Forside

Eksamensinformation
DDD660002C - cs-23-dat-6-12 - project

Besvarelsen afleveres af
Peter Buus Steffensen
psteff19@student.aau.dk

Benjamin Clausen Bennetzen
bbenne20@student.aau.dk

Andreas Tor Mortensen
atmo20@student.aau.dk

Nikolaj Rossander Kristensen
nrkr20@student.aau.dk

Eksamensadministratorer
Lone E. Vriborg
lea@cs.aau.dk
+4599403576

Bedømmere
Hans Hüttel
Eksaminator
hans@cs.aau.dk
+4599408888

Jesper Bengtson
Censor
bengtson@itu.dk
0046733547679

Besvarelsesinformationer

Projekttitel: Indtast den endelige titel på dit projekt. Titlen vil fremgå af dit eksamensbevis: Generalization a

Projekttitel, engelsk. Titlen vil fremgå af dit eksamensbevis : Generalization and lambda encoding of editor calculus

Tro og love-erklæring: Ja

Indeholder besvarelsen fortroligt materiale: Nej

Department Of Computer Science
Aalborg University
http://www.aau.dk

Title:
Generalization and λ encoding of editor calculus

Theme:
Syntax-directed editors

Project Period:
Spring Semester 2023

Project Group:
cs-23-dat-6-12

Participant(s):
Andreas Tor Mortensen
Benjamin Bennetzen
Nikolaj Rossander Kristensen
Peter Buus Steffensen

Supervisor(s):
Hans Hüttel

Copies: 1

Page Numbers: 51

Date of Completion:
May 24, 2023

Abstract:

This paper is our 6th-semester bachelor project, in which we present a generalization of a syntax-directed editor calculus, which can be used to instantiate a specialized syntax-directed editor for any language given by some abstract syntax. The editor calculus guarantees the absence of syntactical errors while allowing incomplete programs. The generalized editor calculus is then encoded into a simply typed lambda calculus, extended with pairs, booleans, pattern matching and fixed points.
We then proceed to prove the soundness of our encoding to the extended lambda calculus. We utilize two lemmas to prove, by induction on the height of the transition tree, that our editor expressions are soundly encoded.

Summary

This report documents our work with designing a generalised syntax-directed editor calculus. It begins with the preliminaries our work is based on. That is, we introduce lambda calculus, and the simply typed extension of this. We then formally introduce abstract syntax and abstract binding trees. After this we present the syntax and semantics of our generalised editor calculus, and explain how it can be used to edit a tree represented as operators.

Then we show how to encode abstract binding trees to the simply typed lambda calculus. Following this we encode contexts, in an extended simply typed lambda calculus that includes pattern matching. We then encode the modal logic and the actual editor expressions in a further expanded lambda calculus, that includes fixed-points. In this chapter we also define the reduction rules and type rules of the pattern matching operator and the fixed points operator.

We then proceed to prove the soundness of our encoding to the extended lambda calculus. To do this, we demonstrate that the encoding of atomic prefix commands applied to the encoding of a given tree, behaves as described by the semantics. Additionally, we employ induction on the height of the derivation trees of judgments to prove that, when a given tree satisfies a modal logic formula, the encoding of the modal logic applied to the encoding of the tree, reduces to true. Conversely, when the given tree does not satisfy the modal logic formula, the encoding reduces to false. Finally, we utilize these two lemmas to prove, using induction on the height of the transition tree, that our editor expressions are soundly encoded.

At last, we conclude that we reached our initial goal of the paper, having presented a generalised syntax-directed editor calculus, that can be soundly encoded to the extended simply typed lambda calculus. With this, we then proceed to mention future work that can be done to expand upon our results. This include proving the completeness of the encoding of the semantics, and expanding the generalised editor calculus with more editor operations such as *copy* and *paste* as well as *undo* and *redo*.

Generalization and λ encoding of editor calculus

Andreas Tor Mortensen, Benjamin Bennetzen, Nikolaj Rossander Kristensen, and Peter Buus Steffensen

Department of Computer Science, Aalborg University

Abstract. This paper is our 6th-semester bachelor project, in which we present a generalization of a syntax-directed editor calculus, which can be used to instantiate a specialized syntax-directed editor for any language given by some abstract syntax. The editor calculus guarantees the absence of syntactical errors while allowing incomplete programs. The generalized editor calculus is then encoded into a simply typed lambda calculus, extended with pairs, booleans, pattern matching and fixed points.
We then proceed to prove the soundness of our encoding to the extended lambda calculus. We utilize two lemmas to prove, by induction on the height of the transition tree, that our editor expressions are soundly encoded.

Keywords: Syntax-directed editors · Lambda calculus · Abstract syntax

1 Introduction

This paper is on the topic of syntax-directed editors. Most current examples of syntax-directed editor calculi, such as the ones presented by Omar et al. [6] and Godiksen et al. [8], are typically tailored specifically towards editing abstract syntax trees from an applied λ-calculus. This fact alone limits the usable scope of these calculi, since they would be rendered useless given any other form of abstract syntax.

Furthermore, these calculi does not account for binding mechanisms in abstract syntax. Many of the editor calculi possesses the ability to interleave construction and evaluation of programs, however, a consequence of this is a complex type system.

In this paper we propose a generalised editor calculus that can be used to create a syntax-directed editor calculus for any abstract syntax. Our editor calculus is inspired by higher-order abstract syntax presented in Pfenning et al. [1], and it will take binding-mechanisms into account. We have encoded our generalized editor calculus into a simply-typed lambda calculus extended with pattern matching and fixed-points. Provided that the encoding itself is sound, we can ensure that our editor calculus has a sound type system, since our extension of the simply-typed lambda calculus has a sound type system. Consequently, we can ensure that the result obtained from performing some editor operation is well-typed, provided the editor operation itself is well-typed.

The remainder of this paper is structured as follows:

– In section 2, we introduce the theory of lambda-calculus and abstract syntax, which plays a central role in the work presented here.
– In section 3, we present a generalized definition of the syntax and semantics of the editor calculus initially proposed by Godiksen et al. [8].
– Moving forward, in section 4, we gradually encode our generalized editor calculus within an extension of the simply-typed lambda calculus.
– Finally, in section 5, we prove the soundness of the lambda encoding.

2 Preliminaries

2.1 Lambda Calculus

Lambda calculus, commonly referred to as λ-calculus, is a model expressing computation as function abstractions and applications. Despite the simple nature of the λ-calculus, it has been shown to be as expressive as other models of computation such as Turing machines, and thus it is Turing-complete. This section serves as a short introduction to the key concepts and notational conventions of the λ-calculus, based on Barendregt et al. [4], Alama et al. [7], and Pierce [3].

Syntax In the simplest form of λ-calculus, the untyped variant, a valid λ-term is built using the abstract syntax shown in fig. 1. Here x ranges over a countable infinite set of variables, the term $M_1 M_2$ represents the application of M_1 to the argument M_2 and the term $\lambda x.M$ represents a function abstraction which binds the variable x inside M.

$$M \quad ::= \quad \lambda x.M \mid M_1 M_2 \mid x$$

Fig. 1: Abstract syntax of untyped λ-calculus

By convention we say that application associates to the left, such that

$$M_1 \dots M_n = (\dots((M_1 M_2) M_3) \dots) M_n)$$

and abstraction associates to the right, such that

$$\lambda x_1 \dots x_n.M = \lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))$$

Variables An abstraction term $\lambda x.M$ is said to *bind* the variable x in the term M. This leads to the notion that variables can be either *free* or *bound* in the context of some term M. The set of free and bound variables of a term M are formally introduced in definitions 1 and 2 respectively.

Definition 1 (Free variables). The set of free variables in term M, denoted by $FV(M)$, is recursively defined as:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(M_1 M_2) &= FV(M_1) \cup FV(M_2) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

A term M with no free variables is said to be *closed*, or a *combinator*.

Definition 2 (Bound variables). The set of bound variables in term M, denoted by $BV(M)$, is recursively defined as:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(M_1 M_2) &= BV(M_1) \cup BV(M_2) \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \end{aligned}$$

Example 1. In the term $\lambda x.xy$ the variable x is bound, whereas y is free. On the other hand the term $\lambda x.x$, known as the identity-function, is closed since it contains no free variables.

By convention names of bound variables are always selected such that they differ from the free variables of a term. For example for the term $x(\lambda x.yx)$ we instead write $x(\lambda x'.yx')$.

Substitution The concept of substitution, introduced in definition 3, is in many ways central to the λ-calculus and provides the basis of its principal rewriting rules of α-conversion and β-reduction.

Definition 3 (Substitution). The substitution of term N for all free occurrences of variable x in term M, denoted by $M[x := N]$, is recursively defined as (where $x \neq y$):

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \\ M_1 M_2[x := N] &\equiv M_1[x := N] M_2[x := N] \\ (\lambda x.M)[x := N] &\equiv \lambda x.M \\ (\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]) \end{aligned}$$

Using the naming convention mentioned previously we can for the most part use substitution in the λ-calculus without a proviso on the free and bound variables. However even if terms are initially written according to the naming convention, once we start rewriting terms we will need a method of renaming bound variables in order to avoid captures during substitution.

***α*-conversion** Terms in the λ-calculus can be identified up to the names of their bound variables, known as α-congruence. For example the terms $\lambda x.x$ and $\lambda y.y$ are α-congruent since they only differ in bound variables, but both describe the identity function. The α-congruent terms of a term N can be obtained through α-conversion as per definition 4.

Definition 4 (α-conversion). α-conversion of a term $\lambda x.M$, denoted by $\rightarrow_\alpha$, is the renaming of the bound variable x such that

$$\lambda x.M \rightarrow_\alpha \lambda y.(M[x := y]), \text{ provided } y \text{ does not occur in } M \qquad (\alpha)$$

Using (α) we can formally define the α-congruence relation.

Definition 5 (α-congruence). Two terms M and N are said to be α-congruent, denoted by $M \equiv_\alpha N$, if there exists a sequence of α-conversions starting from M that ends in N.

α-conversion also provides us a way to ensure capture-avoiding substitutions. When a variable is threatened to be captured from a substitution we simply perform enough α-conversions to avoid the capture.

***β*-reduction** The principal notion of term reduction in the λ-calculus is β-reduction, which informally can be described as the evaluation of applying some abstraction term (or function definition) to an argument. Formally this reduction axiom is defined as per definition 6.

Definition 6 (β-reduction). A term of the form $(\lambda x.M)N$, called a *redex*, can be β-reduced such that

$$(\lambda x.M)N \rightarrow_\beta M[x := N], \qquad (\beta)$$

provided that no free variables in N are captured during the substitution. A term with no redexes cannot be further reduced, and is said to be in β normal form.

Example 2. As mentioned, we can guarantee that no variables are captured by simply performing enough α-conversions prior to the substitution. For example if we naively applied β-reduction to the term $(\lambda x.\lambda y.x)y$ without considering capture of free variables, we would get

$$(\lambda x.\lambda y.x)y \rightarrow_\beta \lambda y.y$$

Here the y is captured during substitution, and consequently the meaning of the inner abstraction term $\lambda y.x$ is changed from the constant function to the identity function $\lambda y.y$.

Instead if we first perform α-conversion to avoid capture, and then use β-reduction to reduce the term, we would correctly get

$$(\lambda x.\lambda y.x)y \rightarrow_\alpha (\lambda x.\lambda z.x)y$$
$$(\lambda x.\lambda z.x)y \rightarrow_\beta \lambda z.y$$

where the meaning of the inner abstraction term is preserved, since no free variables are captured during the reduction step.

Types So far we have only considered the pure untyped λ-calculus. A meaningful extension of this formalism is the introduction of types. The simply typed λ-calculus, abbreviated as $\lambda_{\to}$, is the simplest example of such typed interpretations.

The syntax of $\lambda_{\to}$, shown in fig. 2, closely resembles that of the untyped calculus. Here τ is introduced as a new syntactic sort representing types, $\tau_1 \to \tau_2$ is the function type constructor and t ranges over a fixed set of base types $\mathbb{T}$. Additionally abstraction terms now gets annotated with the type of the argument and c is introduced to range over a fixed set of term constants.

$$\begin{array}{rcl} M & ::= & \lambda x : \tau.M \mid M_1 M_2 \mid x \mid c \\ \tau & ::= & \tau_1 \to \tau_2 \mid t \end{array}$$

Fig. 2: Abstract syntax of $\lambda_{\to}$

By convention the function type constructor $\to$ is right associative, that is, the expression $\tau_1 \to \ldots \to \tau_n$ stands for $\tau_1 \to (\tau_2 \to (\ldots (\tau_{n-1} \to \tau_n) \ldots)$.

The introduction of types allows us to define a set of *well-typed* terms through a typing-relation between types and terms given by the rules shown in fig. 3. Using these a term M can be shown to be well-typed and of type τ if there exists a derivation proving the judgement $\Gamma \vdash M : \tau$, where Γ represents the type-context.

$$\text{(Var)}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{(Const)}\ \frac{c \text{ is a constant of type } t}{\Gamma \vdash c : t}$$

$$\text{(Abs)}\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : \sigma \to \tau} \qquad \text{(App)}\ \frac{\Gamma \vdash M_1 : \sigma \to \tau \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash M_1 M_2 : \tau}$$

Fig. 3: Type rules for $\lambda_{\to}$

An important property of $\lambda_{\to}$ is that it is *strongly normalizing* for all well-typed terms. That is, all well-typed terms can be reduced to their β normal forms regardless of reduction order. Consequently this also means that $\lambda_{\to}$ is not Turing-complete, contrary to the pure variant which is Turing-complete but neither strongly nor weakly normalizing.

2.2 Abstract Syntax

Another important theoretical aspect of this paper is the concept of abstract syntax. Previously we presented the abstract syntax of λ-calculus as a set of formation rules, defining how valid λ-terms can be constructed.

In this section we formally introduce the notion of abstract syntax and how it can be augmented to also describe the binding and scope of variables, inspired by the theory presented in Harper [5].

Abstract Syntax Trees An abstract syntax tree, abbreviated as *ast*, is an ordered tree describing the structure of a piece of syntax.

Ast's are classified by their *sort*, which divides ast's into distinct syntactic categories. For example, a language could have a syntactic distinction between arithmetic and boolean expressions by assigning each of these their own sort.

Structurally ast's are built with *operators*, by composing 0 or more ast's together. Formally an operator o has some arity $(s_1, \ldots, s_n)s$, which specifies the sort s of the operator and the number and sorts $s_1, \ldots, s_n$ of its arguments. For example the application term $M_1 M_2$ from the untyped λ-calculus could be represented as the operator *app* with arity $(M, M)M$, where M is the sort of a λ-term.

Finally ast's also incorporates a notion of *variables*. In this context a variable is simply a placeholder for an unknown ast of some sort. These can be given meaning through substitution, where every occurrence of a variable is replaced by a specific ast. Variables are sort-specific since only ast's of sort s can be plugged in for a variable of sort s. For example, say we had a distinct sort for arithmetic and boolean expressions, trying to replace a boolean variable with an arithmetic expressions would not make sense, since the two are different sorts of syntax.

With the introduction of these three central notions we can give a precise and formal definition of abstract syntax trees.

Definition 7 (Abstract Syntax Trees). For some abstract syntax given as a set of sorts $\mathcal{S}$, an arity-indexed family of operators $\mathcal{O}$ and a sort-indexed family of variables $\mathcal{X}$, the sort-indexed family $\mathcal{A}[\mathcal{X}]$ of abstract syntax trees is the smallest family satisfying the following conditions:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. If o has arity $(s_1, \ldots, s_n)s$ and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

If $a \in \mathcal{A}[\mathcal{X}]_s$ we say that a is an ast of sort s.

