

An Editor Calculus With Undo/Redo

Kjær, Rasmus Rendal; Lundbergh, Magnus Holm ; Nielsen, Magnus Mantzius; Hüttel, Hans

Published in:

Proceedings of 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)

DOI (link to publication from Publisher):

[10.1109/SYNASC54541.2021.00023](https://doi.org/10.1109/SYNASC54541.2021.00023)

Publication date:

2021

Document Version

Publisher's PDF, also known as Version of record

[Link to publication from Aalborg University](#)

Citation for published version (APA):

Kjær, R. R., Lundbergh, M. H., Nielsen, M. M., & Hüttel, H. (2021). An Editor Calculus With Undo/Redo. In C. Schneider, M. Marin, V. Negru, & D. Zaharie (Eds.), *Proceedings of 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (pp. 66-74). Article 9700397 IEEE (Institute of Electrical and Electronics Engineers). <https://doi.org/10.1109/SYNASC54541.2021.00023>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy

If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

An Editor Calculus With Undo/Redo

Rasmus Rendal Kjær

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Magnus Mantzius

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Magnus Holm Lundbergh

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Hans Hüttel

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Abstract—Structure editors provide many potential usability benefits to an end-user by allowing them to edit the AST representation of a program rather than a textual representation of it. In addition, they all but remove syntax errors by only allowing the constructing of programs that are syntactically valid. However, structure editors only rarely include undo/redo functionality into the editor itself, and to the best of our knowledge, an underlying, formal specification for undo/redo has yet to be developed. This paper continues previous work on an editor calculus; we extend the calculus with undo and redo and present a history-based operational semantics of the extension. The history used an underlying graph-based structure, containing a history of user actions in the particular structure editor. We study the expressive power of the calculus, give a simple proof of its Turing-power and use the expressiveness result to show how our history-based extension with undo and redo can be expressed in the original editor calculus.

I. INTRODUCTION

Structure editors view documents not as text but as elements of a formally defined syntax. An editor of this kind operates directly on Abstract Syntax Trees (ASTs); a document is constructed by applying operations that build an AST and move within its structure. The approach arose in the setting of programming environments and was pioneered by Cornell Program Synthesizer [11] and the MENTOR [5] and CENTAUR systems [3]. With this approach it becomes possible to limit the expressiveness to what is syntactically sound in the underlying language without limiting the expressive power of the language to the user.

In a series of papers, Omar et al. have defined an editor calculus called Hazelnut [9], [10] with a formal semantics that describes edit actions on ASTs of a functional programming language as well as the evaluation of incomplete programs and a type system for typing the holes of incomplete programs.

Godiksen et al. [6] define another editor calculus for a simple functional programming language with a resource-aware type system which characterises safe edit sequences: If a calculus expression is well-typed, it will build a syntactically well-formed (possibly incomplete) program.

However, none of these calculi incorporate edit actions that are commonly used. In particular, the useful and common actions of *undo* and *redo* are absent. Undo/redo is a widely used paradigm in user-oriented applications; its first use in a programming environment was in the early work by Teitelman

[12] and has since become common in text editors in general. Undo/redo is typically implemented by incorporating a notion of history that tracks the specific actions that the user takes. Undo and redo actions can then be performed by consulting the history of edit actions; any action that is doable in the program becomes reversible by performing an undo. This paradigm allows the user to correct any errors that they might have made while using the application. If an undo action itself was an error, the user has the ability to reverse that undo with an action known as redo. There are different approaches to modelling the history: one can think of a linear history or record all edit actions and undo/redo actions as a history tree.

The present paper introduces reversibility into the editor calculus of Godiksen et al. [6] and studies the expressive power of the extension within the setting of a branching view of the edit history. While some structure editors such as Scratch implement variations of undo/redo, to the best of our knowledge, this is the first formal semantics of undo/redo in this setting. In order to define the semantics, we take our inspiration from the study of reversible computation, and in particular in the work on reversible process calculus of Danos and Krivine [4]. Godiksen et al. established that their original editor calculus is Turing powerful; we give a much simpler proof of this and use that result to show how our history-based extension with undo and redo can be implemented in the original editor calculus.

The remainder of our paper is structured as follows. Section II describes the syntax of the editor calculus. Section III defines the history-dependent semantics of the calculus. In Section IV we study the expressive power of the editor calculus. Section V is the conclusion and ideas for further work.

II. SYNTAX

In this section we present the syntax of our editor calculus.

A. Abstract Syntax Trees

Expressions in the editor calculus modify expressions in a functional programming language, so we present the syntax of this language first. Programs are expressions in an applied λ -calculus; as we must represent incomplete programs as well, we introduce a notion of holes.

For programs $a \in \mathbf{Ast}$ where \mathbf{Ast} is the set of all ASTs, their syntax is given by the formation rules

$$a ::= x \mid c \mid a_1 a_2 \mid \lambda x. a \mid \llbracket a \rrbracket \mid \langle a \rangle \mid \emptyset$$

The first term constructors are the usual ones from λ -calculus: c ranges over a set of constants. $\lambda x. a$ denotes lambda abstractions, and $a_1 a_2$ denotes application.

The remaining constructs are used in editing. A hole \emptyset represents a part of the program that is not yet constructed. Program ASTs can be evaluated partially even with holes present. Moreover, the syntax allows the use of breakpoints. A breakpoint $\langle a \rangle$ is a node that will halt the evaluation of the program AST once it is reached.

Finally, and importantly, the cursor $\llbracket a \rrbracket$ encapsulates a subtree a , signifying the location at which editor expressions operate.

We say that an AST is *complete* if it is devoid of holes and breakpoints.

