Editing Functional Programs Without Breaking Them

Edward Amsden Ryan Newton Jeremy Siek
Indiana University
{eamsden,rrnewton,jsiek}@indiana.edu

Abstract

We present a set of editing actions on terms in the simply-typed lambda calculus. These actions preserve the well-typedness of terms, and allow the derivation of any well-typed term beginning with any other well-typed term, without resorting to metavariables or other forms of placeholders. We are in the process of proving these properties, and we discuss how general-purpose programming might proceed given this set of editing actions.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Modern typed functional programming languages such as Haskell [11], OCaml [7], SML [13], Idris [2], and Agda [14] offer programmers extremely powerful and flexible type systems to ensure the correctness of their code. However, type systems are useful for far more than checking programs as programmers have already written them. In particular, a type system can be used to guide a term editor such that ill-typed terms are never produced in the first place.

Text editing and submission of the program to a compiler or interpreter for error checking and evaluation is the venerable and proven means of creating and maintaining programs. However, this paradigm has several disadvantages.

- It does not directly integrate semantic knowledge of program components (such as scope and types) into the editing system, requiring local parsing or linking with the language implementation to derive this information from the program text.
- It requires the programmer to reason about the well-typedness of programs either manually, or by trial-and-error by submitting them to a typechecker.
- It is unsuitable for more restrictive human-computer interaction platforms, such as touch-based mobile platforms, game consoles, and accessible interfaces.
- Most importantly, text editing actions do not relate directly to meaningful operations on programs. Insertion or deletion of text will often make a program syntactically invalid. It is likely to introduce scoping errors. It will almost certainly make the program ill-typed.

Languages such as Scratch [10], Kodu [9], and YinYang [12] are first steps toward addressing these issues, but do not offer a clear path toward leveraging the vast body of accomplished and ongoing research in programming languages. We propose instead to take a well-understood language, the lambda calculus, and show how to edit its terms directly.

In particular, we intend to use types to guide the editing operations on terms. Rather than entering programs as text, programmers will have at their disposal a set of actions for modifying, combining, and uncombining complete and well-typed lambda calculus terms. These actions are sufficient to arrive at any well-typed term from any other well-typed term, guarantee the well-typedness of derived terms, and do not require placeholders such as metavariables [14].

For this presentation, we consider the simply-typed lambda calculus. We expect this approach to generalize to polymorphic and dependently-typed calculi. We point out particular and interesting ways in which the application of this approach to more powerful type systems will differ from that presented here.

We make the following contributions:

- We describe a set of actions which allow programmers to construct and deconstruct terms in the simply-typed lambda calculus (Sections 2 and 3).
  - These actions operate only on lambda calculus terms. There are no holes or other placeholders.
- We sketch a proof that this set of actions is *sound*. That is, beginning with well-typed terms, only well-typed terms may be derived using these actions (Section 4.1).
- We sketch a proof that this set of actions is *complete*. That is, any well-typed term may be constructed via these actions, starting with any other well-typed term (Section 4.2).
- We describe how these actions support various general approaches to programming, specifically "top-down" programming (where the programmer begins building the top-level program, binding components for later construction) and "bottom-up" programming (where the programmer begins constructing small components and composes them into the top-level program) (Section 5).

2. Terms, Types and Paths

For this presentation, we consider terms (and associated types) in the simply-typed lambda calculus. We assume that terms and types in the simply-typed lambda calculus are familiar to our readers. However, we include them in our presentation for reference while reading our formulations of paths and actions and our proofs of "soundness" and "completeness" for our editing system. We use De Bruijn indices for this presentation, as this avoids issues of

$$
\begin{array}{rcl}
\text{Variables}(x) & ::= & \mathbb{N} \\
\text{Types}(t) & ::= & \texttt{unit} \mid t \to t \\
\text{Terms}(e) & ::= & x \mid \lambda : t.e \mid e\ e \mid \bullet \\
\text{Paths}(p) & ::= & \texttt{top} \mid p\ \texttt{lamty} \mid p\ \texttt{lambod} \\
 & \mid & p\ \texttt{apprator} \mid p\ \texttt{apprand} \\
 & \mid & p\ \texttt{typein} \mid p\ \texttt{typeout} \\
\text{Environments}(\Gamma) & ::= & \epsilon \mid \Gamma; t
\end{array}
$$

Figure 1. Grammars for terms, types, and paths.