From the first clause of definition 7 we have that any variable of sort s is also an ast of sort s. The second clause tells us that an operator o with arity $(s_1, \ldots, s_n)s$ combines ast's of sort $s_1, \ldots, s_n$ into a compound ast of sort s.

Note 1. While operators, variable and trees are all given as indexed families, for brevity we will sometimes refer to and use these as sets. For example we might write the following $o \in \mathcal{O}$ to signify that the operator o is a member of one of the sets of operators in the family $\mathcal{O}$.

Abstract Binding Trees An abstract binding tree, abbreviated as *abt*, is an enriched variant of an ast, which introduces means of representing binding and scope of variables. Specifically this is achieved by allowing an operator to bind any finite number of variables in each of its arguments.

Operators are assigned a *generalized arity* of the form $(\overrightarrow{s_1}.s_1, \ldots, \overrightarrow{s_n}.s_n)s$, where $\overrightarrow{s_i}.s_i$ stands for $s_1 \ldots s_k.s_i$ specifying the sort of argument i and the number and sorts of variables bound in the scope of that argument. For example in the λ-calculus the abstraction term $\lambda x.M$ binds the variable x inside M. This could be represented as the operator *abs* with arity $(M.M)M$ indicating that it takes one argument of sort M within which is bound a variable of sort M. The term $\lambda x.xx$ could then be represented as the abt:

$$abs(x.app(x; x))$$

Whereas the family of variables $\mathcal{X}$ remains fixed in the definition of ast's, in the context of abt's $\mathcal{X}$ changes when entering the scope of a binding by joining the bound variables to the set of active variables. Variables can be added to $\mathcal{X}$ provided they are fresh for $\mathcal{X}$ as per definition 8.

Definition 8 (Fresh variables). A variable x is said to be *fresh* for $\mathcal{X}$ if $x \notin \mathcal{X}_s$ for any sort $s \in \mathcal{S}$. Furthermore if x is fresh for $\mathcal{X}$, then $\mathcal{X}, x$ is the family of variables obtained by adding x to $\mathcal{X}_s$.

A consequence of the freshness criteria in definition 8 is that it leads to *name clashes*. For example attempting to add x to $\mathcal{X}, x$ is impossible since x is not fresh for $\mathcal{X}, x$. However, analogous to α-congruence in the λ-calculus, choice of bound variable names is irrelevant as long as they describe the same binding. Therefore we introduce the notion of a fresh renaming as per definition 9, which renames the variable bindings of an abt such that they are fresh for $\mathcal{X}$.

Definition 9 (Fresh renamings). A fresh renaming, relative to $\mathcal{X}$, of a finite sequence of variables $\overrightarrow{x}$ is a bijection $\rho : \overrightarrow{x} \leftrightarrow \overrightarrow{x}'$, where $\overrightarrow{x}'$ is fresh for $\mathcal{X}$. $\hat{\rho}(a)$ denotes the result of replacing each occurence of x_i in a with its fresh counterpart $\rho(x_i)$

Using fresh renamings we formally introduce abstract binding trees, similarly to ast's, in definition 10.

Definition 10 (Abstract Binding Trees). For some abstract syntax given as a set of sorts $\mathcal{S}$, an arity-indexed family of operators $\mathcal{O}$ and a sort-indexed family of variables $\mathcal{X}$, the sort-indexed family $\mathcal{B}[\mathcal{X}]$ of abstract binding trees is the smallest family satisfying the following conditions:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. If o is an operator of arity $(\overrightarrow{s_1}.s_1, \ldots, \overrightarrow{s_n}.s_n)s$, and for each $1 \leq i \leq n$ and each fresh renaming ρ_i we have $\hat{\rho}(a_i) \in \mathcal{B}[\mathcal{X}, \overrightarrow{x_i}']_{s_i}$ then $o(\overrightarrow{x_1}.a_1; \ldots; \overrightarrow{x_n}.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

If $a \in \mathcal{B}[\mathcal{X}]_s$ we say that a is an abt of sort s.

Corresponding to the definition of ast's, the first clause says that variables of sort s is also an abt of sort s. The second clause says that an operator o combines abt's of sort $s_1, \ldots, s_n$ into an abt of sort s using fresh renamings of bound variables for each of its arguments.

3 Generalized Editor Calculus

With the principal theory of λ-calculus and abstract syntax introduced, we will now present a generalization of the syntax-directed editor calculus by Godiksen et al. [8]. This specific editor calculus was designed with the language of an applied λ-calculus in mind. Instead, by using the notion of abstract syntax as presented in section 2.2 we generalize the concepts of the editor calculus to account for any language given by some abstract syntax.

3.1 Syntax

Abstract Syntax Firstly, we introduce the general representation of some abstract syntax that will be the focus of the editor calculus. We assume the abstract syntax is given by a set of sorts $\mathcal{S}$, an arity-indexed family of operators $\mathcal{O}$ and a sort-indexed family of variables $\mathcal{X}$.

The notion of cursors and holes is central to the idea of an editor calculus. Whereas the calculus in Godiksen et al. [8] only has one cursor and hole term, in the general case a cursor and hole operator for every sort in the abstract syntax is necessary. These are introduced by extending the abstract syntax with the operators as per definition 11.

Definition 11 (Holes and cursors). For every sort $s \in \mathcal{S}$ we add the following operators to $\mathcal{O}$

1. A $hole_s$ operator with arity $()s$, representing a missing or empty subtree of sort s.
2. A $cursor_s$ operator with arity $(s)s$, representing a subtree of sort s encapsulated by a cursor.

With the set of sorts $\mathcal{S}$, the extended family of operators $\mathcal{O}$ and family of variables $\mathcal{X}$ we can then define the sort-indexed family of abstract binding trees $\mathcal{B}[\mathcal{X}]$ as the smallest family that satisfies the conditions in definition 10.

Example 3. To illustrate how the generalized editor calculus could be instantiated for some language, we will specialize an editor calculus as we proceed through this section for the abstract syntax shown in fig. 4.

Sort		Term	Operator	Arity
s	$::=$	let $x = e$ in s	let	$(e, e.s)s$
	$\mid$	e	exp	$(e)s$
e	$::=$	$e_1 + e_2$	$plus$	$(e, e)e$
	$\mid$	n	$num[n]$	$()e$
	$\mid$	x	$var[x]$	$()e$

Fig. 4: Abstract syntax of arithmetic expressions and local declarations

The syntax in fig. 4 describes a very simple language of statements and expressions. A statement can either be a let statement, which binds a variable x in the scope of another statement s, or it can be an expression e. Similarly an expression can either be the addition of two sub-expressions, a number $n \in \mathbb{N}$ or a variable identifier x. For example, using the abstract syntax we could construct the statement

$$\text{let } x = 5 \text{ in let } y = 10 \text{ in } x + y$$

which, using operators, would be represented as the abt

$$let(5; x.let(10; y.exp(plus(x; y))))$$

Given the abstract syntax in fig. 4 we can extend it with holes and cursors following definition 11. This gives us the new operators shown in fig. 5.

Sort		Term	Operator	Arity
s	$::=$	$[s]$	$cursor_s$	$(s)s$
	$\mid$	$(\!\mid \mid\!)_s$	$hole_s$	$()s$
e	$::=$	$[e]$	$cursor_e$	$(e)e$
	$\mid$	$(\!\mid \mid\!)_e$	$hole_e$	$()e$

Fig. 5: Introduction of cursors and holes to the abstract syntax in fig. 4

With this extension we can now represent statements with holes and cursors as abt's. For example the statement

$$\text{let } x = [5] \text{ in } x + (\!\mid \mid\!)_e$$

would be represented as the abt

$$let(cursor_e(5); x.exp(plus(x; hole_e)))$$

Editor Calculus The abstract syntax of the generalized editor calculus, shown in fig. 6, closely resembles that of the calculus presented by Godiksen et al. [8].

$$
\begin{array}{lcl}
E & ::= & \pi.E \mid \phi \Rightarrow E_1 | E_2 \mid E_1 \ggg E_2 \mid rec\ x.E \mid x \mid nil \\
\pi & ::= & child\ n \mid parent \mid \{o\} \\
\phi & ::= & \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid @o \mid \Diamond o \mid \Box o
\end{array}
$$

Fig. 6: Abstract syntax of general editor calculus

Editor expressions $E \in Edt$, where Edt is the sort of editor expressions, describes the behaviour of the editor calculus. These facilitate ways of composing editor expressions together with conditions, sequential composition and recursion. The means of traversing and modifying an abt are provided by the prefixed expression $\pi.E$, which evaluates π before continuing with E. The conditional expression $\phi \Rightarrow E_1|E_2$ reduces to E_1 if ϕ is satisfied and E_2 otherwise. The sequential expression $E_1 \ggg E_2$ evaluates E_2 only once E_1 has been reduced to nil. The recursive expression $rec\ x.E$ binds the recursion variable x in E, which can then be used to recursively iterate an expression.

The atomic prefix commands $\pi \in Apc$, where Apc is the sort of atomic prefix commands, describes various commands that either traverse or modify the abt encapsulated by the cursor. Specifically, $child\ i$ and $parent$ allows us to move the cursor up and down in the abt. The substitution command $\{o\}$ replaces the abt currently encapsulated by the cursor with o, where o ranges over all operators in $\mathcal{O}$ excluding cursors.

Conditions are introduced as $\phi \in Eec$, where Eec is the sort of conditions, and describes propositional connectives and modal logic with respect to the abt encapsulated by the cursor. The modal operator $@o$ holds when the cursor is currently at the operator o. $\Diamond o$ holds when the operator o is in *a* subtree encapsulated by the cursor. Finally, $\Box o$ holds when the operator o is in *all* subtrees of the abt encapsulated by the cursor.

Evaluation of abt's, which Godiksen et al. [8] introduces through a *eval* construct, is not included in our generalized version. This omission is due to evaluation being entirely dependent on the dynamics of the language in focus, and not the static and structural properties the abstract syntax describes. To define the concept of evaluation in a generalized manner, it would be necessary to devise a formal and concise approach for specifying the semantics of operators.

Example 4. Expanding upon example 3, we can specialize the abstract syntax of the general editor calculus presented in fig. 6 for our simple language by defining the set of operators o ranges over. In this case we have that

$$o \in \{let, exp, hole_s, plus, num[n], var[x], hole_e\}$$

allowing us to write editor expressions such as $(@hole_e \Rightarrow \{plus\}.nil|nil)$, which would substitute the current tree encapsulated by the cursor with the $plus$ operator, if the cursor is at the operator $hole_e$.

Cursor Context In editor expressions, we frequently refer to performing actions on or with the abt that is currently encapsulated by the cursor. To support this

notion we introduce cursor contexts C, inspired by the zipper data structure by Huet [2], as a way to locate the cursor in an abt. Central to this idea is the cursor context $[\,\cdot\,]$, or context hole, which specifies the subtree within which the cursor must reside.

To ensure that the cursor can only reside somewhere within the subtree specified by the context hole, we need to introduce some notion of cursorless abt's. Using the abstract syntax, given by $\mathcal{S}$, $\mathcal{O}$ and $\mathcal{X}$, we define a new corresponding abstract syntax which can only build abt's without cursors. Formally, we define the abstract syntax for cursorless abt's according to definition 12.

Definition 12 (Cursorless trees). We let $\hat{\mathcal{S}} = \{\hat{s}\}_{s \in \mathcal{S}}$ denote the sorts of cursorless trees. Then for every operator $o \in \mathcal{O}$ of arity $(\overrightarrow{s_1}.s_1, \ldots, \overrightarrow{s_n}.s_n)s$, excluding cursors, we add the operator $\hat{o}$ of arity $(\overrightarrow{\hat{s}_1}.\hat{s}_1, \ldots, \overrightarrow{\hat{s}_n}.\hat{s}_n)\hat{s}$ to the family of cursorless operators $\hat{\mathcal{O}}$. Finally, we let $\hat{\mathcal{X}}$ denote the family of cursorless variables.

An abt of sort $\hat{s}$ with no cursors is then any $\hat{a} \in \hat{\mathcal{B}}[\hat{\mathcal{X}}]_{\hat{s}}$ that satisfies the conditions in definition 10

The abstract syntax of cursor contexts can then be defined according to definition 13. Using cursor contexts we can interpret any sort of abt $a \in \mathcal{B}[\mathcal{X}]$ as the cursor context $C[a']$, where the abt a' is substituted for the context hole $[\,\cdot\,]$. Due to definition 13 this also means that the cursor can only reside within a', since it cannot be anywhere in C.

Definition 13 (Cursor Contexts). The abstract syntax of cursor contexts is defined as an extension of the syntax for cursorless abt's. It is given by

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 of arity $()C$.
3. Furthermore, for every operator $\hat{o} \in \hat{\mathcal{O}}$ of arity $(\overrightarrow{\hat{s}_1}.\hat{s}_1, \ldots, \overrightarrow{\hat{s}_n}.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ we also add the operator o_i^C of arity $(\overrightarrow{\hat{s}_1}.\hat{s}_1, \ldots, \overrightarrow{\hat{s}_i}.C, \ldots, \overrightarrow{\hat{s}_n}.\hat{s}_n)C$ to $\mathcal{O}^C$. These represents that $[\,\cdot\,]$ must be somewhere in subtree i of operator $\hat{o}$.
4. The family of variables $\mathcal{X}^C = \hat{\mathcal{X}}$

Example 5. Continuing from example 4, we will now extend the editor calculus for our simple language with cursorless abt's and cursor contexts.

Firstly, we define the abstract syntax of abt's without cursors according to definition 12, the result of which is shown in fig. 7. For brevity the syntax is presented as formation rules, since translating these into the notion of operators, while trivial, is not a very concise way of presenting the abstract syntax.

$$
\begin{array}{lcl}
\hat{s} & ::= & \text{let } x = \hat{e} \text{ in } \hat{s} \mid \hat{e} \mid (\!|\,|\!)_s \\
\hat{e} & ::= & \hat{e}_1 + \hat{e}_2 \mid \text{n} \mid \text{x} \mid (\!|\,|\!)_e
\end{array}
$$

Fig. 7: Abstract syntax of abt's without cursors

Secondly, using the syntax of cursorless abt's we can define the abstract syntax of cursor contexts according to definition 13. This is shown in fig. 8.

$$\begin{array}{rcl} C & ::= & \text{let } x = C \text{ in } \hat{s} \mid \text{let } x = \hat{e} \text{ in } C \\ & \mid & C + \hat{e}_2 \mid \hat{e}_1 + C \mid [\cdot] \end{array}$$

Fig. 8: Abstract syntax of cursor contexts

Although a cursor context accurately specifies the subtree where the cursor must reside, it does not inherently guarantee the presence of only one cursor within that subtree. Therefore, it becomes necessary to introduce the concept of well-formed abt's. Informally, a well-formed abt is simply an abt containing exactly one cursor.

Definition 14 (Well-formed trees). Firstly, we define an abstract syntax that describes abt's with exactly one cursor either as the root or as an argument of the root. The syntax is given by

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 $\hat{s} \in \hat{\mathcal{S}}$. These represent that the cursor encapsulates the root of a cursorless abt of sort $\hat{s}$.
3. Furthermore, for every operator $\hat{o} \in \hat{\mathcal{O}}$ of arity $(\overrightarrow{\hat{s}_1}.\hat{s}_1, \ldots, \overrightarrow{\hat{s}_n}.\hat{s}_n)\hat{s}$ and for every $1 \leq i \leq n$ we also add the operator $\dot{o}_i$ of arity $(\overrightarrow{\hat{s}_1}.\hat{s}_1, \ldots, \overrightarrow{\hat{s}_i}.\hat{s}_i, \ldots, \overrightarrow{\hat{s}_n}.\hat{s}_n)\dot{s}$ to $\dot{\mathcal{O}}$. These represent that the cursor encapsulates the cursorless subtree i of operator $\hat{o}$.
4. The family of variables $\dot{\mathcal{X}} = \hat{\mathcal{X}}$.

