



Master's thesis

# Implementation of a type-safe generalized syntax-directed editor

Sune Skaanning Engtorp

Advisor: Hans Hüttel

Submitted: 31/05/2024

# Contents

<b>1</b>	<b>Problem statement</b>	<b>1</b>
<b>2</b>	<b>Structure editors</b>	<b>3</b>
<b>3</b>	<b>Editor generators</b>	<b>4</b>
<b>4</b>	<b>Partial evaluation</b>	<b>5</b>
<b>5</b>	<b>Abstract Syntax Trees</b>	<b>7</b>
<b>6</b>	<b>Abstract Binding Trees</b>	<b>8</b>
<b>7</b>	<b>Generalized editor calculus</b>	<b>9</b>
7.1	Abstract syntax . . . . .	9
7.1.1	Editor calculus . . . . .	10
7.1.2	Cursor Context . . . . .	10
7.2	Semantics . . . . .	11
<b>8</b>	<b>Encoding the generalized editor calculus in an extended <math>\lambda</math>-calculus</b>	<b>13</b>
8.1	Abstract binding trees . . . . .	14
8.2	Cursor Contexts . . . . .	14
8.3	Atomic Prefix Commands . . . . .	15
8.4	Editor expressions . . . . .	17
8.5	Conditional expressions . . . . .	19
<b>9</b>	<b>Implementation</b>	<b>21</b>
9.1	Representing Syntax . . . . .	21
9.1.1	Specification language . . . . .	22
9.2	Code generation versus generic model . . . . .	26
9.3	Generating source code . . . . .	28
9.4	Decomposing trees . . . . .	33
9.4.1	Cursor path . . . . .	35
9.4.2	Cursor context . . . . .	36
9.4.3	Well-formed tree . . . . .	37

# 1 Problem statement

Structure editors provide a way to manipulate some program's abstract syntax structure directly, in contrast to writing and editing a program's source code 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[18] 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[1].

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[1] 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[8] 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
- $\text{\LaTeX}$

The C language[9] 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[1][4][15] 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.

#### Example 1: 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 semi-colon at line 2:

```
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.

### Example 2: 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';
```

L<sup>A</sup>T<sub>E</sub>X[19] 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.

## 2 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[20] 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[20] 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[15], 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 a program’s dynamic behaviour 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[15] 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 holes.

The Hazel environment is based on the Hazelnut structure editor calculus, defined by operational semantics, that allow finite edit expressions and inserts holes automatically to guarantee that every editor state has some type.

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[KU-bach-missing-ref].

### 3 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 also presented The Synthesizer Generator[18] in 1984, which creates structure editors from language descriptions in the form of attribute grammar specifications.

Another example is the Centaur system[2], which takes formalism described in the Metal language[13], 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 Harper’s definition of abstract syntax[6] in the form of sorts, arity-indexed operators and variables. Operators in both definitions represent a node in an abstract syntax tree and having an associated phyla or being arity-indexed serves the same purpose of constraining the possible children of each node, hence maintaining syntactically correct trees.

However the Centaur system[2] lacks a dedicated type-safe editor calculus. Such a type-safe generalized editor calculus has been proposed by [1], which is a generalization of the work of Godiksen et al. [4]. The generalized editor calculus takes abstract syntax in the form of sorts, arity-indexed operators and variables, as described by Harper[6].

## 4 Partial evaluation

One of the goals of this project is to implement a generic syntax-directed editor based on the editor calculus proposed by [1]. Implementing such a generic editor relates to the question of how to balance between generality and modularity, brought up by Neil D. Jones[12] 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 is very modular and has the advantage of allowing the programmer to focus solely on performance for every smaller program, but it has an obvious disadvantage of being hard to maintain. Highly modular programming might also lead to performance overhead in terms of passing data back and forth between programs and converting among 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[11] 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[11] in-

cludes 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), which can be specialized to having  $n = 5$  (program  $p_5$  in listing 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 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: Two-input program  $p$

```

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