$\boxed{\Gamma \vdash e : t}$

$$\textsc{UnitTy}\quad \Gamma \vdash \bullet : \texttt{unit}$$

$$\textsc{VariableSuccTy}\quad \frac{\Gamma \vdash x : t}{\Gamma; t_{\texttt{ext}} \vdash x + 1 : t}$$

$$\textsc{VariableZeroTy}\quad \Gamma; t \vdash 0 : t$$

$$\textsc{LamTy}\quad \frac{\Gamma; t_{\texttt{in}} \vdash e : t_{\texttt{out}}}{\Gamma \vdash \lambda : t_{\texttt{in}}.e : t_{\texttt{in}} \to t_{\texttt{out}}}$$

$$\textsc{AppTy}\quad \frac{\Gamma \vdash e_{\texttt{rator}} : t_{\texttt{in}} \to t_{\texttt{out}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{in}}}{\Gamma \vdash e_{\texttt{rator}}\ e_{\texttt{rand}} : t_{\texttt{out}}}$$

Figure 2. Typing rules for the Simply-Typed Lambda Calculus

naming in the consideration of actions on terms. The typing rules are given in Figure 2 as we refer to them for proofs of soundness and completeness of the editing actions later in the paper.

In order to designate which part, or subterm, of a term we wish to operate on, we define a notion of paths into terms. *Paths* are simply sequences which recursively describe which subterm of a particular term to pick out. The grammars for terms, types, and paths appear in Figure 1.

2.1 Variables

We require a few operations on variables and variables in terms in support of the definitions of our editing actions. In particular, we will need to compare variables to see if the scope of one variable is within the scope another. We will also need to adjust variables in order to maintain their binding structure as we add and remove bindings. This is done with the $\uparrow_c\ (e)$ and $\downarrow_c\ (e)$ operations. These operations are given in Figure 3. The presentation is due to Pierce [16, p. 79] with a few modifications.

The shift operation $\uparrow_c^d (e)$ given by Pierce is parameterized over the offset d as well as the cutoff c. We will only want to shift by an offset of 1, so d is fixed at 1 and we write $\uparrow_c (e)$, or simply $\uparrow (e)$ for the case where $c = 0$.

We introduce an unshift operation $\downarrow_c\ (e)$. This is a partial function, which is undefined exactly on variables matching the cutoff. Judgements with this function in their premises do not hold in cases where it is undefined. We write $\downarrow (e)$ for the case where $c = 1$. $\downarrow (e)$ is thus undefined on variables which would be bound to the nearest binder surrounding e. Since unshifting is used when replacing a binding which has no bound occurrences with the body of the binding, this is the expected behavior.

2.2 Paths

Paths are intended to mark the part of a term which is under consideration for a particular action. For the purpose of our presentation, we consider paths as separate entities from terms. Paths are described as sequences of atoms, each of which describes a choice of subterm. There are several relations on paths and terms employed

$$
\begin{array}{ll}
\uparrow_c (x) & = \begin{cases} x & x < c \\ x + 1 & x \geq c \end{cases} \\
\uparrow_c (\lambda : t.e) & = \lambda : t.\ \uparrow_{c+1} (e) \\
\uparrow_c (e_1\ e_2) & = \uparrow_c (e_1)\ \uparrow_c (e_2) \\
\uparrow (e) & = \uparrow_0 (e) \\
\downarrow_c (x) & = \begin{cases} x & x < c - 1 \\ x - 1 & x \geq c \end{cases} \\
\downarrow_c (\lambda : t.e) & = \lambda : t.\ \downarrow_{c+1} (e) \\
\downarrow_c (e_1\ e_2) & = \downarrow_c (e_1)\ \downarrow_c (e_2) \\
\downarrow (e) & = \downarrow_1 (e)
\end{array}
$$

Figure 3. Shifting ($\uparrow$) and unshifting ($\downarrow$) of De Bruijn indices. Adapted from the presentation by Pierce [16, p. 79].

in the definitions of the editing actions. These relations are defined in Figure 4.

The relation $p(e)$ extracts the subterm (expression or type) of e to which the path p points. Extraction of subterms is used to determine their type, as well to use them when constructing new terms by λ-abstraction or application.