B. Editor expressions

Changes to the AST are made through editor expressions $E \in \mathbf{Edt}$. The syntax of these is given by the following formation rules.

$$\begin{aligned} \pi &::= \text{eval} \mid \text{undo} \mid \text{redo } n \mid \{D\} \mid \text{child } n \mid \text{parent} \\ \phi &::= \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid @D \mid \Diamond D \mid \Box D \\ E &::= \pi.E \mid \phi \Rightarrow E_1|E_2 \mid E_1 \gg E_2 \mid \text{rec } x.E \mid x \mid 0 \\ D &::= \text{var } x \mid \text{const } c \mid \text{app} \mid \text{lambda } x \mid \text{break} \mid \text{hole} \end{aligned}$$

In this syntax, $\{D\}$ represents substitutions in the AST, while $D \in \mathbf{Aam}$ represents node modifiers. A node modifier replaces the content encapsulated by the cursor with a hole, a breakpoint or a subtree consisting of a term constructor whose children are holes.

$\pi \in \mathbf{Aep}$ represents atomic prefix commands. For any prefix π to an editor expression E in the form $\pi.E$, the prefix will be evaluated before the expression. If it is not a substitution $\{D\}$, a prefix π can either be `eval`, `undo`, `redo n` , `child n` , or `parent`. The `eval` prefix command evaluates an AST depth-first until arriving at a breakpoint. Function application nodes are evaluated from left to right. Editor expressions `child n` and `parent` signify movement of the cursor in the AST. When `child n` is evaluated, the cursor is moved to the n th child of the node currently encapsulated by the cursor. When `parent` is evaluated, the cursor is moved to the parent node.

The new constructs of the editor calculus studied in this paper are `undo`, which reverses the last substitution, moving the cursor to the position it had before the substitution was done, and `redo n` which reverses an `undo`.

Conditional expressions $\phi \Rightarrow E_1|E_2$ allow us to use the shape of an AST to determine how it is supposed to be edited. The intended meaning is that if the condition ϕ holds, E_1 is evaluated, and if it does not hold, E_2 is evaluated instead.

Conditions $\phi \in \mathbf{Eed}$ are given by a spatial logic [1] that can describe the structure of an AST.

Logical expressions can be built using the Boolean connectives $\neg\phi$, $\phi_1 \wedge \phi_2$, and $\phi_1 \vee \phi_2$ by means of the spatial modalities $@D$, $\Diamond D$, and $\Box D$ that allow us to inspect term constructors, holes and breakpoints.

The modality $@D$ is true iff the cursor is currently located at node D . The modality $\Diamond D$ is true iff D is in a sub-tree of the tree that the cursor encapsulates, and finally $\Box D$ is true iff D is located in all subtrees in the tree that the cursor encapsulates.

The sequential operator $E_1 \gg E_2$ specifies that the editor expression E_1 is to be evaluated before editor expression E_2 , such that E_2 will only be evaluated when E_1 has been reduced to 0. The recursion construct $\text{rec } x.E$ specifies that whenever we reach an occurrence of the recursion variable x in the editor expression E , x will be substituted by $\text{rec } x.E$. This allows for recursive expressions.

Sometimes we will define an expression by a family of defining equations $\{x_i \triangleq e_i(x_1, \dots, x_n) \mid 1 \leq i \leq n\}$ where the behaviour of the expression is that of the variable x_1 . Such a family can be rewritten as a single recursion expression using the Scott-Béki technique [2].

A variable x is defined by the expression $\text{rec } x.E$, and when used outside such an expression, it is a free variable and therefore undefined. To determine if free variables exist, we introduce a property called closedness for editor expressions, which is defined in Definition 1.

Definition 1. We say that a given editor expression E is *closed* iff E holds no free variables. We define a function fv as follows [6]:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\text{rec } x.E) &= fv(E) \setminus \{x\} \\ fv(\pi.E) &= fv(E) \\ fv(\phi \Rightarrow E_1|E_2) &= fv(E_1) \cup fv(E_2) \\ fv(E_1 \gg E_2) &= fv(E_1) \cup fv(E_2) \\ fv(0) &= \{\emptyset\} \end{aligned}$$

Iff $fv(E) = \emptyset$, then E is *closed*.

In the following, we use the following shorthands for editor expressions [6]:

$$\phi_1, \phi_2, \dots \Rightarrow E_1, E_2, \dots \triangleq \phi_1 \Rightarrow E_1 | (\phi_2 \Rightarrow E_2 | \dots) \quad (1)$$

$$\phi \Rightarrow E \triangleq \phi \Rightarrow E | 0 \quad (2)$$

$$\pi \triangleq \pi.0 \quad (3)$$

III. SEMANTICS

We now define the semantics of editor expressions. First, we define the logical semantics of conditions. The sections that follow define the transition rules of editor expressions and programs in our version of the λ -calculus.

A. Conditions

We define the satisfaction relation $a \models \phi$ which tells us if a logical condition ϕ holds for an AST a . We do not present the full definition of the satisfaction relation here, as most of them are straightforward. The rules defining the relation for the $@D$ modality can be found in Figure 1.

