator
$query ::=$	"SELECT " id_1 " FROM " id_2 $clause$	$(id_1, id_2, clause)query$	$select$
$cmd ::=$	"INSERT INTO " id_1 " AS " id_2 $query$	$(id_1, id_2.query)cmd$	$insert$
$id ::=$	%string	$()id$	$id[String]$
$const ::=$	%number	$()const$	$num[Int]$
	""" id """	$(id)const$	str
$clause ::=$	"WHERE " $cond$	$(cond)clause$	$where$
	"HAVING " $cond$	$(cond)clause$	$having$
$cond ::=$	exp_1 ">" exp_2	$(exp_1, exp_2)cond$	$greater$
	exp_1 "=" exp_2	$(exp_1, exp_2)cond$	$equals$
$exp ::=$	$const$	$(const)exp$	$econst$
	id	$(id)exp$	$eident$

The $id[String]$ and $num[Int]$ operators represent meta-variables and can be used to represent concrete values of the given type in Elm.

This subset is chosen as it can represent simple select queries and insert commands.

Notably, a binder is used in the $insert$ operator, where the alias of id_1 , specified as id_2 is bound within the sub-query.

Example 6: Syntax of a small L^AT_EX language

Below is the syntax of a subset of the L^AT_EX language:

$$\begin{array}{ll} d \in \textit{Document} & e \in \textit{Environment} \\ cmd \in \textit{Command} & a \in \textit{Argument} \\ id \in \textit{Identifier} & c \in \textit{Content} \end{array}$$

Sort	Term	Arity	Operator
$d ::=$	$\text{"\documentclass{" id "}"}$ $\text{"\newenvironment{" id "}"}$ $a_1 a_2$ $\text{"\begin{document}" } c$ $\text{"\end{document}"}$	$(id, a, a, id.c)d$	$document$
$id ::=$	$\%string$	$()id$	$ident[String]$
$e ::=$	$\text{"\begin{" id "}" } c$ $\text{"\end{" id "}"}$	$(id, c, id)e$	$environment$
$cmd ::=$	"\ " id a	$(id, a)cmd$	$command$
$a ::=$	$\text{"{ " c "}"}$	$(c)a$	$argument$
$c ::=$	$\%text$	$()c$	$textContent[String]$
	$ \quad cmd$	$(cmd)c$	$cmdContent$
	$ \quad e$	$(e)c$	$envContent$
	$ \quad c \ c$	$(c, c)c$	$seqContent$

'%text' and '%identifier' are meta-variables representing any parseable text and identifier in the L^AT_EX language.

This subset is chosen as it can represent a minimal L^AT_EX document with an environment definition, where the name of the environment is bound within the body of the document to demonstrate the use of binders in the abstract syntax.

The $ident[String]$ and $textContent[String]$ operators represent meta-variables and can be used to represent concrete values of the given type in Elm.

Example 7: Syntax of a small C language

Following is a specification of a subset of the C language[10], per the described specification language.

$p \in Prog$	$s \in Stmt$
$vd \in VariableDecl$	$fd \in FunDecl$
$t \in Type$	$id \in Id$
$e \in Exp$	$b \in Block$
$fa \in Funarg$	$cond \in Conditional$

function declarations at the top level, where one of them might represent a main function, the entry point of a C program. The identifier of a function declaration is bound within the following function declarations (e.g. id_1 is bound within fd in the *fundecl1* operator).

A limitation of this specification is recursive function calls. Ideally, the identifier in a function declaration is bound both within the function block and the sequence of following function declarations. However, this would result in the same identifier appearing twice in the arity definition. E.g. the arity for the *fundecl1* operator is $(t_1, id_1.f, t_2, id_2.b).fd$ where id_1 is the identifier of the function, which ideally would be bound in both fd (the following sequence of function declarations) and b (the function block).

Another limitation, or something that might seem unnecessary, is having the *blockdone* and *fundecldone* operators. They are necessary to allow for a block to end with a *vardecl* operator and to end a sequence of function declarations with the *fundecl1* or *fundecl2* operator. This is a pattern that allows operators to bind identifiers within the following terms.

To make the specification language parseable, a more computer-friendly format is presented in Listing 3.1 in BNF-form. Every syntactic category is expected on its own line, followed by a blank line and all derivations. Each derivation is expected to be a syntactic category, followed by '::<=' and every term (which acts as the concrete syntax of an operator), arity and operator name, separated with a vertical bar '|'. Every term, arity and operator are separated with a number-sign '#'.

```

1 <syntax> ::= <synCatList> "\n\n" <synCatRulesList>
2
3 <synCatList> ::= <synCat> "\n" <synCatList> | <synCat>
4 <synCat> ::= <synCatExp> "in" <synCatSet>
5
6 <synCatExp> ::= %string
7 <synCatSet> ::= %string
8
9 <synCatRulesList> ::= <synCatRules> "\n"
10                      <synCatRulesList>
11                      | <synCatRules>
12 <synCatRules> ::= <synCatName> "::<=" <ruleList>
13 <ruleList> ::= <rule> "|" <ruleList> | <rule>
14
15 <rule> ::= <term> "#" <arity> "#" <operator>
16 <term> ::= %string
17 <arity> ::= "(" <arityElementList> ")"
18 <arityElementList> ::= <arityElement>
19                      | <arityElement>
20                      <separator>
21                      <arityElementList>
22 <arityElement> ::= %string
23 <separator> ::= ", " | "."
24 <operator> ::= %string

```

Listing 3.1: The specification language itself in BNF-form

Example 8: Subset of syntax of a small SQL language in a parseable format

```

1 query in Query
2 cmd in Command
3 id in Id
4 clause in Clause
5
6 query ::= " SELECT " id " FROM " id clause # (id,id,clause)query # select
7 cmd ::= " INSERT INTO " id " AS " id query # (id,id.query)cmd # insert
8 ...

```

3.1.2 Code generation versus generic model

Another important thing to consider for the implementation is whether part of the editor's source code should be generated automatically, or if a generic model might suffice.

Automatic generation of source code offers the benefit of directly representing provided operators, along with their arity and sort, within an algebraic data type (referred to as type in Elm and data in Haskell). This ensures that only well-formed terms can be represented using the algebraic data types.

However, opting for this method might require automatic generation or updates to both the definitions and signatures of some functions. This presents a challenge in ensuring that these functions maintain their intended behavior after the updates.

Example 9: Algebraic data types for a small C syntax

Given example Example 11, one can generate the following custom types in Elm:

```
1 type alias Bind a b =  
2   ( List a, b )  
3 type Prog  
4   = Program FunDecls  
5 type Block  
6   = Block BlockItems  
7 type BlockItems  
8   = BlockDecl VarDecls  
9   | BlockStmts Statements  
10  | BlockDone  
11 type VarDecls  
12   = VarDecl Type Id Exp BlockItems  
13 type FunDecls  
14   = FunDecl1 Type (Bind Id FunDecls) Type (Bind Id Block)  
15   | FunDecl2 Type (Bind Id FunDecls) Type (Bind Id (Bind Id Block))  
16   | FunDeclDone  
17 ...
```

The *Bind* type alias is simply a tuple of 2 given type variables, where the first element is a list. This is used to represent binders in the abstract syntax, however with the limitation of only allowing abts of a single sort to be bound.

In contrast, a generic solution without the need for generating new source reduces the risk of syntax errors in the target language for the editor itself. A generic solution might involve designing a model capable of holding the necessary information from any syntax. An approach for this is provided in Listing 3.2. The arity in the Operator type is a tuple, where the first entry is a potential list of identifiers bound

within the term in the second element of the tuple. In Haskell, this corresponds to named fields[7], which have a very similar syntax.

However, this approach requires additional well-formedness checks on any given term concerning the syntax.

```
1 type alias Syntax =  
2 { synCats : [String]  
3   , operators : [Operator]  
4   }  
5  
6 type alias Operator =  
7 { name : String  
8   , arity : (Maybe [String], String)  
9   , concSyn : String  
10 }
```

Listing 3.2: Elm Records for storing syntax information

The implementation will proceed with automatic generation of source code, including algebraic data types, due to their advantage in handling ill-formed terms effectively. If given an ill-formed term, it is considered ill-typed by the editor, which poses an advantage over the generic solution requiring thorough checking to ensure that given terms are well-formed.

3.1.3 Generating source code

Elm CodeGen[18] is an Elm package and CLI tool (command-line interface tool) to generate Elm source code. The tool is an alternative to the otherwise obvious (and arguably tedious) strategy of having a source code template, where certain placeholders are replaced with relevant data or code snippets associated with the parsed syntax.

Besides offering the ability to generate source code, it offers automatic imports and built-in type inference. Example usage of Elm CodeGen from the documentation[18] is given in Fig. 3.1.

Figure 3.1: Elm CodeGen usage

Following declares an Elm record and passes it to a ToString function:

```
1 Elm.declaration "anExample"  
2   (Elm.record  
3     [ ("name", Elm.string "a fancy string!")  
4       , ("fancy", Elm.bool True)  
5     ]  
6   )  
7   |> Elm.ToString.declaration
```

The above will generate following string:

```
1 anExample : { name : String, fancy : Bool }  
2 anExample =  
3   { name = "a fancy string!"  
4     , fancy = True  
5   }
```

Using Elm CodeGen offers the advantage of integrating a parser, which, if implemented in Elm as well, can directly produce the data to enable Elm Codegen to produce source files for the editor.

