Implementing A Type-Safe Structure Editor

Hans Hüttel
hans.huttel@di.ku.dk
Department of Computer Science, University of Copenhagen
2100 København, Denmark

Nicolaj Richs-Jensen
vzq239@alumni.ku.dk
Department of Computer Science, University of Copenhagen
2100 København, Denmark

Thorbjørn Bülow Bringgaard
vwc415@alumni.ku.dk
Department of Computer Science, University of Copenhagen
2100 København, Denmark

Tórur Feilberg Zachariassen
xsz482@alumni.ku.dk
Department of Computer Science, University of Copenhagen
2100 København, Denmark

Abstract

??

CCS Concepts • **Software and its engineering → Semantics**; *Domain specific languages*; • **Theory of computation → Type structures**;

Keywords Structure editors, type systems, functional programming.

ACM Reference Format:
Hans Hüttel, Nicolaj Richs-Jensen, Thorbjørn Bülow Bringgaard, and Tórur Feilberg Zachariassen. 2023. Implementing A Type-Safe Structure Editor. In *Proceedings of the 2023 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Boston, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/???

1 Introduction

Structure editors are an alternative to the standard text editors that eliminate the occurrence of syntax errors by working directly with an abstract syntax tree (AST) and can also be used to give a more intuitive visualization of the program code.

An early example of a structure editor is the Cornell Program Synthesizer from 1981 [3]. It directly edits the ASTs in programs by using a cursor to select nodes and allows for insertions and modifications at the cursor. This editor was not typed meaning that it allowed ill-typed ASTs to be built.

A later example is Hazelnut from 2017 [2]. It is a bidirectionally typed structure editor on ASTs, where it introduces holes that represent uncompleted subtrees. It has dynamic semantics and type consistency of holes are not checked until the hole is built to completeness. The result is that for uncompleted programs the completed parts can still be evaluated and the holes can still be meaningful, even when they are not well-typed.

In their 2020 paper Godiksen et al. [1] described an editor calculus inspired by Hazelnut. The calculus describes the edit actions of a typed structure editor on terms of a simply-typed lambda calculus. The calculus allows for partial evaluation of terms by means of breakpoints and like Hazelnut, terms can be incomplete in the form of holes. However, unlike Hazelnut, the calculus incorporates recursive behaviours and a very expressive spatial logic on ASTs. The type system of the calculus guarantees that well-typed editor expressions will always construct well-typed programs.

In this paper we demonstrate that the editor calculus can serve as the basis of a structure editor by means of an implementation in Elm. This implementation effort involves an implementation of the spatial logic as well as the underlying type system.

In order to create editor expressions that are used to traverse and modify the ASTs, we introduce an editor expression builder that allows us to iteratively build editor expressions.

This paper is structured as follows . Preliminaries are listed in section **??**. The overall principles of Godiksen et al.s editor calculus are described in section **??**. In section **??** we introduce the design of the implementation. Section **??** is about how we model the calculus terms, types and the editor expression builder. The evaluation rules are defined in section **??** together with implementation and execution of said rules. In section **??** we define the type rules and describe the implementation of the type checker. We discuss the user interface and how it combines the calculus in section **??**. In the last section - section 5 - we conclude our project and discuss future work.

PEPM '23, January 2023, Boston, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-8305-9/21/01…$15.00
https://doi.org/???

2 Programs

The program terms that editor expression terms can build and modify are terms of a simply typed λ-calculus extended with breakpoints and holes. The formation rules are

$$D ::= \mathtt{var}\ x \mid \mathtt{const}\ c \mid \mathtt{app} : \tau_1 \longrightarrow \tau_2, \tau_1 \tag{1}$$
$$\mid \mathtt{lambda}\ x : \tau_1 \longrightarrow \tau_2 \mid \mathtt{break} \mid \mathtt{hole} : \tau \tag{2}$$
$$a ::= x \mid c \mid a_1\ a_2 \mid \lambda x : \tau.a \mid [\![a]\!] \mid \langle a \rangle \mid (\!|\,|\!) : \tau \tag{3}$$

The new constructions are $\mathtt{hole} : \tau$ which denotes a hole annotated with type τ, $[\![a]\!]$ which denotes an expression that is directly underneath the cursor and $\langle a \rangle$, which is a breakpoint – meaning that this occurrence of a is left unevaluated.