[AT-VAR]	$\frac{}{x \models @(\text{var } y)}$
[AT-CONST]	$\frac{}{c \models @(\text{const } c)}$
[AT-HOLE]	$\frac{}{\langle \rangle \models @\text{hole}}$
[AT-APP]	$\frac{}{\hat{a}_1 \hat{a}_2 \models @\text{app}}$
[AT-ABS]	$\frac{}{\lambda x. \hat{a} \models @(\text{lambda } y)}$
[AT-BREAK]	$\frac{}{\langle \hat{a} \rangle \models @\text{break}}$

Fig. 1: Rules defining the satisfaction relation for the $@D$ modality

B. Cursor contexts

We represent the location of the cursor by means of cursor contexts, denoted C . Cursor contexts are given by the formation rules

$$C ::= [\cdot] \mid C \hat{a} \mid \hat{a} C \mid \lambda x. C \mid \langle C \rangle$$

Here \hat{a} is a cursor-less AST as given by the following formation rules:

$$\hat{a} ::= x \mid c \mid \hat{a}_1 \hat{a}_2 \mid \lambda x. \hat{a} \mid \langle \hat{a} \rangle \mid \langle \rangle$$

Using these formation rules we can describe an AST a for which the cursor points to the subtree a' as $C[a']$. That is, we say that $a = C[a']$ if and only if replacing $[\cdot]$ with a' in C gives us a . We describe a' as the *open window* of a , which is the part of the AST that can be edited.

Usually, when describing the properties of the document AST, it is important that there is only one cursor. ASTs with this property are called *well-formed*.

Definition 2 (Well-formed ASTs). *An AST a is well-formed iff it is true that $a = C[\hat{a}]$ for some cursor context C and AST \hat{a} , where \hat{a} is derived from the following formation rules:*

$$\hat{a} ::= \llbracket \hat{a} \rrbracket \mid \lambda x. \llbracket \hat{a} \rrbracket \mid \llbracket \hat{a}_1 \rrbracket \hat{a}_2 \mid \hat{a}_1 \llbracket \hat{a}_2 \rrbracket \mid \langle \llbracket \hat{a} \rrbracket \rangle$$

C. Paths

The position of any node in an AST is given by its path, which is a sequence of positions.

Definition 3 (Path). *A path $p \in \mathbf{Pth}$ is defined by the formation rules [6]:*

$$\begin{aligned} T &::= \text{one} \mid \text{two} \\ p &::= p \ T \mid \epsilon \end{aligned}$$

We call T a *singleton path*.

Whenever an undo action is performed, we should move the cursor to the last place where we performed a substitution. In this case, we need to know the path from the root node to that node. To this end we introduce the following useful function.

Definition 4. *We define a function $\text{path}(a)$, which computes the path to the cursor in an AST from the root node.*

$$\text{path}(a) = \begin{cases} \text{one } \text{path}(a_1) & \text{if } a = a_1 \hat{a} \\ \text{two } \text{path}(a_1) & \text{if } a = \hat{a} a_1 \\ \text{one } \text{path}(a_1) & \text{if } a = \lambda x. a_1 \\ \text{one } \text{path}(a_1) & \text{if } a = \langle a_1 \rangle \\ \epsilon & \text{otherwise} \end{cases}$$

The following definitions allows us to succinctly describe large movements in an AST, and describe a cursor-less version of an existing AST.

Definition 5. *We define $a \xRightarrow{T} a'$ if a' is a except that the cursor has moved one level down along the singleton path T .*

Let $p = T_1 \dots T_{n-1}$. We write $a_1 \xrightarrow{p} a_n$ if $a_1 \xRightarrow{T_1} a_2 \dots a_{n-1} \xRightarrow{T_{n-1}} a_n$.

Definition 6. *Given a well-formed AST a , we define the AST a^- as an AST similar to a , except that the cursor has been removed, where the occurrence of $\llbracket a' \rrbracket$ in the AST is replaced with a' .*

Using Definition 5 and Definition 6, it is possible to move the cursor along a specific path p in the AST by the movement $\llbracket a^- \rrbracket \xrightarrow{p} a'$, by removing the cursor from the AST, placing the cursor at the root node, and moving the cursor along the path p .

D. Structural Congruence

We identify expressions that have the same essential structure by means of a notion of structural congruence. The transition [STRUCT] rule in the semantics that follows will then guarantee that structurally congruent expressions have the same behaviour.

Definition 7. *As in [6], we define structural congruence, denoted $E_1 \equiv E_2$, as the least congruence relation over editor expressions that is closed under the axioms:*

$$0 \ggg E \equiv E \tag{4}$$

$$\text{rec } x. E \equiv E \{ \text{rec } x. E / x \} \tag{5}$$

The axiom (4) tells us that terminated components in a sequential composition disappear and the control is passed on to the continuation. The axiom (5) tells us that recursion is carried out by unfolding recursive definitions: We evaluate

$\text{rec } x.E$ by replacing all instances of the recursion variable x in E with $\text{rec } x.E$.

E. Histories

In the transition system that we shall define, we refer to the history of edit actions that have been performed.

When the user performs a substitution in the editor, they should be able to undo it later, and also redo if undo has been used. This places two requirements on the editor: its substitutions must be reversible, and it must keep an ordered memory of which changes have occurred in the AST.

Reversing a substitution is simply a matter of applying the inverse substitution at the same place in the AST—where an inverse substitution is a substitution that replaces the new content with the old content.

A history is built around the tracking of substitutions. We do this by recording the subtree that was added to the document AST as well as the subtree that was consequently removed.

Definition 8. A delta is denoted $\delta \in \Delta$, where $\Delta = \mathbf{Pth} \times \mathbf{Ast} \times \mathbf{Ast}$. An element $\delta \in \Delta$ is represented as $\delta = (p, a, a^{-1})$, representing a path p to a substitution, the inserted subtree a , and the removed subtree a^{-1} .

A history graph is a directed acyclic graph whose nodes correspond to document ASTs. The edges are each labelled with a delta and a natural number. These natural numbers are needed for ordering the children of a node and for distinguishing between different edit action occurrences that involve the same delta.

For any given node v , its associated AST can be built from the initial document by sequentially applying the substitutions described by the edit actions of the deltas along the path from the root node of the history graph to the node v .