More specifically, the implementation will use the built-in Parser Elm library to parse a RawSyntax object (Listing 3.3), which is a direct representation of the syntax as specified in the specification language (Listing 3.1). The RawSyntax represents the collection of syntactic categories RawSyncat and RawSynCatOps, where the latter contains a list of operators RawOp for each syntactic category.

```

1 type alias RawSyntax =
2   { synCats : List RawSynCat
3     , synCaOps : List RawSynCatOps
4   }
5
6 type alias RawSynCatOps =
7   { synCat : String
8     , operators : List RawOp
9   }
10
11 type alias RawOp =
12   { term : String
13     , arity : String
14     , name : String
15   }
16
17 type alias RawSynCat =
18   { exp : String
19     , set : String
20   }

```

Listing 3.3: Raw syntax model in Elm

If parsing of the raw syntax is successful, the raw model will be transformed into a separate model built around the `Syntax` type alias (Listing 3.4). Transformations include converting the string-representation of the arity into its own `Arity` type, which is a simple list of tuples, where the first element is a list of variables to be bound within the second element (line 31-32 in Listing 3.4).

```

1  type alias Syntax =
2      { synCats : List SynCat
3        , synCatOps : List SynCatOps
4      }
5
6  type alias SynCat =
7      { exp : String
8        , set : String
9      }
10
11 type alias SynCatOps =
12     { ops : List Operator
13       , synCat : String
14     }
15
16 type alias SynCatOps =
17     { ops : List Operator
18       , synCat : String
19     }
20
21 type alias Operator =
22     { term : Term
23       , arity : Arity
24       , name : String
25       , synCat : String
26     }
27
28 type alias Term =
29     String
30
31 type alias Arity =
32     List ( List String, String )

```

Listing 3.4: Syntax model

Having all expected sets of sorts and family of operators, i.e. the abstract syntax extended with *hole* and *cursor* operator \mathcal{S}, \mathcal{O} , cursorless trees $\hat{\mathcal{O}}, \hat{\mathcal{S}}$, cursor contexts $\mathcal{S}^C, \mathcal{O}^C$ and well-formed trees $\check{\mathcal{O}}, \check{\mathcal{S}}$, the CodeGen package can generate algebraic data types for every sort and its operators and separate them into their own separate modules (files).

Example 10: From specification parser to Elm CodeGen for small C-language

Given the specification in example Example 11, the parser can produce following Declaration for the Statement algebraic data type in example Example 9:

```

1 Elm.customType "Statement"
2   [ Elm.variantWith "Assignment"
3     [ Elm.Annotation.named [] "Id",
4       Elm.Annotation.named [] "Exp" ]
5   , Elm.variantWith "StmtFunCall"
6     [ Elm.Annotation.named [] "Id",
7       Elm.Annotation.named [] "Funargs" ]
8   , Elm.variantWith "Return"
9     [ Elm.Annotation.named [] "Exp" ]
10  , Elm.variantWith "Conditional"
11    [ Elm.Annotation.named [] "Conditional" ] ]

```

This declaration, if passed to Elm CodeGen’s File function, would generate a source file with following contents:

```

1 type Statement
2   = Assignment Id Exp
3   | StmtFunCall Id Funargs
4   | Return Exp
5   | Conditional Conditional

```

3.1.4 Editor expressions

Having generated all sorts and operators as algebraic data types in Elm, the next step is to implement functionality for manipulating the algebraic data types, representing abt’s in editor. In terms of the editor calculus[16], manipulating an abt involves editor expressions $E \in Edt$ (Fig. 2.1), which include atomic prefix commands $\pi \in Apc$ for abt traversal and modification, and conditions $\phi \in Eec$ for conditional manipulation with respect to modal logic within the encapsulated abt.

Decomposing trees

All editor expressions assume a well-formed abt, which is a tree with exactly one cursor, which cannot be guaranteed by the generated algebraic data types for sorts \mathcal{O} , as every sort has an associated cursor operator.

However, these operations rely on a well-formed abt, which is given by decomposing an abt a into $C[\dot{a}]$, where C is a cursor-context (Definition 3) and \dot{a} is a well-formed abt (Definition 4).

First, it is necessary to know which symbol in the given syntax representation is intended as the starting symbol. Only the syntactic category of the starting symbol would need functions supporting decomposition. In other words, it does not make sense to decompose an abt of some sort which cannot be derived to a well-formed program. This aspect has not been considered yet, and as a temporary solution, a type wrapper called Base has been introduced, which has an entry for every syntactic category in the given syntax.

Example 11: Example of the Base type for the C language

```

1  type Base
2    = Prog Prog
3    | Block Block
4    | BlockItems BlockItems
5    | VarDecls VarDecls
6    | FunDecls FunDecls
7    ...

```

This allows for any operator of any sort to be decomposed, although the ideal solution would be having a way to specify the starting symbol in the syntax representation.

The generalized editor calculus[16] defines well-formed trees as abt's with exactly one cursor either as the root or as an argument of the root. However, this definition, in conjunction with the cursor context definition, leads to multiple valid decompositions for an abt in certain scenarios, which is also mentioned in the Aalborg project[16]. This can occur when the cursor is located at one of the immediate children of the root. In that case, the cursor context can be interpreted as either the tree where the cursor has been replaced with a context hole, or it can be interpreted as an empty context. For example, consider a small tree created from the syntax given in example Example 5 (including cursor and hole operators) and their two possible decompositions in figure Fig. 3.2.

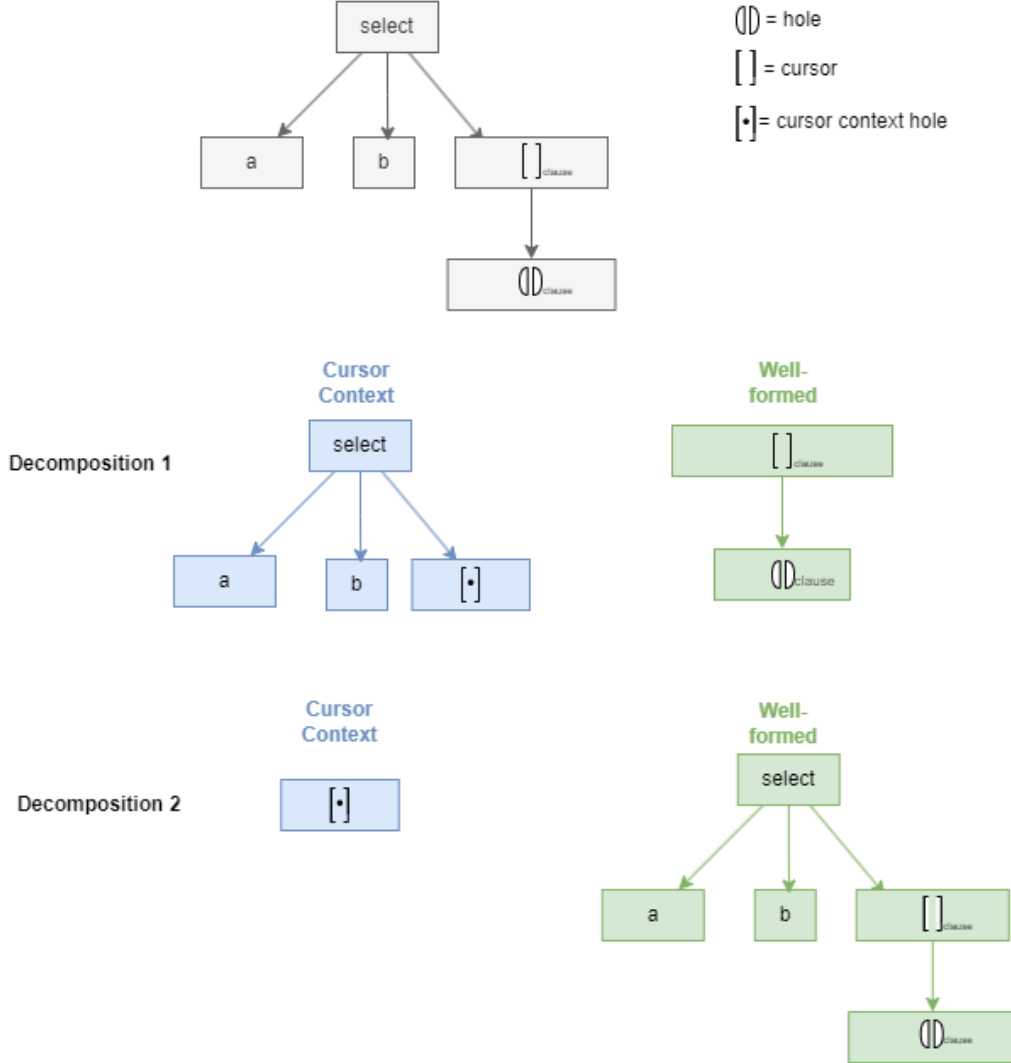


Figure 3.2: Two different decompositions of the same term

In order to generate a decomposition function which works for any sort, it is necessary to specify how we for any term in any sort, can decompose uniquely into a cursor context and well-formed-tree pair.

Unique decomposition of an abt can be defined algorithmically and divided into following sub-tasks:

- Locate the cursor in the tree to be decomposed and generate a path to the cursor
- Generate an abt of sort $s^C \in \mathcal{S}^C$ based on the cursor path

- Generate an abt of sort $\dot{s} \in \dot{\mathcal{S}}$ based on the rest of the tree that was not traversed when generating the cursor context

The next section will explain in more details how the steps above can be done, and how we always get a unique decomposition, as long as the abt to be decomposed is well-formed.

This way of decomposing an abt always places the cursor at the root of the well-formed tree. This makes it impossible for the cursor to encapsulate the immediate child of the root, making the property 3 in the definition of well-formed trees (Definition 4) redundant. For this reason, the code generation has been revised to only generate an operator of arity $(\hat{s})\dot{s}$ (property 4), representing that the cursor only encapsulates the root of a cursorless abt of sort \hat{s} .

