UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Implementation of a type-safe generalized syntax-directed editor

Sune Skaanning Engtorp

Advisor: Hans Hüttel

Submitted: 31/05/2024

Contents

1 Abstract

2 Introduction

2.1 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[18] 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[18] 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[14], 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[14] 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**].

2.2 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 also presented The Synthesizer Generator[17] 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[12], 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].

2.3 Partial evaluation

One of the goals of this project is to implement a generic syntax-directed editor based on the editor calculus proposed by [1]. Implementing such a generic editor relates to the question of how to balance between generality and modularity, brought up by Neil D. Jones[11] 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[10] 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[10] 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), 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.

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

Listing 1: Two-input program p

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

Listing 2: Specialization of program p

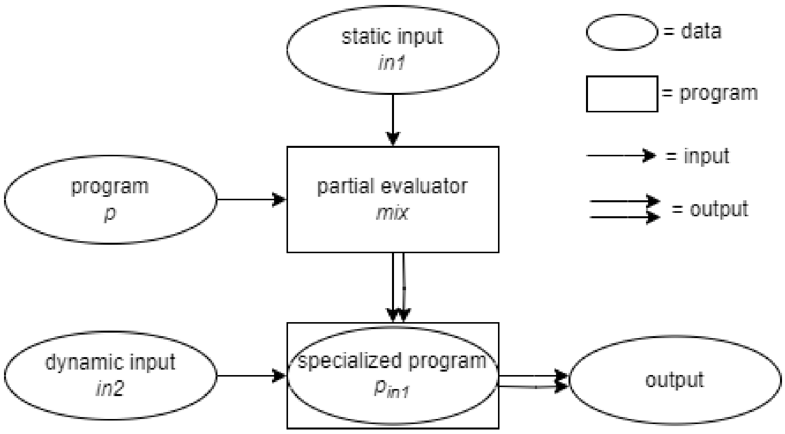

Figure 1: Visualisation of partial evaluation of a two-input program

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:

$$f : t_1 \to t_2 \to t_3$$

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 $t_2 \to t_3$, 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.

3 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 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 $(s_1, ..., s_n)s$, 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 $(s_1, ..., s_n)s$ 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 α. When o is an operator with arity $(s_1, ..., s_n)s$, we say o is an operator of sort s with n arguments, each of sort $s_1, ..., s_n$.

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 $(s_1, ..., s_n)s$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, ..., a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; ...; a_n) \in \mathcal{A}[\mathcal{X}]_s$

4 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 $x_1, ..., x_k.a$ where variables $x_1, ..., x_k$ are bound within the abt a. A finite sequence of bound variables $x_1, ..., x_k$ is represented as $\vec{x}$ for short.

Operators are assigned a generalized arity of the form $(v_1, ..., v_n)s$ where a valence v has the form $s_1, ..., s_k.s$, 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 $(\vec{s}_1.s_1, ..., \vec{s}_n.s_n)s$, 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 $o(\vec{x}_1.a_1; ...; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$

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 $\mathcal{O}$.

5 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[4] et al. which is tailored towards an applied λ-calculus.

5.1 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
$s ::=$	let $x = e$ in s	let	$(e, e.s)s$
$\mid$	e	exp	$(e)s$
$e ::=$	$e_1 + e_2$	$plus$	$(e, e)e$
$\mid$	n	$num[n]$	$()e$
$\mid$	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

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

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

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
$s ::=$	$[s]$	$cursor_s$	$(s)s$
$\mid$	$\llbracket\,\rrbracket_s$	$hole_s$	$()s$
$e ::=$	$[e]$	$cursor_e$	$(e)e$
$\mid$	$\llbracket\,\rrbracket_e$	$hole_e$	$()e$

5.1.1 Editor calculus

The abstract syntax of the general editor calculus (Fig. 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

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

5.1.2 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.

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 $(\vec{\hat{s}}_1.\hat{s}_1, ..., \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ for every $o \in \mathcal{O}$ of arity $(\vec{s}_1.s_1, ..., \vec{s}_n.s_n)s$

3. The family of variables $\hat{\mathcal{X}}$

5.1.3 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.

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

4. The family of variables $\mathcal{X}^C = \hat{\mathcal{X}}$

5.1.4 Well-formed trees

Well-formed trees are trees with a single cursor, where the cursor is located either at the root or one of the immediate children. The rest of the tree is cursorless, hence being well-formed since it only contains a single cursor.