Conversely, the initial document can be obtained from v by using the substitutions defined by a^{-1} along the reverse path from v to the root node of the graph.

Definition 9. A history graph is a directed acyclic graph $H \subseteq \mathbb{V}_{\mathbb{H}} \times \mathbb{E}_{\mathbb{H}}$, where $\mathbb{E}_{\mathbb{H}} = \mathbb{V}_{\mathbb{H}} \times \mathbb{V}_{\mathbb{H}} \times \mathbb{N} \times \Delta$.

We use the degree of a node to enumerate the outgoing edges of a node.

Definition 10. The degree of a node, denoted $\text{deg}_H(v)$ is defined as the number of outgoing edges of v in the history graph H .

This is useful in the case where we construct new edges through substitution editor expressions, to ensure that their values are not identical to the values of previously created edges.

To denote the child or parent of an edge, we use the following notation:

Definition 11. We use $H \vdash v \xrightarrow{\delta}_i v'$ to denote that for a history graph $H = (\mathbf{V}_H, \mathbf{E}_H)$, there exist nodes $v, v' \in \mathbf{V}_H$, and there exists an edge $(v, v', i, \delta) \in \mathbf{E}_H$.

This notation tells us that there exists an edge between v and v' .

To create a new edge in the history graph, we use the following notation:

Definition 12. We use $H[v \xrightarrow{\delta} v']$ to denote a history $H' = (\mathbf{V}_{H'}, \mathbf{E}_{H'})$ as an extension of the history $H = (\mathbf{V}_H, \mathbf{E}_H)$ such that:

$$\begin{aligned} \mathbf{V}_{H'} &= \mathbf{V}_H \cup \{v'\} \\ \mathbf{E}_{H'} &= \mathbf{E}_H \cup \{(v, v', \text{deg}_H(v) + 1, \delta)\} \end{aligned}$$

Here, we create a new node v' in the graph and connect it through an edge to the node v containing the delta δ , where the new node v' represents a new change made to the document AST.

When navigating the history graph, the convention shall be to represent the currently active node v —the current state of the edited document—and the history graph H simply as the pair $\mathcal{H} = (H, v)$ for the sake of brevity.

F. The history-enabled transition system

We now introduce the transition system for editor expressions $(S_E, \mathcal{L}_E, \rightarrow)$, where the set of states $S_E = (\mathbf{Hst} \times \mathbb{V}_{\mathbb{H}}) \times \mathbf{Edt} \times \mathbf{Ast}$, the set of labels $\mathcal{L}_E = \mathbf{Val} \cup \mathbf{Aep} \cup \{\epsilon\}$, and transitions (\rightarrow) have the form $\langle \mathcal{H}, E, a \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E', a' \rangle$, where E is a closed editor expression and a is a well-formed AST.

These transition rules are extended versions of those found in [6]. The extended transition rules for editor expressions can be seen in full in Figure 2.

The transition rules for editor expressions make use of an auxiliary labelled transition system for cursor movements and substitutions. This system is of the form $(S_\pi, \mathcal{L}_\pi, \rightarrow)$, where the set of states $S_\pi = \mathbf{Ast}$ is the set of ASTs, $\mathcal{L}_\pi = \mathbf{Aep}$ is the set of labels, and where the rules for the transition relation (\rightarrow) are described by Figure 3 and Figure 4.

The transition rules for these auxiliary labelled transition systems are identical to those found in [6].

G. Description of transition rules

Most rules in our semantics are simple extensions of those of Godiksen et al. [6] insofar as the history is not directly involved. We shall therefore focus on describing the rules that consult the edit history, namely the rules for undo and redo.

a) *Context rules:* The contextual rules with which we operate are different from those of Godiksen et al. [6]. There, all changes to the AST are performed through their *(CONTEXT)* rule. Instead, we split this into two separate rules: [CONTEXT-M] for cursor movements, and [CONTEXT-S] for substitutions. This allows us to track substitutions in the history while ignoring cursor movements.

The [CONTEXT-S] rule describes how substitutions are performed. By using the cursor context representation of ASTs, the cursor is first located, and the substitution is subsequently performed on the subtree encapsulated by the cursor. We record the substituted subtree as well as the

replacement subtree in history, thus ensuring reversibility. There is generally more than one way of viewing an AST as a completed cursor context; by requiring that the cursor is at the top of the open window in the [CONTEXT-S] rule, we ensure that undo restores the substituted subtree at the correct location.

The second side condition, $H' = H[v \xrightarrow{(path(C[a]), a, a')} v']$, says that H' is H with an added node v' connected to v by an edge from v to v' . We label this edge with the path to the cursor, the substituted subtree and the replacement subtree.

The [CONTEXT-M] rule allows for cursor movement in the document AST. This rule calls on the transition rules for cursor movement and is used in the case where π is `child` n or `parent`.

Cursor movements do not modify the document as such, rather they specify where in the document changes are to take place. For this reason they do not interact with the document history directly and recording them is therefore not necessary.

If the history was removed, [CONTEXT-S] and [CONTEXT-M] combined would work similarly to the (*CONTEXT*) rule from [6], which is self evident in the case of [CONTEXT-M], as they are identical, save for the history.

For [CONTEXT-S], removing the history, and the side condition requiring a substitution, would also make it identical to (*CONTEXT*), except that [CONTEXT-S] specifies the exact position of the cursor, whereas (*CONTEXT*) does not. However, specifying the position of the cursor here does not modify its behavior; in both cases, substitutions can only happen to the exact node that the cursor encapsulates, because of how the substitution is defined in Figure 3.

b) Undo rule: The [UNDO] rule changes the current node v into v' , representing movement back through the history. This rule does not modify the history graph H and no nodes are added, removed or modified; instead it reverts the document back to a previous state.