An abt with exactly one cursor of sort $\dot{s}$ is then any $\dot{a} \in \dot{\mathcal{B}}[\dot{\mathcal{X}}]_{\dot{s}}$ that satisfies the conditions in definition 10. Given this, a well-formed abt $a \in \mathcal{B}[\mathcal{X}]$ is any abt that can be interpreted as $C[\dot{a}]$, since C cannot contain any cursors and $\dot{a}$ contains exactly one.

Example 6. We will now expand upon example 5, and define the notion of well-formed trees according to definition 14 for our simple language. In fig. 9 we show the abstract syntax for trees containing exactly one cursor. Notice that this abstract syntax is an extension of the syntax of cursorless trees from example 5.

$$\begin{array}{rcl} \dot{s} & ::= & \text{let } x = [\hat{e}] \text{ in } \hat{s} \mid \text{let } x = \hat{e} \text{ in } [\hat{s}] \mid [\hat{e}] \mid [\hat{s}] \\ \dot{e} & ::= & [\hat{e}_1] + \hat{e}_2 \mid \hat{e}_1 + [\hat{e}_2] \mid [\hat{e}] \end{array}$$

Fig. 9: Abstract syntax of abt's with exactly one cursor

Given the syntax in fig. 9 and the previously introduced cursor context, we can now determine the well-formedness of any abt in our language. For example

the statement

$$\text{let } x = [(\!|\ |\!)_e] \text{ in } x + 5$$

is well-formed since it can be written as $C[\dot{e}]$ where

$$C = \text{let } x = [\,\cdot\,] \text{ in } x + 5$$
$$\dot{e} = [(\!|\ |\!)_e]$$

It should be noted that there often are two valid interpretations of a tree a as $C[\dot{a}]$. In this example, the other valid interpretation would be:

$$C' = [\,\cdot\,]$$
$$\dot{e}' = \text{let } x = [(\!|\ |\!)_e] \text{ in } x + 5$$

3.2 Semantics

In this section we present the transition systems and the general forms of transition rules defining these systems, based on the generalized syntax of the editor calculus previously presented.

Editor Expressions The labelled transition system for editor expressions is defined as $(Edt \times \mathcal{B}[\mathcal{X}], Apc \cup \{\epsilon\}, \Rightarrow)$. Transitions are of the form $\langle E, a\rangle \stackrel{\alpha}{\Rightarrow} \langle E', a'\rangle$ where the editor expression $E \in Edt$ is closed and the abt $a \in \mathcal{B}[\mathcal{X}]$ is well-formed. The labels of the transitions, α, are the atomic prefix commands Apc and the silent transition ϵ. The transition relation $\Rightarrow$ is defined by the rules shown in fig. 10.

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

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

$$\text{(Seq)}\ \frac{\langle E_1, a\rangle \stackrel{\alpha}{\Rightarrow} \langle E_1', a'\rangle}{\langle E_1 \ggg E_2, a\rangle \stackrel{\alpha}{\Rightarrow} \langle E_1' \ggg E_2, a'\rangle}$$

$$\text{(Seq-Trivial)}\ \frac{}{\langle \text{nil} \ggg E_2, a\rangle \stackrel{\epsilon}{\Rightarrow} \langle E_2, a\rangle}$$

$$\text{(Recursion)}\ \frac{}{\langle \text{rec } x.E, a\rangle \stackrel{\epsilon}{\Rightarrow} \langle E[x := rec\ x.E], a\rangle}$$

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

Fig. 10: Reduction rules for editor expressions

Unlike the editor calculus by Godiksen et al. [8], we do not utilize any form of structural congruence on editor expressions to define the semantics of trivial sequential composition or recursion. Instead we simply define these notions with their own transition rule (Seq-Trivial) and (Recursion).

Substitution and cursor movement The labelled transition system for substitutions and cursor movement is defined as $(\mathcal{B}[\mathcal{X}], Apc, \Rightarrow)$. This system describes the semantics of modifying the well-formed abt a encapsulated by the cursor, by either substituting it with an operator or moving the cursor up or down the tree. Transitions are therefore of the form $a \overset{\pi}{\Rightarrow} a'$, where $a, a' \in \mathcal{B}[\mathcal{X}]$ and $\pi \in Apc$.

For substitution we define a transition rule for every label $\{o\} \in Apc$, which substitutes the abt currently encapuslated by the cursor with o. The general case of this rule is shown in fig. 11. The side-condition ensures that we can only substitute operators of sort s with abt's of sort s.

$$\text{(Insert-op)} \quad \frac{}{[\hat{a}] \overset{\{o\}}{\Rightarrow} [o(\overrightarrow{x_1}.(\!|\,|\!)_{s_1}; \ldots; \overrightarrow{x_n}.(\!|\,|\!)_{s_n})]} \; \hat{a} \in \mathcal{B}[\mathcal{X}]_s, \text{ where } s \text{ is the sort of } o$$

Fig. 11: General form of reduction rule for substitution

For cursor movement we define two transition rules *child* i and *parent* for every operator o of arity $(\overrightarrow{s_1}.s_1, \ldots, \overrightarrow{s_n}.s_n)s$ and for every $1 \leq i \leq n$. These rules, shown in fig. 12, facilitate cursor movement from the parent operator to child i and from child i back to the parent operator, respectively.

$$\text{(Child-i)} \quad \frac{}{[\hat{o}(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)] \overset{child\ i}{\Rightarrow} o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_i}.[a_i]; \ldots; \overrightarrow{x_n}.\hat{a}_n)}$$

$$\text{(Parent)} \quad \frac{}{o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_i}.[\hat{a}_i]; \ldots; \overrightarrow{x_n}.\hat{a}_n) \overset{parent}{\Rightarrow} [\hat{o}(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]}$$

Fig. 12: General form of reduction rules for cursor movement

Example 7. Continuing from example 6, we now define the semantics of the editor calculus for our simple language. We do not show the transition rules for editor expressions, since these are equivalent to the ones presented in fig. 10.

The transition system for cursor movement and substitution is defined with respect to the operators o in our language. Based on the general form of substitution rules shown in fig. 11, we define a specialised rule for substituting the tree encapsulated by the cursor for every operator o. A selection of these are shown in fig. 13.

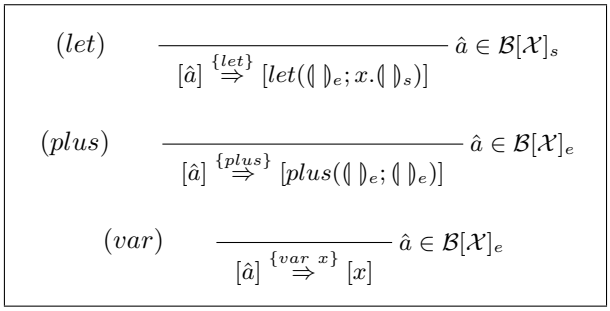


Fig. 13: Selected reduction rules for substitution

Notice that in every rule we ensure that the substitution can only be performed if the abt $\hat{a}$ is of the same sort as the operator. For example, given the configuration $\langle \{let\}.nil, \text{let } x = [\langle\!|\,|\!\rangle_e] \text{ in } x + x \rangle$ we cannot substitute in a statement, as shown below:

$$\text{(Context)}\frac{(let)\ \dfrac{}{[\langle\!|\,|\!\rangle_e] \overset{\{let\}}{\nRightarrow}}\ \langle\!|\,|\!\rangle_e \notin \mathcal{B}[\mathcal{X}]_s}{\langle \{let\}.nil, \text{let } x = [\langle\!|\,|\!\rangle_e] \text{ in } x + x \rangle \overset{\{let\}}{\nRightarrow}}$$

Similarly, using the general form of cursor movement transition rules in fig. 12, we define specialised rules for every argument i of every operator o. In fig. 14 we show the *parent* and *child* rules for the *let* operator specifically. The transition rules for the remaining operators would be defined analogous to these.

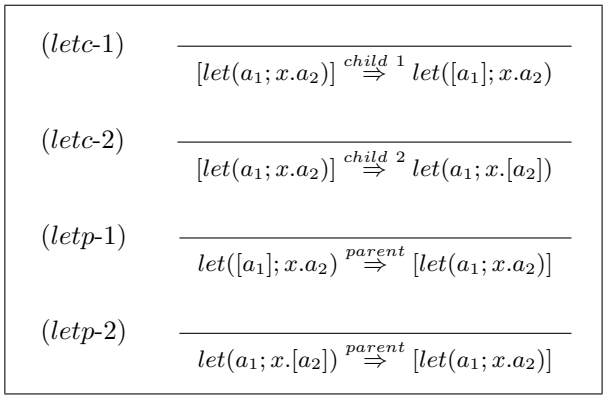


Fig. 14: Reduction rules for cursor movement on the *let* operator

Conditions Finally, we define the satisfaction relation for conditions ϕ in our editor calculus. The propositional connectives are defined as expected in fig. 15.

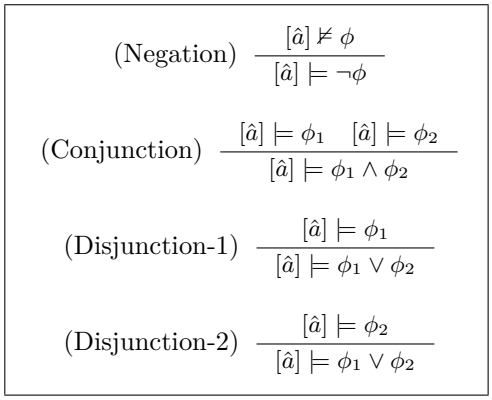


Fig. 15: Satisfaction relation for propositional connectives

The general form of the satisfaction rules for the modalitites $@o$, $\Diamond o$ and $\Box o$ are defined in fig. 16.

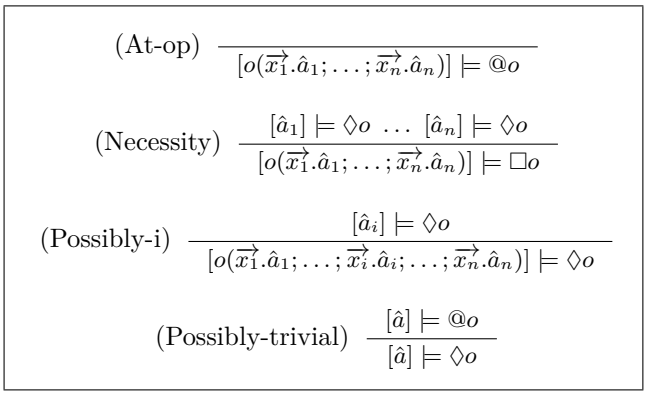


Fig. 16: Satisfaction relation for modal operators

The modal operator $@o$ is satisfied when the root of the abt encapsulated by the cursor is the operator o. Therefore we define the rule (At-op) for every operator o.

The modal operator $\Diamond o$ is satisfied when the operator o is anywhere within the abt encapsulated by the cursor. For the trivial case, when the root of the tree encapsulated by the cursor is o, we only define the one rule (Possibly-trivial). For the non-trivial case, where o is in one of the subtrees $\hat{a}_i$, we define a rule (Possibly-i) for every argument of every operator o.

The modal operator $\Box o$ is satisfied when o is somewhere in every subtree of the abt encapsulated by the cursor. To describe this, we define the rule (Necessity) for every operator o.

Example 8. We finalize our specialized editor calculus from example 7, by defining the satisfaction relation. The propositional connectives are defined as pre-

viously shown in fig. 15. In fig. 17 we have shown the satisfaction rules for all modalities for the *plus* operator. Rules for the remaining operators o are defined accordingly.

$$\text{(At-plus)}\ \frac{}{[plus(\hat{a}_1;\hat{a}_2)] \models @plus} \qquad \text{(Pos1-plus)}\ \frac{[\hat{a}_1] \models \Diamond o}{[plus(\hat{a}_1;\hat{a}_2] \models \Diamond o}$$

$$\text{(Nec-plus)}\ \frac{[\hat{a}_1] \models \Diamond o \quad [\hat{a}_2] \models \Diamond o}{[plus(\hat{a}_1;\hat{a}_2)] \models \Box o} \qquad \text{(Pos2-plus)}\ \frac{[\hat{a}_2] \models \Diamond o}{[plus(\hat{a}_1;\hat{a}_2] \models \Diamond o}$$

Fig. 17: Satisfaction relation for modal operators on *plus*

4 Encoding the generalised editor calculus in an extended lambda calculus

In this section we attempt to soundly encode the generalised editor calculus using the simply typed λ-calculus ($\lambda_{\rightarrow}$) as the base calculus. The calculus will be extended with various constructs such as pattern matching, pairs and the recursive *fix* operator. A concrete overview of the calculus and the constructs necessary to encode every central concept of the editor calculus can be seen in table 1.

Editor calculus	Encoding
Abstract Binding Trees	Simply typed lambda calculus, $\lambda_{\rightarrow}$
Atomic Prefix Commands	$\lambda_{\rightarrow}$ with pattern matching, $\lambda_{\rightarrow,p}$
Modal Logic	$\lambda_{\rightarrow}$ with pattern matching and recursion, $\lambda_{\rightarrow,p,fix}$
Editor Expressions	$\lambda_{\rightarrow}$ with pattern matching and recursion, $\lambda_{\rightarrow,p,fix}$

Table 1: Calculus requirements to encode the generalized editor calculus

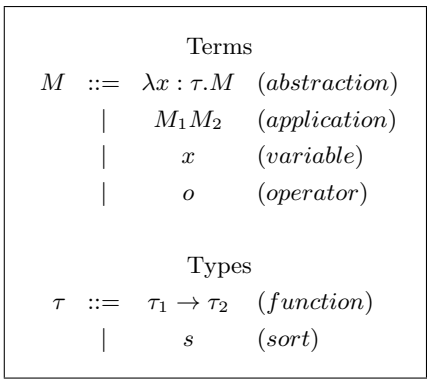
4.1 Motivation

The motivation behind encoding the generalized editor calculus in $\lambda_{\rightarrow,p,fix}$ (or subsets thereof) is most importantly the type system it provides. If our encoding is sound, then any instance of the editor calculus will have a sound type system, regardless of the abstract syntax. This follows from the fact that the type system of $\lambda_{\rightarrow,p,fix}$ is sound. Our editor calculus having a sound type system means that it will reject incorrect editor expressions. A sound type system means that for any well-typed terms, they will remain well-typed during reduction [3].

Secondly a $\lambda_{\rightarrow,p,fix}$ encoding provides a clear strategy for implementing any instantiation of the generalized editor calculus in a functional programming language like Haskell.

4.2 Abstract Binding Trees

To encode abstract binding trees we will use the simply typed lambda-calculus, as presented previously and again on fig. 18. We extend the lambda-calculus with term constants o for every $o \in \mathcal{O}$ excluding cursors, and the base types s for every $s \in \mathcal{S}$.