```

Listing 2: Specialization of program  $p$

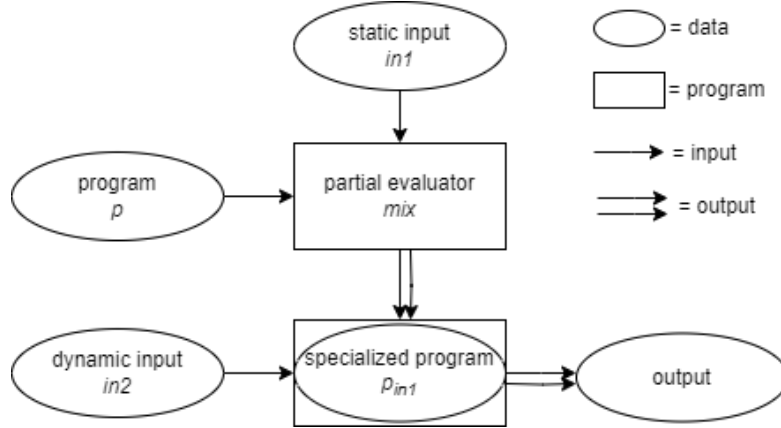


Figure 1: Visualisation of partial evaluation of a two-input program

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.

## 5 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  $\mathcal{O}_\alpha$  of arity  $\alpha$  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  $\mathcal{O}_\alpha$  of arity  $\alpha$ . 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 3: 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  $o(a_1; \dots; a_n) \in \mathcal{A}[\mathcal{X}]_s$

## 6 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 4: 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  $\mathcal{O}_\alpha$  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  $\mathcal{O}$ .

## 7 Generalized editor calculus

The generalized editor calculus[1] 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 et al. which is tailored towards an applied  $\lambda$ -calculus.

### 7.1 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  $\mathcal{X}$ .

#### Example 5: Abstract syntax of a simple language

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

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

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

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

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

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

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

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  $\mathcal{S}$ .

Formally, 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 6: Introduction of cursors and holes

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

Sort	Term	Operator	Arity
$s ::=$	$[s]$	$cursor_s$	$(s)s$
	$  \quad []_s$	$hole_s$	$()s$
$e ::=$	$[e]$	$cursor_e$	$(e)e$
	$  \quad []_e$	$hole_e$	$()e$

### 7.1.1 Editor calculus

The abstract syntax of the general editor calculus (figure 2) 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.

Figure 2: Abstract syntax of general editor calculus

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

### 7.1.2 Cursor Context

A cursor context  $C$  enables the editor to locate the cursor in an abt.

The notion of cursorless abts is introduced to support the cursor only being able to reside within the specified subtree.

Cursorless abts are defined as any  $\hat{a} \in \hat{\mathcal{B}}[\hat{\mathcal{X}}]_{\hat{s}}$  with sort  $\hat{s}$ , where  $\hat{s}$  is a sort of cursorless trees and  $\hat{\mathcal{X}}$  is the family of cursorless variables. We also have the family of cursorless operators  $\hat{\mathcal{O}}$ , which is relevant for defining cursor contexts themselves.

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  $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_n.\hat{s}_n)$  and for every  $1 \leq i \leq n$  the operator  $o_i^C$  of arity  $(\vec{\hat{s}}_1.\hat{s}_1, \dots, \vec{\hat{s}}_i.C, \dots, \vec{\hat{s}}_n.\hat{s}_n)$  to  $\mathcal{O}^C$
4. The family of variables  $\mathcal{X}^C = \hat{\mathcal{X}}$

The calculus proposed by Godiksen et. al and the generalized calculus relies on a single cursor. For that reason, it is also necessary to define the notion of well-formed trees, which describes the syntax of abts with exactly one cursor:

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  $C[\dot{a}]$ .

## 7.2 Semantics

Reduction rules for editor expressions are presented in figure ??, substitution in figure ?? and cursor movement in figure ??.

**Figure 3: 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 4: Reduction rules for substitution**

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

**Figure 5: 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 are presented in figure ?? and modalities in figure ??.

**Figure 6: 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 7: Satisfaction rules for modalities**

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

## 8 Encoding the generalized editor calculus in an extended $\lambda$ -calculus

The following sections will show how to encode the generalized editor calculus in a simply typed  $\lambda$ -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  $\lambda$ -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 figure ??.

