Master's thesis

Implementation of a type-safe generalized syntax-directed editor

Sune Skaanning Engtorp

Advisor: Hans Hüttel

UNIVERSITY OF COPENHAGEN

Agenda

- Motivation and background
- Goal of the project
- Background
- Implementation
- Editor examples
- Conclusion
- Questions

Motivation and background

- Structure editors
  - Avoid syntax errors
  - (Arguably) Improved code overview
  - With ability to support
    - Typed holes
    - Context-sensitive syntax

Motivation and background (Continued)

- Cornell Program Synthesizer (1981)[1]

placeholder. The following file segment shows with underlines all the possible stopping points for the cursor when the **up** and **down** keys are used:

```
IF (k > 0 )
    THEN statement
    ELSE PUT SKIP LIST ('not positive');
```

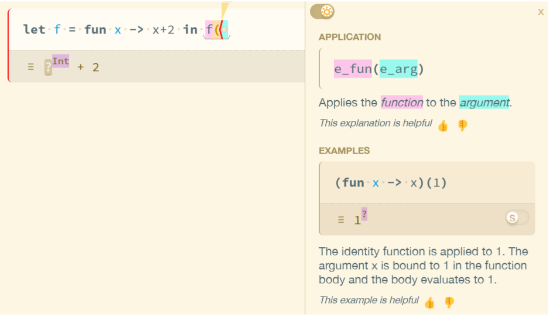
Left and **right** differ from **up** and **down** by also moving the cursor to every character within a phrase:

```
IF ( k > 0 )
    THEN statement
    ELSE PUT SKIP LIST ('not positive' );
```

[1] Teitelbaum and Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment".

Motivation and background (Continued)

- Hazel programming environment (2019)[2]



[2] Omar et al., "Live functional programming with typed holes".

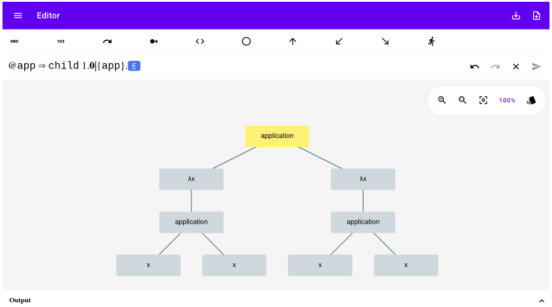
Motivation and background (Continued)

- Type-Safe Structure Editor Calculus[3]

[3]Godiksen et al., "A type-safe structure editor calculus".
[4]Richs-Jensen, Bringgaard, and Zachariasen, "Implementation of a Type-Safe Structure Editor".
[5]Mortensen et al., "A type-safe generalized editor calculus (Short Paper)".

Motivation and background (Continued)

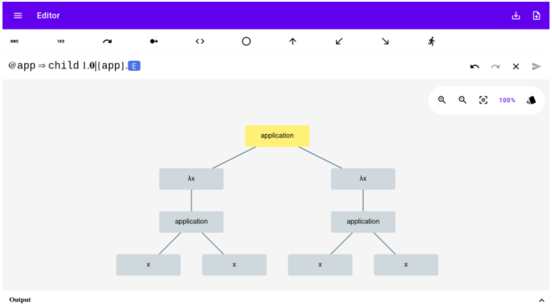
- Type-Safe Structure Editor Calculus[3]
- Implemented by a group of UCPH students[4]



[3] Godiksen et al., "A type-safe structure editor calculus".
[4] Richs-Jensen, Bringgaard, and Zachariasen, "Implementation of a Type-Safe Structure Editor".
[5] Mortensen et al., "A type-safe generalized editor calculus (Short Paper)".

Motivation and background (Continued)

- Type-Safe Structure Editor Calculus[3]
- Implemented by a group of UCPH students[4]
- Generalized by a group of AAU students[5]



[3] Godiksen et al., "A type-safe structure editor calculus".
[4] Richs-Jensen, Bringgaard, and Zachariasen, "Implementation of a Type-Safe Structure Editor".
[5] Mortensen et al., "A type-safe generalized editor calculus (Short Paper)".

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Goal of the Project

- Implement an editor based on the generalized calculus
- Criteria for a good implementation:
  - Editing the abstract syntax of any program directly
  - Generic editing
  - Handling context-sensitive syntax
  - Multiple views of code being edited
  - Non-challenging way of specifying syntax
- Minimum viable product:
  - Editing the abstract syntax of any program directly
  - Generic editing

Picking good language examples

- What makes a good set of examples?
    - Different paradigms and purposes:
        - General purpose programming language
        - Domain-specific language
        - Markup language
    - Popular (present in GitHub top 30 ranking[6])

[6] GitHub Inc. *Programming languages.* https://innovationgraph.github.com/global-metrics/programming-languages. Accessed: 27/02/2024.