Terms

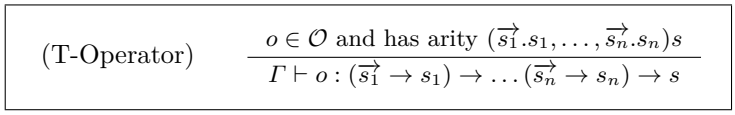
$$
\begin{array}{rcll}
M & ::= & \lambda x : \tau . M & (abstraction) \\
 & | & M_1 M_2 & (application) \\
 & | & x & (variable) \\
 & | & o & (operator)
\end{array}
$$

Types

$$
\begin{array}{rcll}
\tau & ::= & \tau_1 \to \tau_2 & (function) \\
 & | & s & (sort)
\end{array}
$$

Fig. 18: Abstract syntax to encode abstract binding trees, by introducing sorts as base types and operators as term constants.

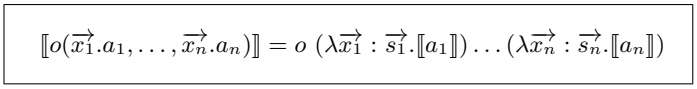
Typing rules for operators can be seen on fig. 19, where we can infer the type of an operator in the lambda-calculus by its arity.



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

Fig. 19: Typing rules for operators.

With the operators added as term constants and sorts added as base types, encoding abts is straightforward as we just have to curry the operator o, encode the children $a_1, \ldots, a_n$ and add typings. This can be seen on fig. 20.



$$[\![o(\overrightarrow{x_1}.a_1, \ldots, \overrightarrow{x_n}.a_n)]\!] = o\ (\lambda \overrightarrow{x_1} : \overrightarrow{s_1}.[\![a_1]\!]) \ldots (\lambda \overrightarrow{x_n} : \overrightarrow{s_n}.[\![a_n]\!])$$

Fig. 20: Encoding of abstract binding trees.

Example 9. Continuing from example 8 we will now encode the operators in our small language. Following our singular rule for encoding abt's we get the resulting encoding seen on fig. 21, and the type for each operator on fig. 22.

$$[\![plus(a_1; a_2)]\!] = plus[\![a_1]\!][\![a_2]\!] \quad [\![num[n]]\!] = num[n]$$

$$[\![var[x]]\!] = var[x] \qquad [\![hole_e]\!] = hole_e$$

$$[\![exp(a_1)]\!] = exp[\![a_1]\!] \qquad [\![hole_s]\!] = hole_s$$

$$[\![let(a_1; x.a_2)]\!] = let\ ([\![a_1]\!])(\lambda x : e.[\![a_2]\!])$$

Fig. 21: Encoding of operators.

$$plus : e \to e \to e \quad num[n] : e$$

$$var[x] : e \qquad hole_e : e$$

$$exp : e \to s \qquad hole_s : s$$

$$let : e \to (e \to s) \to s$$

Fig. 22: Types for the encoded operators.

4.3 Cursor Contexts

To represent abts with cursor contexts we make use of a 2-tuple. Such a construct can be encoded as $\lambda x.\lambda y.\lambda f.fxy$, however, to improve readability we introduce the pair construct similarly to Pierce [3]. In fig. 23 we introduce the new syntactic forms which include the pair construct, first projection, second projection and the product type.

Terms

M	::=	(M_1, M_2)	(pair)
	|	$M.1$	(first projection)
	|	$M.2$	(second projection)

Types

τ	::=	$\tau_1 \times \tau_2$	(product type)

Fig. 23: Abstract syntax for representing cursor contexts as pairs.

The reduction rules for the pair construct can be seen in fig. 24, where the axioms for first and second projection are defined.

$$\text{(E-Proj1)}\ \frac{}{(M_1, M_2).1 \to M_1} \qquad \text{(E-Proj2)}\ \frac{}{(M_1, M_2).2 \to M_2}$$

Fig. 24: Reduction rules for pairs.

Typing rules can be found in fig. 25, where we see that the type of a first projection is the type of the first term in a pair, the type of a second projection is the type of the second term of a pair and the type of a pair is the product type of the first and second term in the pair.

$$\text{(T-Proj1)}\ \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash M.1 : \tau_1} \qquad \text{(T-Proj2)}\ \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash M.2 : \tau_2}$$

$$\text{(T-Pair)}\ \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

Fig. 25: Typing rules for pairs.

In fig. 26 the encoding for cursor contexts can be seen. We can encode a cursor context $C[a]$ as a pair of two encoded trees the first being a regular abt and the second being an abt with a context hole $[\,\cdot\,]$, which we encode as an operator $[\,\cdot\,]$. We also introduce a type alias such that $[\![C[a]]\!]$ has type $Ctx = s \times s$.

$$[\![C[a]]\!] = ([\![a]\!], [\![C]\!])$$
$$[\![[\,\cdot\,]]\!] = [\,\cdot\,]$$

Fig. 26: Encoding of cursor contexts.

Example 10. In example 6 we saw that the expression.

$$\text{let } x = [\!(\,)\!]_e] \text{ in } x + 5$$

is well-formed since it can be written as $C[\dot{e}]$ where

$$\begin{aligned} C = &\quad \text{let } x = [\,\cdot\,] \text{ in } x + 5 \\ \dot{e} = &\quad [\!(\,)\!]_e] \end{aligned}$$

Encoding $C[\dot{e}]$ following the rules for encoding contexts and the rules for encoding abts in example 9 yields the following encoding:

$$\begin{aligned}
[\![C[\dot{e}]]\!] &= ([\![\dot{e}]\!], [\![C]\!]) \\
&= ([\![cursor_e(hole_e)]\!], [\![let([\cdot]; x.plus(var[x], num[5]))]\!]) \\
&= (cursor_e([\![hole_e]\!]), let\ [\![[\cdot]]\!]\ \lambda x.[\![plus(var[x], num[5]]\!])) \\
&= (cursor_e(hole_e), let\ [\cdot]\ (\lambda x.plus\ [\![var[x]]\!]\ [\![num[5]]\!])) \\
&= (cursor_e(hole_e), let\ [\cdot]\ (\lambda x.plus\ var[x]\ num[5]))
\end{aligned}$$

4.4 Atomic Prefix Commands

To encode the atomic prefix commands we further extend the lambda calculus with pattern matching. As seen in fig. 27 we introduce the match construct where we match the term M on the patterns p resulting in the corresponding term N if there is a match. As mentioned p is the syntactic category describing the pattern we are trying to match. These patterns can be a variable which will be bound to whatever is left of the pattern, a wildcard matching anything, operators, pairs, and bindings.

Terms

M, N	::=	$match\ M\ \overrightarrow{p \to N}$	(match construct)
p	::=	x	(variable)
	|	$\_$	(wildcard)
	|	$o\ \overrightarrow{p}$	(operator)
	|	(p_1, p_2)	(pair)
	|	$.p$	(binding)

Fig. 27: Abstract syntax for representing atomic prefix commands, extending the lambda-calculus with a match construct.

The reduction rules for the match construct can be seen in fig. 28. The reduction rule says that if there exists a matching pattern, we reduce to the corresponding term while also binding the variables to the matches found in the pattern. If multiple patterns match, we choose the pattern appearing first in the vector of patterns. The reduction rule uses the auxiliary function *binds* which takes a term and a pattern as input and evaluates either to a function of variable bindings or a failure value.

$$\text{(E-Match)} \quad \frac{\sigma_i = binds(M, p_i) \neq fail \quad \forall j < i. binds(M, p_j) = fail}{match\ M\ \overrightarrow{p \to N} \to N_i \sigma_i}$$

$$binds : \ M \times p \to (Var \to M) \cup \{fail\}$$

$$\begin{aligned}
binds(M, x) &= [x \to M] \\
binds(M, _) &= [] \\
binds(o\ a_1\ \ldots\ a_n, o\ p_1\ \ldots\ p_n) &= binds(c_1, p_1) \circ \cdots \circ binds(c_n, p_n) \\
binds((M, N), (p_1, p_2)) &= binds(M, p_1) \circ binds(N, p_2) \\
binds(M, .p) &= binds(M, p) \\
binds(\lambda x.M, .p) &= binds(M, .p)
\end{aligned}$$

for remaining values in the domain of $binds$ the result is defined as $fail$.

$fail$ is defined as the function that always returns $fail$.

Fig. 28: Reduction rules for the match construct.

In fig. 29 we define the typing rules for the match construct. A match construct has type T if for all matching patterns p_i their corresponding term N_i also has type T given the typings for the pattern variables $x_1, \ldots x_n$ are added to the typing context Γ.

$$\text{(T-Match)} \quad \frac{\text{forall } i \text{ satisfying } \sigma_i = binds(M, p_i) \neq fail \quad \Gamma \vdash N_i \sigma_i : T}{\Gamma \vdash match\ M\ \overrightarrow{p \to N} : T}$$

Fig. 29: Typing rules for the match construct.

We now define the auxiliary functions as seen in fig. 30. These function as the operations of a zipper data structure, which helps encode atomic prefix commands on abts. We also introduce the abbreviation $[a]$ to mean the cursor operator with the child a.

$$
\begin{array}{ll}
\mathit{down} & \stackrel{def}{=} \lambda x : s.\text{match } x \\
& \quad [o\ (.a_1) \ldots (.a_n)] \to o\ (\lambda \overrightarrow{x_1}.[a_1]) \ldots (\overrightarrow{x_n}.a_n) \\
\\
\mathit{right} & \stackrel{def}{=} \lambda x : s.\text{match } x \\
& \quad o\ (.a_1)\ \ldots (.[a_i])\ \ldots (.a_n) \to o\ (\lambda \overrightarrow{x_1}.a_1)\ \ldots (\lambda \overrightarrow{x_{i+1}}.[a_{i+1}])\ \ldots (\overrightarrow{x_n}.a_n) \\
\\
\mathit{up} & \stackrel{def}{=} \lambda x : s.\text{match } x \\
& \quad o\ (.a_1) \ldots (.[a_i]) \ldots (.a_n) \to [o\ (\lambda \overrightarrow{x_1}.a_1) \ldots (\lambda \overrightarrow{x_n}.a_n)] \\
\\
\mathit{set} & \stackrel{def}{=} \lambda a : s.\lambda x : s.\text{match } x \\
& \quad [a'] \to [a]
\end{array}
$$

Fig. 30: Function definitions for cursor movement and substitution on abstract binding trees.

The encoding of atomic prefix commands can be seen in fig. 31. The *child* command can be encoded as a base case and a recursive case. In the base case we want to move to the first *child*, which means we just go down. In the recursive case we want to move to *child* n, we can encode this as moving to *child* $n-1$ and then moving right. *parent* and *insert* can directly be encoded as *up* and *set* respectively.

$$
\begin{array}{ll}
[\![\mathit{child}\ 1]\!] = \mathit{down} & [\![\mathit{child}\ n]\!] = \mathit{right}\ [\![\mathit{child}\ n-1]\!] \\
\\
[\![\mathit{parent}]\!] = \mathit{up} & [\![\mathit{insert}\ a]\!] = \mathit{set}\ [\![a]\!]
\end{array}
$$

Fig. 31: Encoding of atomic prefix commands.

4.5 Control Structures and Modal Logic

To encode control structures and modal logic, we need to further extend the lambda-calculus with a *fix* operator as well as the *Bool* base type. We introduce the *fix* operator as seen in Pierce [3]. The new syntactic forms can be seen on fig. 32, in which the *fix* operator and boolean values are added.

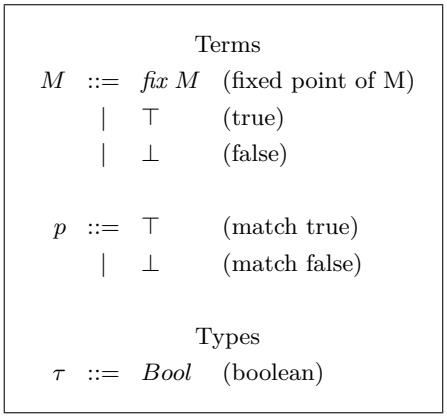


Fig. 32: Abstract syntax for control structures and modal logic, extending the calculus with the *fix* operator and boolean types and term constants.

To be able to pattern match on boolean values we also extend the *binds* function, which can be seen in fig. 33.

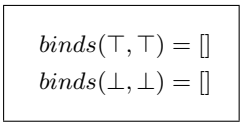


Fig. 33: Extension of binds function to account for booleans.

In fig. 34 the reduction rules for *fix* can be seen. E-FixBeta is an axiom stating that we can unravel another layer of recursion. E-Fix states that if a term M has a transition, then $fix\ M$ also has a transition. The typing rules for *fix* and booleans are shown on fig. 35, which state that *fix* has type T if its argument has type $T \to T$, and that the terms $\top$ and $\bot$ have type $Bool$.

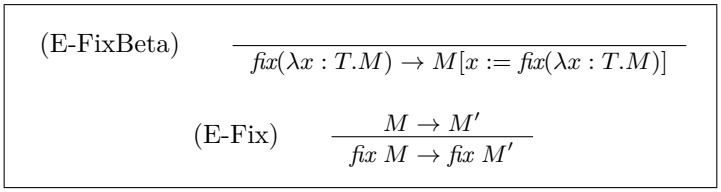


Fig. 34: Reduction rules for the *fix* operator.

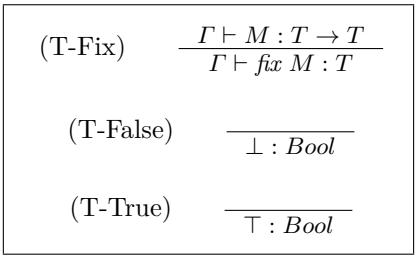


Fig. 35: Typing rules for the *fix* operator and booleans.

On fig. 36 the encoding of editor expressions can be seen. For atomic prefixes the atomic prefix command should be applied before the rest of the editor expression. The *nil* editor expression will become the identity function. Sequential editor expressions will apply the encoding of the editor expressions in the correct order. Recursion works by using the newly introduced *fix* operator. Lastly, conditional editor expressions are encoded such that we match on the evaluation of ϕ on the first projection of a context C and apply the encoding of E_1 or E_2 on C accordingly.

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

$$[\![E_1 \ggg E_2]\!] = \lambda C : Ctx.[\![E_2]\!]([\![E_1]\!]C) \qquad [\![\text{Rec } x.E]\!] = \mathit{fix}(\lambda x : (Ctx \to Ctx).[\![E]\!])$$

$$[\![\phi \Rightarrow E_1 | E_2]\!] = \lambda C : Ctx.\text{match } ([\![\phi]\!]\ C.1) \qquad [\![\langle E, C[a'] \rangle]\!] = [\![E]\!]\ ([\![a']\!], [\![C]\!])$$
$$\mid \top \to [\![E_1]\!]\ C$$
$$\mid \bot \to [\![E_2]\!]\ C$$

Fig. 36: Encoding of editor expressions and context configuration.

In fig. 37 we define auxiliary functions that will help with the encoding of conditional expressions. The auxiliary functions *and*, *or* and *neg* function as their name would suggest. In our semantics, when checking if a formula ϕ holds, we do so against an abt a (i.e $a \vDash \phi$). When constructing the proof tree for such a formula we see that this abt a is propagated up through the proof tree, thus we need to have the ability to propagate an abt throughout our encoded formula as well. To propagate this abt through *and*, *or* and *neg* we define the combinators *phoenix* (liftA2/liftM2 in Haskell) and *bluebird* (function composition / B combinator).

$$
\begin{array}{lcl}
\mathit{phoenix} & \stackrel{def}{=} & \lambda f : (\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}). \\
 & & \lambda g : (s \to \mathit{Bool}). \\
 & & \lambda h : (s \to \mathit{Bool}). \\
 & & \lambda a : s. \\
 & & f(ga)(ha) \\
 & & \\
\mathit{bluebird} & \stackrel{def}{=} & \lambda f : (\mathit{Bool} \to \mathit{Bool}). \\
 & & \lambda g : (s \to \mathit{Bool}). \\
 & & a : s. \\
 & & f(ga) \\
 & & \\
\mathit{or} & \stackrel{def}{=} & \lambda b_1 : \mathit{Bool}.\lambda b_2 : \mathit{Bool}.\text{match } (b1, b2) \\
 & | & (\bot, \bot) \to \bot \\
 & | & (\_, \_) \to \top) \\
 & & \\
\mathit{and} & \stackrel{def}{=} & \lambda b_1 : \mathit{Bool}.\lambda b_2 : \mathit{Bool}.\text{match } (b1, b2) \\
 & | & (\top, \top) \to \top \\
 & | & (\_, \_) \to \bot) \\
 & & \\
\mathit{neg} & \stackrel{def}{=} & \lambda b : \mathit{Bool}.\text{match } b \\
 & | & \top \to \bot \\
 & | & \bot \to \top
\end{array}
$$

Fig. 37: Function definitions for auxiliary functions.