Cursor path

Generating a path to the cursor in the abt of sort $s \in \mathcal{S}$ extended with hole and cursor operators (Definition 1) simplifies the process of generating the cursor context. The path tells us which operator $o^C \in \mathcal{O}^C$ replaces each $o \in \mathcal{O}$. The list can be generated by performing pre-order traversal of the tree to be decomposed, extending the list with every i , representing which argument in an operator of arity $(\vec{s}_1.s_1, \dots, \vec{s}_i.s_i, \dots, \vec{s}_n.s_n)\mathcal{S}$ was followed to locate the cursor. See Listing 3.5 for pseudocode demonstrating this process.

```

1 getCursorPath op path =
2   case op of
3     cursor -> path
4   _ ->
5     case length op.arity of
6       0 -> []
7     _ ->
8       map ++
9         (index_map
10          (\i child ->
11            getCursorPath child (path ++ [i]))
12          op.arity)

```

Listing 3.5: Pseudocode for generating cursor path

The implementation of such a function depends on the set of sorts \mathcal{S} and arity-indexed family of operators \mathcal{O} given by the abstract syntax of a language. The Elm CodeGen package[18] has been used to generate a `getCursorPath` function for a Base type (Listing 3.6). The function makes use of the helper function `getBranchList` which is left out for brevity, but its purpose is to generate a case expression for every syntactic category in the given syntax. See Example 12 for an excerpt of the generated `getCursorPath` function for the small C language (Example 11).

```

1 createGetCursorPath : Syntax -> Elm.Declaration
2 createGetCursorPath syntax =
3   Elm.declaration "getCursorPath" <|
4     Elm.withType
5       (Type.function
6         [ Type.list Type.int
7           , Type.named [] "Base"
8         ]
9         (Type.list Type.int)
10      )
11     (Elm.fn2
12       ( "path", Nothing )
13       ( "base", Nothing )
14       (\_ base ->
15         Elm.Case.custom base
16           (Type.named [] "Base")
17           (getBranchList syntax)
18       )
19     )

```

Listing 3.6: getCursorPath function generator

Example 12: Generated cursor path finder function

Following is an excerpt of the generated `getCursorPath` function for the small C language:

```
1 getCursorPath : List Int -> Base -> List Int
2 getCursorPath path base =
3   case base of
4     P p ->
5       case p of
6         Program arg1 ->
7           getCursorPath (path ++ [ 1 ]) (Fd arg1)
8
9         Hole_p ->
10          []
11
12        Cursor_p _ ->
13          path
14
15      ...
16    Vd vd ->
17      case vd of
18        Vardecl arg1 arg2 ( boundVars3, arg3 ) ->
19          (getCursorPath
20            (path ++ [ 1 ])
21            (T arg1)
22            ++ getCursorPath
23              (path ++ [ 2 ])
24              (E arg2)
25            )
26          ++ getCursorPath
27            (path ++ [ 3 ])
28            (Bi arg3)
29
30        Hole_vd ->
31          []
32
33        Cursor_vd _ ->
34          path
```

Notable for this example is the `Vardecl` branch, which represents an operator with three arguments, hence the need for three recursive calls to `getCursorPath`. The third argument is deconstructed since it is a binder, although the bound variables are not used in the cursor path generation, as the cursor cannot be in a bound variable.

Cursor context

Having the cursor path, the cursor context can be generated by replacing every operator $o \in \mathcal{O}$ with its corresponding cursor context operator $o^C \in \mathcal{O}^C$, with respect to which argument in the operator was followed to locate the cursor. This is also done by performing pre-order traversal of the tree, but it will stop when the cursor is reached (i.e. when the cursor path is empty) and replace the operator reached with the context hole operator $[\cdot] \in C$. The rest of the tree which has not been traversed will be passed to the next step, building the well-formed tree. See Listing 3.7 for pseudocode demonstrating this process.

```

1 toCCtx op path =
2   case path of
3     [] -> (context_hole, op)
4     pathi :: pathrest ->
5       case op of
6         cursor -> error "invalid path"
7         _ ->
8           case length op.arity of
9             0 -> error "invalid path"
10            _ ->
11              case pathi of
12                1 ->
13                  let (cctxarg1, restTree) =
14                      toCCtx op.args[1] rest
15                  in
16                    (Cctx1_opName cctxarg1
17                     , restTree)
18                2 ->
19                  let clessarg1 =
20                      toCLess op.args[2]
21                  let (cctxarg2, restTree) =
22                      toCCtx op.args[2] rest
23                  in
24                    (Cctx2_opName clessarg1 cctxarg2
25                     , restTree)
26                ...
27 toCLess op =
28   case op of
29     Cursor _ -> error "not wellformed"
30     _ ->
31       case length op.arity of
32         0 -> CLess_opName
33         1 -> CLess_opName
34             (toCLess op.args[1])
35         2 -> CLess_opName
36             (toCLess op.args[1])
37             (toCLess op.args[2])

```

Listing 3.7: Pseudocode for building cursor context

Functions supporting building cursor contexts are generated by Elm CodeGen, and can be seen in Listing 3.8, where a `toCCtx` function is generated for the `Base` type in conjunction with a `toCCtx_s` for every s syntactic category in the given syntax, where the algorithm described above is implemented for every sort.


```

1 createToCCtxFuns : Syntax -> List Elm.Declaration
2 createToCCtxFuns syntax =
3   List.map createToCCtxFun syntax.synCatOps ++
4   [ Elm.declaration "toCCtx" <|
5     Elm.withType
6       (Type.function
7         [ Type.named [] "Base"
8           , Type.list Type.int ]
9         (Type.tuple
10          (Type.named [] "Cctx")
11          (Type.named [] "Base"))
12        ) <|
13     Elm.fn2
14       ( "base", Nothing )
15       ( "path", Nothing )
16       (\base path ->
17         Elm.Case.custom base
18           (Type.named [] "Base")
19           (List.map
20             (\synCatOp ->
21               Elm.Case.branchWith
22                 synCatOp.synCat
23                 1
24                 (\exps ->
25                   Elm.apply
26                     (Elm.val <|
27                       "toCCtx_" ++ synCatOp.synCat
28                       (exps ++ [ path ]))
29                   )
30             )
31             syntax.synCatOps
32           )
33       )
34 ]

```

Listing 3.8: toCCtx function generator

Well-formed tree

The well-formed tree is generated by performing pre-order traversal of the rest of the tree that was not traversed when generating the cursor context. This is done by first replacing the cursor operator $o \in \mathcal{O}$ with the well-formed operator $\hat{o} \in \hat{\mathcal{O}}$ of arity $(\hat{s})\hat{s}$ indicating that the cursor encapsulates a cursorless abt of sort \hat{s} , being the cursorless equivalent of the child of o . After this, the rest of the tree is traversed, and every operator $o \in \mathcal{O}$ is replaced with its corresponding cursorless operator $\hat{o} \in \hat{\mathcal{O}}$. See Listing 3.9 for pseudocode demonstrating this process.

```

1 toWellformed op =
2   let op = consumeCursor op
3   in
4     case length op.arity of
5       0 -> error "not wellformed"
6       1 -> Cursor_opName
7           (toCLess op.args[1])
8       2 -> Cursor_opName
9           (toCLess op.args[1])
10          (toCLess op.args[2])
11     ...
12
13 consumeCursor op =
14   case op of
15     Cursor under -> under
16     _ -> op

```

Listing 3.9: Pseudocode for generating well-formed tree

Like when generating functions supporting cursor context, a very similar approach is taken here, and can be seen in listing Listing [3.10](#).

```

1 createToWellFormedFun : Syntax -> Elm.Declaration
2 createToWellFormedFun syntax =
3   Elm.declaration "toWellformed" <|
4   Elm.withType
5     (Type.function
6       [ Type.named [] "Base" ]
7       (Type.named [] "Wellformed")) <|
8   Elm.fn
9     ( "base", Nothing )
10    (\base ->
11      Elm.Case.custom
12        (Elm.apply
13          (Elm.val "consumeCursor")
14          [ base ])
15        (Type.named [] "Base")
16        (List.map
17          (\synCatOp ->
18            branchWith synCatOp.synCat
19              1
20              (\exps ->
21                Elm.apply
22                  (Elm.val <|
23                    "Root_" ++
24                    synCatOp.synCat ++
25                    "_CLess")
26                  [ Elm.apply
27                    (Elm.val <|
28                      "toCLess_" ++
29                      synCatOp.synCat)
30                    exps
31                  ]
32                )
33              )
34          syntax.synCatOps
35        )
36    )

```

Listing 3.10: toWellFormed function generator

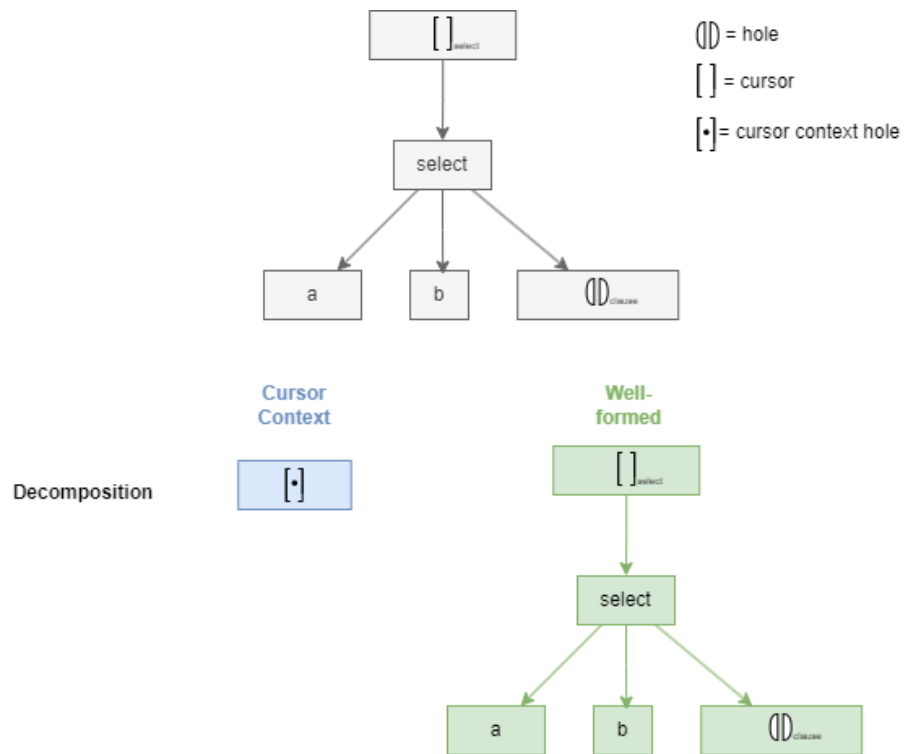
The cursor context and well-formed tree pair as defined above will decompose any well-formed abt into a unique pair of a cursor context and a well-formed tree.

