

A Type-Safe Structure Editor Calculus

Hans Hüttel

hans.huttel@di.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

Nicolaj Richs-Jensen

vzq239@alumni.ku.dk

Department of Computer Science, University of
Copenhagen

2100 København, Denmark

Tórir Feilberg Zachariassen

xsx482@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 Tórir Feilberg Zachariassen. 2023. A Type-Safe Structure Editor Calculus. In *Proceedings of the 2023 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Boston, USA*. ACM, New York, NY, USA, 3 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 [5]. 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 [3]. 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. [2] 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 4 - we conclude our project and discuss future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

$$D ::= \text{var } x \mid \text{const } c \mid \text{app} : \tau_1 \rightarrow \tau_2, \tau_1 \quad (1)$$

$$\mid \text{lambda } x : \tau_1 \rightarrow \tau_2 \mid \text{break} \mid \text{hole} : \tau \quad (2)$$

$$a ::= x \mid c \mid a_1 a_2 \mid \lambda x : \tau. a \mid \llbracket a \rrbracket \mid \langle a \rangle \mid \langle \rangle : \tau \quad (3)$$

The new constructions are $\text{hole} : \tau$ which denotes a hole annotated with type τ , $\llbracket a \rrbracket$ 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.

$$\pi ::= \text{eval} \mid \{D\} \mid \text{child } n \mid \text{parent} \quad (4)$$

$$\phi ::= \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid @D \mid \diamond D \mid \square D \quad (5)$$

$$E ::= \pi.E \mid \phi \Rightarrow E_1 | E_2 \mid E_1 \gg E_2 \mid \text{rec } x.E \mid x \mid \mathbf{0} \quad (6)$$

$$D ::= \text{var } x \mid \text{const } c \mid \text{app} \mid \text{lambda } x \mid \text{break} \mid \text{hole} \quad (7)$$

Expressions E involve

3.2 Semantics

The semantics of the editor calculus is given by a labelled transition system $(S_E, \mathcal{L}_E, \rightarrow)$, 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.

The rules (COND-1) and (COND-2) describe that for a biconditional editor expression, $\phi \Rightarrow E_1 | E_2$, 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 . The rule (EVAL) describes that if we have the expression $\text{eval}.E$, we evaluate the AST to some value in \mathbf{Val} which we return, and the expression of the new configuration is E . The rule (SEQ) describes that for a composition $E_1 \gg E_2$ we reduce E_1 until it becomes $\mathbf{0}$, while updating the configuration and returning accordingly. The trivial case $E_1 = \mathbf{0}$ in sequential composition is caught by the rule (STRUCT), as it describes that if two editor expressions are equivalent, then two configurations using the editor expressions respectively will also be reduced equivalently, e.g.

we have that the sequential composition $\mathbf{0} \gg E_2$ is equivalent to E_2 and thus reducing the two expressing on an AST is also equivalent. This rule also takes care of recursive editor expressions. The last rule (CONTEXT) describes that if we have a substitution or cursor movement expression of the form $\pi.E$, then we reduce the AST according to the labeled transition system for substitution and cursor movement expressions, and update the configuration to contain the editor expression E and the new AST.

4 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. [2]. 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.

4.1 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.

4.1.1 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,

$$\begin{array}{ll}
\text{(COND-1)} & \frac{a \models \phi}{\langle \phi \Rightarrow E_1 | E_2, a \rangle \xrightarrow{\epsilon} \langle E_1, a \rangle} \\
\text{(EVAL)} & \frac{a \rightarrow v}{\langle \text{eval}.E, a \rangle \xrightarrow{v} \langle E, a \rangle} \\
\text{(STRUCT)} & \frac{E_1 \equiv E_2 \quad \langle E_2, a \rangle \xrightarrow{\alpha} \langle E'_2, a' \rangle \quad E'_2 \equiv E'_1}{\langle E_1, a \rangle \xrightarrow{\alpha} \langle E'_1, a' \rangle} \\
\text{(COND-2)} & \frac{a \not\models \phi}{\langle \phi \Rightarrow E_1 | E_2, a \rangle \xrightarrow{\epsilon} \langle E_2, a \rangle} \\
\text{(SEQ)} & \frac{\langle E_1, a \rangle \xrightarrow{\alpha} \langle E'_1, a' \rangle}{\langle E_1 \gg E_2, a \rangle \xrightarrow{\alpha} \langle E'_1 \gg E_2, a' \rangle} \\
\text{(CONTEXT)} & \frac{a \xrightarrow{\pi} a'}{\langle \pi.E, C[a] \rangle \xrightarrow{\pi} \langle E, C[a'] \rangle} \\
& \text{where } \pi \neq \text{eval}
\end{array}$$

Figure 1. Editor Expression reduction rules

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.

4.1.2 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.

4.1.3 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] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. 2000. Alice: Lessons Learned from Building a 3D System for Novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*. Association for Computing Machinery, New York, NY, USA, 486–493. <https://doi.org/10.1145/332040.332481>
- [2] 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>
- [3] 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)*.
- [4] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Siverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. <https://doi.org/10.1145/1592761.1592779>
- [5] 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>