The encoding of conjunction, disjunction and negation, shown in fig. 38, comes quite naturally using the previously defined auxiliary functions.

$$
\begin{array}{l}
[\![\phi_1 \wedge \phi_2]\!] = \mathit{phoenix}\ \mathit{and}\ [\![\phi_1]\!]\ [\![\phi_2]\!] \\
 \\
[\![\phi_1 \vee \phi_2]\!] = \mathit{phoenix}\ \mathit{or}\ [\![\phi_1]\!]\ [\![\phi_2]\!] \\
 \\
[\![\neg\phi]\!] = \mathit{bluebird}\ \mathit{neg}\ [\![\phi]\!] \\
 \\
[\![@o]\!] = \lambda x : s.\text{match } x \\
\quad | \ [o\ \_\ \ldots\ \_] \to \top \\
\quad | \ \_ \to \bot
\end{array}
$$

Fig. 38: Encoding of propositional connectives

In fig. 39 we show the encoding of modal logic operators.

$$
\begin{aligned}
&[\![\Diamond o]\!] = \mathit{fix}(\lambda f : (s \to \mathit{Bool}).\lambda x : s.\text{match } x \\
&\quad \mid [o \ \_ \ \ldots \ \_] \to \top \\
&\quad \mid [\_(\_.a_1, \ldots, \_.a_n)] \to \mathit{or}(\ldots(\mathit{or}\ (f\ [a_1])\ (f\ [a_2]))\ \ldots)\ (f\ [a_n]) \\
&\quad \mid \_ \to \bot) \\
\\
&[\![\Box o_1]\!] = \lambda x : s.\text{match } x \\
&\quad \mid \_(\_.a_1, \ldots, \_.a_n) \to \mathit{and}(\ldots(\mathit{and}([\![\Diamond o]\!]\ a_1)([\![\Diamond o]\!]\ a_2))\ldots)\ ([\![\Diamond o]\!]\ a_n) \\
&\quad \mid \_ \to \top
\end{aligned}
$$

Fig. 39: Encoding of modal logic and boolean logic operators.

@o is encoded such that we pattern match on the on the operator o, and if there is a match the resulting value is true, otherwise it is false.

$\Diamond o$ uses the *fix* operator, as the function is recursively defined. We have three patterns to check;

1. If the operator matches the one we are looking for return true.
2. If the current operator has any children, recursively check them and combine their result through *or*. This pattern should technically be multiple patterns, one for each number of children that are possible in a given sort, but we use a bit of syntactic sugar to circumvent this.
3. The current node is not a match nor does it have any children, therefore we must return false.

$\Box o$ consists of two patterns;

1. Does the current operator have any children? If so apply all of them to $[\![\Diamond o]\!]$ and combine their result through *and*.
2. The current operator does not have any children, therefore we can return true.

Example 11. To illustrate how editor expressions are encoded, we provide an encoding of the following expression

$$\mathit{child}\ 2.\mathit{nil} \ggg \mathit{parent}.\mathit{nil}$$

using the encoding rules seen on fig. 36 and fig. 31. For readability purposes we will not add explicit typings, just note that any variable C_i has type Ctx.

$$
\begin{aligned}
& [\![child\ 2.nil \ggg parent.nil]\!] \\
= {} & \lambda C_1.[\![parent.nil]\!]([\![child\ 2.nil]\!]C_1) \\
= {} & \lambda C_1.(\lambda C_2.[\![nil]\!]([\![parent]\!]C_2.1, C_2.2))([\![child\ 2.nil]\!]C_1) \\
= {} & \lambda C_1.(\lambda C_2.[\![nil]\!](up\ C_2.1, C_2.2))([\![child\ 2.nil]\!]C_1) \\
= {} & \lambda C_1.(\lambda C_2.(\lambda C_3.C_3)(up\ C_2.1, C_2.2))([\![child\ 2.nil]\!]C_1) \\
\to_\beta {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))([\![child\ 2.nil]\!]C_1) \\
= {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))(\lambda C_3.[\![nil]\!]([\![child\ 2]\!]C_3.1, C_3.2)C_1) \\
= {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))(\lambda C_3.(\lambda C_4.C_4)([\![child\ 2]\!]C_3.1, C_3.2)C_1) \\
\to_\beta {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))(\lambda C_3.([\![child\ 2]\!]C_3.1, C_3.2)C_1) \\
= {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))(\lambda C_3.right([\![child\ 1]\!](C_3.1, C_3.2))C_1) \\
= {} & \lambda C_1.(\lambda C_2.(up\ C_2.1, C_2.2))(\lambda C_3.right(down(C_3.1, C_3.2))C_1)
\end{aligned}
$$

As can be seen, the editor expression could be fully encoded to the extended lambda calculus.

5 Proof of soundness

To prove that the encoding of our generalised editor-calculus is sound, we will need to show that our encoding essentially works in the same way as the transition rules for the editor calculus. To do that, we need to show that the encoding of our atomic prefix operations, when applied to a tree, returns the correct modified tree. Furthermore, we need to show that the encoding of our modal logic return the correct boolean value, when applied to a tree. These two properties will be used to show that the encoding of the editor expressions is sound.

5.1 Atomic prefix commands

Lemma 1 (Soundness of atomic prefix commands). For every abt a, if there exists a transition

$$a \overset{\pi}{\Rightarrow} a' \tag{1}$$

then the following β-reduction is possible on the encoding

$$[\![\pi]\!]\ [\![a]\!] \to_\beta^* [\![a']\!] \tag{2}$$

Proof. To prove lemma 1, we will show that for each atomic prefix command, the lemma holds. To do this, we need to show that the encoding of the atomic prefix operation applied on the encoding of the original tree, will lead to the encoding of the modified tree. We begin with the rule for the *parent* atomic prefix command:

(Parent) The rule for the *parent* atomic prefix command is:

$$\text{(Parent)}\ \frac{}{o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_i}.[\hat{a}_i];\ldots;\overrightarrow{x_n}.\hat{a}_n) \overset{parent}{\Rightarrow} [\hat{o}(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_n}.\hat{a}_n)]}$$

To show that lemma 1 holds for the case of the (Parent)-rule, we first encode the left-hand side of the transition:

$$\begin{aligned}&[\![parent]\!][\![o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_i}.[\hat{a}_i];\ldots;\overrightarrow{x_n}.\hat{a}_n)]\!]\\&= up\ (o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!]))\end{aligned} \tag{3}$$

The encoding of the right-hand side of the $\overset{parent}{\Rightarrow}$ transition would be:

$$[\![[o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_n}.\hat{a}_n)]]\!] = [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])] \tag{4}$$

We will now show that by β-reduction, the first encoding will reach the second encoding:

$$\begin{aligned}&up\ (o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]]),\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!]))\\&= \Big(\lambda x : s.\text{match } x\ (o\ (.a_1)\ldots(.[a_i])\ldots(.a_n)) \to [o\ (\lambda\overrightarrow{x_1}.a_1)\ldots(\lambda\overrightarrow{x_n}.a_n)]\Big)\\&\quad\ \big(o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\big)\\&\to_\beta \Big(\text{match } o\ (\lambda\overrightarrow{x}_1.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]])\ldots(\lambda\overrightarrow{x}_n.[\![a_n]\!])\\&\quad\ (o\ (.a_1)\ldots(.[a_i])\ldots(.a_n)) \to [o\ (\lambda\overrightarrow{x_1}.a_1)\ldots(\lambda\overrightarrow{x_n}.a_n)]\Big)\\&\to_\beta [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])]\end{aligned}$$

Which was exactly the encoding of the first transition. Therefore the lemma holds in the case of the *parent*-rule.

For the full proof of lemma 1 we refer to appendix A.

5.2 Modal logic

We also need to show that the encoding of the modal logic has the property that it always evaluates to true when the given tree satisfies ϕ, and to false when it does not:

Lemma 2 (Modal logic encoding). For every tree a and every ϕ, if in the editor calculus we have:

$$a \models \phi \tag{5}$$

then we can reduce the encoding such that

$$[\![\phi]\!]\ [\![a]\!] \to_\beta^* \top \tag{6}$$

Correspondingly, if we have that

$$a \nvDash \phi \tag{7}$$

then we can reduce the encoding such that

$$[\![\phi]\!]\ [\![a]\!] \to_\beta^* \bot \tag{8}$$

Proof. To prove that lemma 2 holds, we need to show that for each operator-rule in the modal logic, and each boolean operator, the lemma holds. To do this, we will use induction on the height of the derivation trees of the judgement $a \models \phi$. We will begin with the base-cases, and show that the lemma holds for each of the modal logic operators. Then we will use induction to prove the lemma also holds for the propositional connectives negation, conjunction and disjunction. We will begin with the @o-operator:

(At-op) To prove that lemma 2 holds for the (At-Op)-rule, we will encode the operator, and show that when the tree satisfies the judgement, the encoding of the tree and the @o operator evaluates to $\top$. The rule is:

$$\text{(At-Op)} \qquad \frac{}{[o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)] \models @o}$$

We show that the encoding evaluates to $\top$, if the operator is the given operator, and $\bot$ if it is not. The encoding of the operator in the tree and the @o-operator are:

$$\begin{aligned}
&[\![@o]\!][\![[o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]]\!] \\
&= (\lambda x : s.\text{match } x\ [o\ _\ \ldots\ _] \to \top \mid _ \to \bot)\ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
&\to_\beta (\text{match } [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])]\ [o\ _\ \ldots\ _] \to \top \mid _ \to \bot) \\
&\to_\beta \top
\end{aligned}$$

We see that it evaluates to $\top$, and therefore the lemmas first part holds for the (At-Op)-rule. We will show it returns false, when the operator is a different operator, o':

$$\begin{aligned}
&[\![@o]\!][\![[o'(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]]\!] \\
&= (\lambda x : s.\text{match } x\ [o\ _\ \ldots\ _] \to \top \mid _ \to \bot)(o'\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])) \\
&\to_\beta (\text{match } (o'\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!]))\ [o\ _\ \ldots\ _] \to \top \mid _ \to \bot) \\
&\to_\beta \bot
\end{aligned}$$

Therefore, the second part of the lemma holds for the @o-operator, and therefore, the whole lemma holds for the (At-op)-rule.

For the full proof of lemma 2 we refer to appendix B.

5.3 Editor expressions

We will now look at the soundness theorem for editor expressions:

Theorem 1 (Soundness of editor expressions). For every editor expression, if we have the transition:

$$\langle E, a\rangle \to \langle E', a'\rangle \tag{9}$$

Then the encoding will also be able to do the following beta-reduction:

$$[\![\langle E, a\rangle]\!] \to_\beta^* [\![\langle E', a'\rangle]\!] \tag{10}$$

Proof. To prove theorem 1, we will use induction on the height of the derivation tree, and make use of lemma 1 and lemma 2. There is only one rule in the editor calculus, the (Seq)-rule, that is not an axiom, but has as premise that a transition for another editor expression is possible. Therefore, we first prove that the theorem holds for all axioms, as these rules does not need an inductive hypothesis. Finally, we show that the theorem holds for the rule (Seq), provided the theorem holds for the derivation tree of the premise.

(Conditional) We now show that the 2 conditional rules are sound. The conditional rules are:

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

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

The encoding of the left-hand side of both conditional rules are:

$$\begin{aligned}[\![\langle \phi \Rightarrow E_1 | E_2, C[a]\rangle]\!] = \big(&\lambda C' : Ctx.\text{match } ([\![\phi]\!]\ C'.1)\\ &\mid \top \to [\![E_1]\!]\ C'\\ &\mid \bot \to [\![E_2]\!]\ C'\big)\ ([\![a]\!], [\![C]\!])\end{aligned}$$

The encoding of the right-hand side in the first rule is:

$$[\![\langle E_1, C[a]\rangle]\!] = [\![E_1]\!][\![C[a]]\!] = [\![E_1]\!]([\![a]\!], [\![C]\!]) \tag{11}$$

The encoding of the right-hand side in the second rule is:

$$[\![\langle E_2, C[a]\rangle]\!] = [\![E_2]\!][\![C[a]]\!] = [\![E_2]\!]([\![a]\!], [\![C]\!]) \tag{12}$$

We will look at the two cases of the conditional-rule. The first is where $a \models \phi$. From lemma 2 we know that if that is the case, then $[\![\phi]\!]\ [\![a]\!]$ reduces to $\top$.

We will use this to prove that the first rule is soundly encoded:

$$\begin{aligned}
[\![\langle \phi \Rightarrow E_1 | E_2, C[a] \rangle]\!] = {} & \big(\lambda C' : Ctx.\text{match } ([\![\phi]\!]\ C'.1) \\
& \quad \mid \top \to [\![E_1]\!]\ C' \\
& \quad \mid \bot \to [\![E_2]\!]\ C'\big)\ ([\![a]\!], [\![C]\!]) \\
\to_\beta {} & \text{match } ([\![\phi]\!]\ \ ([\![a]\!], [\![C]\!]).1) \\
& \quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\
& \quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\
\to_\beta {} & \text{match } ([\![\phi]\!]\ [\![a]\!]) \\
& \quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\
& \quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\
\to_\beta^* {} & \text{match } (\top) \\
& \quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\
& \quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\
\to_\beta {} & [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!])
\end{aligned}$$

This is exactly the same as the right-hand side of the first rule, and therefore, we can say that the conditionals first rule is soundly encoded. We now look at the case where we have that $a \not\models \phi$. From lemma 2 we know that this means that $[\![\phi]\!]\ [\![a]\!]$ reduces to $\bot$. Therefore, the reduction of the encoding in this case will be:

$$\begin{aligned}
& \text{match } ([\![\phi]\!]\ [\![a]\!]) \\
& \quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\
& \quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\
\to_\beta^* {} & \text{match } (\bot) \\
& \quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\
& \quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\
\to_\beta {} & [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!])
\end{aligned}$$

This is exactly the right-hand side of the second rule encoded, and therefore the second rule is also soundly encoded. As the second rule is also soundly encoded, we have proven that the conditional statement is soundly encoded.

For the full proof of theorem 1 we refer to appendix C.

6 Conclusion & Further Work

We have developed a generalized editor calculus that enables the creation of a syntax-directed editor calculus for a specific abstract syntax. Subsequently, we

encoded this editor calculus into an extended version of the lambda calculus, which incorporates recursion and pattern matching.

Furthermore, we have proven the soundness of the encoding of the generalized editor calculus into the extended lambda calculus. As a consequence, given that the type system of the extended lambda calculus is sound, it logically follows that our editor calculus also possesses a sound type system.

We have identified several possibilities for expanding our work on the generalized editor calculus. First of all, it would be excellent to prove that our encoding is complete. However, in the scope of this paper, we decided to only include the proof of the soundness of the encoding, as that would give us the sound type system.