Cursor movement - Child

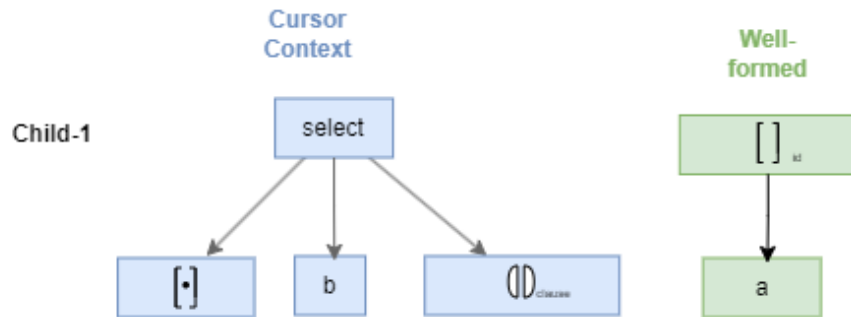
Cursor movement is a fundamental operation, allowing the editor to move the cursor to a different position, changing the target of editor expressions. This is defined as two operations in the editor calculus in the form of Child-i and Parent operations (Fig. 2.4).

For moving to child i of an operator, a child function is generated which takes an integer i and a decomposed tree represented by a tuple of a cursor context and well-formed tree. It returns a new tuple, where the context hole in the cursor context has been replaced with the operator of sort \mathcal{S}^C corresponding to the cursorless operator of sort $\hat{\mathcal{S}}$ under the cursor in the well-formed tree before moving the cursor. The integer i determines which argument of the cursor context operator contains the new context hole. The new well-formed tree is built by wrapping the i th argument of the cursorless operator current root operator in the well-formed tree.

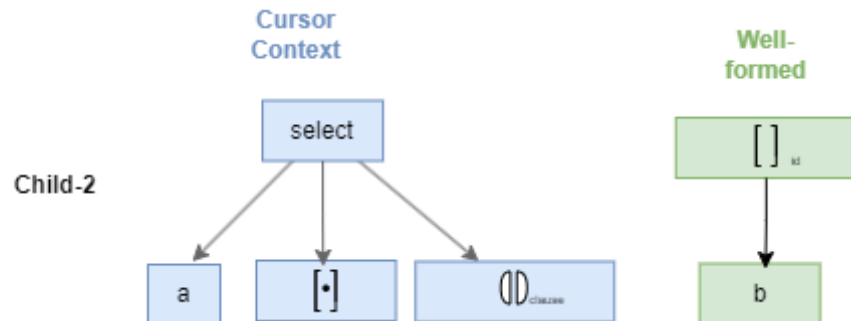
For a visualization of this, see Fig. 3.3 showing the result of applying various child operations to a decomposed tree. Pseudocode explaining the overall algorithm is provided in Listing 3.11.



(a) Decomposition of a small SQL program



(b) Applying the Child-1 operation



(c) Applying the Child-2 operation

Figure 3.3: Example of cursor movement

```

1 child(i, (cctx, wellformed)) =
2   case wellformed of
3     Root_Sort1 underCursor ->
4       case underCursor.arity of
5         0 -> error "operator has no children"
6         1 ->
7           case i of
8             1 ->
9               (replaceCCtxHole cctx i underCursor
10                -- X: sort of child
11                , Root_SortX underCursor.args[1])
12             _ -> error "Invalid child"
13         2 ->
14           case i of
15             1 ->
16               (replaceCCtxHole cctx i underCursor
17                , Root_SortX underCursor.args[1])
18             2 ->
19               (replaceCCtxHole cctx i underCursor
20                , Root_SortX underCursor.args[2])
21             _ -> error "Invalid child"
22         ...
23     Root_ClessOp2 ->
24     ...
25
26 replaceCCtxHole cctx i underCursor =
27   case cctx of
28     CCtxHole ->
29       case underCursor arg1 arg2 ... argn of
30         Root_Sort1 _ ->
31           case i of
32             1 -> CCtx1_Sort1 CctxHole ... argn
33             2 -> CCtx2_Sort1 arg1 CctxHole ... argn
34             ...
35         Root_Sort2 _ _ ->
36           case i of
37             1 -> CCtx1_Sort2 CctxHole ... argn
38             2 -> CCtx2_Sort2 arg1 CctxHole ... argn
39             ...
40         ...
41     CCtxOp1 arg1 .. argn ->
42       CCtxOp1
43         (replaceCCtxHole arg1 i underCursor)
44       ...
45       argn
46     CCtxOp2 args ->
47       CCtxOp2
48       arg1

```

```

49     (replaceCCtxHole arg2 i underCursor)
50     ...
51     argn
52     ...

```

Listing 3.11: Pseudocode for applying child operator

This has been implemented by having Elm CodeGen generate a child function which takes an integer i and a decomposed tree, and returns a new cursor context and well-formed tree pair, where the cursor has been moved to the i th child.

Cursor movement - Parent

Moving the cursor to the parent of the cursor is done by first updating the cursor context by moving the context hole a level up, such that the parent of the context hole becomes the new context hole. Then, the parent of the cursor context which has been updated (operator of sort \mathcal{S}^C) is inserted as its equivalent well-formed operator (operator of sort $\dot{\mathcal{S}}$) at the root of the well-formed tree, where the prior root of the well-formed tree becomes an argument to the new root operator.

To visualize this, consider the example already given for the child operations, but where applying the Parent operation to the tree in Fig. 3.3c would result in the tree in Fig. 3.3a.

The pseudocode for this operation is shown in Listing 3.12. In this pseudocode, the parent function first updates the cursor context by calling the moveCCtxHoleUp function, which moves the context hole one level up in the cursor context tree. The moveCCtxHoleUp function recursively traverses the cursor context based on the cursor context hole's path. When the path contains two elements, we are one level above the cursor context hole's parent, which is then replaced with the new context hole (line 11-14).

When the path contains only one element, the cursor context hole is an immediate child of the root, and in that case it can be replaced directly with the new context hole (line 15).

If the path contains more than two elements, the function recurses further down the cursor context tree, until the grand-parent of the cursor context hole is reached (line 16-28).

Next, the insertAsRoot function is called to insert the removed context as the new root of the well-formed tree. This function adds the prior root of the well-formed tree as an argument to the removed cursor context's equivalent well-formed operator, which is then placed at the root of the well-formed tree. The checks being performed in the insertAsRoot function ensure that the insertion is well-formed, i.e. it checks that if an operator of arity $(\vec{s}_1.\hat{s}_1, \dots, \vec{s}_i.C, \dots, \vec{s}_n.\hat{s}_n)C$ was removed

from the cursor context, the root of the well-formed tree must be an operator of sort $(\hat{s}_i.\hat{s}_i)\dot{s}$.

```

1 parent((cctx, wellformed)) =
2   let (newCCtx, removed) = moveCCtxHoleUp holepath cctx
3   let newWellformed = insertAsRoot removed wellformed
4   in
5     (newCCtx, newWellformed)
6
7 moveCCtxHoleUp holepath cctx =
8   case holepath of
9     [_,_] -> case cctx of
10       CCtxHole -> error "invalid path"
11       CCtxOp1 arg1 ... argn ->
12         ((CCtxOp1 CCtxHole ... argn), arg1)
13       CCtxOp2 arg1 arg2 ... argn ->
14         ((CCtxOp2 arg1 CCtxHole ... argn), arg2)
15     [] -> (CCtxHole, cctx)
16     _ : restpath -> case cctx of
17       CCtxOp1 arg1 ... argn ->
18         let (newCCtx, removed) =
19           moveCCtxHoleUp restpath arg1
20         in
21           ((CCtxOp1 newCCtx ... argn)
22            , removed)
23       CCtxOp2 arg1 arg2 ... argn ->
24         let (newCCtx, removed) =
25           moveCCtxHoleUp restpath arg2
26         in
27           ((CCtxOp2 arg1 newCCtx ... argn)
28            , removed)
29     [] -> error "empty path"
30
31 insertAsRoot removed wellformed =
32   case removed of
33     CCtxHole -> error "invalid cctx"
34     CCtxOp1_SortY arg1_SortX ... argn ->
35       case wellformed of
36         Root_SortX underCursor ->
37           Root_SortY (CLessOp_SortY underCursor)
38         _ -> error "not wellformed"
39     CCtxOp2_SortY arg1 arg2_SortX ... argn ->
40       case wellformed of
41         Root_SortX underCursor ->
42           Root_SortY (CLessOp_SortY underCursor)
43         _ -> error "not wellformed"
44     ...

```

Listing 3.12: Pseudocode for applying parent operator

Substitution

Substitution allow for replacing the abt encapsulated by the cursor with another abt of the same sort, ensuring that well-formedness is maintained. This is defined as the Insert-op operation in the editor calculus (Fig. 2.3).

For this a substitution function is generated by Elm CodeGen, which takes a decomposed tree and a new cursorless operator, and returns a new decomposed tree where the cursor has been replaced with the new operator, if substitution is valid. See Listing 3.13 for pseudocode demonstrating this process and ?? for an illustration of the substitution operation.

```
1 substitution(sub, (cctx, wellformed)) =  
2   case wellformed of  
3     Root_SortX underCursor ->  
4       case sub of  
5         CursorLess_SortX sub_ ->  
6           (cctx, Root_SortX sub_)  
7         _ -> error "invalid substitution"  
8     Root_SortY underCursor ->  
9       case sub of  
10        CursorLess_SortY sub_ ->  
11          (cctx, Root_SortY sub_)  
12        _ -> error "invalid substitution"  
13     ...
```

Listing 3.13: Pseudocode for substitution operator

Conditional expressions, Sequence, Recursion and Context

Conditional expressions provide a way to perform different operations based on modal logic of the encapsulated abt, such as the @o operator which holds if the cursor is encapsulating the operator o. This is defined as At-op along Necessity, Possibly-i and Possibly-i operators in the editor calculus[16] (Fig. 2.6). Conditionals can be connected with logical operators, forming the operators Negation, Conjunction, Disjunction-1 and Disjunction-2 (Fig. 2.5).