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 $(\vec{\hat{s}}_1.\hat{s}_1, ..., \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ the operator $\dot{o}_i$ of arity $(\vec{\hat{s}}_1.\hat{s}_1, ..., \vec{\hat{s}}_i.\dot{s}_i, ..., \vec{\hat{s}}_n.\hat{s}_n)\dot{s}$ 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.

5.2 Semantics

Reduction rules for editor expressions are presented in Fig. 3, substitution in Fig. 4 and cursor movement in Fig. 5.

Figure 3: Reduction rules for editor expressions

$$\text{(Cond-1)}\ \frac{a \models \phi}{\langle \phi \Rightarrow E_1 | E_2, C[a]\rangle \overset{\epsilon}{\Rightarrow} \langle E_1, C[a]\rangle}$$

$$\text{(Cond-2)}\ \frac{a \not\models \phi}{\langle \phi \Rightarrow E_1 | E_2, C[a]\rangle \overset{\epsilon}{\Rightarrow} \langle E_2, C[a]\rangle}$$

$$\text{(Seq)}\ \frac{\langle E_1, a\rangle \overset{\alpha}{\Rightarrow} \langle E_1', a'\rangle}{\langle E_1 \ggg E_2, a\rangle \overset{\alpha}{\Rightarrow} \langle E_1' \ggg E_2, a'\rangle}$$

$$\text{(Seq-Trivial)}\ \frac{}{\langle nil \ggg E_2, a\rangle \overset{\epsilon}{\Rightarrow} \langle E_2, a\rangle}$$

$$\text{(Recursion)}\ \frac{}{\langle \text{rec}x.E, a\rangle \overset{\epsilon}{\Rightarrow} \langle E[x := rec\ x.E], a\rangle}$$

$$\text{(Context)}\ \frac{a \overset{\pi}{\Rightarrow} a'}{\langle \pi.E, C[a]\rangle \overset{\pi}{\Rightarrow} \langle E, C[a']\rangle}$$

Figure 4: Reduction rules for substitution

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

Figure 5: Reduction rules for cursor movement

$$\text{(Child-i)}\ \frac{}{[\hat{o}(\vec{x}_1.\hat{a}_1; ...; \vec{x}_n.\hat{a}_n)] \overset{child\ i}{\Rightarrow} o(\vec{x}_1.\hat{a}_1; ...; \vec{x}_i.[\hat{a}_i]; ...; \vec{x}_n.\hat{a}_n)}$$

$$\text{(Parent)}\ \frac{}{o(\vec{x}_1.\hat{a}_1; ...; \vec{x}_i.[\hat{a}_i]; ...; \vec{x}_n.\hat{a}_n) \overset{parent}{\Rightarrow} [\hat{o}(\vec{x}_1.\hat{a}_1; ...; \vec{x}_n.\hat{a}_n)]}$$

Satisfaction rules for the propositional connectives are presented in Fig. 6 and modalitites in Fig. 7.

Figure 6: Satisfaction rules for propositional connectives

$$\text{(Negation)}\ \frac{[\hat{a}] \not\models \phi}{[\hat{a}] \models \neg\phi}$$

$$\text{(Conjunction)}\ \frac{[\hat{a}] \models \phi_1 \quad [\hat{a}] \models \phi_2}{[\hat{a}] \models \phi_1 \wedge \phi_2}$$

$$\text{(Disjunction-1)}\ \frac{[\hat{a}] \models \phi_1}{[\hat{a}] \models \phi_1 \vee \phi_2}$$

$$\text{(Disjunction-2)}\ \frac{[\hat{a}] \models \phi_2}{[\hat{a}] \models \phi_1 \vee \phi_2}$$

Figure 7: Satisfaction rules for modalities

$$\text{(At-op)}\ \frac{}{[o(\vec{x}_1.\hat{a}_1; ...; \vec{x}_n.\hat{a}_n)] \models @o}$$

$$\text{(Necessity)}\ \frac{[\hat{a}_1] \models \Diamond ... [\hat{a}_n] \models \Diamond o}{[o(\vec{x}_1.\hat{a}_1; ...; \vec{x}_n.\hat{a}_n)] \models \Box o}$$

$$\text{(Possibly-i)}\ \frac{[\hat{a}_i] \models \Diamond o}{[o(\vec{x}_1.\hat{a}_1; ...; \vec{x}_i.\hat{a}_i; ...; \vec{x}_n.\hat{a}_n)] \models \Diamond o}$$

$$\text{(Possibly-trivial)}\ \frac{[\hat{a}] \models @o}{[\hat{a}] \models \Diamond o}$$

5.3 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 $[\![a]\!]$ 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. 8.