One expansion we also attempted involved implementing a *copy* and *paste* operation capable of storing an entire tree in a separate clipboard and subsequently pasting it into a new location within the tree. However, we encountered issues related to name clashes. This problem arises from the fact that when copying a subtree, moving the cursor to a different position, and then attempting to paste it there, the free variables could become bound to different bindings than their original ones. This would result in them functioning as different variables. An example can be seen in eq. (13), where an occurrence of x gets copied into another location, where it would refer to another binding than the one it originally referred to.

$$\begin{aligned} \mathit{copy} &: \langle \lambda x.(x + x)\ (\lambda x.(x - [x])\ 4), \_ \rangle \\ \mathit{move} &: \langle \lambda x.(x + x)\ (\lambda x.(x - [x])\ 4), x \rangle \\ \mathit{paste} &: \langle \lambda x.([x + x])\ (\lambda x.(x - x)\ 4), x \rangle \\ \mathit{result} &: \langle \lambda x.([x])\ (\lambda x.(x - x)\ 4), x \rangle \end{aligned} \tag{13}$$

We could fix this by performing α-conversion on the whole tree. However, sometimes we wanted the free variables of the copied tree to be captured, for example if you copied a subtree with free variabels, that in an outer tree were bound variables, and then tried to paste the copied tree into the same location, as we copied it from. An example of this can be seen in eq. (14).

$$\begin{aligned} \mathit{copy} &: \langle (\lambda x.(x + [x])\ 4), \_ \rangle \\ \mathit{paste} &: \langle (\lambda x.(x + [x])\ 4), x \rangle \\ \mathit{result} &: \langle (\lambda z.(z + [x])\ 4), x \rangle \end{aligned} \tag{14}$$

In this case it would not make sense to α-convert the tree, as the free variables originally were meant to be bound by that binding.

We could have made the design choice that, when pasting a tree with free variables that might be captured, we would α-convert the tree and not concern ourselves with whether it would be captured by its original binding. However, this would have meant sacrificing much of the functionality of the *copy* and *paste* operations. We did attempt to incorporate both possibilities of pasting a tree into its original bindings while still avoiding name clashes when pasted elsewhere. However, our attempt required that all variable bindings in the edited tree had

to have different names. This imposed significant requirements on inserting variable bindings, as the newly inserted bindings could not share the same variable name as any other bindings in the tree, nor have the same name as a removed binding to which variables in the copied tree still originally referred. This property introduced a considerable amount of overhead and removed the elegance of the insert rule.

Moreover, properly encoding this property in the extended lambda calculus proved to be rather challenging. Therefore, in this paper, we chose to abandon it for future research, where a more elegant way of incorporating *copy* and *paste* operations could potentially be devised.

Another avenue for future work could be the introduction of an *undo* operation, allowing users to reverse operations performed on the tree. This feature has been implemented in the original editor calculus as presented in Kjær et al. [9] and could potentially be extended to the generalized editor calculus as well.

References

[1] F. Pfenning and C. Elliott. "Higher-Order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN* 1988 *Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 199–208. ISBN: 0897912691. DOI: 10.1145/53990.54010. URL: https://doi.org/10.1145/53990.54010.

[2] Gérard Huet. "The Zipper". In: *J. Funct. Program.* 7 (Sept. 1997), pp. 549–554. DOI: 10.1017/S0956796897002864.

[3] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[4] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139032636.

[5] Robert Harper. *Practical Foundations for Programming Languages*. 2nd. USA: Cambridge University Press, 2016. ISBN: 1107150302.

[6] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. "Hazelnut: A Bidirectionally Typed Structure Editor Calculus". In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 86–99. ISSN: 0362-1340. DOI: 10.1145/3093333.3009900. URL: https://doi.org/10.1145/3093333.3009900.

[7] Jesse Alama and Johannes Korbmacher. "The Lambda Calculus". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[8] Christian Godiksen, Thomas Herrmann, Hans Hüttel, Mikkel Korup Lauridsen, and Iman Owliaie. "A Type-Safe Structure Editor Calculus". In: *Proceedings of the* 2021 *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 1–13. ISBN: 9781450383059. DOI: 10.1145/3441296.3441393. URL: https://doi.org/10.1145/3441296.3441393.

[9] Rasmus Rendal Kjær, Magnus Holm Lundbergh, Magnus Mantzius Nielsen, and Hans Hüttel. “An Editor Calculus With Undo/Redo”. English. In: *Proceedings of* 23*rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (*SYNASC*). Ed. by Carsten Schneider, Mircea Marin, Viorel Negru, and Daniela Zaharie. Proceedings - 2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2021. Publisher Copyright: © 2021 IEEE.; 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), SYNASC ; Conference date: 07-12-2021 Through 10-12-2021. United States: IEEE, Dec. 2021, pp. 66–74. DOI: 10.1109/SYNASC54541.2021.00023. URL: https://synasc.ro/2021/.

A Soundness proof of atomic prefix commands

Lemma 1 (Soundness of atomic prefix commands). For every abt a, if there exists a transition

$$a \overset{\pi}{\Rightarrow} a' \tag{15}$$

then the following β-reduction is possible on the encoding

$$[\![\pi]\!]\ [\![a]\!] \to_\beta^* [\![a']\!] \tag{16}$$

Proof. To prove lemma 1, we will show that for each atomic prefix command, the lemma holds. To do this, we need to show that the encoding of the atomic prefix command applied on the encoding of the original tree, will lead to the encoding of the modified tree. We will begin with the rule for the *parent*-atomic prefix operation:

(Parent) The rule for the *parent* atomic prefix command is:

$$\text{(Parent)}\ \frac{}{o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_i}.[\hat{a}_i]; \ldots; \overrightarrow{x_n}.\hat{a}_n) \overset{parent}{\Rightarrow} [\hat{o}(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]}$$

To show that lemma 1 holds for the case of the (Parent)-rule, we first encode the left-hand side of the transition:

$$\begin{aligned} &[\![parent]\!][\![o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_i}.[\hat{a}_i]; \ldots; \overrightarrow{x_n}.\hat{a}_n)]\!] \\ &= up\ (o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_i}.[[\![a_i]\!]]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])) \end{aligned} \tag{17}$$

The encoding of the right-hand side of the $\overset{parent}{\Rightarrow}$ transition would be:

$$[\![[o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]]\!] = [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \tag{18}$$

We will now show that by β-reduction, the first encoding will reach the second encoding:

$$\begin{aligned} &up\ (o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_i}.[[\![a_i]\!]]), \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])) \\ &= \Big(\lambda x : s.\text{match } x\ (o\ (.a_1) \ldots (.[a_i]) \ldots (.a_n)) \to [o\ (\lambda\overrightarrow{x_1}.a_1) \ldots (\lambda\overrightarrow{x_n}.a_n)]\Big) \\ &\quad \big(o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_i}.[[\![a_i]\!]]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])\big) \\ &\to_\beta \Big(\text{match } o\ (\lambda\overrightarrow{x}_1.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_i}.[[\![a_i]\!]]) \ldots (\lambda\overrightarrow{x}_n.[\![a_n]\!]) \\ &\quad (o\ (.a_1) \ldots (.[a_i]) \ldots (.a_n)) \to [o\ (\lambda\overrightarrow{x_1}.a_1) \ldots (\lambda\overrightarrow{x_n}.a_n)]\Big) \\ &\to_\beta [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \end{aligned}$$

Which was exactly the encoding of the first transition. Therefore the lemma holds in the case of the (Parent)-rule.

(Child) We do the same for the transition rule for *child i*:

$$\text{(Child-i)}\quad \frac{}{[\hat{o}(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_n}.\hat{a}_n)] \overset{child\ i}{\Rightarrow} o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_i}.[\hat{a_i}];\ldots;\overrightarrow{x_n}.\hat{a}_n)}$$

We can show that the lemma will also hold for the encoding of using the *child i* rule in the following way. The left-hand side encoded would be:

$$\begin{aligned}&[\![child\ i]\!][\![[o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_n}.\hat{a}_n)]]\!]\\ &=\ right_1\ \ldots\ right_{i-1}\ down\ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])]\end{aligned} \tag{19}$$

The encoding of the right-hand side would be:

$$\begin{aligned}&[\![o(\overrightarrow{x_1}.\hat{a}_1;\ldots;\overrightarrow{x_i}.[\hat{a_i}];\ldots;\overrightarrow{x_n}.\hat{a}_n)]\!]\\ &= o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\end{aligned} \tag{20}$$

We will now show that by β-reduction, the first encoding will reach the second encoding:

$$\begin{aligned}&right_1\ \ldots\ right_{i-1}\ down\ \big([o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[\![a_i]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])]\big)\\ &=\ right_1\ \ldots\ right_{i-1}\ \Big(\lambda x:s.\text{match}\ x\ [o\ (.a_1)\ldots(.a_n)]\\ &\qquad\to o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ldots(\lambda\overrightarrow{x_i}.a_i)\ldots(\lambda\overrightarrow{x_n}.a_n)\Big)\\ &\qquad\big([o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[\![a_i]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])]\big)\\ &\to_\beta\ right_1\ \ldots\ right_{i-1}\ \Big(\text{match}\ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])]\\ &\qquad[o\ (.a_1)\ldots(.a_n)]\to o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ldots(\lambda\overrightarrow{x_n}.a_n)\Big)\\ &\to_\beta\ right_1\ \ldots\ right_{i-1}\ \big(o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\big)\end{aligned}$$

From here, we will only show one step of the i applications of the *right*-function, as the others would intuitively follow:

$$\begin{aligned}&right_1\ \ldots\ right_{i-1}\ \big(o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ (\lambda\overrightarrow{x_2}.[\![a_2]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\big)\\ &=\ right_1\ \ldots\ right_{i-2}\ \Big(\lambda x:s.\text{match}\ x\\ &\qquad o\ (.[a_1])\ \ldots(.a_2)\ \ldots(.a_n)\to\big(o\ (\lambda\overrightarrow{x_1}.a_1)\ (\lambda\overrightarrow{x_2}.[a_2])\ldots(\lambda\overrightarrow{x_n}.a_n)\big)\Big)\\ &\qquad\big(o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ (\lambda\overrightarrow{x_2}.[\![a_2]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\big)\\ &\to_\beta\ right_1\ \ldots\ right_{i-2}\ \Big(\text{match}\ o\ (\lambda\overrightarrow{x_1}.[[\![a_1]\!]])\ (\lambda\overrightarrow{x_2}.[\![a_2]\!])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\\ &\qquad o\ (.[a_1])\ \ldots(.a_2)\ \ldots(.a_n)\to o\ (\lambda\overrightarrow{x_1}.a_1)\ (\lambda\overrightarrow{x_2}.[a_2])\ldots(\lambda\overrightarrow{x_n}.a_n)\Big)\\ &\to_\beta right_1\ \ldots\ right_{i-2}\ \big(o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ (\lambda\overrightarrow{x_2}.[[\![a_2]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!])\big)\end{aligned}$$

Applying the *right* function the remaining $i-2$ times, would then yield:

$$o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\ldots(\lambda\overrightarrow{x_i}.[[\![a_i]\!]])\ldots(\lambda\overrightarrow{x_n}.[\![a_n]\!]) \tag{21}$$

Which was exactly the encoding of the right hand side. Therefore the lemma also holds for the case of the *child* i-rule.

(Insert-op) We will do the same for inserting operators:

$$\text{(Insert)}\ \frac{}{[\hat{a}] \overset{\{o\}}{\Rightarrow} [o(\overrightarrow{x_1}.(\!|\,|\!)_{s_1}; \ldots; \overrightarrow{x_n}.(\!|\,|\!)_{s_n})]}\ \hat{a} \in \mathcal{B}[\mathcal{X}]_s, \text{ where } s \text{ is the sort of } o$$

We encode the left-hand side of the (Insert)-rule:

$$[\![insert\ o]\!][\![a]\!] = set[\![o]\!][\![a]\!] \tag{22}$$

And we encode the right-hand side of the transition:

$$\begin{aligned}[\![[o(\overrightarrow{x_1}.(\!|\,|\!)_{s_1}; \ldots; \overrightarrow{x_n}.(\!|\,|\!)_{s_n})]]\!] &= [o\ (\lambda\overrightarrow{x_1}.[\![(\!|\,|\!)_{s_1}]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![(\!|\,|\!)_{s_n}]\!])] \\ &= [o\ (\lambda\overrightarrow{x_1}.(\!|\,|\!)_{s_1}) \ldots (\lambda\overrightarrow{x_n}.(\!|\,|\!)_{s_n})]\end{aligned} \tag{23}$$

We will now reduce the left-hand side encoding, so it reaches the encoding of the right hand side.

$$\begin{aligned}&set\ [\![o]\!]\ [[\![a]\!]] \\ &= (\lambda a : s.\lambda x : s.\text{match } x\ ([a']) \to [a])\ (o\ (\lambda\overrightarrow{x_1}.(\!|\,|\!)_{s_1}) \ldots (\lambda\overrightarrow{x_n}.(\!|\,|\!)_{s_n}))\ [[\![a]\!]] \\ &\to_\beta (\lambda x : s.\text{match } x\ ([a']) \to [o\ (\lambda\overrightarrow{x_1}.(\!|\,|\!)_{s_1}) \ldots (\lambda\overrightarrow{x_n}.(\!|\,|\!)_{s_n})])\ [[\![a]\!]] \\ &\to_\beta (\text{match } [[\![a]\!]]\ ([a']) \to [o\ (\lambda\overrightarrow{x_1}.(\!|\,|\!)_{s_1}) \ldots (\lambda\overrightarrow{x_n}.(\!|\,|\!)_{s_n})]) \\ &\to_\beta [o\ (\lambda\overrightarrow{x_1}.(\!|\,|\!)_{s_1}) \ldots (\lambda\overrightarrow{x_n}.(\!|\,|\!)_{s_n})]\end{aligned}$$

This is exactly the same as the encoding of the right hand side. Furthermore, because of the type of the *set*-function being "$Set[s] : s \to s \to s$", we also have the side condition from the transition rule, that the inserted operator, and the replaced tree, should be of the same sort, as inserting the wrong sort would be a type error. With this, we have proven that the lemma holds for all allowed atomic prefix commands.

B Soundness proof of modal logic

Lemma 2 (Modal logic encoding). For every tree a and every ϕ, if in the editor calculus we have:

$$a \models \phi \tag{24}$$

then we can reduce the encoding such that

$$[\![\phi]\!]\ [\![a]\!] \to_\beta^* \top \tag{25}$$

Correspondingly, if we have that

$$a \not\models \phi \tag{26}$$

then we can reduce the encoding such that

$$[\![\phi]\!]\ [\![a]\!] \rightarrow_\beta^* \bot \tag{27}$$

Proof. To prove that lemma 2 holds, we need to show that for each operator-rule in the modal logic, and for the boolean operators, the lemma holds. To do this, we will use induction on the height of the deriviation-trees of the jugdements $a \models \phi$. We will begin with the base-cases, and show that the lemma holds for each of the modal logic operators. Then we will prove the lemma also holds for the propositional connectives negation, conjunction and disjunction. We will begin with the @o-operator:

(At-op) To prove that lemma 2 holds for the (At-op)-rule, we will encode the operator, and show that when the tree satisfies the judgement, the encoding of the tree and the @o operator evaluates to $\top$. The rule is:

$$\text{(At-Op)} \qquad \frac{}{[o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)] \models @o}$$

We show that the encoding evaluates to $\top$, if the operator is the given operator, and $\bot$ if it is not. The encoding of the operator in the tree and the @o-operator are:

$$\begin{aligned}
&[\![@o]\!][\![[o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]]\!] \\
&= (\lambda x : s.\text{match } x\ [o\ _\ \ldots\ _] \rightarrow \top \mid _ \rightarrow \bot)\ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
&\rightarrow_\beta (\text{match } [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])]\ [o\ _\ \ldots\ _] \rightarrow \top \mid _ \rightarrow \bot) \\
&\rightarrow_\beta \top
\end{aligned}$$

We see that it evaluates to $\top$, and therefore the lemmas first part holds for the (At-Op)-rule. We will show it returns false, when the operator is a different operator, o':

$$\begin{aligned}
&[\![@o]\!][\![[o'(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_n}.\hat{a}_n)]]\!] \\
&= (\lambda x : s.\text{match } x\ [o\ _\ \ldots\ _] \rightarrow \top \mid _ \rightarrow \bot)\ [o'\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
&\rightarrow_\beta (\text{match } [o'\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])]\ [o\ _\ \ldots\ _] \rightarrow \top \mid _ \rightarrow \bot) \\
&\rightarrow_\beta \bot
\end{aligned}$$

Therefore, the second part of the lemma holds for the @o-operator, and therefore, the whole lemma holds for the (At-op)-rule.