**Figure 8: Abstract syntax of extended  $\lambda$ -calculus**

Terms	
$M ::= \lambda x : \tau. M$	(abstraction)
$M_1 M_2$	(application)
$x$	(variable)
$o$	(operator)
Types	
$\tau ::= \tau_1 \rightarrow \tau_2$	(function)
$s$	(sort)

## 8.1 Abstract binding trees

The types of operators in the  $\lambda$ -calculus can be inferred by their arity. The typing rules for this is provided in figure ??.

**Figure 9: Typing rules for  $\lambda$ -calculus operators**

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

Then, any abt can be encoded in the extended simply typed  $\lambda$ -calculus by the encoding in figure 10.

**Figure 10: Encoding of abts**

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

## 8.2 Cursor Contexts

Cursor contexts will be represented in the  $\lambda$ -calculus as pairs, hence it is extended to support this notion. The abstract syntax is given in ??.

**Figure 11: Abstract syntax of extended  $\lambda$ -calculus**

Terms	
$M ::= (M_1, M_2)$	(pair)
$M.1$	(first projection)
$M.2$	(second projection)
Types	
$\tau ::= \tau_1 \times \tau_2$	(product type)



Reduction rules for projecting the first and second value in a pair are given in figure ??.

**Figure 12: 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  $\lambda$ -calculus are given in figure ??.

**Figure 13: 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 figure ?. For short, in the following encodings, the cursor context is also given a type alias  $Ctx = s \times s$ .

**Figure 14: Encoding of cursor contexts**

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

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

### 8.3 Atomic Prefix Commands

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

**Figure 15: Abstract syntax of extended  $\lambda$ -calculus**

$M, N$	$::= \text{match } M \xrightarrow{p} \vec{N}$	(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 figure ??, 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 16: Reduction rules for pattern matching**

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

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

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

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 figure ??.

**Figure 17: Typing rules for pattern matching**

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

With the *match* construct, auxiliary functions for cursor movement (figure ??) 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 18: 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 figure ???. 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 19: Encoding of cursor movement**

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

## 8.4 Editor expressions

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

A *fix* operator (figure ??) 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 20: *fix* operator**

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

The boolean term constants and base types are defined directly in the abstract syntax of the  $\lambda$ -calculus (figure ??). 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 (figure ??).

**Figure 21: Abstract syntax of extended  $\lambda$ -calculus**

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

**Figure 22: *binds* extended with boolean constructs**

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

Typing rules for *fix* and booleans can be seen in figure ??.

**Figure 23: Typing rules for *fix* and booleans**

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

With prior definitions, encoding of editor expressions and context configuration is given in figure ??.

**Figure 24: Encoding of editor expressions and context configuration**

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

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

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

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

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

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

## 8.5 Conditional expressions

To encode conditional expressions  $\phi \in Eec$ , auxiliary functions *checkboth*, *checkone*, *or*, *and* and *neg* are defined in figure ??.

**Figure 25: Auxiliary functions for conditionals**

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

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

**Figure 26: Encoding of propositional connectives**

$$\begin{aligned}
 \llbracket \phi_1 \wedge \phi_2 \rrbracket &= \text{checkboth and } \llbracket \phi_1 \rrbracket \llbracket \phi_2 \rrbracket \\
 \\
 \llbracket \phi_1 \vee \phi_2 \rrbracket &= \text{checkboth or } \llbracket \phi_1 \rrbracket \llbracket \phi_2 \rrbracket \\
 \\
 \llbracket \neg \phi \rrbracket &= \text{checkone neg } \llbracket \phi \rrbracket
 \end{aligned}$$

Figure 27: Encoding of modal logic

```

[[@o]] = λx : smatch x
      | [o ... _] → ⊤
      | _ → ⊥

[[◇o]] = fix(λf : (s → Bool).λx : s.match x
      | [o ... _] → ⊤
      | [_(a1, ..., an)] → or(...(or(f [a1])(f [a2])) ...) (f [an])
      | _ → ⊥)

[[□o]] = λx : s.match x
      | [_(a1, ..., an)] → and(...(and([[◇o]] a1)([[◇o]] a2))...) ([[◇o]] an)
      | _ → ⊤

```

## 9 Implementation

### 9.1 Representing Syntax

The generalized editor calculus[1] 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[13] has been used in the Mentor [3] and CENTAUR [2] 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[10] 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)[21], 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 [14].