Figure 8: Abstract syntax of extended λ-calculus

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

5.3.1 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. 9.

Figure 9: Typing rules for λ-calculus operators

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

Then, any abt can be encoded in the extended simply typed λ-calculus be the encoding in Fig. 10.

Figure 10: Encoding of abts

$$
[\![o(\vec{x}_1.a_1, ..., \vec{x}_n.a_n)]\!] = o(\lambda \vec{x}_1 : \vec{s}_1.[\![a_1]\!])...(\lambda \vec{x}_n : \vec{s}_n.[\![a_n]\!])
$$

5.3.2 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. 11.

Figure 11: Abstract syntax of extended λ-calculus

Terms

$$
\begin{array}{rcll}
M & ::= & (M_1, M_2) & \text{(pair)} \\
 & | & M.1 & \text{(first projection)} \\
 & | & M.2 & \text{(second projection)}
\end{array}
$$

Types

$$
\tau \quad ::= \tau_1 \times \tau_2 \quad \text{(product type)}
$$

Reduction rules for projecting the first and second value in a pair are given in Fig. 12.

Figure 12: Reduction rules for pairs

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

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

Typing rules for pairs in the λ-calculus are given in Fig. 13.

Figure 13: Typing rules for pairs

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

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

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

The actual encoding of cursor contexts can be seen in Fig. 14. 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

$$
[\![C[a]]\!] = ([\![a]\!], [\![C]\!])
$$

$$
[\![[\cdot]]\!] = [\cdot]
$$

5.3.3 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. 15.

Figure 15: Abstract syntax of extended λ-calculus

$$\begin{array}{llr}
M, N & ::= match\ M\ \overrightarrow{p \to N} & \text{(match construct)} \\
p & ::= x & \text{(variable)} \\
 & \mid\ _ & \text{(wildcard)} \\
 & \mid\ o\ \vec{p} & \text{(operator)} \\
 & \mid\ (p_1, p_2) & \text{(pair)} \\
 & \mid\ .p & \text{(binding)}
\end{array}$$

The reduction rules for the match construct are given in Fig. 16, which uses an auxiliary function *binds* which takes a term M and a pattern p and returns a function of variable bindings, e.g. $[x \to M]$ 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

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

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

$$\begin{array}{ll}
binds(M, x) & = [x \to M] \\
binds(M, _) & = [] \\
binds(o\ a_1\ ...\ a_n,\ o\ p_1\ ...\ p_n) & = binds(a_1, p_1) \circ ... \circ binds(a_n, p_n) \\
binds((M, N), (p_1, p_2)) & = binds(m, p_1) \circ binds(N, p_2) \\
binds(M, .p) & = binds(M, p) \\
binds(\lambda x.M, .p) & = binds(M, .p)
\end{array}$$

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. 17.

Figure 17: Typing rules for pattern matching

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

With the *match* construct, auxiliary functions for cursor movement (Fig. 18) 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

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

The encoding of atomic prefix commands $\pi \in Apc$ is done in terms of the cursor movement functions in Fig. 19. 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

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

5.3.4 Editor expressions

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

A fix operator (Fig. 20) 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

$$\text{(E-FixBeta)} \quad \frac{}{fix(\lambda x : T.M) \rightarrow M[x := fix(\lambda x : T.M)]}$$

$$\text{(E-Fix)} \quad \frac{M \rightarrow M'}{fix\ M \rightarrow fix\ M'}$$

The boolean term constants and base types are defined directly in the abstract syntax of the λ-calculus (Fig. 21). 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. 22).

Figure 21: Abstract syntax of extended λ-calculus

Terms

$$\begin{array}{lcll} M & ::= & fix\ M & \text{(fixed point of M)} \\ & | & \top & \text{(true)} \\ & | & \bot & \text{(false)} \\ \\ p & ::= & \top & \text{(match true)} \\ & | & \bot & \text{(match false)} \end{array}$$

Types

$$\begin{array}{lcll} \tau & ::= & Bool & \text{(boolean)} \end{array}$$

Figure 22: *binds* extended with boolean constructs

$$binds(\top, \top) = []$$
$$binds(\bot, \bot) = []$$

Typing rules for fix and booleans can be seen in Fig. 23.

Figure 23: Typing rules for fix and booleans

$$\text{(T-Fix)} \quad \frac{\Gamma \vdash M : T \to T}{\Gamma \vdash fix\ M : T}$$

$$\text{(T-False)} \quad \frac{}{\bot : Bool}$$

$$\text{(T-True)} \quad \frac{}{\top : Bool}$$

With prior definitions, encoding of editor expressions and context configuration is given in Fig. 24.