Unfortunately, this project has not yet implemented conditional expressions, but it is entirely possible by generating functions for each of the conditional operators, which would take a decomposed tree and return a boolean value based on the modal logic of the encapsulated abt.

Like conditional expressions, sequence and recursion operators have not been implemented yet in this project.

In other words, given the current state of the implementation, $\pi.nil$ is the only supported expression in the editor calculus (Fig. 2.1).

Generic implementation

Given the generic nature of the editor calculus, the implementation could benefit from type classes, which would allow for more general functions to be written, without the need for explicit references to specific function implementations for different sorts.

For example, consider the cursor substitution operator (Fig. 2.3), where the editor calculus enforces that operators can only be substituted with operators of same sort. Initially, a straightforward approach involves having a substitution function for each sort $s \in \mathcal{S}$. This approach of course leads an implementer to consider generalization, i.e. how a single function can take any cursor-encapsulated abt and replace it with an operator of the same sort. A solution to this could be using type classes, where we might have a type class called `substitutable` and having an instance for each sort. Having a typeclass is possible in Haskell (Listing 3.14), but typeclasses are not directly supported by Elm. They can however be simulated with Elm records, as shown in the "typeclasses" Elm package[19]. An example of such a simulation is shown in Listing 3.15. This typeclass simulation in Elm has the disadvantage of forcing an explicit reference to the typeclass "instance" in a generic function, in contrast to Haskell.

```
1 class Substitutable a where
2     substitute :: a -> a -> a
3
4 instance Substitutable a where
5     substitute _ replacement = replacement
6
7 doIntSub :: Int
8 doIntSub = substitute 1 2
```

Listing 3.14: Haskell typeclass example

```
1 type alias Substitutable a =
2     { substitute : a -> a -> a }
3
4 substituteAny : Substitutable a
5 substituteAny =
6     { substitute = \_ replacement -> replacement }
7
8 substitute : Substitutable a -> a -> a -> a
9 substitute substitutable expression replacement =
10     substitutable.substitute expression replacement
11
12 doIntSub : Int
13 doIntSub =
14     substitute substitutableAny 1 2
```

Listing 3.15: Elm typeclass simulation example

Unfortunately, this leads to more verbose code and it does not solve the rigid source code generation, since direct references to the typeclass "instance" is needed, which are made out of manipulated strings (although with assistance from the Elm CodeGen package). In other words, this does not provide any benefits over referring to a concrete function for each sort, which is the current approach.

Chapter 4

Editor Examples

The following will show from start to finish, how the implementation can be used to instantiate a syntax-directed editor for the C language (Example 11), SQL (Example 5) and L^AT_EX (Example 6).

Each section will first show what the computer-friendly and parseable syntax for each language looks like, followed by an excerpt of the generated source code for the editor instance. Only an excerpt is shown, as the generated source code can be quite large, and over 4000 lines for the C language syntax specifically, with the current implementation. The complete generated editors can be found in the "examples" directory of the source code repository.

Finally, a demonstration of the editor in action will be shown using the Elm REPL, where a sample tree built from the generated algebraic data types is manipulated with the generated functions.

4.1 C

The syntax for the C language is given in Example 11. The same syntax, but in the parseable format (Listing 3.1) is given in Listing 4.1.

```

1  p in Prog
2  s in Stmt
3  vd in VariableDecl
4  fd in FunDecl
5  t in Type
6  id in Id
7  e in Exp
8  b in Block
9  bi in BlockItem
10 fa in Funarg
11 cond in Conditional
12
13 p ::= fd # (fd)p # program
14 b ::= bi # (bi)b # block
15 bi ::= vd # (vd)bi # blockdecls
16      | s # (s)bi # blockstmts
17      | epsilon # ()bi # blockdone
18 vd ::= t id \ "=" e; bi # (t,e,id,bi)vd # vardecl
19 fd ::= t_1 id_1 \ "(" t_2 id_2 \ ")" \ "{" b \ "}\\"
20      # (t,id,fd,t,id,b)fd
21      # fundecl1
22      | t_1 id_1 \ "(" t_2 id_2, t_3 id_3 \ ")" \ "{" b \ "}\\"
23      # (t,id,fd,t,t,id,id,b)fd
24      # fundecl2
25      | epsilon # ()fd # fundecldone
26 s ::= id \ "=" e ";" \ # (id,e)s # assignment
27      | id \ "(" fa \ ")" ";" \ # (id,fa)s # stmtfuncall
28      | \ "return \ " e ";" \ # (e)s # return
29      | cond # (cond)s # conditional
30      | s_1 s_2 # (s,s)s # compstmt
31 fa ::= t id # (t,id)fa # funarg
32      | t id, fa # (t,id,fa)fa # funargs
33 cond ::= \ "if ( \ " e \ ")" \ "{" b_1 \ "} else { \ " b_2 \ "}\\"
34         # (e,b,b)cond
35         # ifelse
36 t ::= \ "int\ " # ()t # tint
37      | \ "char\ " # ()t # tchar
38      | \ "bool\ " # ()t # tbool
39 e ::= %int # ()e # int[Int] | %char # ()e # char[Char]
40      | %bool # ()e # bool[Bool]
41      | e_1 \ "+" e_2 # (e,e)e # plus
42      | e_1 \ "=" e_2 # (e,e)e # equals
43      | id \ "(" fa \ ")" \ # (id,fa)e # expfuncall
44      | id # (id)e # expident
45 id ::= %string # ()id # ident[String]

```

Listing 4.1: Parseable format of C language syntax

When passed the parseable format, the parser will produce the algebraic data types

given in Listing 4.2, alongside relevant functions for cursor movement (Listing 4.3) and substitution (Listing 4.4).

```

1  type P
2    = Program Fd
3    | Hole_p
4    | Cursor_p P
5
6  type S
7    = Assignment Id E
8    | Stmtfuncall Id Fa
9    | Return E
10   | Conditional Cond
11   | Compstmt S S
12   | Hole_s
13   | Cursor_s S
14   ...
15  type P_CLess
16    = Program_CLess Fd_CLess
17    | Hole_p_CLess
18
19
20  type S_CLess
21    = Assignment_CLess Id_CLess E_CLess
22    | Stmtfuncall_CLess Id_CLess Fa_CLess
23    | Return_CLess E_CLess
24    | Conditional_CLess Cond_CLess
25    | Compstmt_CLess S_CLess S_CLess
26    | Hole_s_CLess
27    ...
28  type Cctx
29    = Cctx_hole
30    | Program_CLess_cctx1 Cctx
31    ...
32    | Vardecl_CLess_cctx1 Cctx E_CLess (Bind Id_CLess Bi_CLess)
33    | Vardecl_CLess_cctx2 T_CLess Cctx (Bind Id_CLess Bi_CLess)
34    | Vardecl_CLess_cctx3 T_CLess E_CLess (Bind Id_CLess Cctx)
35    ...
36
37  type Wellformed
38    = Root_p_CLess P_CLess
39    | Root_s_CLess S_CLess
40    | Root_vd_CLess Vd_CLess
41    | Root_fd_CLess Fd_CLess
42    | Root_t_CLess T_CLess
43    ...

```

Listing 4.2: Generated ADT for the C language

```

1  getCursorPath : List Int -> Base -> List Int
2  getCursorPath path base =
3      case base of
4          P p ->
5              case p of
6                  Program arg1 ->
7                      getCursorPath (path ++ [ 1 ]) (Fd arg1)
8
9                  Hole_p ->
10                     []
11
12                 Cursor_p _ ->
13                     path
14
15          B b ->
16              case b of
17                  Block arg1 ->
18                      getCursorPath (path ++ [ 1 ]) (Bi arg1)
19
20                  Hole_b ->
21                     []
22
23                  Cursor_b _ ->
24                     path
25  ...
26
27
28  parent : ( Cctx, Wellformed ) -> Maybe ( Cctx, Wellformed )
29  parent decomposed =
30      let
31          ( cctx, wellformed ) =
32              decomposed
33      in
34      case moveCctxHoleUp cctx (getCctxPath cctx []) of
35          Nothing ->
36              Nothing
37
38          Just ( newCctx, removedCctx ) ->
39              case addParent removedCctx wellformed of
40                  Nothing ->
41                      Nothing
42
43                  Just newWellformed ->
44                      Just ( newCctx, newWellformed )
45  ...
46
47  child : Int -> ( Cctx, Wellformed ) -> Maybe ( Cctx, Wellformed )
48  child i decomposed =

```

```

49 let
50   ( cctx, wellformed ) =
51     decomposed
52 in
53 case wellformed of
54   Root_p_CLess underCursor ->
55     case underCursor of
56       Program_CLess arg1 ->
57         case i of
58           1 ->
59             Just
60               ( replaceCctxHole i cctx (P_CLess underCursor)
61                 , Root_fd_CLess arg1
62               )
63           _ ->
64             Nothing
65       Hole_p_CLess ->
66         Nothing
67   Root_s_CLess underCursor ->
68     case underCursor of
69       Assignment_CLess arg1 arg2 ->
70         case i of
71           1 ->
72             Just
73               ( replaceCctxHole i cctx (S_CLess underCursor)
74                 , Root_id_CLess arg1
75               )
76           2 ->
77             Just
78               ( replaceCctxHole i cctx (S_CLess underCursor)
79                 , Root_e_CLess arg2
80               )
81           _ ->
82             Nothing
83   ...