Picking good language examples

- What makes a good set of examples?
  - Different paradigms and purposes:
    - General purpose programming language
    - Domain-specific language
    - Markup language
  - Popular (present in GitHub top 30 ranking[6])
- Examples:
  - C
  - SQL
  - LaTeX

[6] GitHub Inc. *Programming languages.* https://innovationgraph.github.com/global-metrics/programming-languages. Accessed: 27/02/2024.

Picking good language examples (Continued)

- A C program with syntax errors

```
int main() {
    int x = 0;
    for (int i; i < 5; i++) {
        x++;
    }
    return 0;
}
```

Picking good language examples (Continued)

- A SQL query with syntax errors

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

Picking good language examples (Continued)

- A LaTeX document with syntax errors

```
...
\begin{equation}
    \|\vect{v}\| = \sqrt{\sum_{i=1}^n v_i^2}
\end{equation}
...
```

Background

- Abstract syntax
- Generalized editor calculus

Abstract Syntax Trees

- Described by Harper[7]
- Set of sorts $\mathcal{S}$
- Arity-indexed family of operators $\mathcal{O}$
- Sort-indexed family of variables $\mathcal{X}$

[7] Harper, *Practical Foundations for Programming Languages (2nd. Ed.)*

Abstract Syntax Trees

- Described by Harper[7]
- Set of sorts $\mathcal{S}$
- Arity-indexed family of operators $\mathcal{O}$
- Sort-indexed family of variables $\mathcal{X}$

- $\mathcal{S} = \{\mathit{exp}\}$
- $\mathit{plus} \in \mathcal{O}_\alpha$ with arity $\alpha = (\mathit{exp}_1, \mathit{exp}_2)\mathit{exp}$

[7] Harper, *Practical Foundations for Programming Languages (2nd. Ed.)*

Abstract Binding Trees

- Enriched AST with bindings
- All operators are assigned generalized arity $(\vec{x_1}.x_1, \ldots, \vec{x_n}.x_n)s$

Abstract Binding Trees

- Enriched AST with bindings
- All operators are assigned generalized arity $(\vec{x_1}.x_1, \ldots, \vec{x_n}.x_n)s$
- $\mathcal{S} = \{exp, stmt\}$
- $let \in \mathcal{O}_\alpha$ with arity $\alpha = (exp_1, exp_2.stmt)stmt$

Generalized editor calculus

- Generalization of a type-safe structure editor calculus[8]
  - Evaluate programs partially with breakpoints and typed holes
  - Type system: If $p, \Gamma_e \vdash \langle E, a \rangle : ok$ and reduction $\langle E, a \rangle \xrightarrow{\alpha} \langle E', a' \rangle$, then $(p, \Gamma_e) \downarrow_\alpha \vdash \langle E', a' \rangle : ok$

[8] 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.

Abstract syntax of a language

- Assumes that abstract syntax of a language is given by:
  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 $(s)s$

Editor calculus

- Abtract syntax of general editor calculus

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

Cursorless trees

- Trees without cursors - crucial for defining cursor contexts and well-formed trees

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 $(\vec{\hat{s}}_1.\hat{s}_1, \ldots, \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ for every $o \in \mathcal{O}$ of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, excluding cursors
3. The family of variables $\hat{\mathcal{X}}$

Cursor context

- Holds information about the current tree, up until a context hole

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

Well-formed trees

- Well-formed: contains only a single 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 $(\vec{\hat{s}}_1.\hat{s}_1, \ldots, \vec{\hat{s}}_n.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ the operator $\dot{o}_i$ of arity $(\vec{\hat{s}}_1.\hat{s}_1, \ldots, \vec{\hat{s}}_i.\dot{s}_i, \ldots, \vec{\hat{s}}_n.\hat{s}_n)\dot{s}$ is added to $\dot{\mathcal{O}}$
4. The family of variables $\dot{\mathcal{X}} = \hat{\mathcal{X}}$

Semantics (Editor Expressions)

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

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

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

Semantics (Substitution and cursor movement)

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

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

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

Semantics (Conditionals and modal logic)

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

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

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

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

Encoding the generalized editor calculus in an extended λ-calculus

- Simply-typed λ-calculus with pairs, pattern matching and recursion
- Assuming that:
  - Type system of the simply-typed λ-calculus is sound
  - Encoding is correct
  - Then any instance of the editor will have a sound type system