Figure 24: Encoding of editor expressions and context configuration

$$[\![\pi.E]\!] = \lambda CC : Ctx.[\![E]\!](([\![\pi]\!]C.1), C.2)$$

$$[\![nil]\!] = \lambda C : Ctx.C$$

$$[\![E_1 \ggg E_2]\!] = \lambda C : Ctx.[\![E_2]\!]([\![E_1]\!]C)$$

$$[\![\mathrm{Rec}\ x.E]\!] = fix(\lambda x : (Ctx \to Ctx).[\![E]\!])$$

$$\begin{aligned}[\![\phi \Rightarrow E_1|E_2]\!] = \lambda C : Ctx.&\mathrm{match}([\![\phi]\!]C.1)\\ &| \top \to [\![E_1]\!]\ C\\ &| \bot \to [\![E_2]\!]\ C\end{aligned}$$

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

5.3.5 Conditional expressions

To encode conditional expressions $\phi \in Eec$, auxiliary functions *checkboth*, *checkone*, *or*, *and* and *neg* are defined in Fig. 25.

Figure 25: Auxiliary functions for conditionals

$$\begin{aligned}
\mathit{checkboth} &\stackrel{def}{=} \lambda f:(\mathit{Bool}\to\mathit{Bool}\to\mathit{Bool}).\\
&\quad \lambda g:(s\to\mathit{Bool}).\\
&\quad \lambda h:(s\to\mathit{Bool}).\\
&\quad \lambda a:s.\\
&\quad f(ga)(ha)
\end{aligned}$$

$$\begin{aligned}
\mathit{checkone} &\stackrel{def}{=} \lambda f:(\mathit{Bool}\to\mathit{Bool}).\\
&\quad \lambda g:(s\to\mathit{Bool}).\\
&\quad \lambda a:s.\\
&\quad f(ga)
\end{aligned}$$

$$\begin{aligned}
\mathit{or} &\stackrel{def}{=} \lambda b_1:\mathit{Bool}.\lambda b_2:\mathit{Bool}.\mathrm{match}\ (b_1,b_2).\\
&\quad (\bot,\bot)\to\bot\\
&\quad (_,_)\to\top
\end{aligned}$$

$$\begin{aligned}
\mathit{and} &\stackrel{def}{=} \lambda b_1:\mathit{Bool}.\lambda b_2:\mathit{Bool}.\mathrm{match}\ (b_1,b_2).\\
&\quad (\top,\top)\to\top\\
&\quad (_,_)\to\bot
\end{aligned}$$

$$\begin{aligned}
\mathit{neg} &\stackrel{def}{=} \lambda b:\mathit{Bool}.\mathrm{match}\ b.\\
&\quad \top\to\bot\\
&\quad \bot\to\top
\end{aligned}$$

With the auxiliary functions, the encoding of propositional connectives is given in Fig. 26 and the encoding of modal logic is given in Fig. 27, which also make use of the fix and $match$ functions.

Figure 26: Encoding of propositional connectives

$$[\![\phi_1 \wedge \phi_2]\!] = \textit{checkboth and } [\![\phi_1]\!][\![\phi_2]\!]$$

$$[\![\phi_1 \vee \phi_2]\!] = \textit{checkboth or } [\![\phi_1]\!][\![\phi_2]\!]$$

$$[\![\neg\phi]\!] = \textit{checkone neg } [\![\phi]\!]$$

Figure 27: Encoding of modal logic

$$\begin{aligned}[\![@o]\!] = \lambda x : \text{smatch } x \\ &\quad \mid [o\ \_\ \ldots\ \_] \to \top \\ &\quad \mid \_ \to \bot\end{aligned}$$

$$\begin{aligned}[\![\Diamond o]\!] = \textit{fix}(\lambda f : (s \to \textit{Bool}).\lambda x : s.\text{match } x \\ &\quad \mid [o\ \_\ldots\ \_] \to \top \\ &\quad \mid [\_(\_.a_1, \ldots, \_.a_n)] \to \textit{or}(\ldots(\textit{or}(f\ [a_1])(f\ [a_2]))\ \ldots)(f\ [a_n]) \\ &\quad \mid \_ \to \bot)\end{aligned}$$

$$\begin{aligned}[\![\Box o]\!] = \lambda x : s.\text{match } x \\ &\quad \mid \_(\_.a_1, \ldots, \_.a_n) \to \textit{and}(\ldots(\textit{and}([\![\Diamond o]\!]\ a_1)([\![\Diamond o]\!]\ a_2))\ldots)([\![\Diamond o]\!]\ a_n) \\ &\quad \mid \_ \to \top\end{aligned}$$

6 Implementation

6.1 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[12] 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[9] 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)[19], 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[13].