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.

### 9.1.1 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  $s$ .

From this specification language, it is possible to extract what is assumed by the generalized editor calculus[1]. 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 7: Syntax of a small C language

Following is a specification of a subset of the C language[9], per the described



specification language.

$p \in Prog$	$s \in Stmt$
$vd \in VariableDecl$	$fd \in FunDecl$
$t \in Type$	$id \in Id$
$e \in Exp$	$b \in Block$
$fa \in Funarg$	$cond \in Conditional$
$int \in Int$	$char \in Char$
$bool \in Bool$	$string \in String$



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 a pattern that allows operators to bind identifiers within the following terms.

### Example 8: Syntax of a small SQL language

Below is the syntax of a subset of the PostgreSQL[5] dialect of SQL:

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

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

where '%string' and '%number' and '%char' are placeholders for any parsable sequence of characters and numbers by the PostgreSQL language.

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.

To make the specification language parseable, a more computer-friendly format is presented in figure Fig. 28. 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 '#'. See figure Fig. 29 for an example.

**Figure 28: Specification language in BNF notation**

**Figure 29: 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
```

It is also assumed that the first non-terminal from the derivations is the starting symbol.

## 9.2 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 updates to both the definitions and signatures of some functions. This presents a challenge in ensuring that these functions maintain their intended behaviour after the updates.

### Example 9: Algebraic data types for a small C syntax

Given example Example 7, one can generate the following custom types in Elm:

```
1 type alias Bind a b =
2   ( 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
15     Block)
16   | FunDecl2 Type (Bind Id FunDecls) Type (Bind Id (
17     Bind Id Block))
18   | FunDeclDone
19 type Statement
20   = Assignment Id Exp
21   | StmtFunCall Id Funargs
22   | Return Exp
23   | Conditional Conditional
24 type Funargs
25   = ArgSingle Funarg
26   | ArgCompound Funargs Funarg
27 type Funarg
28   = Funarg Type Id
29 type Conditional
30   = IfElse Exp Block Block
31 type Statements
32   = SSingle Statement
33   | SCompound Statements Statement
34 type Type
35   = TInt
36   | TChar
37   | TBool
38 type Exp
39   = Num
40   | Char
41   | Bool
42   | Plus Exp Exp
43   | Equals Exp Exp
44   | ExpFunCall Id Funargs
45   | ExpId Id
```

```

44 type Id
45     = Ident String

```

The *Bind* type alias is simply a tuple of 2 given type variables. This would be a part of the generated source code.

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 involves designing a model capable of holding the necessary information from any syntax. An approach for this is provided in figure ?? . However, this approach requires additional well-formedness checks on any given term concerning the syntax.

As an alternative to having specific types for each given syntactic category (as in example Example 9), one can design a generic model with records in Elm, as in listing Listing 3.

```

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: Elm Records for storing syntax information

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.

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.

## 9.3 Generating source code

Elm CodeGen[16] 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[16] is given in figure Fig. 30.

Figure 30: 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 4), which is a direct representation of the syntax as specified in the specification language (Fig. 28).

```
1 type alias RawSyntax =
2   { synCats : List RawSynCat
3     , synCatRules : List RawSynCatRules
4   }
5
6 type alias RawSynCatRules =
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 4: 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 Listing 5). 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.

```

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 5: 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  $\hat{\mathcal{O}}, \hat{\mathcal{S}}$ , 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 7, 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" ]
12    ]

```

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

```

Having generated all sorts and operators as algebraic data types in Elm, the next step is to implement functionality for editor expressions. For example, consider the cursor substitution operator (definition ??), 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 (see listing Listing 6), but typeclasses are not directly supported by Elm. They can however be simulated with Elm records, as shown in the "typeclasses" Elm package[17]. An example of such a simulation is shown in listing Listing 7. 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. This leads to more verbose code and more source code generation.

```

1  -- typeclass
2  class Substitutable a where
3      substitute :: a -> a -> a
4
5  -- instance of typeclass
6  instance Substitutable a where
7      substitute _ replacement = replacement
8
9  -- example usage
10 doIntSub :: Int
11 doIntSub = substitute 1 2