Extended λ-calculus

Terms

$$
\begin{array}{llr}
M & ::= \lambda x : \tau.M & (\mathit{abstraction}) \\
& \mid \; M_1 M_2 & (\mathit{application}) \\
& \mid \; x & (\mathit{variable}) \\
& \mid \; o & (\mathit{operator}) \\
& \mid \; (M_1, M_2) & (\text{pair}) \\
& \mid \; M.1 & (\text{first projection}) \\
\ldots & & \\
M, N & ::= \mathit{match}\ M\ \overrightarrow{p \to N} & (\text{match construct}) \\
p & ::= x & (\text{variable}) \\
\ldots & &
\end{array}
$$

Types

$$
\begin{array}{llr}
\tau & ::= \tau_1 \to \tau_2 & (\mathit{function}) \\
& \mid \; s & (\mathit{sort}) \\
& \mid \; \tau_1 \times \tau_2 & (\text{product type}) \\
& \mid \; \mathit{Bool} & (\text{boolean})
\end{array}
$$

Encoding abts and editor expressions

- Typing rules for operators

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

- Encoding of abts

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

- Encoding of editor expressions

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

$$\dots$$

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

Implementation

- Representing syntax
- Code generation vs. generic model
- Generating source code
- Editor expressions

Representing syntax

- Abstract syntax per Robert Harper[9]
  - Set of sorts $\mathcal{S}$
  - Arity-indexed family of operators $\mathcal{O}$
  - Sort-indexed family of variables $\mathcal{X}$
  - Binders: $(\vec{x_1}.x_1)s$

[9] Harper, *Practical Foundations for Programming Languages (2nd. Ed.)*

Representing syntax

- Abstract syntax per Robert Harper[9]
  - Set of sorts $\mathcal{S}$
  - Arity-indexed family of operators $\mathcal{O}$
  - Sort-indexed family of variables $\mathcal{X}$
  - Binders: $(\vec{x_1}.x_1)s$
- How can a user provide this in a non-challenging way?

[9] Harper, *Practical Foundations for Programming Languages (2nd. Ed.)*

Representing syntax (Continued)

- Metal[10]

[10] Kahn et al., "Metal: A Formalism to Specify Formalisms".
[11] Wang et al., "The Zephyr Abstract Syntax Description Language".
[12] Li and Jain, "Abstract Syntax Notation One".

Representing syntax (Continued)

- Metal[10]
- Zephyr ASDL[11]

[10] Kahn et al., "Metal: A Formalism to Specify Formalisms".
[11] Wang et al., "The Zephyr Abstract Syntax Description Language".
[12] Li and Jain, "Abstract Syntax Notation One".

Representing syntax (Continued)

- Metal[10]
- Zephyr ASDL[11]
- ASN.1[12]

[10] Kahn et al., "Metal: A Formalism to Specify Formalisms".
[11] Wang et al., "The Zephyr Abstract Syntax Description Language".
[12] Li and Jain, "Abstract Syntax Notation One".

Representing syntax (Continued)

- Metal[10]
- Zephyr ASDL[11]
- ASN.1[12]
- Common problem: no support for binders

[10] Kahn et al., "Metal: A Formalism to Specify Formalisms".
[11] Wang et al., "The Zephyr Abstract Syntax Description Language".
[12] Li and Jain, "Abstract Syntax Notation One".

Representing syntax (Continued)

- Let's make our own specification language

$$q \in Query$$
$$cmd \in Command \qquad id \in Id$$
$$const \in Const \qquad clause \in Clause$$
$$cond \in Condition \qquad exp \in Expression$$

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

Representing syntax (Continued)

- Let's make our own specification language

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
...

$q \in Query$	
$cmd \in Command$	$id \in Id$
$const \in Const$	$clause \in Clause$
$cond \in Condition$	$exp \in Expression$

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

Generic model or code generation?

- Generic model:
  - No need for code generation (less work and error prone)
  - However less efficient and needs thorough well-formedness checks
- Code generation:
  - Take advantage of algebraic data types (only well-formed terms can be created)
  - However requires code generation

Generating source code

- Elm CodeGen package[13]
- Can be useful if integrated with language specification parser

```
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
    }
```

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

Generating source code (Continued)