(Possibly) We will now prove that the lemma holds for the $\Diamond o$-operator. The rules for this operator are:

$$\text{(Possibly-trivial)}\ \frac{\hat{a} \models @o}{\hat{a} \models \Diamond o}$$

$$\text{(Possibly-i)}\ \frac{\hat{a}_i \models \Diamond o}{o(\overrightarrow{x_1}.\hat{a}_1; \ldots; \overrightarrow{x_i}.\hat{a}_i; \ldots; \overrightarrow{x_n}.\hat{a}_n) \models \Diamond o}$$

To prove that lemma 2 holds for $\Diamond o$-operator, we will first encode the tree and the operator:

$$
\begin{aligned}
[\![\Diamond o]\!][\![[a]]\!] = \ & \mathit{fix}(\lambda f : (s \to \mathit{Bool}).\lambda x : s.\text{match } x \\
& \quad \mid [o \ \_ \ \ldots \ \_] \to \top \\
& \quad \mid \_ \ (.a_1) \ldots (.a_n) \to \mathit{or}(\ldots(\mathit{or}\ (f\ [a_1])\ (f\ [a_2]))\ \ldots)\ (f\ [a_n]) \\
& \quad \mid \_ \to \bot)\ [\![[a]]\!] \\
\to_\beta \ & (\lambda x : s.\text{match } x \\
& \quad \mid [o \ \_ \ \ldots \ \_] \to \top \\
& \quad \mid \_ \ (.a_1) \ldots (.a_n) \to \mathit{or}(\ldots(\mathit{or}\ (\mathit{fix}(\ldots)\ [a_1])\ (\mathit{fix}(\ldots)\ [a_2]))\ \ldots) \\
& \quad (\mathit{fix}(\ldots)\ [a_n]) \\
& \quad \mid \_ \to \bot)\ [\![[a]]\!] \\
\to_\beta \ & \text{match } [\![[a]]\!] \\
& \quad \mid [o \ \_ \ \ldots \ \_] \to \top \\
& \quad \mid \_ \ (.a_1) \ldots (.a_n) \to \mathit{or}(\ldots(\mathit{or}\ (\mathit{fix}(\ldots)\ [a_1])\ (\mathit{fix}(\ldots)\ [a_2]))\ \ldots) \\
& \quad (\mathit{fix}(\ldots)\ [a_n]) \\
& \quad \mid \_ \to \bot
\end{aligned}
$$

To prove the lemmas first part holds for the (Possibly-trivial) rule, we use the insight from the previous rule, that if $[\hat{a}] \models @\mathrm{o}$, then $[\![[\hat{a}]]\!]$ will match on $[o \ \_ \ldots \_]$. Therefore we have the following reduction, when the precondition holds:

$$
\begin{aligned}
& \text{match } [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \ldots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
& \quad \mid [o \ \_ \ \ldots \ \_] \to \top \\
& \quad \mid \_ \ (.a_1) \ldots (.a_n) \to \mathit{or}(\ldots(\mathit{or}\ (\mathit{fix}(\ldots)\ [a_1])\ (\mathit{fix}(\ldots)\ [a_2]))\ \ldots) \\
& \quad (\mathit{fix}(\ldots)\ [a_n]) \\
& \quad \mid \_ \to \bot \\
& \to_\beta \top
\end{aligned}
$$

So the lemmas first part holds for the trivial rule.

For the (Possibly-i)-rule, we need to use induction on the height of the derivation tree of the judgement. We have shown the base case holds, when the cursor is at the operator. Now we show that given a tree $[a_i]$ of height h, and given that $[a_i] \models \Diamond o$, then the parent of this tree, having height $h + 1$, will also $\models \Diamond o$. This means we will assume that the lemma holds for $[a_i] \models \Diamond o$.

This means that $fix(\dots)[a_i]$, can evaluate to $\top$:

$$
\begin{array}{ll}
& \text{match } [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!])\dots(\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
& \quad \mid [o\ \_\ \dots\ \_] \to \top \\
& \quad \mid \_\ (.a_1)\dots(.a_n) \to or(\dots(or\ (\mathit{fix}(\dots)\ [a_1])\ (\mathit{fix}(\dots)\ [a_2])) \\
& \qquad \dots\ (\mathit{fix}(\dots)\ [a_i]))\dots)\ (\mathit{fix}(\dots)\ [a_n]) \\
& \quad \mid \_ \to \bot \\
\to_\beta & \ or(\dots(or\ (\mathit{fix}(\dots)\ [a_1])\ (\mathit{fix}(\dots)\ [a_2])) \\
& \qquad \dots(\mathit{fix}(\dots)\ [a_i]))\dots)\ (\mathit{fix}(\dots)\ [a_n]) \\
\to_\beta & \ or(\dots(or\ (\mathit{fix}(\dots)\ [a_1])\ (\mathit{fix}(\dots)\ [a_2]))\ \dots\top)\dots)\ (\mathit{fix}(\dots)\ [a_n]) \\
\to_\beta & \ \top
\end{array}
$$

The *or*-row will evaluate to $\top$, so long as a single of the arguments evaluate to $\top$. Now we have shown that given a tree a_i with height h, that satisfy the lemmas first part, then the parent of that tree, a new tree with length $h+1$ also satisfies the lemmas first part. Therefore, through induction, we know that the lemmas first part holds in both cases of the $\Diamond o$-operators rules. If none of the children, called a_i, have that $[a_i] \models \Diamond o$, then the whole expression would evaluate to $\bot$, as the whole row of *or*'s would consist of $\bot$, and therefore the second part of the lemma would be true. If it is an operator with no child nodes, and it does not satisfy $@o$, then it will also evaluate to $\bot$, as it would match on the last pattern. Therefore, both rules of $\Diamond o$ satisfy the lemma.

(Necessity) We will now prove that the lemma holds for the rule of $\Box o$. The proof will be very similar to the proof of the $\Diamond o$-rules. We only have one rule for $\Box o$:

$$\text{(Necessity)}\ \frac{[\hat{a}_1] \models \Diamond o\ \dots\ [\hat{a}_n] \models \Diamond o}{[o(\overrightarrow{x_1}.\hat{a}_1;\dots;\overrightarrow{x_n}.\hat{a}_n)] \models \Box o}$$

We will show the encoding of the operator and the tree:

$$
\begin{array}{l}
[\![\Box o_1]\!]\ [\![[a]]\!] = \big(\lambda x : s.\text{match } x \\
\quad \mid \_\ (.a_1)\dots(.a_n) \to and(\dots(and([\![\Diamond o]\!]\ a_1)([\![\Diamond o]\!]\ a_2))\dots)\big)\ ([\![\Diamond o]\!]\ a_n) \\
\quad \mid \_ \to \top)\ [\![([a])]\!]
\end{array}
$$

As we have shown, when $[a] \models \Diamond o$, then $[\![\Diamond o]\!]\ [\![[a]]\!]$ evaluates to $\top$. We now use the premise of the rule for $\Box o$, that says that for each child a_i, $[a_i] \models \Diamond o$. This we will use in our reduction, as we then can evaluate all $[\![\Diamond o]\!]\ [\![[a]]\!]$ to $\top$. Furthermore, we will use the fact that the encoding of $[\![[a]]\!] = [[\![a]\!]]$. This

means we can say that for each a_i, we have that $[\![\Diamond o]\!]\ [[\![a_i]\!]]$ evaluates to $\top$:

$$
\begin{aligned}
&(\lambda x : s.\text{match } x \\
&\ \_\ (.a_1) \dots (.a_n) \to and(\dots(and([\![\Diamond o]\!]\ a_1)([\![\Diamond o]\!]\ a_2))\dots)\ ([\![\Diamond o]\!]\ a_n) \\
&|\_ \to \top)\ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \dots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
=&(\text{match}\ \ [o\ (\lambda\overrightarrow{x_1}.[\![a_1]\!]) \dots (\lambda\overrightarrow{x_n}.[\![a_n]\!])] \\
&\ \_\ (.a_1) \dots (.a_n) \to and(\dots(and([\![\Diamond o]\!]\ a_1)([\![\Diamond o]\!]\ a_2))\dots)\ ([\![\Diamond o]\!]\ a_n) \\
&|\ \_ \to \top) \\
=&and(\dots(and([\![\Diamond o]\!]\ [[\![a_1]\!]])([\![\Diamond o]\!]\ [[\![a_2]\!]]))\dots)\ ([\![\Diamond o]\!]\ [[\![a_n]\!]]) \\
=&and(\dots(and(\top)(\top))\dots)\ (\top) \\
=&\top
\end{aligned}
$$

Therefore, the (Necessity)-rule satisfy the first part of the lemma. The second part can be seen from the fact that, if there ever is one of the diamonds that become $\bot$, then the whole row of *and* would evaluate to $\bot$. If it does not satisfy the match, meaning the operator has no children, it returns true, which is also the case for the transition rule.

(Negation) We will now prove that the theorem also holds for the negation operator:

$$\text{(Negation)}\ \ \frac{[\hat{a}] \nvDash \phi}{[\hat{a}] \models \neg\phi}$$

We will encode the negation of ϕ:

$$
\begin{aligned}
[\![\neg\phi]\!][[\![a]\!]] &= \mathit{bluebird}\ \mathit{neg}\ [\![\phi]\!] \\
&=(\lambda b : \mathit{Bool}.\text{match } (b)\ \top \to \bot \mid \bot \to \top)\ ([\![\phi]\!]\ [[\![a]\!]])
\end{aligned}
$$

We will now show that it reduces to the opposite. We will use induction on the height of the derivation tree of the judgement $[\hat{a}] \models \phi$. We will show that if the lemma holds for every ϕ with a derivation tree of height h, then when we apply the negation operator to it, the lemma still holds for the new tree with height $h+1$. We begin with the case where $[\hat{a}] \models \phi$, meaning we from the lemma know that $[\![\phi]\!]\ [[\![a]\!]]$ will evaluate to $\top$. This will be the reverse of the (Negation)-rule, and show that the lemmas second part holds for the (Negation)-rule:

$$
\begin{aligned}
&(\lambda b : \mathit{Bool}.\text{match } (b)\ \top \to \bot \mid \bot \to \top)\ ([\![\phi]\!]\ [[\![a]\!]]) \\
=&(\lambda b : \mathit{Bool}.\text{match } (b)\ \top \to \bot \mid \bot \to \top)\top \\
=&(\text{match } (\top)\ \top \to \bot \mid \bot \to \top) \\
=&\bot
\end{aligned}
$$

This shows that it upholds the second part of the lemma. We now show it for case where $[\hat{a}] \not\models \phi$, where we then must have that $[\![\phi]\!]\ [\![[a]]\!]$ evaluates to $\bot$:

$$\begin{aligned}
&(\lambda b : \mathit{Bool}.\text{match } (b)\ \top \to \bot \mid \bot \to \top)\ [\![\phi]\!]\ [\![[a]]\!] \\
=&(\lambda b : \mathit{Bool}.\text{match } (b)\ \top \to \bot \mid \bot \to \top)\bot \\
=&(\text{match } (\bot)\ \top \to \bot \mid \bot \to \top) \\
=&\top
\end{aligned}$$

With this, we have shown that for both parts of the lemma, when given a ϕ with a derivation tree of the judgement with height h, where the lemma holds, then it also holds for $\neg\phi$, having a derivation tree of the judgement with height $h + 1$. Therefore, we have inductively shown that the lemma holds for the negation-operator.

(Conjunction) We will now show the rule of conjunction is sound:

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

We encode the formula:

$$\begin{aligned}
[\![\phi_1 \wedge \phi_2]\!]\ [\![[a]]\!] = &\ \mathit{phoenix\ and}\ [\![\phi_1]\!]\ [\![\phi_2]\!]\ [\![[a]]\!] \\
= &\ (\lambda b_1 : \mathit{Bool}.\lambda b_2 : \mathit{Bool}.\text{match } (b_1, b_2) \\
&\quad \mid (\top, \top) \to \top \\
&\quad \mid (_, _) \to \bot)([\![\phi_1]\!][\![[a]]\!])([\![\phi_2]\!][\![[a]]\!])
\end{aligned}$$

Now we will prove that the lemma holds for the (Conjunction)-rule, using induction. We assume that the lemma holds for derivation trees of judgements with height of maximum h, where h is the height of the highest derivation trees of ϕ_1 and ϕ_2. Then we will show that the lemma also holds for the conjunction operator, which will bring the height of the derivation trees to $h + 1$. We use the lemma to say that $[\![\phi_1]\!]\ [\![[a]]\!]$ and $[\![\phi_2]\!]\ [\![[a]]\!]$ will evaluate to either $\top$ or $\bot$. We will begin with the case of the rule, where $[a] \models \phi_1, \phi_2$,

and we then know that they both will evaluate to $\top$:

$$
\begin{aligned}
&(\lambda b_1 : Bool.\lambda b_2 : Bool.\text{match } (b_1, b_2) \\
&\quad (\top, \top) \to \top \\
&\quad \mid (\_, \_) \to \bot)([\![\phi_1]\!]\ encode[a])([\![\phi_2]\!]\ encode[a]) \\
=&(\lambda b_1 : Bool.\lambda b_2 : Bool.\text{match } (b_1, b_2) \\
&\quad (\top, \top) \to \top \\
&\quad \mid (\_, \_) \to \bot)\top\ \top \\
=&(\text{match } (\top, \top) \\
&\quad (\top, \top) \to \top \\
&\quad \mid (\_, \_) \to \bot) \\
=&\top
\end{aligned}
$$

So, the first part of the lemma holds for the conjunctive operator, given that the lemma holds for trees of height h. We can easily see that any other combinations of values would evaluate to $\bot$, and therefore the lemma's second part also hold for the conjunctive operator. We now have shown that given the lemma holds for derivation trees of judgements with height h, it also holds for derivation trees of judgements with height $h+1$, if the last operator is a conjunction operator, meaning the lemma holds for the conjunction operator.