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.

6.1.1 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 5: Syntax of a small C language

Following is a specification of a subset of the C language[8], per the described specification language.

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

Sort	Term	Arity	Operator
p ::=	fd	$(fd)p$	*program*
b ::=	bi	$(bi)b$	*block*
bi ::=	vd	$(vd)bi$	*blockdecls*
|	s	$(s)bi$	*blockstmts*
|	ϵ	$()bi$	*blockdone*
vd ::=	t id "=" e ";" bi	$(t, e, id.bi)s$	*vardecl*
fd ::=	t_1 id_1 "(" t_2 id_2 ")"	$(t_1, id_1.fd,$	*fundecl1*
	"{" b "}" fd	$t_2, id_2.b)fd$	
|	t_1 id_1 "(" t_2 id_2 ","	$(t_1, id_1.fd, t_2,$	*fundecl2*
	t_3 id_3 ") {" b "}" fd	$t_3, id_2.id_3.b)fd$	
|	ϵ	$()fd$	*fundecldone*
s ::=	id "=" e ";"	$(id, e)s$	*assignment*
|	id "(" fa ");"	$(id, fa)s$	*stmtfuncall*
|	"return " e ";"	$(e)s$	*return*
|	$cond$	$(cond)s$	*conditional*
|	s s	$(s, s)s$	*compstmt*
fa ::=	t id	$(t, id)fa$	*funarg*
|	t id "," fa	$(t, id, fa)fa$	*funargs*
$cond$ ::=	"if (" e ") {" b_1 "} else {" b_2 "}"	$(e, b_1, b_2)cond$	*ifelse*
t ::=	"int"	$()t$	*tint*
|	"$char$"	$()t$	*tchar*
|	"$bool$"	$()t$	*tbool*
e ::=	int	$(int)e$	*int*
|	$char$	$(char)e$	*char*
|	$bool$	$(bool)e$	*bool*
|	e_1 "+" e_2	$(e_1, e_2)e$	*plus*
|	e_1 "==" e_2	$(e_1, e_2)e$	*equals*
|	id "(" fa ")"	$(id, fa)e$	*expfuncall*
|	id	$(id)e$	*expident*
id ::=	%string	$()id$	*ident*

'%int', '%char', '%string' and '%bool' are meta-variables representing any parseable integer, character, sequence of characters and boolean constant by the C language.
This subset is chosen as it can represent a C program consisting of only

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 $(t_1, id_1.fd, t_2, id_2.b)fd$ 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 6: Syntax of a small SQL language

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

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

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
$const$ ::=	%number	$()const$	num
|	"""%string"""	$()const$	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 ::=	$const$	$()exp$	$econst$
|	id	$()exp$	eid

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

```
query in Query
cmd in Command
id in Id
clause in Clause

query ::= " SELECT " id " FROM " id clause # (id,id,
    clause)query # select
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.

6.2 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 behavior after the updates.

Example 7: Algebraic data types for a small C syntax

Given example Example 5, one can generate the following custom types in Elm:

```
type alias Bind a b =
    ( List a, b )
type Prog
    = Program FunDecls
type Block
    = Block BlockItems
type BlockItems
    = BlockDecl VarDecls
    | BlockStmts Statements
```

```
    | BlockDone
type VarDecls
    = VarDecl Type Id Exp BlockItems
type FunDecls
    = FunDecl1 Type (Bind Id FunDecls) Type (Bind Id
   Block)
    | FunDecl2 Type (Bind Id FunDecls) Type (Bind Id (
   Bind Id Block))
    | FunDeclDone
type Statement
    = Assignment Id Exp
    | StmtFunCall Id Funargs
    | Return Exp
    | Conditional Conditional
type Funargs
    = ArgSingle Funarg
    | ArgCompound Funargs Funarg
type Funarg
    = Funarg Type Id
type Conditional
    = IfElse Exp Block Block
type Statements
    = SSingle Statement
    | SCompound Statements Statement
type Type
    = TInt
    | TChar
    | TBool
type Exp
    = Num
    | Char
    | Bool
    | Plus Exp Exp
    | Equals Exp Exp
    | ExpFunCall Id Funargs
    | ExpId Id
type Id
    = Ident String
```

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 involves designing a model capable of holding the necessary information from any syntax. An approach for this is provided in Listing 3.

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