```

Listing 6: Haskell typeclass example

```

1  {-| Simulate a type class
2  -}
3  type alias Substitutable a =
4      { substitute : a -> a -> a }
5
6  {-| Generic instance of the typeclass, we don't need any
7      specific implementation for each type/sort, we just
8      want to assure that the expression and replacement
9      are of the same type. This is constrained by the '
10     substitute' function signature in the (simulated)
11     typeclass.
12 -}
13 substituteAny : Substitutable a
14 substituteAny =
15     { substitute = \_ replacement -> replacement }
16
17 {-| Polymorphic function that can be used with any type
18     that has an instance of the 'Substitutable' typeclass

```

```

13  .
14  -}
15  substitute : Substitutable a -> a -> a -> a
16  substitute substitutable expression replacement =
17      substitutable.substitute expression replacement
18  {-| Example usage
19  -}
20  doIntSub : Int
21  doIntSub =
22      substitute substituteAny 1 2

```

Listing 7: Elm typeclass simulation example

## 9.4 Decomposing trees

The implementation needs to be able to check if a given abt is well-formed, or in other words, if it can be decomposed into  $C[\dot{a}]$ , where  $C$  is a cursor-context (definition ??) and  $\dot{a}$  is a well-formed abt (definition ??).

First of all, 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. For example, see listing Listing 8 for an excerpt of the **Base** type generated for the C language in example Example 7.

```

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

```

Listing 8: Example of the Base type

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[1] 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 [1]. 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 8 (including cursor and hole operators) and their two possible decompositions in figure Fig. 31.