A movement in the history graph results in a corresponding change in the AST. Namely, the current AST a changes into $C[[a']]$, where a' is defined in the side condition $H \vdash v' \xrightarrow{(p, a'', a')} v$ which states that there exists an edge from the node v' to v with $\delta = (p, a'', a')$ in the history graph.

The purpose of this rule is to remove a'' from the AST and substitute it with a' —the exact reverse of what [CONTEXT-S] does. To define which parts of the AST that we do not wish to modify, we use the cursor context C , which we define in the side condition $[a^-] \xrightarrow{p} C[[a']]$. The notation a^- is defined in Definition 6, and is used to remove the cursor from the tree a such that it can be put at the root of AST. The transition \xrightarrow{p} then, as defined in Definition 5, places the cursor at the location decided by the path p . This lets us define the cursor context, which is all nodes in the AST a not encapsulated by the cursor. This lets us move the cursor to the position before the substitution.

c) Redo rule: The [REDO] rule describes how we can go from a node representing a state v to a state v' iff a valid path

(p, a', a'') through the n th outgoing edge from node v exists and given that a , where the cursor has been moved to position p , is equal to $C[[a']]$. Essentially, this transition substitutes the subtree a'' at path p in a with a' . These side conditions ensure that you can only redo to a child node of the current node in the history graph.

The [REDO]-rule is similar to the [UNDO]-rule in its content and structure. This is because they are symmetrical opposites, in the sense that they reverse each other; [UNDO] moves backward in the history graph and removes changes to the AST, while [REDO] moves forward in the history, and re-enacts changes. These properties will be discussed and proven later.

IV. THE EXPRESSIVE POWER OF THE EDITOR CALCULUS

In this section we investigate the expressive power of the editor calculus. First, we give a simple proof that the calculus is Turing-power. Next, we use this to show that our editor calculus is no more powerful than that of [6]: The undo and redo actions can be simulated by representing the edit history directly in the AST.

A. Turing completeness

Godiksen et al. [6] mention in their paper that their version of the editor calculus is Turing powerful. The proof of this (not included in [6]) consists in an encoding of a simple imperative while-language.

In this section, we give a much more straightforward proof; we show that any two-counter machine can be expressed in the original calculus. As we shall see in the following section, the implication is that the calculus with undo and redo is equiexpressive to the original calculus.

Two-counter machines were first defined in [8]. They are very simple imperative programs that constitute a universal model of computation.

A two-counter machine uses two integer-valued variables called *counters*. Its behaviour is defined by a sequence of labelled instructions of the following form.

- The **INC** instruction: $c_i := c_i + 1$; **goto** k
- The **DEC** instruction: **if** $c_i = 0$ **then goto** k_1 **else** $c_i := c_i - 1$; **goto** k_2
- Halt instruction

Here, k, k_1 and k_2 are natural numbers that label instructions.

Formally, two-counter machines can be defined as follows.

Definition 13. A two-counter machine M is a mapping from a finite set \mathbf{Addr}_M of natural numbers to the set of instructions:

$$\{\mathbf{Inc}_{i,k} \mid i \in \{0, 1\}, k \in \mathbf{Addr}_M \cup \{\perp\}\} \\ \cup \{\mathbf{Dec}_{i,k_1,k_2} \mid i \in \{0, 1\}, k_1, k_2 \in \mathbf{Addr}_M \cup \{\perp\}\}$$

We require that $0 \in \mathbf{Addr}_M$.

A *configuration* of a two-counter machine is a triple $\langle k, m_0, m_1 \rangle$. The *initial configuration* σ_I of a machine is $\langle 0, 0, 0 \rangle$.

The transition relation \xrightarrow{l}_M (where $l \in \{\mathbf{inc}_i, \mathbf{then}_i, \mathbf{else}_i \mid i \in \{0, 1\}\}$) is defined by the following inference rules.

$$\begin{array}{c}
[SEQ] \frac{\langle \mathcal{H}, E_1, a \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E'_1, a' \rangle}{\langle \mathcal{H}, E_1 \gg E_2, a \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E'_1 \gg E_2, a' \rangle} \quad [EVAL] \frac{a \rightarrow v}{\langle \mathcal{H}, \text{eval}.E, a \rangle \xrightarrow{v} \langle \mathcal{H}, E, a \rangle} \\
[COND-1] \frac{a \models \phi}{\langle \mathcal{H}, \phi \Rightarrow E_1 | E_2, a \rangle \xrightarrow{\phi} \langle \mathcal{H}, E_1, a \rangle} \quad [COND-2] \frac{a \not\models \phi}{\langle \mathcal{H}, \phi \Rightarrow E_1 | E_2, a \rangle \xrightarrow{\epsilon} \langle \mathcal{H}, E_2, a \rangle} \\
[STRUCT] \frac{E_1 \equiv E_2 \quad \langle \mathcal{H}, E_2, a \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E'_2, a' \rangle \quad E'_2 \equiv E'_1}{\langle \mathcal{H}, E_1, a \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E'_1, a' \rangle} \\
[CONTEXT-S] \frac{\llbracket a \rrbracket \xrightarrow{\pi} \llbracket a' \rrbracket}{\langle (H, v), \pi.E, C[\llbracket a \rrbracket] \rangle \xrightarrow{\pi} \langle (H', v'), E, C[\llbracket a' \rrbracket] \rangle} \quad \text{Where } \pi \notin \{\text{child } n, \text{parent}, \text{undo}, \text{redo}, \text{eval}\} \\
\text{and } H' = H[v, \xrightarrow{(\text{path}(C[a]), a, a')} v'] \\
[CONTEXT-M] \frac{a \xrightarrow{\pi} a'}{\langle \mathcal{H}, \pi.E, C[a] \rangle \xrightarrow{\pi} \langle \mathcal{H}, E, C[a'] \rangle} \quad \text{Where } \pi \in \{\text{child } n, \text{parent}\} \\
[REDO] \frac{}{\langle (H, v), \text{redo } n.E, a \rangle \xrightarrow{\text{redo } n} \langle (H, v'), E, C[\llbracket a' \rrbracket] \rangle} \quad \text{Where } H \vdash v \xrightarrow{(p, a', a'')} \xrightarrow{n} v' \\
\text{and } \llbracket a^- \rrbracket \xrightarrow{p} C[\llbracket a'' \rrbracket] \\
[UNDO] \frac{}{\langle (H, v), \text{undo}.E, a \rangle \xrightarrow{\text{undo}} \langle (H, v'), E, C[\llbracket a' \rrbracket] \rangle} \quad \text{Where } H \vdash v' \xrightarrow{(p, a'', a')} \xrightarrow{p} v, \\
\text{and } \llbracket a^- \rrbracket \xrightarrow{p} C[\llbracket a'' \rrbracket]
\end{array}$$