```
type alias Syntax =
{ synCats : [String]
, operators : [Operator]
}

type alias Operator =
{ name : String
, arity : (Maybe [String], String)
, concSyn : String
}
```

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.

6.3 Generating source code

Elm CodeGen[15] 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 offers automatic imports and built-in type inference. Example usage of Elm CodeGen from the documentation[15] is given in Fig. 30.

Figure 30: Elm CodeGen usage

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

```
Elm.declaration "anExample"
    (Elm.record
        [ ("name", Elm.string "a fancy string!")
        , ("fancy", Elm.bool True)
        ]
    )
    |> Elm.ToString.declaration
```

The above will generate following string:

```
anExample : { name : String, fancy : Bool }
anExample =
    { name = "a fancy string!"
    , fancy = True
    }
```

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).

```
type alias RawSyntax =
    { synCats : List RawSynCat
    , synCatRules : List RawSynCatRules
    }

type alias RawSynCatRules =
    { synCat : String
    , operators : List RawOp
    }

type alias RawOp =
    { term : String
    , arity : String
    , name : String
    }

type alias RawSynCat =
    { exp : String
    , set : String
```

```
    }
```

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 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.

```
type alias Syntax =
    { synCats : List SynCat
    , synCatOps : List SynCatOps
    }

type alias SynCat =
    { exp : String
    , set : String
    }

type alias SynCatOps =
    { ops : List Operator
    , synCat : String
    }

type alias SynCatOps =
    { ops : List Operator
    , synCat : String
    }

type alias Operator =
    { term : Term
    , arity : Arity
    , name : String
    , synCat : String
    }

type alias Term =
    String

type alias Arity =
    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 $\dot{\mathcal{O}}, \dot{\mathcal{S}}$, the CodeGen package can generate algebraic data types for every sort and its operators and separate them into their own separate modules (files).

Example 8: From specification parser to Elm CodeGen for small C-language

Given the specification in example Example 5, the parser can produce following `Declaration` for the `Statement` algebraic data type in example Example 7:

```
Elm.customType "Statement"
            [ Elm.variantWith "Assignment"
                [ Elm.Annotation.named [] "Id",
                  Elm.Annotation.named [] "Exp" ]
            , Elm.variantWith "StmtFunCall"
                [ Elm.Annotation.named [] "Id",
                  Elm.Annotation.named [] "Funargs" ]
            , Elm.variantWith "Return"
                [ Elm.Annotation.named [] "Exp" ]
            , Elm.variantWith "Conditional"
                [ Elm.Annotation.named [] "Conditional" ]
    ]
```

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

```
type Statement
    = Assignment Id Exp
    | StmtFunCall Id Funargs
    | Return Exp
    | 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 (Fig. 4), 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 6), but typeclasses are not directly supported by Elm. They can however be simulated with Elm records, as shown in the ”typeclasses” Elm package[16]. An example of such a simulation is shown in 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.

```haskell
-- typeclass
class Substitutable a where
    substitute :: a -> a -> a

-- instance of typeclass
instance Substitutable a where
    substitute _ replacement = replacement

-- example usage
doIntSub :: Int
doIntSub = substitute 1 2
```

Listing 6: Haskell typeclass example

```elm
{-| Simulate a type class
-}
type alias Substitutable a =
    { substitute : a -> a -> a }

{-| Generic instance of the typeclass, we don't need any specific implementation for each type/sort, we just want to assure that the expression and replacement are of the same type. This is constrained by the `substitute` function signature in the (simulated) typeclass.
-}
substituteAny : Substitutable a
substituteAny =
    { substitute = \_ replacement -> replacement }

{-| Polymorphic function that can be used with any type that has an instance of the `Substitutable` typeclass.
-}
substitute : Substitutable a -> a -> a -> a
substitute substitutable expression replacement =
    substitutable.substitute expression replacement

{-| Example usage
```

```
-}
doIntSub : Int
doIntSub =
    substitute substituteAny 1 2
```
Listing 7: Elm typeclass simulation example

6.4 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 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. For example, see Listing 8 for an excerpt of the `Base` type generated for the C language in Example 5.

```
type Base
    = Prog Prog
    | Block Block
    | BlockItems BlockItems
    | VarDecls VarDecls
    | 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 6 (including cursor and hole operators) and their two possible decompositions in figure Fig. 31.