The relation $\gamma(p, e)$ gives the typing environment at the subterm of e to which the path p points. This relation is used to determine what bindings are in scope at a particular point in support of the variable replacement operation. It is also used by the $\texttt{c}(p, e)$ operation when determining what types are valid at a path. Finally, it is employed in the definitions of editing actions to check that new terms do in fact meet their typing constraints.

The relation $\texttt{c}(p, e)$ gives the set of types which the subterm of e to which the path p points may match. This relation allows the definitions of editing actions to ensure that they do not make the term surrounding the subterm on which they operate ill-typed by changing the type of that subterm. For instance, if e_1 is applied to e_2, then λ-abstracting e_2 will yield a well-typed term derived from e_2, but the application will no longer be well-typed in the STLC.

The relation $e_{\texttt{full}}[e_{\texttt{sub}}/p]$ or $e_{\texttt{full}}[t/p]$ gives a new term in which $e_{\texttt{sub}}$ or t is substituted for the subterm of $e_{\texttt{full}}$ to which p points. This relation is used to define the operation of actions on subterms. In general, the $p(e)$ relation is used to extract a subterm, the subterm is suitably modified, and the modified term put back in its place by the $e_{\texttt{full}}[e_{\texttt{sub}}/p]$ relation.

The relation $\texttt{appable}(p)$ asserts that a path is suitable for constructing an application with. This is used to ensure that the application operation is not employed in subterms of applications. Were this allowed, it is not clear which of several possible outcomes of this operation would be the correct one. Further, disallowing this does not affect the soundness or completeness of the editing operations. However, allowing application under applications is almost certain to be a desirable feature in the implementation, so future work will describe a resolution of this ambiguity and lift the restriction on application operations under applications.

3. Editing Actions

The core of our contributions is a set of editing actions, which describe how to combine and manipulate lambda calculus terms in a way that maintains well-typedness. These actions are shown in Figure 5.

Not all actions are available at all paths into a subterm. Actions will usually change the type of a subterm, and may not do so in a way that would make the containing term ill-typed. This may seem an onerous restriction. However, we are able to show that any well-typed term may be reached from any other well-typed term using our restriction. Further, the mechanism of constraints which we use to judge whether a type-change will make the containing term ill-

$\boxed{p(e) = e_{\text{sub}}}$

TopPath
$\texttt{top}(e) = e$

LamTyPath
$$\frac{p(e) = \lambda : t_{\text{ty}}.e_{\text{bod}}}{p\ \texttt{lamty}(e) = t_{\text{ty}}}$$

LamBodPath
$$\frac{p(e) = \lambda : t_{\text{ty}}.e_{\text{bod}}}{p\ \texttt{lambod}(e) = e_{\text{bod}}}$$

AppRatorPath
$$\frac{p(e) = e_{\text{rator}}\ e_{\text{rand}}}{p\ \texttt{apprator}(e) = e_{\text{rator}}}$$

AppRandPath
$$\frac{p(e) = e_{\text{rator}}\ e_{\text{rand}}}{p\ \texttt{apprand}(e) = e_{\text{rand}}}$$

TypeInPath
$$\frac{p(e) = t_{\text{in}} \to t_{\text{out}}}{p\ \texttt{typein}(e) = t_{\text{in}}}$$

TypeOutPath
$$\frac{p(e) = t_{\text{in}} \to t_{\text{out}}}{p\ \texttt{typeout}(e) = t_{\text{out}}}$$

$\boxed{\gamma(p, e) = \Gamma}$

TopPathCtx
$\gamma(\texttt{top}, e) = \epsilon$

LamBodPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = \lambda : t.e_{\text{bod}}}{\gamma(p\ \texttt{lambod}, e) = \Gamma; t}$$

AppRatorPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}}}{\gamma(p\ \texttt{apprator}, e) = \Gamma}$$

AppRandPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}}}{\gamma(p\ \texttt{apprand}, e) = \Gamma}$$

$\boxed{\texttt{c}(p, e) = C}$

TopPathTypes
$\texttt{c}(\texttt{top}, e) = \mathcal{L}(t)$

LamTyPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad \gamma(p, e) = \Gamma p(e) = \lambda : t.e_{\text{bod}}}{\texttt{c}(p\ \texttt{lamty}, e) = \{t_{\text{in}} | \Gamma; t_{\text{in}} \vdash e_{\text{bod}} : t_{\text{out}} \wedge t_{\text{in}} \to t_{\text{out}} \in C\}}$$

LamBodPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = \lambda : t_{\text{in}}.e_{\text{bod}}}{\texttt{c}(p\ \texttt{lambod}, e) = \{t_{\text{out}} | t_{\text{in}} \to t_{\text{out}} \in C\}}$$

AppRatorPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}} \qquad \gamma(p, e) = \Gamma \qquad \Gamma \vdash e_{\text{rand}} : t_{\text{in}}}{\texttt{c}(p\ \texttt{apprator}, e) = \{t_{\text{in}} \to t_{\text{out}} | t_{\text{out}} \in C\}}$$

AppRandPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}} \qquad \gamma(p, e) = \Gamma \qquad \Gamma \vdash e_{\text{rator}} : t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{apprand}, e) = \{t_{\text{in}}\}}$$

TypeInPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{typein}, e) = \{t | t \to t_{\text{out}} \in C\}}$$

TypeOutPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{typeout}, e) = \{t | t_{\text{in}} \to t \in C\}}$$

$\boxed{e_{\texttt{full}}[e_{\text{sub}}/p] = e_{\text{new}}}$

TopPathSub
$e_{\texttt{full}}[e_{\text{sub}}/\texttt{top}] = e_{\text{sub}}$

LamTyPathSub
$$\frac{p(e_{\texttt{full}}) = \lambda : t.e \qquad e_{\texttt{full}}[\lambda : t_{\text{sub}}.e/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{lamty}] = e_{\text{new}}}$$

LamBodPathSub
$$\frac{p(e_{\texttt{full}}) = \lambda : t.e \qquad e_{\texttt{full}}[\lambda : t.e_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{lambod}] = e_{\text{new}}}$$

AppRatorPathSub
$$\frac{p(e_{\texttt{full}}) = e_{\text{rator}}\ e_{\text{rand}} \qquad e_{\texttt{full}}[e_{\text{sub}}\ e_{\text{rand}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{apprator}] = e_{\text{new}}}$$

AppRandPathSub
$$\frac{p(e_{\texttt{full}}) = e_{\text{rator}}\ e_{\text{rand}} \qquad e_{\texttt{full}}[e_{\text{rator}}\ e_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{apprand}] = e_{\text{new}}}$$

TypeInPathSub
$$\frac{p(e_{\texttt{full}}) = t_{\text{in}} \to t_{\text{out}} \qquad e_{\texttt{full}}[t_{\text{sub}} \to t_{\text{out}}/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{typein}] = e_{\text{new}}}$$

TypeOutPathSub
$$\frac{p(e_{\texttt{full}}) = t_{\text{in}} \to t_{\text{out}} \qquad e_{\texttt{full}}[t_{\text{in}} \to t_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{typeout}] = e_{\text{new}}}$$

$\boxed{\texttt{appable}(p)}$

TopPathAppable
$\texttt{appable}(\texttt{top})$

LamBodAppable
$$\frac{\texttt{appable}(p)}{\texttt{appable}(p\ \texttt{lambod})}$$

Figure 4. Definitions of path relations. $\mathcal{L}(t)$ denotes the language of the nonterminal t.

Actions(a) ::=

Usage	Action	Denoted By
Construction	λ-abstract	λ
	$\rightarrow$-abstract	$\rightarrow$
	Replace $p(e)$ with x	$\texttt{replace}_x$
	Replace $p(e)$ with $\bullet$	$\texttt{replace}_\bullet$
	Apply	`apply`
Destruction	Delete binding	`unbind`
	Unapply	`unapply`
Movement	Type of a lambda	`lamty`
	Body of a lambda	`lambod`
	Operator of an app	`apprator`
	Operand of an app	`apprand`
	Input of a type arrow	`tyin`
	Output of a type arrow	`tyout`
	Up	`up`

Action Sequences(s) ::= $\epsilon \mid s\ a(p,e) \mid s\ a(p,e,e)$

Figure 5. Editing Actions

typed will also allow us to describe exactly how circumscribed the set of actions is for any term, and see how these boundaries will be extended when our editing theory is extended to polymorphic calculi.