```

Listing 4.3: Generated functions for cursor movement for the C language


```

1 substitute : ( Cctx, Wellformed ) -> CursorLess -> Maybe ( Cctx, Wellformed )
2 substitute decomposed sub =
3   let
4     ( cctx, wellformed ) =
5       decomposed
6   in
7   case wellformed of
8     Root_p_CLess _ ->
9       case sub of
10        P_CLess sub0 ->
11          Just ( cctx, Root_p_CLess sub0 )
12
13        _ ->
14          Nothing
15
16     Root_s_CLess _ ->
17       case sub of
18        S_CLess sub0 ->
19          Just ( cctx, Root_s_CLess sub0 )
20
21        _ ->
22          Nothing
23   ...

```

Listing 4.4: Generated functions for cursor substitution for the C language

An editing session of a sample program within the C language in the Elm REPL can be seen in Listing 4.5. The sample program consists of a function declaration for a function called `main` which has an input parameter called `"x"`. The body of the function consists of two statements, an assignment to `"x"` and a return statement, returning `"x"`. The cursor is located at the second child of the Assignment operator, encapsulating a hole.

Decomposition results in a cursor context of the whole tree, except the second child of the `Assignment_CLess_cctx2` operator, which contains a cursor context hole. The well-formed tree consists only of the hole at the second child of the assignment operator.

Applying the parent function to the decomposed tree results in a new cursor context and well-formed tree pair, where the cursor has been moved to the assignment operator itself, as expected.

Ultimately, the substitution function is demonstrated, where the cursor is located at the hole of the assignment operator, which is replaced with an `Int_CLess` operator containing the value 2.

```

1 > example = P <|
2 | Program <|

```

```

3 | Fundecl1
4 | Tint
5 | ( [ Ident "main" ], Fundecldone )
6 | Tint
7 | ( [ Ident "x" ]
8 | , Block
9 | (Blockstmts
10 | (Compstmt
11 | (Assignment (Ident "x") (Cursor_e Hole_e))
12 | (Return (Expident (Ident "x"))))
13 | )
14 | )
15 | )
16 |
17 | P (Program (Fundecl1 Tint ([Ident "main"],Fundecldone) Tint ([Ident "x"],Block
    (Blockstmts (Compstmt (Assignment (Ident "x") (Cursor_e Hole_e)) (Return
    (Expident (Ident "x"))))))))
18 | : Base
19 |
20 | > decomposed = decompose example
21 | (Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess
    "main"],Fundecldone_CLess) Tint_CLess ([Ident_CLess "x"],Block_CLess_cctx1
    (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 (Assignment_CLess_cctx2
    (Ident_CLess "x") Cctx_hole) (Return_CLess (Expident_CLess (Ident_CLess
    "x"))))))),Root_e_CLess Hole_e_CLess)
22 | : ( Cctx, Wellformed )
23 |
24 | > new = parent decomposed
25 | Just (Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess
    "main"],Fundecldone_CLess) Tint_CLess ([Ident_CLess "x"],Block_CLess_cctx1
    (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 Cctx_hole (Return_CLess
    (Expident_CLess (Ident_CLess "x"))))))),Root_s_CLess (Assignment_CLess
    (Ident_CLess "x") Hole_e_CLess))
26 | : Maybe ( Cctx, Wellformed )
27 |
28 | > child 1 (Maybe.withDefault (Cctx_hole, Root_p_CLess Hole_p_CLess) new)
29 | Just (Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess
    "main"],Fundecldone_CLess) Tint_CLess ([Ident_CLess "x"],Block_CLess_cctx1
    (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 (Assignment_CLess_cctx1
    Cctx_hole Hole_e_CLess) (Return_CLess (Expident_CLess (Ident_CLess
    "x"))))))),Root_id_CLess (Ident_CLess "x"))
30 | : Maybe ( Cctx, Wellformed )
31 |
32 | > substitute decomposed (E_CLess (Int_CLess 2))
33 | Just (Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess
    "main"],Fundecldone_CLess) Tint_CLess ([Ident_CLess "x"],Block_CLess_cctx1
    (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 (Assignment_CLess_cctx2
    (Ident_CLess "x") Cctx_hole) (Return_CLess (Expident_CLess (Ident_CLess
    "x"))))))),Root_e_CLess (Int_CLess 2))

```

34 : Maybe (Cctx, Wellformed)

Listing 4.5: Elm REPL demonstration of C language editor

4.2 SQL

The syntax for the SQL language is given in Example 5.

Parseable format for the SQL language is given in Listing 4.11.

```

1 q in Query
2 cmd in Command
3 id in Id
4 const in Const
5 clause in Clause
6 cond in Condition
7 exp in Expression
8
9 q ::= \ " SELECT \ " id_1 \ " FROM \ " id_2 clause # (id,id,clause)q # select
10 cmd ::= \ " INSERT INTO \ " id_1 \ " AS \ " id_2 q # (id,id.q)cmd # insert
11 id ::= %string # ()id # ident[String]
12 const ::= %number # ()const # num[Int]
13         | \ "' \ " %string \ "' \ " # ()const # str[String]
14 clause ::= \ " WHERE \ " cond # (cond)clause # where
15         | \ " HAVING \ " cond # (cond)clause # having
16 cond ::= exp_1 \ ">\ " exp_2 # (exp,exp)cond # greater
17         | exp_1 \ "=\ " exp_2 # (exp,exp)cond # equals
18 exp ::= const # (const)exp # econst | id # (id)exp # eident

```

Listing 4.6: Parseable format of the SQL language syntax

When passed the parseable format, the parser will produce the algebraic data types given in Listing 4.7, alongside relevant functions for cursor movement (Listing 4.8) and substitution (Listing 4.9).

```

1  type Q
2    = Select Id Id Clause
3    | Hole_q
4    | Cursor_q Q
5
6  type Cmd
7    = Insert Id (Bind Id Q)
8    | Hole_cmd
9    | Cursor_cmd Cmd
10 ...
11
12 type Base
13   = Q Q
14   | Cmd Cmd
15   | Id Id
16   | Const Const
17   | Clause Clause
18   | Cond Cond
19   | Exp Exp
20
21 type Q_CLess
22   = Select_CLess Id_CLess Id_CLess Clause_CLess
23   | Hole_q_CLess
24
25 type Cmd_CLess
26   = Insert_CLess Id_CLess (Bind Id_CLess Q_CLess)
27   | Hole_cmd_CLess
28 ...
29
30 type Cctx
31   = Cctx_hole
32   | Select_CLess_cctx1 Cctx Id_CLess Clause_CLess
33   | Select_CLess_cctx2 Id_CLess Cctx Clause_CLess
34   | Select_CLess_cctx3 Id_CLess Id_CLess Cctx
35   | Insert_CLess_cctx1 Cctx (Bind Id_CLess Q_CLess)
36   ...
37
38 type Wellformed
39   = Root_q_CLess Q_CLess
40   | Root_cmd_CLess Cmd_CLess
41   | Root_id_CLess Id_CLess
42   | Root_const_CLess Const_CLess
43   | Root_clause_CLess Clause_CLess
44   | Root_cond_CLess Cond_CLess
45   | Root_exp_CLess Exp_CLess

```

Listing 4.7: Generated ADT for the SQL language

```

1  getCursorPath : List Int -> Base -> List Int
2  getCursorPath path base =
3      case base of
4          Q q ->
5              case q of
6                  Select arg1 arg2 arg3 ->
7                      (getCursorPath (path ++ [ 1 ]) (Id arg1)
8                      ++ getCursorPath
9                      (path ++ [ 2 ])
10                     (Id arg2)
11                     )
12                     ++ getCursorPath (path ++ [ 3 ]) (Clause arg3)
13
14             Hole_q ->
15                 []
16
17             Cursor_q _ ->
18                 path
19
20 ...
21 parent : ( Cctx, Wellformed ) -> Maybe ( Cctx, Wellformed )
22 parent decomposed =
23     let
24         ( cctx, wellformed ) =
25             decomposed
26     in
27     case moveCctxHoleUp cctx (getCctxPath cctx []) of
28         Nothing ->
29             Nothing
30
31         Just ( newCctx, removedCctx ) ->
32             case addParent removedCctx wellformed of
33                 Nothing ->
34                     Nothing
35
36                 Just newWellformed ->
37                     Just ( newCctx, newWellformed )
38
39 addParent : Cctx -> Wellformed -> Maybe Wellformed
40 addParent cctx wellformed =
41     case cctx of
42         Cctx_hole ->
43             Nothing
44
45         Select__CLess__cctx1 arg1 arg2 arg3 ->
46             case wellformed of
47                 Root_id__CLess underCursor ->
48                     Just (Root_q__CLess (Select__CLess underCursor arg2 arg3))

```

```

49         _ ->
50             Nothing
51     ...
52
53
54 moveCCtxHoleUp : Cctx -> List Int -> Maybe ( Cctx, Cctx )
55 moveCCtxHoleUp cctx path =
56     case path of
57         [ _, _ ] ->
58             case cctx of
59                 Cctx_hole ->
60                     Nothing
61
62                 Select_CLess_cctx1 arg1 arg2 arg3 ->
63                     Just ( Select_CLess_cctx1 Cctx_hole arg2 arg3, arg1 )
64     ...
65
66 substitute : ( Cctx, Wellformed ) -> CursorLess -> Maybe ( Cctx, Wellformed )
67 substitute decomposed sub =
68     let
69         ( cctx, wellformed ) =
70             decomposed
71     in
72     case wellformed of
73         Root_q_CLess _ ->
74             case sub of
75                 Q_CLess sub0 ->
76                     Just ( cctx, Root_q_CLess sub0 )
77
78         _ ->
79             Nothing
80     ...

```

Listing 4.8: Generated functions for cursor movement for the SLQ language

```

1 substitute : ( Cctx, Wellformed ) -> CursorLess -> Maybe ( Cctx, Wellformed )
2 substitute decomposed sub =
3 let
4   ( cctx, wellformed ) =
5     decomposed
6 in
7 case wellformed of
8   Root_q_CLess _ ->
9     case sub of
10      Q_CLess sub0 ->
11        Just ( cctx, Root_q_CLess sub0 )
12
13      _ ->
14        Nothing
15
16   Root_cmd_CLess _ ->
17     case sub of
18      Cmd_CLess sub0 ->
19        Just ( cctx, Root_cmd_CLess sub0 )
20
21      _ ->
22        Nothing
23 ...