3 The editor calculus

3.1 Syntax

Editor expressions are given by the following formation rules.

$$\pi ::= \mathtt{eval} \mid \{D\} \mid \mathtt{child}\ n \mid \mathtt{parent} \tag{4}$$
$$\phi ::= \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid @D \mid \Diamond D \mid \Box D \tag{5}$$
$$E ::= \pi.E \mid \phi \Rightarrow E_1 \mid E_2 \mid E_1 \ggg E_2 \mid \mathtt{rec}\ x.E \mid x \mid \mathbf{0} \tag{6}$$
$$D ::= \mathtt{var}\ x \mid \mathtt{const}\ c \mid \mathtt{app} \mid \mathtt{lambda}\ x \mid \mathtt{break} \mid \mathtt{hole} \tag{7}$$

Editor calculus expressions E can examine or modify the subtree of the AST that is currently pointed to by the cursor. Expressions are built from atomic edit action prefixes π. The $\{D\}$ prefix substitutes the content under the cursor by the term constructor D. Moreover, conditional expressions $\phi \Rightarrow E_1 \mid E_2$ will proceed as E_1 if the condition ϕ holds for the current subtree and as E_2 otherwise. We allow for recursive expressions $\mathtt{rec}\ x.E$ where x ranges over recursion variables (we only consider expressions where every x is bound by some $\mathtt{rec}\ x.E$) and sequential composition $E_1 \ggg E_2$.

Conditions ϕ are conditions from a spatial logic on ASTs that contains the usual propositional connectives as well as the modalities $@\phi$, $\Diamond\phi$ and $\Box\phi$. The formula $@\phi$ expresses that ϕ holds for the current subtree, while $\Diamond\phi$ expresses that ϕ in some subtree of the current subtree and $\Box\phi$ expresses that ϕ holds everywhere in the current subtree.

Finally, the `eval` primitive allows us to evaluate the entire AST (up to possible breakpoints).

3.2 Semantics

The semantics of the editor calculus is given by a labelled transition system $(S_E, \mathcal{L}_E, \longrightarrow)$, where $S_E = \mathbf{Edt} \times \mathbf{Ast}$ and $\mathcal{L}_E = \mathbf{Val} \cup \mathbf{Aep} \cup \{\epsilon\}$ and where transitions take the form $\langle E, a \rangle \xrightarrow{\alpha} \langle E', a' \rangle$ with a being well-formed, E being completed and $\alpha \in \mathcal{L}_E$.

Editor expressions are always reduced in configurations of the form $\langle E, a \rangle$, where E is the given editor expression and a is the currently build AST to which E is applied. In terms of the labeled transition system, we incrementally reduce the configuration until E' becomes $\mathbf{0}$, and returns α for each increment. Figure 1 show the different reduction rules for editor expressions.

As an example, the rules (COND-1) and (COND-2) describe that for a biconditional editor expression, $\phi \Rightarrow E_1 \mid E_2$, we either use E_1 or E_2 as editor expression for the next reduction increment, based on whether or not ϕ holds for the AST in the configuration. Hence it just returns ϵ.

4 Implementing the editor calculus

The implementation of the editor calculus requires

- implementing the semantics of the calculus and allowing for an appropriate visualization of editing
- implementing a type checker based on the type system

but perhaps surprisingly, it involves finding a way to script editor calculus expressions. In this section we outline this.

4.1 Implementing the editor calculus semantics

We model the formation rules of the editor calculus by a collection of algebraic datatypes. As an example, expressions are given by the datatype

Listing 1. Formation rules (4) modeled in Elm

```
type Aep
    = Eval
    | Sub Aam
    | Child Ast.Child
    | Parent
```

The rest of the formation rules are defined similarly.

4.2 Abstract syntax trees

We are interested in visualizing the AST to make the editor intuitive for a user. We are not necessarily interested in finding the "best" visualization, nor in implementing a vast number of different representations.

A simple representation would be to represent the AST in a textual form, as seen in figure 2.

$$[\![(\langle(\lambda x(x\ x))\rangle\ (\lambda x(x\ x)))]\!]$$

Figure 2. An AST visualized in textual form.

The notation used for this representation follows that of Godiksen et al. [1]. It is straightforward but can also be difficult to reason about, especially as the AST grows large.

We also implement another representatio, namely as a tree as seen in figure 3.

Switching between the two visualizations is immediate in the user interface.

$$\text{(COND-1)}\quad \frac{a \vDash \phi}{\langle \phi \Rightarrow E_1 | E_2,\, a\rangle \xrightarrow{\epsilon} \langle E_1,\, a\rangle} \qquad \text{(COND-2)}\quad \frac{a \nvDash \phi}{\langle \phi \Rightarrow E_1 | E_2,\, a\rangle \xrightarrow{\epsilon} \langle E_2,\, a\rangle}$$

$$\text{(EVAL)}\quad \frac{a \rightarrow v}{\langle \texttt{eval}.E,\, a\rangle \xrightarrow{v} \langle E,\, a\rangle} \qquad \text{(SEQ)}\quad \frac{\langle E_1,\, a\rangle \xrightarrow{\alpha} \langle E_1',\, a'\rangle}{\langle E_1 \ggg E_2,\, a\rangle \xrightarrow{\alpha} \langle E_1' \ggg E_2,\, a'\rangle}$$