(Disjunction) Now we show the same for the disjunction-operator:

$$\text{(Disjunction-1)}\ \frac{\hat{a} \models \phi_1}{\hat{a} \models \phi_1 \vee \phi_2} \qquad \text{(Disjunction-2)}\ \frac{\hat{a} \models \phi_2}{\hat{a} \models \phi_1 \vee \phi_2}$$

We encode the formula, and then we will show that it only returns true, when either of the ϕ's applied to the tree, evaluate to $\top$. We use the same induction method as before:

$$
\begin{aligned}
[\![\phi_1 \vee \phi_2]\!]\ [\![[a]]\!] &= \textit{phoenix or } [\![\phi_1]\!]\ [\![\phi_2]\!]\ [\![[a]]\!] \\
&= (\lambda b_1 : Bool.\lambda b_2 : Bool.\text{match } (b_1, b_2) \\
&\quad (\bot, \bot) \to \bot \\
&\quad \mid (\_, \_) \to \top)([\![\phi_1]\!]\ [\![[a]]\!])\ ([\![\phi_2]\!]\ [\![[a]]\!])
\end{aligned}
$$

Now we will prove that the lemma holds for the disjunction-operator, using induction. We assume that the lemma holds for derivation trees of judgements with height of maximum h, where h is the height of the highest derivation tree of ϕ_1 and ϕ_2. Then we will show that the lemma also holds for the disjunction operator, which will bring the height of the derivation tree to $h+1$. We use the lemma to say that $[\![\phi_1]\!]\ [\![[a]]\!]$ and $[\![\phi_2]\!]\ [\![[a]]\!]$ will evaluate to either $\top$ or $\bot$. We will only show the case where neither of them evaluate to $\top$, as that is the only case where the disjunction should return $\bot$, if the

lemma holds:

$$
\begin{aligned}
&(\lambda b_1 : Bool.\lambda b_2 : Bool.\text{match } (b_1, b_2)\ (\bot, \bot) \to \bot \\
&\mid (\_, \_) \to \top)([\![\phi_1]\!]\ [\![[a]]\!])\ ([\![\phi_2]\!]\ [\![[a]]\!]) \\
=&(\lambda b_1 : Bool.\lambda b_2 : Bool.\text{match } (b_1, b_2) \\
&(\bot, \bot) \to \bot \\
&\mid (\_, \_) \to \top)\bot\ \bot \\
=&(\text{match } (\bot, \bot) \\
&(\bot, \bot) \to \bot \\
&\mid (\_, \_) \to \top) \\
=&\bot
\end{aligned}
$$

Therefore, the second part of the lemma holds for the conjunction-operator. It can easily be seen that if either $[\![\phi_1]\!]\ [\![[a]]\!]$ or $[\![\phi_2]\!]\ [\![[a]]\!]$ had evaluated to $\top$, the conjunction operator would also evaluate to $\bot$, as the pair would not have matched on $(\bot, \bot)$. Therefore, we have shown that when the lemma holds for derivation trees with height up to h, then it also holds for trees with height $h + 1$, given the last operator is a disjunction.
As we have now shown that the lemma holds for all base cases, and that if it holds for derivation trees of height h, then it also holds for trees of height $h + 1$, as we have proven it holds no matter what operator the last one to increase the height is, the lemma is therefore proven to hold in all cases.

C Soundness proof of editor expressions

Theorem 1 (Soundness of editor expressions). For every editor expression, if we have the transition:

$$\langle E, a\rangle \to \langle E', a'\rangle \tag{28}$$

Then the encoding will also be able to do the following β-reduction:

$$[\![\langle E, a\rangle]\!] \to^*_\beta [\![\langle E', a'\rangle]\!] \tag{29}$$

Proof. To prove theorem 1, we will use induction on the height of the derivation tree, and make use of lemma 1 and lemma 2. There is only one rule in the editor calculus, the (Seq)-rule, that is not an axiom but has as premise that a transition for another editor expression is possible. Therefore, we first prove that the theorem holds for all axioms, as these rules does not need an inductive hypothesis. Finally, we show that the theorem holds for the rule (Seq), provided the theorem holds for the derivation tree of the premise.

(Conditional) We now show that the 2 conditional rules are sound.

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

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

The encoding of the left-hand side of both conditional rules are:

$$\begin{aligned}[\![\langle \phi \Rightarrow E_1 | E_2, C[a] \rangle]\!] = \big(&\lambda C' : Ctx.\text{match } ([\![\phi]\!]\ C'.1) \\ &\mid \top \to [\![E_1]\!]\ C' \\ &\mid \bot \to [\![E_2]\!]\ C'\big)\ ([\![a]\!], [\![C]\!])\end{aligned}$$

The encoding of the right-hand side in the first rule is:

$$[\![\langle E_1, C[a] \rangle]\!] = [\![E_1]\!][\![C[a]]\!] = [\![E_1]\!]([\![a]\!], [\![C]\!]) \tag{30}$$

The encoding of the right-hand side in the second rule is:

$$[\![\langle E_2, C[a] \rangle]\!] = [\![E_2]\!][\![C[a]]\!] = [\![E_2]\!]([\![a]\!], [\![C]\!]) \tag{31}$$

We will look at the two cases of the conditional-rule. The first is where $a \models \phi$. From lemma 2 we know that if that is the case, then $[\![\phi]\!]\ [\![a]\!]$ reduces to $\top$. We will use this to prove that the first rule is soundly encoded:

$$\begin{aligned}[\![\langle \phi \Rightarrow E_1 | E_2, C[a] \rangle]\!] = \big(&\lambda C' : Ctx.\text{match } ([\![\phi]\!]\ C'.1) \\ &\quad \mid \top \to [\![E_1]\!]\ C' \\ &\quad \mid \bot \to [\![E_2]\!]\ C'\big)\ ([\![a]\!], [\![C]\!]) \\ \to_\beta\ &\text{match } ([\![\phi]\!]\ \ ([\![a]\!], [\![C]\!]).1) \\ &\quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\ &\quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\ \to_\beta\ &\text{match } ([\![\phi]\!]\ [\![a]\!]) \\ &\quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\ &\quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\ \to_\beta^*\ &\text{match } (\top) \\ &\quad \mid \top \to [\![E_1]\!]\ \ ([\![a]\!], [\![C]\!]) \\ &\quad \mid \bot \to [\![E_2]\!]\ \ ([\![a]\!], [\![C]\!]) \\ \to_\beta\ &[\![E_1]\!]\ \ ([\![a]\!], [\![C]\!])\end{aligned}$$

This is exactly the same as the right-hand side of the first rule, and therefore, we can say that the (Cond-1)-rule is soundly encoded. We now look at the case where we have that $a \not\models \phi$. From lemma 2 we know that this means that $[\![\phi]\!]\ [\![a]\!]$ reduces to $\bot$. Therefore, the reduction of the encoding in this case

will be:

$$\begin{aligned}
&\text{match } (\llbracket\phi\rrbracket\ \llbracket a\rrbracket) \\
&\quad \mid \top \to \llbracket E_1\rrbracket\ \ (\llbracket a\rrbracket, \llbracket C\rrbracket) \\
&\quad \mid \bot \to \llbracket E_2\rrbracket\ \ (\llbracket a\rrbracket, \llbracket C\rrbracket) \\
&\to^*_\beta \text{match } (\bot) \\
&\quad \mid \top \to \llbracket E_1\rrbracket\ \ (\llbracket a\rrbracket, \llbracket C\rrbracket) \\
&\quad \mid \bot \to \llbracket E_2\rrbracket\ \ (\llbracket a\rrbracket, \llbracket C\rrbracket) \\
&\to_\beta \llbracket E_2\rrbracket\ \ (\llbracket a\rrbracket, \llbracket C\rrbracket)
\end{aligned}$$

This is exactly the right-hand side of the (Cond-2)-rule encoded, and therefore the second rule is also soundly encoded. As this second rule is also soundly encoded, we have proven that the conditional statement is now soundly encoded.

(Recursion) We move on to prove the soundness of recursion.

$$\text{(Recursion)}\ \ \frac{}{\langle \text{rec } x.E, C[a]\rangle \overset{\epsilon}{\Rightarrow} \langle E[x := rec\ x.E], C[a]\rangle}$$

To prove the soundness of recursion, we need to do two cases. The one where there in the editor expression E is one or more occurrences of x, and one where there is no x in E. We will begin with the case of the editor expressions with one or more x. That means, $E = \ldots x \ldots$ The left-hand side encoded is:

$$\begin{aligned}
&\llbracket\langle \text{rec } x.(\ldots x \ldots) C[a]\rangle\rrbracket = \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket \ldots x \ldots \rrbracket)\ \llbracket C[a]\rrbracket \\
&\to_\beta (\lambda x : (Ctx \to Ctx).\llbracket\ldots\rrbracket\ x\ \llbracket\ldots\rrbracket)\ \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket \ldots x \ldots)\rrbracket)\ \llbracket C[a]\rrbracket \\
&\to_\beta \llbracket\ldots\rrbracket\ \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket \ldots x \ldots)\rrbracket)\ \llbracket\ldots\rrbracket)\ \llbracket C[a]\rrbracket
\end{aligned} \tag{32}$$

The right-hand side encoded is:

$$\begin{aligned}
&\llbracket(\ldots x \ldots)[x := rec\ x.(\ldots x \ldots)]\rrbracket\llbracket C[a]\rrbracket = \llbracket\ldots\rrbracket\llbracket rec\ x.(\ldots x \ldots)\rrbracket\llbracket\ldots\rrbracket(\llbracket[a]\rrbracket, \llbracket C\rrbracket) \\
&= \llbracket\ldots\rrbracket\ \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket \ldots x \ldots \rrbracket)\ \llbracket\ldots\rrbracket)\ \llbracket C[a]\rrbracket
\end{aligned} \tag{33}$$

As we can see, the left hand side can evaluate to the right hand side. We will now look at the case with no occurrences of x in E. In that case we can remove the substitution on the right hand side, and we use the fact that the x is not present in E, so that when a value is substituted into x, it simply removes the λx. The left-hand side encoded is in this case:

$$\begin{aligned}
&\llbracket\text{rec } x.E\rrbracket\llbracket C[a]\rrbracket = \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket E\rrbracket)\ \llbracket C[a]\rrbracket \\
&\to_\beta (\lambda x : (Ctx \to Ctx).\llbracket E\rrbracket)\ \mathit{fix}(\lambda x : (Ctx \to Ctx).\llbracket \ldots x \ldots)\rrbracket)\ \llbracket C[a]\rrbracket \\
&\to_\beta \llbracket E\rrbracket\ \llbracket C[a]\rrbracket
\end{aligned} \tag{34}$$

The right-hand side encoded is:

$$[\![E[x := rec\ x.E]]\!][\![C[a]]\!] = [\![E]\!][\![C[a]]\!] \tag{35}$$

As they are exactly the same, we can show that the theorem holds for the (Recursion)-rule, no matter if the recursive variable is present in the expression or not, and therefore, the theorem holds for the whole (Recursion)-rule.

(Context) We will now prove the soundness of the context rule.

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

To prove the soundness of the context rule, we show that the encoding of the left-hand side can reach the right-hand side. The left-hand side encoded would be:

$$[\![\langle \pi.E, C[a] \rangle]\!] = \lambda C : Ctx.([\![E]\!](([\![\pi]\!]\ C.1), C.2)\ ([\![a]\!], [\![C]\!]))) \tag{36}$$

The encoding of the right-hand side would be:

$$[\![\langle E, C[a'] \rangle]\!] = [\![E]\!]\ ([\![a']\!], [\![C]\!]) \tag{37}$$

To show the (Context)-rule is soundly encoded, we will reduce the left-hand side encoded to the right-hand side. We will use lemma 1, that states that if a transition $a \overset{\pi}{\Rightarrow} a'$ exists, then $[\![\pi]\!]\ [\![a]\!]$ can be reduced to $[\![a']\!]$. We now reduce the lefthand side:

$$\begin{aligned}[\![E]\!]\lambda C : Ctx.(([\![\pi]\!]\ C.1), C.2)\ ([\![a]\!], [\![C]\!]) &= [\![E]\!](([\![\pi]\!]\ [\![a]\!]), [\![C]\!]) \\ &\rightarrow_\beta [\![E]\!]([\![a']\!], [\![C]\!])\end{aligned}$$

This is exactly the right-hand side, and therefore the (Context)-rule can be encoded to our lambda calculus soundly. It should be noted, that this interpretation requires a reinterpretation of the context between operations, just like in the normal editor calculus between transitions.

(Seq-Trivial) We now go on to prove the soundness of the rule (Seq-Trivial).

$$\text{(Seq-Trivial)}\ \frac{}{\langle nil \ggg E_2, C[a] \rangle \overset{\epsilon}{\Rightarrow} \langle E_2, C[a] \rangle}$$

This rule can be proven to be soundly encoded, by reducing the left-hand side to the right-hand side. The encoding of the left-side is:

$$[\![\langle nil \ggg E_2, C[a] \rangle]\!] = [\![E_2]\!][\![nil]\!][\![C[a]]\!] \tag{38}$$

The encoding of the right-side is:

$$[\![\langle E_2, C[a] \rangle]\!] = [\![E_2]\!][\![C[a]]\!] \tag{39}$$

The encoding of nil is determined by the rules in fig. 36, where nil is encoded as the identity function, which reduces to nothing, and we end with the right-hand side.

$$\begin{aligned}[\![E_2]\!][\![nil]\!][\![C[a]]\!] &= [\![E_2]\!](\lambda C : Ctx.C)[\![C[a]]\!] \\ &= [\![E_2]\!][\![C[a]]\!]\end{aligned} \tag{40}$$

Therefore, the theorem holds for the (Seq-Trivial)-rule.

(Seq) We will now show that the rule (Seq) is sound:

$$\text{(Seq)} \quad \frac{\langle E_1, C[a]\rangle \overset{\alpha}{\Rightarrow} \langle E_1', C[a']\rangle}{\langle E_1 \ggg E_2, C[a]\rangle \overset{\alpha}{\Rightarrow} \langle E_1' \ggg E_2, C[a']\rangle}$$

We will use induction on the height of the transition tree to prove that this rule is soundly encoded. We will begin by encoding the left-hand side of the transition:

$$[\![\langle E_1 \ggg E_2, C[a]\rangle]\!] = (\lambda C' : Ctx.[\![E_2]\!]([\![E_1]\!]\ C'))\ [\![C[a]]\!] = [\![E_2]\!]([\![E_1]\!]\ [\![C[a]]\!]) \tag{41}$$

The encoding of the right-side would be:

$$[\![\langle E_1' \ggg E_2, C[a']\rangle]\!] = (\lambda C' : Ctx.[\![E_2]\!]([\![E_1']\!]C'))[\![C[a']]\!] = [\![E_2]\!]([\![E_1']\!][\![C[a']]\!]) \tag{42}$$

To show that the sequential rule is sound, we assume that theorem 1 holds for all transition trees of height h, where h is the height of the transition tree for $\langle E_1, C[a]\rangle \overset{\alpha}{\Rightarrow} \langle E_1', C[a']\rangle$. Then we will show that the theorem also holds for transition trees of height $h+1$, given the last transition is the sequential rule. We will use that the theorem states that if $\langle E_1, C[a]\rangle \overset{\alpha}{\Rightarrow} \langle E_1', C[a']\rangle$, then we also have that $[\![E_1, C[a]]\!]$ can be reduced to $[\![E_1', C[a']]\!]$. We now reduce the lefthand side:

$$[\![E_2]\!]([\![E_1]\!][\![C[a]]\!]) \rightarrow_\beta [\![E_2]\!]([\![E_1']\!][\![C[a']]\!]) \tag{43}$$

This was exactly the right-hand side, and therefore, we have shown that the (Sequential)-rule is encoded soundly.

With this, we have proven theorem 1, as we have shown for each axiom editor expression, the theorem holds, and that for the sequential rule, with the assumption that the theorem holds for all transition trees up to height h, where h is the heigth of the transition tree in the precondition, the theorem also holds for the sequential rule, bringing the height to $h+1$. Therefore, for all possible editor expressions, we can encode them soundly.