A Type-Safe Structure Editor Calculus

Hans Hüttel
hans@cs.aau.dk
Department of Computer Science, University of Copenhagen
2100 København, Denmark

Nicolaj

Department of Computer Science, University of Copenhagen
2100 København, Denmark

Thorbjørn

Department of Computer Science, University of Copenhagen
2100 København, Denmark

Tórur

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, Thorbjørn, and Tórur. 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

In their 2020 paper Godiksen et al. [**?**] described an editor calculus for a typed structure editor that works directly with abstract syntax trees (ASTs) making syntax errors impossible. Their editor calculus has structural operational semantics and works for a simple lambda calculus that allows evaluation of unfinished programs through the notion of holes and breakpoints. The calculus uses editor expressions to modify ASTs and has bidirectional communication between editor expressions and ASTs such that an editor expression modifies an AST and an AST provides information that the editor expression needs in order to reduce. Our project aims to work as a proof of concept for the calculus.

PEPM '23, January 2023, Boston, USA

ACM ISBN 978-1-4503-8305-9/21/01…$15.00
https://doi.org/???

There are a multitude of reasons as to why structure editors can work as an alternative to the standard text editors. This includes the elimination of syntax errors by working directly with ASTs and in other instances giving a more intuitive visualization of the program.

An early example of such a structure editor is the Cornell Program Synthesizer from 1981 [**?**]. 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 [**?**]. 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 comparison, the type-safe editor calculus described by Godiksen et al. has static semantics and does assign meaning to ill-typed holes, but still allows partial evaluation of programs through the insertion of breakpoints.

The visualization of the Cornell Program Synthesizer and Hazelnut implementations are both textual and close to that of a text editor. Some structure editors that are motivated by unconventional (non text-based) visualization include Scratch [**?**], where programming is done by moving blocks of different color and combining blocks like a puzzle. Another inclusion is Alice [**?**], which is a programming language for writing 3D graphics where the structure editor is visualized as 3D renderings.

In this paper we will introduce the implementation details of our structure editor derived from the editor calculus described in the paper by Godiksen et al. and argue for why our implementation retains all guarantees from the calculus, even where the calculus may not be directly modeled.

We want to implement the editor in a functional programming language, as they provide expressive types that lend themselves well to modeling the formal descriptions of the

editor. In particular, we want a typed language that has algebraic data types. We considered using either Haskell or Elm, since both are fairly popular functional programming languages. Haskell would be a good choice, but we ended up choosing Elm, primarily because it would be easier to create a user interface in Elm.

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

We want to visualize ASTs and editor expressions as well as interact with them. By interacting we mean to be able to modify an AST by evaluating configurations consisting of an AST and an editor expression, where the editor expression can be any desired expression. Furthermore we want choices available in our visual representation of ASTs, making it possible to change between different visualizations.

Godiksen et al. also describe a way to acquire type-safety in their calculus, by way of their type system. The implementation of their type system should make it impossible to get runtime errors from problems such as invalid ASTs as well as expressions acting upon non-existing subtrees et cetera.

Going from a completely theoretical calculus to a practical implementation, we also have to consider the usability of the editor. While not the main focus we will go through approaches where we make it easier to use certain aspects of the editor, such as using editor expressions and navigating the ASTs.

This paper is structured the following way. 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 2 - we conclude our project and discuss future work.

2 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. [**?**]. 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.

2.1 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.

2.1.1 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.

2.1.2 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.

2.1.3 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