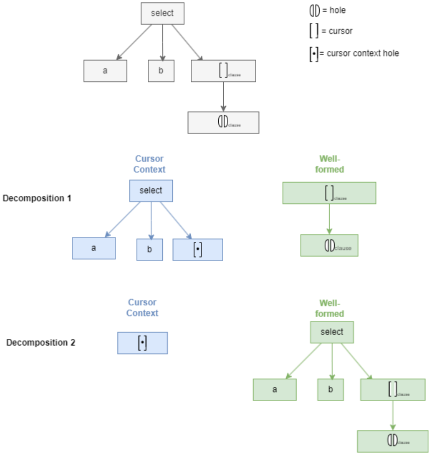
```
Elm.customType "Statement"
                [ Elm.variantWith "Assignment"
                    [ Elm.Annotation.named [] "Id",
                      Elm.Annotation.named [] "Exp" ]
                , Elm.variantWith "StmtFunCall"
                    [ Elm.Annotation.named [] "Id",
                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

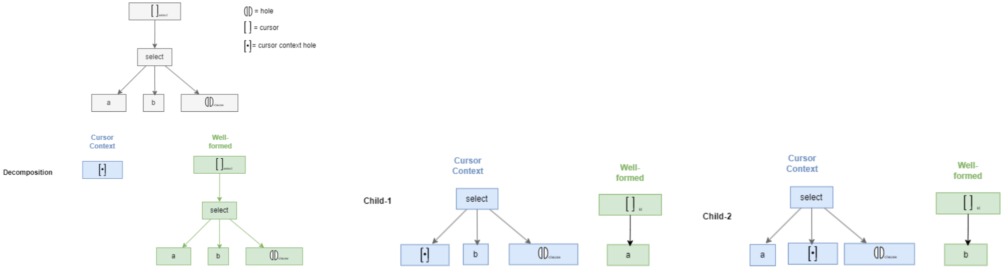
Generating Editor Expression Code

- Unique decomposition
  - Generate a path to the cursor
  - Generate an abt of sort $s^C \in \mathcal{S}^C$ based on the path and stop at the cursor
  - Generate an abt of sort $\dot{s} \in \dot{\mathcal{S}}$ based on the rest of the tree that was not traversed



Generating Editor Expression Code (Continued)

- Cursor movement (child and parent)



Generating Editor Expression Code (Continued)

- Substitution
- Conditionals
- Sequence
- Recursion

Editor examples

- C
- SQL
- LaTeX

C

```
> example = P <| Program <|Fundecl1 Tint ( [ Ident "main" ], Fundecldone ) Tint ( [ Ident "x" ], Block (Blockstmts)
P (Program (Fundecl1 Tint ([Ident "main"],Fundecldone) Tint ([Ident "x"],Block (Blockstmts (Compstmt (Assignment (I
dent "x") (Cursor_e Hole_e)) (Return (Expident (Ident "x"))))))))
    : Base
> decomposed = decompose example
(Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess "main"],Fundecldone_CLess) Tint_CLess ([Ident_C
Less "x"],Block_CLess_cctx1 (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 (Assignment_CLess_cctx2 (Ident_CLess "x")
 Cctx_hole) (Return_CLess (Expident_CLess (Ident_CLess "x"))))))),Root_e_CLess Hole_e_CLess)
    : ( Cctx, Wellformed )
> movedup = Maybe.withDefault (Cctx_hole, Root_e_CLess Hole_e_CLess) <| parent decomposed
(Program_CLess_cctx1 (Fundecl1_CLess_cctx4 Tint_CLess ([Ident_CLess "main"],Fundecldone_CLess) Tint_CLess ([Ident_C
Less "x"],Block_CLess_cctx1 (Blockstmts_CLess_cctx1 (Compstmt_CLess_cctx1 Cctx_hole (Return_CLess (Expident_CLess (
Ident_CLess "x"))))))),Root_s_CLess (Assignment_CLess (Ident_CLess "x") Hole_e_CLess))
    : ( Cctx, Wellformed )
> evalCond movedup <| At <| S_CLess <| Assignment_CLess Hole_id_CLess Hole_e_CLess
True : Bool
> evalCond movedup <| Neg <| At <| S_CLess <| Assignment_CLess Hole_id_CLess Hole_e_CLess
False : Bool
> evalCond movedup <| Possibly <| Id_CLess <| Ident_CLess "x"
True : Bool
> evalCond movedup <| Necessarily <| T_CLess <| Tint_CLess
False : Bool
> |
```

SQL