```

Listing 4.9: Generated functions for cursor substitution for the SQL language

Sample editing session in the Elm REPL for the generated SQL editor is shown in Listing 4.10. The sample tree consists of a SQL query selecting every column from a table called "table-b" where the value in "col-a" is greater than 2. For the purpose of demonstration, this sample has no cursor, showing that the decomposition of the tree will result in a cursor context containing only the context hole and a well-formed tree with the cursor at the root of the tree. Attempting to move the cursor to the parent will result in Nothing, as the cursor is already at the root of the tree. Moving the cursor to child 1 moves the cursor to the first child of the Select operator, as expected. Finally, simple substitution is shown where the cursor is encapsulating the root of tree, resulting in the substitute being the only element under the updated cursor.

```

1 > example = Q (Select (Ident "col-a") (Ident "table-b") (Where (Greater (Eident
    (Ident "col-a")) (Econst (Num 2)))))
2 Q (Select (Ident "col-a") (Ident "table-b") (Where (Greater (Eident (Ident "col-a"))
    (Econst (Num 2)))))
3 : Base
4
5 > decomposed = decompose example
6 (Cctx_hole,Root_q_CLess (Select_CLess (Ident_CLess "col-a") (Ident_CLess
    "table-b") (Where_CLess (Greater_CLess (Eident_CLess (Ident_CLess "col-a"))
    (Econst_CLess (Num_CLess 2)))))
7 : ( Cctx, Wellformed )
8
9 > new = parent decomposed
10 Nothing : Maybe ( Cctx, Wellformed )
11
12 > child 1 decomposed
13 Just (Select_CLess_cctx1 Cctx_hole (Ident_CLess "table-b") (Where_CLess
    (Greater_CLess (Eident_CLess (Ident_CLess "col-a")) (Econst_CLess
    (Num_CLess 2)))),Root_id_CLess (Ident_CLess "col-a"))
14 : Maybe ( Cctx, Wellformed )
15
16 > substitute decomposed (Q_CLess Hole_q_CLess)
17 Just (Cctx_hole,Root_q_CLess Hole_q_CLess)
18 : Maybe ( Cctx, Wellformed )

```

Listing 4.10: Elm REPL demonstration of the SQL editor

4.3 L^AT_EX

The syntax for the L^AT_EX language is given in Example 6.

Parseable format for the L^AT_EX language is given in Listing 4.11.

```
1 d in LatexDocument
2 e in Environment
3 cmd in Command
4 a in Argument
5 id in Identifier
6 c in Content
7
8 d ::= "\"\documentclass{\" id_1 \"}
9       \"\newenvironment{\" id_2 \"}\" a_1 a_2
10      "\"\begin{document} \" c \"\end{document}\"
11      # (id,a,a,id.b)d
12      # latexdoc
13 id ::= %string # ()id # ident[String]
14 e ::= "\"\begin{\" id \"}\" c
15      "\"\end{\" id \"}\"
16      # (id,c,id)e
17      # environment
18 cmd ::= "\"\\\" id a # (id,a)cmd # command
19 a ::= "\"{\\" c \"}\" # (c)a # argument
20 c ::= %string # ()c # textContent[String]
21      | cmd # (cmd)c # cmdContent
22      | e # (e)c # envContent
23      | c c # (c,c)c # seqContent
```

Listing 4.11: Parseable format of L^AT_EX language syntax

When passed the parseable format, the parser will produce the algebraic data types given in Listing 4.12, alongside relevant functions for cursor movement and substitution (Listing 4.13).

```

1 type D
2   = Latexdoc Id E A A (Bind Id B)
3   | Hole_d
4   | Cursor_d D
5
6
7 type E
8   = Environment Id C Id
9   | Hole_e
10  | Cursor_e E
11
12 ...
13
14 type D_CLess
15   = Latexdoc_CLess Id_CLess E_CLess A_CLess A_CLess (Bind Id_CLess
16     C_CLess)
17   | Hole_d_CLess
18
19 type E_CLess
20   = Environment_CLess Id_CLess C_CLess Id_CLess
21   | Hole_e_CLess
22
23 ...
24
25 type Cctx
26   = Cctx_hole
27   | Latexdoc_CLess_cctx1 Cctx E_CLess A_CLess A_CLess (Bind Id_CLess
28     B_CLess)
29   | Latexdoc_CLess_cctx2 Id_CLess Cctx A_CLess A_CLess (Bind Id_CLess
30     B_CLess)
31   ...
32
33 type Wellformed
34   = Root_d_CLess D_CLess
35   | Root_e_CLess E_CLess
36   | Root_cmd_CLess Cmd_CLess
37   | Root_a_CLess A_CLess
38   | Root_id_CLess Id_CLess
39   | Root_c_CLess C_CLess

```

Listing 4.12: Generated ADT for the \LaTeX language

```

1  getCursorPath : List Int -> Base -> List Int
2  getCursorPath path base =
3      case base of
4          D d ->
5              case d of
6                  Latexdoc arg1 arg2 arg3 arg4 ( boundVars5, arg5 ) ->
7                      (((getCursorPath (path ++ [ 1 ]) Id arg1
8                          ++ getCursorPath (path ++ [ 2 ]) (E arg2)
9                          )
10                         ++ getCursorPath (path ++ [ 3 ]) (A arg3)
11                         )
12                        ++ getCursorPath (path ++ [ 4 ]) (A arg4)
13                        )
14                        ++ getCursorPath (path ++ [ 5 ]) (B arg5)
15
16                  Hole_d ->
17                      []
18
19                  Cursor_d _ ->
20                      path
21
22  ...
23  parent : ( Cctx, Wellformed ) -> Maybe ( Cctx, Wellformed )
24  parent decomposed =
25      let
26          ( cctx, wellformed ) =
27              decomposed
28      in
29      case moveCctxHoleUp cctx (getCctxPath cctx []) of
30          Nothing ->
31              Nothing
32
33          Just ( newCctx, removedCctx ) ->
34              case addParent removedCctx wellformed of
35                  Nothing ->
36                      Nothing
37
38                  Just newWellformed ->
39                      Just ( newCctx, newWellformed )
40
41  ...
42  moveCctxHoleUp : Cctx -> List Int -> Maybe ( Cctx, Cctx )
43  moveCctxHoleUp cctx path =
44      case path of
45          [ _, _ ] ->
46              case cctx of
47                  Cctx_hole ->
48                      Nothing

```

```

49         Latexdoc_CLess_cctx1 arg1 arg2 arg3 arg4 ( boundVars5, arg5 ) ->
50         Just
51         ( Latexdoc_CLess_cctx1
52           Cctx_hole
53           arg2
54           arg3
55           arg4
56           ( boundVars5, arg5 )
57           , arg1
58         )
59     ...
60
61 addParent : Cctx -> Wellformed -> Maybe Wellformed
62 addParent cctx wellformed =
63     case cctx of
64     Cctx_hole ->
65         Nothing
66
67     Latexdoc_CLess_cctx1 arg1 arg2 arg3 arg4 ( boundVars5, arg5 ) ->
68     case wellformed of
69     Root_id_CLess underCursor ->
70         Just
71         (Root_d_CLess
72          (Latexdoc_CLess
73           underCursor
74           arg2
75           arg3
76           arg4
77           ( boundVars5, arg5 )
78          )
79         )
80
81     _ ->
82         Nothing
83
84     ...
85
86 child : Int -> ( Cctx, Wellformed ) -> Maybe ( Cctx, Wellformed )
87 child i decomposed =
88     let
89         ( cctx, wellformed ) =
90             decomposed
91     in
92     case wellformed of
93     Root_d_CLess underCursor ->
94     case underCursor of
95     Latexdoc_CLess arg1 arg2 arg3 arg4 ( boundVars5, arg5 ) ->
96     case i of
97     1 ->
98         Just

```

```

98             ( replaceCtxHole i cctx (D_CLess underCursor)
99             , Root_id_CLess arg1
100             )
101         ...
102
103 substitute : ( Cctx, Wellformed ) -> CursorLess -> Maybe ( Cctx, Wellformed )
104 substitute decomposed sub =
105     let
106         ( cctx, wellformed ) =
107             decomposed
108     in
109     case wellformed of
110         Root_d_CLess _ ->
111             case sub of
112                 D_CLess sub0 ->
113                     Just ( cctx, Root_d_CLess sub0 )
114             _ ->
115                 Nothing
116

```

Listing 4.13: Generated functions for cursor movement for the \LaTeX language

Sample editing session in the Elm REPL for the generated \LaTeX editor is shown in Listing 4.14. The example tree represents a \LaTeX document of type "article" with an environment called "myenv", where the definition of this environment contains holes. The body of the document contains the text "Hello World!". The cursor is located at the text of the document body. Decomposing the tree results in a cursor context containing the whole tree, except for the text content of the document body, which is under the root of the well-formed tree. Moving the cursor to the parent results in the cursor being moved to the root of the sample tree. Simple substitution is shown where if the cursor is at the text content, we can replace it with another text content operator with an updated string value.

```

1 > example = D (Latexdoc (Ident "article") Hole_a Hole_a ( [ Ident "myenv" ],
    Cursor_c (TextContent "Hello World!") ))
2 D (Latexdoc (Ident "article") Hole_a Hole_a ([Ident "myenv"],Cursor_c (TextContent
    ("Hello World!"))))
3 : Base
4
5 > decomposed = decompose example
6 (Latexdoc_CLess_cctx4 (Ident_CLess "article") Hole_a_CLess Hole_a_CLess
    ([Ident_CLess "myenv"],Cctx_hole),Root_c_CLess (TextContent_CLess ("Hello
    World!")))
7 : ( Cctx, Wellformed )
8
9 > new = parent decomposed
10 Just (Cctx_hole,Root_d_CLess (Latexdoc_CLess (Ident_CLess "article")
    Hole_a_CLess Hole_a_CLess ([Ident_CLess "myenv"],TextContent_CLess
    ("Hello World!"))))
11 : Maybe ( Cctx, Wellformed )
12
13 > child 1 (Maybe.withDefault (Cctx_hole, Root_d_CLess Hole_d_CLess) new)
14 Just (Latexdoc_CLess_cctx1 Cctx_hole Hole_a_CLess Hole_a_CLess ([Ident_CLess
    "myenv"],TextContent_CLess ("Hello World!")),Root_id_CLess (Ident_CLess
    "article"))
15 : Maybe ( Cctx, Wellformed )
16
17 > substitute decomposed (C_CLess (TextContent_CLess "Updated content"))
18 Just (Latexdoc_CLess_cctx4 (Ident_CLess "article") Hole_a_CLess Hole_a_CLess
    ([Ident_CLess "myenv"],Cctx_hole),Root_c_CLess (TextContent_CLess
    ("Updated content")))
19 : Maybe ( Cctx, Wellformed )