Fig. 2: Transition rules for editor expressions

$$\begin{array}{c}
[VAR] \frac{}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{var } x\}} \llbracket x \rrbracket} \quad [HOLE] \frac{}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{hole}\}} \llbracket () \rrbracket} \quad [CONST] \frac{}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{const } c\}} \llbracket c \rrbracket} \quad [APP] \frac{}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{app}\}} \llbracket () () \rrbracket} \\
[BREAK-1] \frac{\hat{a} \neq \langle \hat{a}' \rangle}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{break}\}} \llbracket \langle \hat{a}' \rangle \rrbracket} \quad [LAMBDA] \frac{}{\llbracket \hat{a} \rrbracket \xrightarrow{\{\text{lambda } a\}} \llbracket \lambda x. () \rrbracket} \quad [BREAK-2] \frac{}{\llbracket \langle \hat{a} \rangle \rrbracket \xrightarrow{\{\text{break}\}} \llbracket \hat{a} \rrbracket}
\end{array}$$

Fig. 3: Transition rules for substitution

$$\frac{M(k) = \mathbf{Inc}_{i,k'} \quad m'_i = m_i + 1 \quad m'_{1-i} = m_{1-i}}{\langle k, m_0, m_1 \rangle \xrightarrow{\text{inc}_i}_M \langle k', m'_0, m'_1 \rangle}$$

$$\frac{M(k) = \mathbf{Dec}_{i,k_1,k_2} \quad m_i = 0}{\langle k, m_0, m_1 \rangle \xrightarrow{\text{then}_i}_M \langle k_1, m_0, m_1 \rangle}$$

$$\frac{M(k) = \mathbf{Dec}_{i,k_1,k_2} \quad m_i > 0 \quad m'_i = m_i - 1 \quad m'_{1-i} = m_{1-i}}{\langle k, m_0, m_1 \rangle \xrightarrow{\text{else}_i}_M \langle k_2, m'_0, m'_1 \rangle}$$

We write $\sigma \xrightarrow{l_1 \dots l_n}_M \sigma'$ if $\sigma \xrightarrow{l_1}_M \dots \xrightarrow{l_n}_M \sigma'$ for some l_1, \dots, l_n . We also write $\sigma \Rightarrow_M \sigma'$ if $\sigma \Rightarrow_M \sigma'$ for some

s .

The machine M *halts* if there is a reduction sequence $\sigma_I \Rightarrow_M \langle \perp, m_1, m_2 \rangle$ for some m_1, m_2 .

a) *Encoding machine configurations*: Let M be a program for a two-counter machine. With no loss of generality, we assume that the set of labels is an interval $[0..n]$ for some n , that the address 0 is the initial address, and that the address n is the halt address.

The initial configuration of the machine is encoded as the AST in Figure 5.

The left-hand side of the AST represents the current configuration of the machine. It consists of a hole, which has $n - 1$ breakpoints above it. If the cursor is placed below the application, this encodes the current instruction as 0. Placing the cursor below the top-most breakpoint encodes the address

$$\begin{array}{c}
[APPC-1] \frac{}{\llbracket \hat{a}_1 \hat{a}_2 \rrbracket \xrightarrow{\text{child 1}} \llbracket \hat{a}_1 \rrbracket \hat{a}_2} \quad [APPC-2] \frac{}{\llbracket \hat{a}_1 \hat{a}_2 \rrbracket \xrightarrow{\text{child 2}} \hat{a}_1 \llbracket \hat{a}_2 \rrbracket} \quad [APPP-1] \frac{}{\llbracket \hat{a}_1 \rrbracket \hat{a}_2 \xrightarrow{\text{parent}} \llbracket \hat{a}_1 \hat{a}_2 \rrbracket} \\
[APPP-2] \frac{}{\hat{a}_1 \llbracket \hat{a}_2 \rrbracket \xrightarrow{\text{parent}} \llbracket \hat{a}_1 \hat{a}_2 \rrbracket} \quad [BREAKP] \frac{}{\langle \llbracket \hat{a} \rrbracket \rangle \xrightarrow{\text{parent}} \llbracket \langle \hat{a} \rangle \rrbracket} \quad [BREAKC] \frac{}{\llbracket \langle \hat{a} \rangle \rrbracket \xrightarrow{\text{child 1}} \langle \llbracket \hat{a} \rrbracket \rangle} \\
[ABSC] \frac{}{\llbracket \lambda x. \hat{a} \rrbracket \xrightarrow{\text{child 1}} \lambda x. \llbracket \hat{a} \rrbracket} \quad [ABSP] \frac{}{\lambda x. \llbracket \hat{a} \rrbracket \xrightarrow{\text{parent}} \llbracket \lambda x. \hat{a} \rrbracket}
\end{array}$$

Fig. 4: Transition rules for cursor movements

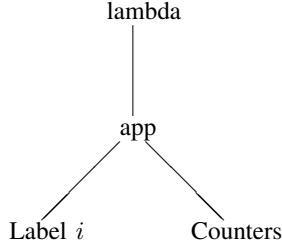


Fig. 5: An AST representing a two-counter machine

as 1, and so on and so forth.