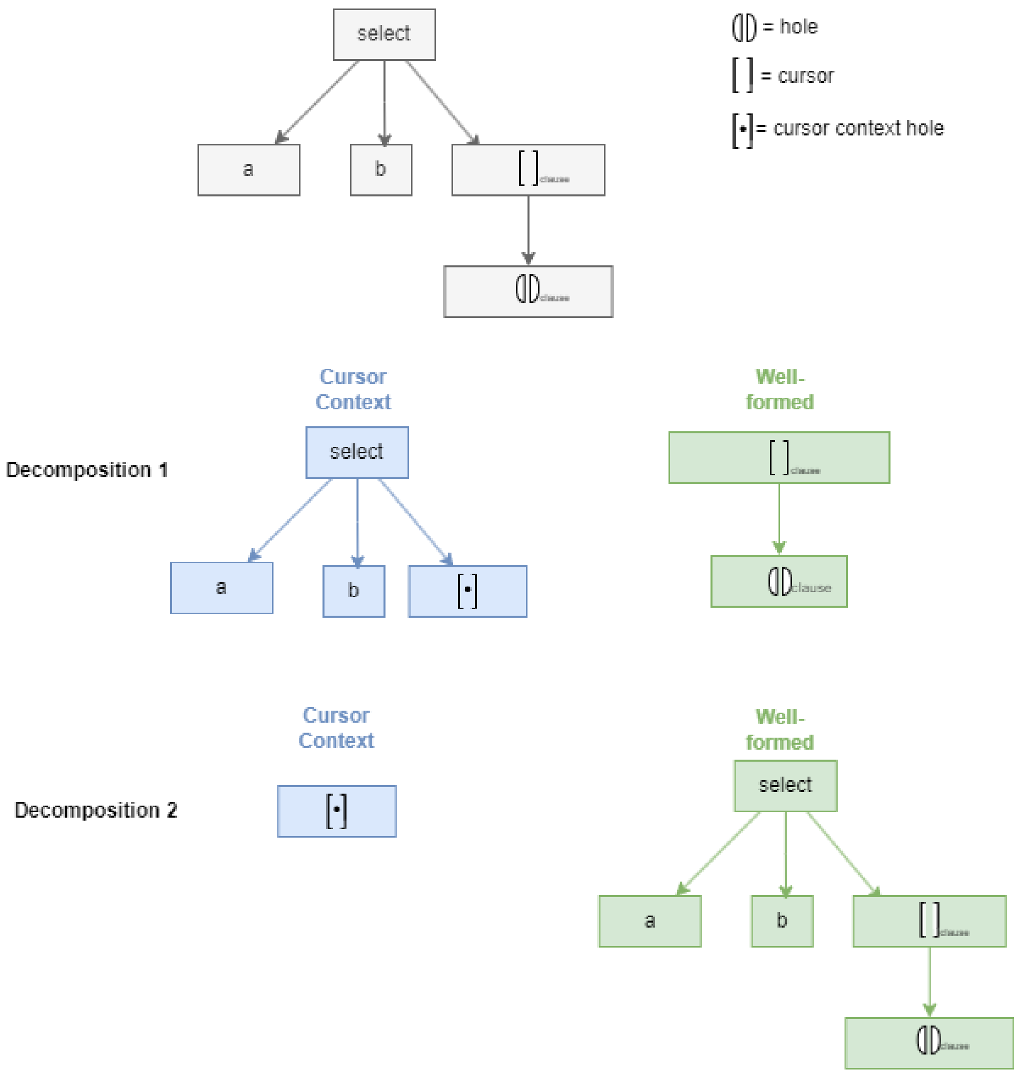

Figure 31: Two different decompositions of the same term

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.

6.4.1 Cursor path

Generating a path to the cursor in the abt of sort $s \in \mathcal{S}$ extended with hole and cursor operators (Definition 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 $(\vec{s}_1.s_1, ..., \vec{s}_i.s_i, ..., \vec{s}_n.s_n)\mathcal{S}$ was followed to locate the cursor. See listing Listing 9 for pseudocode demonstrating how the function generator determines what the switch cases should be for each operator in the syntax.

```
genPath : Op -> [Int] -> [Int]
genPath op path =
  map (\op ->
    if op is Cursor then
      branch path
    else
      case (length op.arity) of
        0 -> branch []
        1 -> recursive call to "cursorPath" [op.child1, path ++ [1]]
        2 -> recursive call to "cursorPath" with op.child1 and path ++ [
             ++
           recursive call to "cursorPath" with op.child2 and path ++ [
        3 -> ...
  ) operators
```