```

Listing 4.14: Editing sessions with the \LaTeX editor in the Elm REPL

Chapter 5

Conclusion

This project was exploratory in nature, aiming to implement a generic editor based on the work of the Aalborg project[16] and related work. The primary objective was to create a type-safe, generalized syntax-directed editor capable of supporting any language, including but not limited to programming languages. The implementation achieved a minimal viable product that can instantiate a syntax-directed editor for any language, with basic editing functionalities such as cursor movement and substitutions. This was demonstrated through practical examples involving subsets of different languages, including C, SQL, and LaTeX.

Only atomic prefix commands have been implemented at the time of writing. This however serves as a necessary and big first step toward completing the remaining editor expressions.

The implementation does not handle context-sensitive syntax, specifically binding mechanisms, at the time of writing. However, the model in the implementation has been designed to hold information about binders, and the specification language supports binders.

The implementation also lacks views of the code being edited. For now, the editor can only be interacted with through the Elm REPL, as shown in the examples.

Ultimately, the specification language enables the user to define the syntax (both abstract and concrete) of any language in a non-challenging way, assuming the user is familiar with syntax and language constructs in general.

5.1 Future work

Editor expressions for the remaining commands should be implemented. This would allow for a more complete editor, capable of handling all editor expression provided

by the calculus. This is possible by following a similar approach to the one used for the atomic prefix commands, i.e. by implementing function generators for each command based on the semantics provided in the editor calculus.

Different views is also a feature that should be implemented, as this also was a criteria for a good solution. This would allow the editor to display the syntax tree in different ways, such as a tree view or string-format (pretty-printing). The implementation of the non-generic structure editor[21] was successful in having a tree view, which can serve as inspiration. The model in this implementation is already designed to hold the concrete syntax of an operator (the term field in the Syntax record), allowing for pretty-printing to be implemented with relative ease.

There is currently no active use of binders in the editor, leaving out the possibility of potentially checking for use-before-declarations or shadowing. Implementing this is a non-trivial task, as it requires a more complex model of the syntax tree, as well as a more complex implementation of the editor expressions. Implementing this would bring the implementation closer to being a complete realization of the editor calculus.

Given the generic nature of the editor, it would be interesting to see an implementation in a language with support for typeclasses, such as Haskell. This should allow for a more concise implementation overall, being less rigid than the current implementation, since it relies on references to functions and data types via a low-level interface in the Elm CodeGen package, which refers to them with manipulated strings. This is in contrast to typeclass instances, which would allow for a more abstract representation, with editor functions supporting any tree of any sort, without the need for explicit references to functions implementing actions for each sort.

As the last point of future work, a minor improvement to the specification language would be to add support for defining the starting symbol of the language. This would shrink the amount of code in the source code generator, but also significantly reduce the amount of generated lines of code, since this would eliminate the need for supporting all sorts in some cases.

Bibliography

- [1] Patrick Borras et al. “CENTAUR: The System”. In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Massachusetts, USA, November 28-30, 1988*. Ed. by Peter B. Henderson. ACM, 1988, pp. 14–24. DOI: [10.1145/64135.65005](https://doi.org/10.1145/64135.65005). URL: <https://doi.org/10.1145/64135.65005>.
- [2] Thierry Despeyroux. “Executable Specification of Static Semantics”. In: *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*. Ed. by Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin. Vol. 173. Lecture Notes in Computer Science. Springer, 1984, pp. 215–233. DOI: [10.1007/3-540-13346-1_11](https://doi.org/10.1007/3-540-13346-1_11). URL: https://doi.org/10.1007/3-540-13346-1_11.
- [3] Véronique Donzeau-Gouge, Bernard Lang, and Bertrand Melese. “Practical Applications of a Syntax Directed Program Manipulation Environment”. In: *Proceedings, 7th International Conference on Software Engineering, Orlando, Florida, USA, March 26-29, 1984*. Ed. by Terry A. Straeter, William E. Howden, and Jean-Claude Rault. IEEE Computer Society, 1984, pp. 346–357. URL: <http://dl.acm.org/citation.cfm?id=801990>.
- [4] Christian Godiksen et al. “A type-safe structure editor calculus”. In: *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*. Ed. by Sam Lindley and Torben Æ. Mogensen. ACM, 2021, pp. 1–13. DOI: [10.1145/3441296.3441393](https://doi.org/10.1145/3441296.3441393). URL: <https://doi.org/10.1145/3441296.3441393>.
- [5] The PostgreSQL Global Development Group. *About PostgreSQL*. <https://www.postgresql.org/about/>. Accessed: 11/03/2024.

- [6] Robert Harper. *Practical Foundations for Programming Languages* (2nd. Ed.) Cambridge University Press, 2016. ISBN: 9781107150300. URL: <https://www.cs.cmu.edu/~rwh/pfpl.html>.
- [7] WikiBooks: Haskell. *Named Fields (Record Syntax)*. https://en.wikibooks.org/wiki/Haskell/More_on_datatypes. Accessed: 13/03/2024.
- [8] Gérard P. Huet. “The Zipper”. In: *J. Funct. Program.* 7.5 (1997), pp. 549–554. DOI: [10.1017/S0956796897002864](https://doi.org/10.1017/S0956796897002864). URL: <https://doi.org/10.1017/s0956796897002864>.
- [9] GitHub Inc. *Programming languages*. <https://innovationgraph.github.com/global-metrics/programming-languages>. Accessed: 27/02/2024.
- [10] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth, p. 520. URL: <https://www.iso.org/standard/74528.html>.
- [11] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. <https://www.cs.utexas.edu/users/novak/yaccpaper.htm>. Accessed: 26/03/2024.
- [12] Neil D. Jones. “An Introduction to Partial Evaluation”. In: *ACM Comput. Surv.* 28.3 (1996), pp. 480–503. DOI: [10.1145/243439.243447](https://doi.org/10.1145/243439.243447). URL: <https://doi.org/10.1145/243439.243447>.
- [13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN: 978-0-13-020249-9.
- [14] Gilles Kahn et al. “Metal: A Formalism to Specify Formalisms”. In: *Sci. Comput. Program.* 3.2 (1983), pp. 151–188. DOI: [10.1016/0167-6423\(83\)90009-6](https://doi.org/10.1016/0167-6423(83)90009-6). URL: [https://doi.org/10.1016/0167-6423\(83\)90009-6](https://doi.org/10.1016/0167-6423(83)90009-6).
- [15] “Abstract Syntax Notation One”. In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil K. Jain. Springer US, 2009, p. 1. DOI: [10.1007/978-0-387-73003-5_670](https://doi.org/10.1007/978-0-387-73003-5_670). URL: https://doi.org/10.1007/978-0-387-73003-5_670.
- [16] Andreas Tor Mortensen et al. “A type-safe generalized editor calculus (Short Paper)”. In: ACM, 2023, pp. 1–7.
- [17] Cyrus Omar et al. “Live functional programming with typed holes”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 14:1–14:32. DOI: [10.1145/3290327](https://doi.org/10.1145/3290327). URL: <https://doi.org/10.1145/3290327>.
- [18] Elm packages. *Elm CodeGen*. <https://package.elm-lang.org/packages/mdgriffith/elm-codegen/latest/>. Accessed: 18/03/2024.

- [19] Elm packages. *Elm Typeclasses*. <https://package.elm-lang.org/packages/nikita-volkov/typeclasses/latest/>. Accessed: 17/04/2024.
- [20] Thomas W. Reps and Tim Teitelbaum. “The Synthesizer Generator”. In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*. Ed. by William E. Riddle and Peter B. Henderson. ACM, 1984, pp. 42–48. DOI: [10.1145/800020.808247](https://doi.org/10.1145/800020.808247). URL: <https://doi.org/10.1145/800020.808247>.
- [21] Nicolaj Richs-Jensen, Thorbjørn Bülow Bringgaard, and Tórir Feilberg Zachariasen. “Implementation of a Type-Safe Structure Editor”. In: 2023.
- [22] LaTeX Project Team. *Introduction to LaTeX*. <https://www.latex-project.org/about/>. Accessed: 11/03/2024.
- [23] Tim Teitelbaum and Thomas W. Reps. “The Cornell Program Synthesizer: A Syntax-Directed Programming Environment”. In: *Commun. ACM* 24.9 (1981), pp. 563–573. DOI: [10.1145/358746.358755](https://doi.org/10.1145/358746.358755). URL: <https://doi.org/10.1145/358746.358755>.
- [24] Daniel C. Wang et al. “The Zephyr Abstract Syntax Description Language”. In: *Proceedings of the Conference on Domain-Specific Languages, DSL’97, Santa Barbara, California, USA, October 15-17, 1997*. Ed. by Chris Ramming. USENIX, 1997, pp. 213–228. URL: <http://www.usenix.org/publications/library/proceedings/dsl97/wang.html>.