On the right-hand side is an application, with two holes as children. The number of breakpoints between the hole and the application represent the value of the counters.

We denote an AST that encodes a configuration with the current label k and counter values m_1 and m_2 by $a(k, m_1, m_2)$.

b) Encoding the behaviour: For each instruction of the machine we introduce a recursion variable. This allows us to describe an expression that defines the program of the machine.

For every label i we introduce a recursively defined expression $P_i(x)$, For the halt instruction with label n we let

$$P_n \triangleq 0$$

For every other label, the representation depends on the kind of instruction. If the instruction is a **Dec** instruction, we define

$$\begin{aligned}
P_i(x) \triangleq \text{parent}.\text{@lambda} \Rightarrow & \\
& (\text{child 2.child } i \gg \text{@hole} \Rightarrow \\
& (\text{parent.parent.child 1.(child 1)}^{k_1} | \\
& (\{\text{break}\} \gg \text{parent.parent.child 1.(child 1)}^{k_2}) \\
&) | P_{i+1}(x) \text{ if } \mathbf{Dec}_{i,k_1,k_2} \in M
\end{aligned}$$

If the instruction is an **Inc** instruction, we define

$$\begin{aligned}
P_i(x) \triangleq \text{parent}.\text{@lambda} \Rightarrow & \\
& (\text{child 2.child } i. \\
& \text{rec } y.(\text{@break} \Rightarrow ((\text{child 1}).y \gg \text{parent}) \gg \\
& \quad \text{@hole} \Rightarrow (\text{break})) \\
& \gg \text{parent.parent.child 1.(child 1)}^k \\
& | P_{i+1}(x) \text{ if } \mathbf{Inc}_{i,k} \in M
\end{aligned}$$

The whole expression representing a program is then defined as $\text{rec } x.P_0(x)$.

We have that this encoding is sound and complete. The following theorems are both proved by induction in the length of the transition sequences.

Theorem IV.1 (Soundness). *If*

$$\langle k, m_0, m_1 \rangle \rightarrow^* \langle k', m'_0, m'_1 \rangle$$

then

$$\langle \mathcal{H}_\epsilon, P_0, a(m_0, m_1) \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E', a(k'.m'_0.m'_1) \rangle$$

for some editor expression E' .

Theorem IV.2 (Completeness). *If*

$$\langle \mathcal{H}_\epsilon, P_0, a(m_0, m_1) \rangle \xrightarrow{\alpha} \langle \mathcal{H}', E', a(k'.m'_0.m'_1) \rangle$$

for some editor expression E' , then

$$\langle k, m_0, m_1 \rangle \rightarrow^* \langle k', m'_0, m'_1 \rangle$$

B. Representing the history-based semantics

We can encode almost all of the extended editor calculus in the simple calculus without undo and redo. The exception is that the `eval` primitive cannot be represented. The reason for this is that `eval` evaluates the entire AST; since we represent the history as well as the current AST in a single AST, evaluation of the entire tree does not make sense.

In the semantics of the original editor calculus, configurations are of the form $\langle E, a \rangle$ and transitions are of the form

$$\langle E, a \rangle \xrightarrow{\alpha} \langle E', a' \rangle$$

We encode a configuration $\langle \mathcal{H}, E, a \rangle$ from our semantics by encoding each of its components.

We represent a history-AST pair $\langle \mathcal{H}, v \rangle$ where v is the node representing the current AST, as a single AST. An example is shown in Figure 6. The AST associated with the current node v is the immediate right subtree of an `app` term constructor. We refer to this subtree as `dec(v)`. The current node v , which can be represented as a natural number, is the twice-left child of the root. The history \mathcal{H} is kept in the right subtree of the left subtree. Each node of the history tree is encoded with metadata that assigns a unique ID, and describes the child and parents of the tree. We refer to this subtree as `enc(H)`.

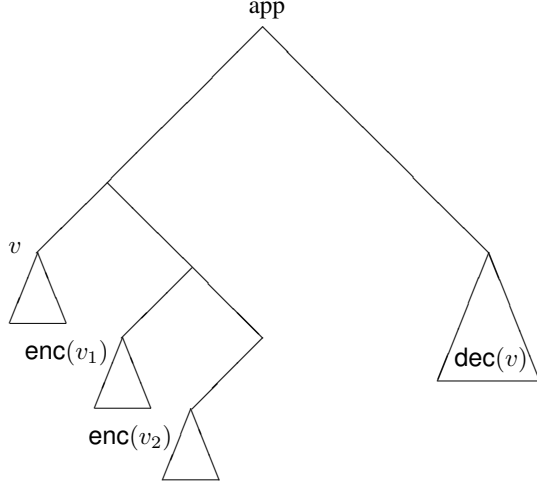


Fig. 6: The representation of a history-AST pair $\langle \mathcal{H}, a \rangle$ where $\mathcal{H} = (\{v_1, v_2\}, \mathbb{E}_{\mathbb{H}})$, and the currently selected node is v

The idea of the encoding of E is to store the result of each atomic edit action in the history, unless the action is `undo` or `redo`. In these cases, we consult the history.