Listing 9: Cursor path generation pseudocode

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[15] has been used to generate a `getCursorPath` function for a `Base` type (Listing 10). 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 9 for an excerpt of the generated `getCursorPath` function for the small C language (Example 5).

```
createGetCursorPath : Syntax -> Elm.Declaration
createGetCursorPath syntax =
    Elm.declaration "getCursorPath" <|
        Elm.withType
            (Type.function
                [ Type.list Type.int
                , Type.named [] "Base"
                ]
                (Type.list Type.int)
            )
            (Elm.fn2
                ( "path", Nothing )
                ( "base", Nothing )
                (\_ base ->
                    Elm.Case.custom base
                        (Type.named [] "Base")
                        (getBranchList syntax)
                )
```

```
            )
```

Listing 10: getCursorPath function generator

Example 9: Generated cursor path finder function

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

```
getCursorPath : List Int -> Base -> List Int
getCursorPath path base =
  case base of
    P p ->
      case p of
        Program arg1 ->
          getCursorPath (path ++ [ 1 ]) (Fd arg1)

        Hole_p ->
          []

        Cursor_p _ ->
          path

    B b ->
      case b of
        Block arg1 ->
          getCursorPath (path ++ [ 1 ]) (Bi arg1)

        Hole_b ->
          []

        Cursor_b _ ->
          path
```

6.4.2 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 $[\,\cdot\,] \in C$. 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 11, 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.

```
createToCCtxFuns : Syntax -> List Elm.Declaration
createToCCtxFuns syntax =
  List.map createToCCtxFun syntax.synCatOps ++
  [ Elm.declaration "toCCtx" <|
    Elm.withType
      (Type.function
        [ Type.named [] "Base"
        , Type.list Type.int ]
        (Type.tuple
          (Type.named [] "Cctx")
          (Type.named [] "Base"))
      ) <|
      Elm.fn2
        ( "base", Nothing )
        ( "path", Nothing )
        (\base path ->
          Elm.Case.custom base
            (Type.named [] "Base")
            (List.map
              (\synCatOp ->
                Elm.Case.branchWith
                  synCatOp.synCat
                  1
                  (\exps ->
                    Elm.apply
                      (Elm.val <|
                      "toCCtx_" ++ synCatOp.synCat)
                      (exps ++ [ path ])
                  )
              )
              syntax.synCatOps
              )
        )
  ]
```

Listing 11: toCCtx function generator

6.4.3 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 12.

```
createToWellFormedFun : Syntax -> Elm.Declaration
createToWellFormedFun syntax =
  Elm.declaration "toWellformed" <|
    Elm.withType
      (Type.function
        [ Type.named [] "Base" ]
        (Type.named [] "Wellformed")) <|
      Elm.fn
          ( "base", Nothing )
          (\base ->
            Elm.Case.custom
                (Elm.apply
                  (Elm.val "consumeCursor")
                  [ base ])
                (Type.named [] "Base")
                (List.map
                  (\synCatOp ->
                    branchWith synCatOp.synCat
                        1
                        (\exps ->
                          Elm.apply
                              (Elm.val <|
                                "Root_" ++
                                synCatOp.synCat ++
                                "_CLess")
                              [ Elm.apply
                                (Elm.val <|
                                  "toCLess_" ++
                                  synCatOp.synCat)
                                exps
                              ]
                        )
                  )
```

```
                        syntax.synCatOps
                    )
            )
```

Listing 12: 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.

7 Editor Examples

8 Conclusion

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. 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. 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] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth, p. 520. URL: https://www.iso.org/standard/74528.html.

[9] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. https://www.cs.utexas.edu/users/novak/yaccpaper.htm. Accessed: 26/03/2024.

[10] Neil D. Jones. "An Introduction to Partial Evaluation". In: *ACM Comput. Surv.* 28.3 (1996), pp. 480–503. DOI: 10.1145/243439.243447. URL: https://doi.org/10.1145/243439.243447.

[11] 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.

[12] 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. URL: https://doi.org/10.1016/0167-6423(83)90009-6.

[13] "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. URL: https://doi.org/10.1007/978-0-387-73003-5_670.

[14] 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. URL: https://doi.org/10.1145/3290327.

[15] Elm packages. *Elm CodeGen*. https://package.elm-lang.org/packages/mdgriffith/elm-codegen/latest/. Accessed: 18/03/2024.

[16] Elm packages. *Elm Typeclasses*. https://package.elm-lang.org/packages/nikita-volkov/typeclasses/latest/. Accessed: 17/04/2024.

[17] 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. URL: https://doi.org/10.1145/800020.808247.

[18] 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. URL: https://doi.org/10.1145/358746.358755.

[19] 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.