$$\text{(STRUCT)}\quad \frac{E_1 \equiv E_2 \qquad \langle E_2,\, a\rangle \xrightarrow{\alpha} \langle E_2',\, a'\rangle \qquad E_2' \equiv E_1'}{\langle E_1,\, a\rangle \xrightarrow{\alpha} \langle E_1',\, a'\rangle} \qquad \text{(CONTEXT)}\quad \frac{a \xrightarrow{\pi} a'}{\langle \pi.E,\ C[a]\rangle \xrightarrow{\pi} \langle E,\ C[a']\rangle}$$

where $\pi \neq \texttt{eval}$

Figure 1. Editor Expression reduction rules

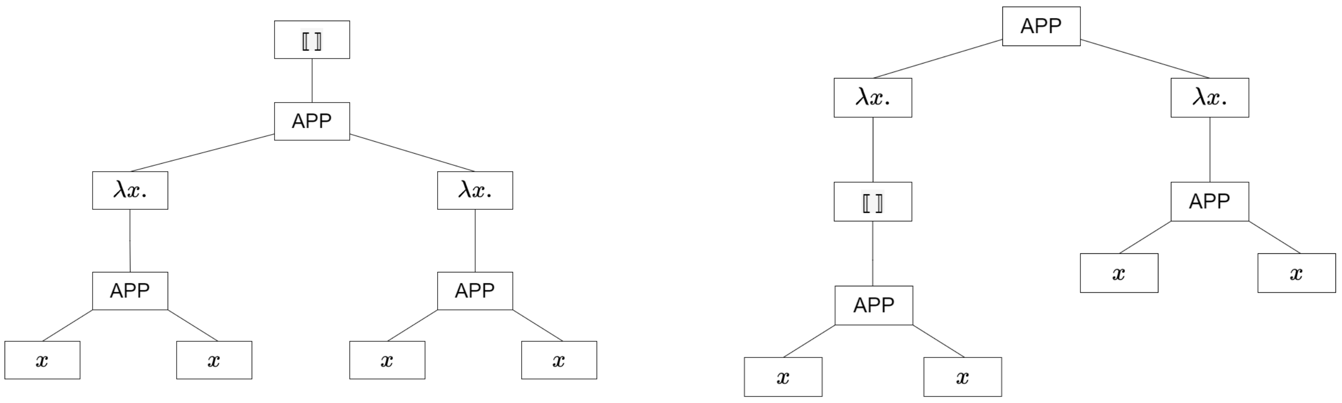

Figure 3. An AST before and after cursor movement visualized in tree form

4.3 Implementing a type checker

We type check with respect to a configuration. We say that a configuration is well-typed iff; the AST is both well-formed and well-typed in the empty AST context and the editor expression is both well-formed and well-typed for the path to the cursor in the AST and the editor context containing all valid paths in the AST [1]. I.e. equation (**??**) where $\emptyset$ is the empty AST context, p is the path from the root to the cursor in a, and Γ_e is the editor context containing some pair for each valid path in a.

$$\frac{\emptyset \vdash a : \tau \quad p, \Gamma_e \vdash E : ok}{p, \Gamma_e \vdash \langle E, a\rangle : ok} \tag{8}$$

4.4 Building editor expressions

To build editor expressions, we need to introduce holes to the editor expressions. We define a hole term constructor for each type of editor expression, as seen in figure 4

```
type Aep                     type Eed
  = Eval                       = Neg Eed
       ⋮                            ⋮
  | AepHole                    | EedHole
type Edt                     type Aam
  = Pre Aep Edt                = Var Var.Id
       ⋮                            ⋮
  | EdtHole                    | AamHole
```

Figure 4. Editor expression definitions with holes

We are now able to build editor expressions by initializing the editor expression builder with an `EdtHole`, and then allow the user to substitute holes with appropriate expressions. As with ASTs, we have the concept of atomic editor expressions, which are editor expressions with holes as children. The user can substitute holes until no holes are left. An editor expression without any holes is said to be *completed*.

We are only interested in evaluating completed editor expressions. Therefore, we need to know when an editor expression is completed, and only then allow the user to evaluate it. To do this, we introduce a type variable to each editor expression type. The type variable is used for recursively defining the type in terms of the type variable, but also as an argument to each hole constructor, as seen in the following listings.

```
type Aep a
    = Eval
    | Sub (Aam a)
    | Child Ast.Child
    | Parent
    | AepHole a

type Eed a
    = Neg (Eed a)
    | Conjunction (Eed a) (Eed a)
    | Disjunction (Eed a) (Eed a)
    | At (Aam a)
    | Possibly (Aam a)
    | Necessarily (Aam a)
    | EedHole a

type Edt a
    = Pre (Aep a) (Edt a)
    | Bicond (Eed a) (Edt a) (Edt a)
    | SeqComp (Edt a) (Edt a)
    | Rec Var.Id (Edt a)
    | Call Var.Id
    | Nil
    | EdtHole a

type Aam a
    = Var Var.Id
    | Con Const.Value
    | App ATyp ATyp
    | Lambda Var.Id ATyp ATyp
    | Break
    | Hole ATyp
    | AamHole a
```