```
> example = Q (Select (Ident "col-a") (Ident "table-b") (Where (Greater (Eident (Ident "col-a")) (Econst (Num 2)))))
Q (Select (Ident "col-a") (Ident "table-b") (Where (Greater (Eident (Ident "col-a")) (Econst (Num 2)))))
    : Base
> decomposed = decompose example
(Cctx_hole,Root_q_CLess (Select_CLess (Ident_CLess "col-a") (Ident_CLess "table-b") (Where_CLess (Greater_CLess (Ei
dent_CLess (Ident_CLess "col-a")) (Econst_CLess (Num_CLess 2))))))
    : ( Cctx, Wellformed )
> new = parent decomposed
Nothing : Maybe ( Cctx, Wellformed )
> child 1 decomposed
Just (Select_CLess_cctx1 Cctx_hole (Ident_CLess "table-b") (Where_CLess (Greater_CLess (Eident_CLess (Ident_CLess "
col-a")) (Econst_CLess (Num_CLess 2)))),Root_id_CLess (Ident_CLess "col-a"))
    : Maybe ( Cctx, Wellformed )
> substitute decomposed (Q_CLess Hole_q_CLess)
Just (Cctx_hole,Root_q_CLess Hole_q_CLess)
    : Maybe ( Cctx, Wellformed )
> evalCond decomposed ◁ At ◁ Q_CLess ◁ Select_CLess Hole_id_CLess Hole_id_CLess Hole_clause_CLess
True : Bool
> evalCond decomposed ◁ Neg ◁ At ◁ Q_CLess ◁ Select_CLess Hole_id_CLess Hole_id_CLess Hole_clause_CLess
False : Bool
> evalCond decomposed ◁ Possibly ◁ Const_CLess ◁ Num_CLess 1
True : Bool
> evalCond decomposed ◁ Necessarily ◁ Const_CLess ◁ Num_CLess 1
False : Bool
```

LaTeX

```
> example = D (Latexdoc (Ident "article") Hole_e Hole_a Hole_a ( [ Ident "myenv" ], Cursor_c (TextContent "Hello W)
D (Latexdoc (Ident "article") Hole_e Hole_a Hole_a ([Ident "myenv"],Cursor_c (TextContent ("Hello World!"))))
    : Base
> decomposed = decompose example
(Latexdoc_CLess_cctx5 (Ident_CLess "article") Hole_e_CLess Hole_a_CLess Hole_a_CLess ([Ident_CLess "myenv"],Cctx_ho
le),Root_c_CLess (TextContent_CLess ("Hello World!")))
    : ( Cctx, Wellformed )
> new = parent decomposed
Just (Cctx_hole,Root_d_CLess (Latexdoc_CLess (Ident_CLess "article") Hole_e_CLess Hole_a_CLess Hole_a_CLess ([Ident
_CLess "myenv"],TextContent_CLess ("Hello World!"))))
    : Maybe ( Cctx, Wellformed )
>  child 1 (Maybe.withDefault (Cctx_hole, Root_d_CLess Hole_d_CLess) new)
|
Just (Latexdoc_CLess_cctx1 Cctx_hole Hole_e_CLess Hole_a_CLess Hole_a_CLess ([Ident_CLess "myenv"],TextContent_CLes
s ("Hello World!")),Root_id_CLess (Ident_CLess "article"))
    : Maybe ( Cctx, Wellformed )
> substitute decomposed (C_CLess (TextContent_CLess "Updated content"))
Just (Latexdoc_CLess_cctx5 (Ident_CLess "article") Hole_e_CLess Hole_a_CLess Hole_a_CLess ([Ident_CLess "myenv"],Cc
tx_hole),Root_c_CLess (TextContent_CLess ("Updated content")))
    : Maybe ( Cctx, Wellformed )
> evalCond decomposed ◁ At ◁ C_CLess ◁ TextContent_CLess "x"
True : Bool
> evalCond decomposed ◁ At ◁ C_CLess ◁ CmdContent_CLess Hole_cmd_CLess
False : Bool
> evalCond decomposed ◁ Conjunction (At ◁ C_CLess ◁ CmdContent_CLess Hole_cmd_CLess) (At ◁ C_CLess ◁ TextCont)
False : Bool
> evalCond decomposed ◁ Disjunction (At ◁ C_CLess ◁ CmdContent_CLess Hole_cmd_CLess) (At ◁ C_CLess ◁ TextCont)
True : Bool
> |
```

Conclusion of the project

- MVP has been achieved

Conclusion of the project

- MVP has been achieved
- Some editor expressions are not yet implemented

Conclusion of the project

- MVP has been achieved
- Some editor expressions are not yet implemented
- Missing criteria for a ”good” implementation:
  - Handling context-sensitive syntax
  - Views of code being edited

Future work

- Implement the missing editor expressions
- Handling context-sensitive syntax
- Views of code being edited
- Consider a more concise implementation (maybe in Haskell)
- Add support for adding starting symbol to the specification language

Questions

Thank you for your attention!