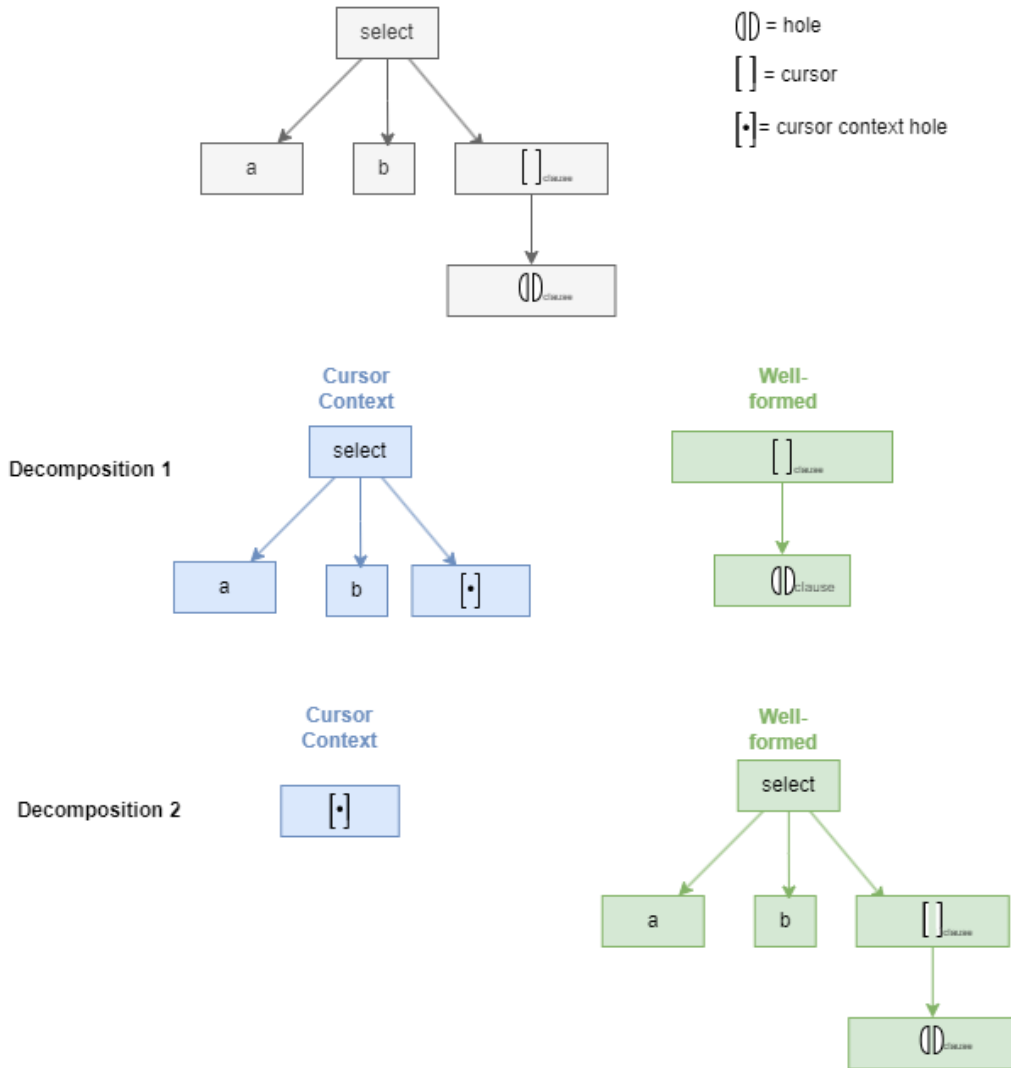


Figure 31: Two different decompositions of the same term

Decomposition should be generic and be possible on a valid abt of any given syntax. For this we have a typeclass called `decomposable` that contains a method called `decompose` which takes a statement made up by the operators of sort  $\mathcal{S}$ , and returns a cursor-context and well-formed-tree pair, if decomposition is possible.

In order to instantiate the `decomposable` type class for any sort, it is then 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 following 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.

#### 9.4.1 Cursor path

Generating a path to the cursor in the abt of sort  $s \in \mathcal{S}$  extended with hole and cursor operators (definition ??) 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.

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[16] has been used to generate a `getCursorPath` function for a `Base` type (Listing 9). 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 11 for an excerpt of the generated `getCursorPath` function for the small C language (Example 7).

```
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 9: getCursorPath function generator

#### Example 11: Generated cursor path finder function

### 9.4.2 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  $[\cdot] \in \mathcal{C}$ . The rest of the tree which has not been traversed will be passed to the next step, generating the well-formed tree.

Functions supporting this are generated by Elm CodeGen, and can be seen in listing Listing 10, 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.

```

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 10: toCctx function generator

### 9.4.3 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 with the well-formed operator  $\dot{o} \in \dot{\mathcal{O}}$  of arity  $(\hat{s})\dot{s}$  indicating that the cursor encapsulates the root of a cursorless abt of sort  $\hat{s}$ . After this, the rest of the tree is traversed, and every operator  $o \in \mathcal{O}$  is replaced with its corresponding cursorless operator  $\hat{o} \in \hat{\mathcal{O}}$ . Like when generating functions supporting cursor context, a very similar approach is taken here, and can be seen in listing Listing 11.

```

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 11: 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.



## References

- [1] Anonymous. “A type-safe generalized editor calculus (Short Paper)”. In: ACM, 2023, pp. 1–7.
- [2] 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>.
- [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](https://en.wikibooks.org/wiki/Haskell/More_on_datatypes). Accessed: 13/03/2024.
- [8] GitHub Inc. *Programming languages*. <https://innovationgraph.github.com/global-metrics/programming-languages>. Accessed: 27/02/2024.
- [9] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth, p. 520. URL: <https://www.iso.org/standard/74528.html>.

- [10] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. <https://www.cs.utexas.edu/users/novak/yaccpaper.htm>. Accessed: 26/03/2024.
- [11] 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>.
- [12] 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.
- [13] 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).
- [14] “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](https://doi.org/10.1007/978-0-387-73003-5_670).
- [15] 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>.
- [16] Elm packages. *Elm CodeGen*. <https://package.elm-lang.org/packages/mdgriffith/elm-codegen/latest/>. Accessed: 18/03/2024.
- [17] Elm packages. *Elm Typeclasses*. <https://package.elm-lang.org/packages/nikita-volkov/typeclasses/latest/>. Accessed: 17/04/2024.
- [18] 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>.
- [19] LaTeX Project Team. *Introduction to LaTeX*. <https://www.latex-project.org/about/>. Accessed: 11/03/2024.
- [20] 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>.

- [21] 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>.