The language of actions is given by the non-terminal a. Editor states $\mathcal{E}$ are subsets of $\mathcal{L}(p) \times \mathcal{L}(e)$[1]. To define an action, we give a rule for one of three relations: $a(p,e) \rightsquigarrow \mathcal{E}$, $a(p,e_1,e_2) \rightsquigarrow \mathcal{E}$, or $a(p_1,e_1) \overset{!}{\rightsquigarrow} (p_2,e_2)$. The non-terminal s describes sequences of actions. The relation $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ defines how a sequence of actions takes one editor state to another, in terms of the relations on individual actions. The definitions of these relations are given in Figure 6.

The λ-abstract action wraps a λ binding around the subterm of e at path p. This does not affect which binders the variables of the subterm reference, since the action shifts the variable indices of variables which are free in the subterm. Thus, in the new λ-abstracted subterm, there are no variable occurrences bound by the new binding.

The $\rightarrow$-abstract action replaces the type t at path p by the type $\texttt{unit} \rightarrow t$. This is how types for bindings are built up.

The replacement action replaces the subterm at path p with a variable x which is in scope at that path, or with the unit term $\bullet$. This is how elements of base types and references to bindings are introduced after the bindings are introduced by λ-abstraction.

The apply action takes the subterms of e_1 and e_2 pointed to by the path p, and constructs their application under the same sequence of binders. It cannot be applied when p goes into an application, as discussed in the description of the $\texttt{appable}(p)$ relation in Section 2.

The delete binding action replaces a λ-bound subterm with the body of the binding, assuming that their are no occurences of the bound varialbe. The variable occurences in the body are adjusted to continue to point to the same bindings. This is how terms with λ bindings may be deconstructed.

The unapply action splits a term at an application site, producing a term with the operator substituted for the application, and another term with the operand substituted for the application. This is how terms with applications may be deconstructed into their component terms.

The movement operations are straightforward. Movement operations extend paths with the atoms corresponding to their names, in the case that the extended path is valid on the corresponding term. The exception is the up action, which removes the last atom from a path, corresponding to selecting the parent subterm of a subterm.

[1] $\mathcal{L}$ denotes the set of trees matched by a nonterminal.

4. Properties

We define the properties of "soundness" and "completeness" for editing semantics with respect to typing semantics, and prove that they hold for the set of actions described in Section 3. These concepts are analogous to soundness and completeness for type systems, with the crucial difference that a sound and complete editing system (with respect to a particular type system) is in fact possible.

4.1 Soundness

The soundness theorem (Theorem 1) states that if all of the terms input to an action are well-typed, then all terms in its output are well-typed as well.

In support of the statement of this theorem, we define a predicate which is true if and only if all terms in an editing state are well-typed:

Definition 1.

$$\texttt{welltyped}(\mathcal{E}) \equiv \forall (p,e) \in \mathcal{E}. \exists t \in \mathcal{L}(t). \vdash e : t$$

The formal statement of the soundness theorem is:

Theorem 1.

$$\texttt{welltyped}(\mathcal{E}_1) \wedge s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \Rightarrow \texttt{welltyped}(\mathcal{E}_2)$$

Proof. (See Section A.2) □

Informally, this theorem states that we do not "break" the well-typedness of programs. Together with the absence of holes in the terms, this theorem means that only complete and well-typed programs can occur in an editing derivation which started with a set of well-typed terms.

The proof strategy is induction over sequences of actions. In the base case (the empty sequence), no terms are added to or removed from the set, and so the preservation of well-typedness holds trivially. In the inductive step, we show that for each action, either the new terms are well-typed, or the single step relations $\rightsquigarrow$ and $\overset{!}{\rightsquigarrow}$ do not hold, and thus the action is impossible at that step. This is the case because the judgements for the $\rightsquigarrow$ relation restrict substituted terms to those whose types are in the set $c(p,e) = C$, and we can show that for any type in C, the context will typecheck given a term of that type.

4.2 Completeness

The completeness theorem (Theorem 2) says that any well-typed term can be reached from the unit term.

Theorem 2.

$$\vdash e : t \Rightarrow \exists s, \mathcal{E}. s(\{(\texttt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge (\texttt{top}, e) \in \mathcal{E}$$

Proof. (See Section A.3) □

Informally, this theorem states that we do not give up the ability to derive any well-typed program. This property is of course important for a general software development tool, so it is encouraging to demonstrate that our editing system maintains it.