We utilize the `()` and the `Never` types to represent completed and uncompleted editor expressions. If for example `a` in `Edt a` is replaced with `()`, the editor expression can contain holes. Conversely, if we replace `a` with `Never`, it cannot contain holes, since we need a `Never` value to construct a hole. We can therefore use `Edt ()` to represent uncompleted editor expressions, and `Edt Never` to represent completed editor expressions. We have created a type alias for completed and uncompleted editor expressions of every type. For example, the following are the type aliases for `Edt a`.

```
type alias Uncompleted =
    Edt ()

type alias Completed =
    Edt Never
```

This approach creates stronger guarantees by making impossible states impossible. The `toCompleted` function cannot take an `uncompleted` editor expression, return a `completed` editor expression and still have bugs, since we have constrained the types to create type level guarantees.

5 Conclusion

The intention of our project was to act as a form of proof of concept of the editor calculus described by Godiksen et al. [1]. We have described the implementation details of our editor, which models the formal editor calculus quite precisely. Part of the reason we could achieve this was our choice of writing in Elm. We modeled the calculus terms precisely in Elm using algebraic data types and the rules using pattern matches. Furthermore, we are able to construct programs in our editor by extending ASTs iteratively with editor expressions.

With our implemented editor expression builder, we have made it possible to build editor expressions from scratch in such a way that only allows syntactically valid editor expressions to be made and completed editor expressions to be evaluated. Certain commonly used expressions are available as predefined macros.

We partially implemented a type system that — if fully implemented — makes reductions of ill-typed configurations, ill-typed ASTs and runtime errors impossible. The unfinished implementations of the type system resulted in the type system not being used in the final implementation, but the majority of the system is implemented.

Lastly, we implemented visual representations of ASTs and editor expressions. The representations of ASTs can be switched between textual and tree representations. Furthermore, we made it possible to move the view around and zoom in and out of the tree visualization.

5.1 Future work

The most crucial future work would be to fully implement the type system. In particular to implement the last cases of `Eed.types` and `Edt.typed`. As we do not have more details regarding this issue, we will focus on additional features we had considered during development, that we never got around to.

5.1.1 Collapsing subtrees

For the purpose of making the tree-view more visually intuitive, one might want to collapse subtrees. This comes with certain challenges. One problem is that if a subtree can be collapsed, we likely want to ensure that the cursor is not in that subtree, as the cursor would then be hidden. Similarly, if a subtree is collapsed, the cursor should not be able to navigate into that subtree.

To address the latter problem, we can add additional data to every AST term constructor that keeps track of whether the "node" is collapsed. With this data, we can ensure that the function that navigates to a child node first checks if the child is collapsed, and then navigates only if the child is not collapsed.

The solution to the former problem depends on how we let the user collapse a tree. One way would be to add a formation rule to editor expressions that represents toggling the collapsed state of the node under the cursor. If this approach is taken, we do not need to think about the cursor being in the subtree that is being collapsed, since it per definition surrounds the subtree.

A second approach would be to let the user click on a node in the user interface, and then toggle whether that node is collapsed or not. With this approach we would have to check whether the cursor is inside the subtree, if it is being collapsed. If the cursor is not in the subtree, we would simply collapse the subtree, otherwise, we could either do nothing, or move the cursor up each parent until it encapsulates the subtree to be collapsed.

Another consideration we have had for this feature, is that we could try and evaluate the collapsed subtree. If this subtree is evaluated to a constant, variable or some other short expression, we can show this as a summary instead of showing the root of the subtree we have collapsed.

5.1.2 View following cursor

Another feature that would improve usability is to make the default of the view to follow the cursor when moving through the tree. There are a few different ways to do it. The editor can follow the cursor with it being centered in the view. Alternatively it can have the view follow whenever the cursor is near the edge. For either of these cases, the editor could have a button that moves the focus back to the cursor, making it possible to toggle between the locked and the free view.

5.1.3 Saving and loading project

Another feature that would be nice to have is being able to save and load a project. This could for example be implemented as JSON encoders and decoders, where the user could store to or load from a local JSON file.

References

[1] Christian Godiksen, Thomas Herrmann, Hans Hüttel, Mikkel Korup Lauridsen, and Iman Owliaie. 2021. A Type-Safe Structure Editor Calculus. https://doi.org/10.1145/3441296.3441393. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2021)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3441296.3441393

[2] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*.

[3] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (sep 1981), 563–573. https://doi.org/10.1145/358746.358755