These operations on the encoding of the history are implemented by means of the following collection of auxiliary operations on subtrees.

copyto copies $dec(v)$ to a new subtree in $enc(\mathcal{H})$, adds v as a child of the previous node; v becomes the current node
child (i) gets the i th child of v
parent gets the parent of v
copyfrom (i) replaces $dec(v)$ with $dec(v_i)$ from the history tree

Since the original editor calculus is Turing-powerful, we can use it to implement each of these operations. One can either store the subtrees directly or use a binary representation.

We use the auxiliary operations to encode the atomic edit actions; the encoding is homomorphic wrt. composite editor expressions. For a subset of editor expressions, they can be encoded in the history-free semantics as follows.

$enc(E) = E.\text{copyto}$ where $E \notin \{\text{eval}, \text{undo}, \text{redo}\}$
 $enc(\text{undo}) = \text{copyfrom}(\text{parent})$
 $enc(\text{redo } n) = \text{copyfrom}(\text{child}(n))$

V. CONCLUSIONS AND FURTHER WORK

A. Results

The present work extends the editor calculus of Godiksen et al. [6] with a branching notion of `undo/redo` actions. To the best of our knowledge, these systems have, prior to the present paper, not yet been formalized. We do this by introducing two new editor expressions: `undo` and `redo`, and associated transition rules. `undo` reverts the document (a possibly incomplete λ -calculus expression) to a prior state, while `redo` reverts an `undo`.

Our semantics is history-based: we introduce a history graph that records the modifications to the document, from which it is possible to infer the state of the present document.

Our `undo/redo` system is fully reversible in the sense that, for any given state, there exists exactly one *prior* state, such that it is always possible to uniquely identify the prior state that resulted in the current.

Godiksen et al. [6] show that their semantic is Turing-powerful, but we provide an alternative, much simpler proof of this by expressing a two-counter machine in the original semantics. We use the expressive power of the original calculus to embed an encoded version of the history graph into the document; thereby allowing the two-counter machine to emulate the history graph as defined by our semantics. This result shows that the extended calculus is equiexpressive with the original, that is, that every document expressible in the original semantic is also expressible in the extended and vice-versa. A notable exception is that of the `eval`-action, since its semantics is that of evaluating the entire AST. As the encoding stores the history graph in the AST, this would not make sense.

The `undo/redo` paradigm in the traditional sense is interesting in its own right; however, our extension makes it powerful still: Since `undo` and `redo` are introduced as first-class editor expressions in the extended semantic, it is possible and convenient to use these actions in a broader sense: `undo` and `redo` can be executed conditionally, sequentially or recursively to provide non-trivial functionality—directly in relation to manipulating the history, or in the context of broader editor expressions not necessarily related to the history, e.g. storing and retrieving data. In this way our extension allows the `undo/redo` paradigm to symbiotically integrate into the existing structure editor calculus.

B. Future Work

A central aspect of [6] is that of a type system which guarantees that a well-typed editor expression can only produce a well-typed program. A natural next step is to show that our extension also has this safety property.

The `eval`-action is not covered in our expressiveness result, but we conjecture that this can be dealt with by modifying the semantics of `eval`, such that evaluation is relative to the position of the cursor.

Another avenue for further work is to extend the notions of `undo` and `redo`. Jakubec et al. [7] describes selective `undo/redo` system as a system which allows the user to `undo` and `redo` any action in the history in arbitrary order. Our history model could be extended with a model for history that allows for exactly that.

A selective `undo/redo` system is especially suited for this editor calculus as the structure of the history graph provides detailed information in deciding whether a particular selective `undo/redo` makes sense in the context of the document. In principle, by looking at the path of a delta, it can be switched with another delta if none of the paths are sub-paths of each other.

REFERENCES

- [1] AIELLO, M., PRATT-HARTMANN, I. E., AND BENTHEM, J. F. V. *Handbook of Spatial Logics*. Springer-Verlag, Berlin, Heidelberg, 2007.
- [2] BEKIĆ, H. *Definable operations in general algebras, and the theory of automata and flowcharts*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984, pp. 30–55.
- [3] BORRAS, P., CLEMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. Centaur: The system. *SIGSOFT Softw. Eng. Notes* 13, 5 (Nov. 1988), 14–24.
- [4] DANOS, V., AND KRIVINE, J. Reversible communicating systems. In *International Conference on Concurrency Theory* (2004), Springer, pp. 292–307.
- [5] DONZEAU-GOUGE, J. V., HUET, G., KAHN, G., T, W. K. A. I., LANG, E. I., DTIC, T., SIGNIFICANT, C. A., DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structured editors: The MENTOR experience. Tech. rep., 1980.
- [6] GODIKSEN, C., HERRMANN, T., HÜTTEL, H., LAURIDSEN, M. K., AND OWLIAIE, I. A type-safe structure editor calculus. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (2021), pp. 1–13.
- [7] JAKUBEC, K., POLÁK, M., NEČASKÝ, M., AND HOLUBOVÁ, I. Undo/redo operations in complex environments. *Procedia Computer Science* 32 (2014), 561–570.
- [8] MINSKY, M. L. *Computation: Finite and infinite Machines*. Prentice-Hall, 1967.
- [9] OMAR, C., VOYSEY, I., CHUGH, R., AND HAMMER, M. A. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019).
- [10] OMAR, C., VOYSEY, I., HILTON, M., ALDRICH, J., AND HAMMER, M. A. Hazelnut: a bidirectionally typed structure editor calculus. *ACM SIGPLAN Notices* 52, 1 (2017), 86–99.
- [11] TEITELBAUM, T., AND REPS, T. The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573.
- [12] TEITELMAN, W. Automated programming: The programmer’s assistant.