The proof strategy for the construction lemma (Lemma **??**) is to first prove, by induction on sizes of sets and induction over STLC terms without applications, that all terms in the unzipping of the term targeted for construction can be constructed. Informally, the unzipping is the set of all variables and constants from the term in

$\boxed{a(p,e) \rightsquigarrow \mathcal{E}}$

LAMABST

$$\frac{\uparrow(e_{\texttt{bod}}) = e_{\texttt{new}} \qquad \gamma(p,e) = \Gamma \qquad \Gamma; \texttt{unit} \vdash e_{\texttt{new}} : t \qquad \begin{array}{c} p(e) = e_{\texttt{bod}} \\ \texttt{c}(p,e) = C \end{array} \qquad \texttt{unit} \to t \in C \qquad e[\lambda : \texttt{unit}.e_{\texttt{new}}/p] = e_{\texttt{newnew}}}{\lambda(p,e) \rightsquigarrow \{(p, e_{\texttt{newnew}})\}}$$

ARRABST

$$\frac{p(e) = t \qquad \texttt{c}(p,e) = C \qquad \texttt{unit} \to t \in C \qquad e[\texttt{unit} \to t/p] = e_{\texttt{new}}}{\to (p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

REPLACE

$$\frac{\texttt{c}(p,e) = C \qquad \gamma(p,e) = \Gamma \qquad \Gamma \vdash x : t \qquad t \in C \qquad e[x/p] = e_{\texttt{new}}}{\texttt{replace}_x(p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

REPLACEUNIT

$$\frac{\texttt{c}(p,e) = C \qquad \texttt{unit} \in C \qquad e[\bullet/p] = e_{\texttt{new}}}{\texttt{replace}_\bullet(p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

UNBIND

$$\frac{p(e) = \lambda : t.e_{\texttt{bod}} \qquad \downarrow(e_{\texttt{bod}}) = e_{\texttt{new}} \qquad \texttt{c}(p,e) = C \qquad \gamma(p,e) = \Gamma \qquad \Gamma \vdash e_{\texttt{new}} : t_{\texttt{new}} \qquad t_{\texttt{new}} \in C \qquad e[e_{\texttt{new}}/p] = e_{\texttt{newnew}}}{\texttt{unbind}(p,e) \rightsquigarrow \{(p, e_{\texttt{newnew}})\}}$$

UNAPPLY

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}} \qquad \Gamma \vdash e_{\texttt{rator}} : t_{\texttt{rator}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{rand}} \qquad \begin{array}{c} \texttt{c}(p,e) = C \\ t_{\texttt{rator}} \in C \end{array} \qquad \begin{array}{c} \gamma(p,e) = \Gamma \\ t_{\texttt{rand}} \in C \end{array} \qquad e[e_{\texttt{rator}}/p] = e_1 \qquad e[e_{\texttt{rand}}/p] = e_2}{\texttt{unapply}(p,e) \rightsquigarrow \{(p,e_1),(p,e_2)\}}$$

$\boxed{a(p,e_1,e_2) \rightsquigarrow \mathcal{E}}$

APPLY

$$\frac{\begin{array}{c} p(e_1) = e_{\texttt{rator}} \qquad p(e_2) = e_{\texttt{rand}} \qquad \texttt{appable}(p) \qquad \texttt{c}(p,e_1) = C \\ \gamma(p,e_1) = \Gamma \qquad \gamma(p,e_2) = \Gamma \qquad \Gamma \vdash e_{\texttt{rator}} : t_{\texttt{in}} \to t_{\texttt{out}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{in}} \qquad t_{\texttt{out}} \in C \end{array} \qquad e_1[e_{\texttt{rator}}\ e_{\texttt{rand}}/p] = e_{\texttt{new}}}{\texttt{app}(p,e_1,e_2) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

$\boxed{a(p_1,e_1) \overset{!}{\rightsquigarrow} (p_2,e_2)}$

LAMTYMOVE

$$\frac{p(e) = \lambda t : e_{\texttt{bod}}}{\texttt{lamty}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{lamty}, e)}$$

LAMBODMOVE

$$\frac{p(e) = \lambda t : e_{\texttt{bod}}}{\texttt{lambod}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{lambod}, e)}$$

APPRATORMOVE

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}}}{\texttt{apprator}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{apprator}, e)}$$

APPRANDMOVE

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}}}{\texttt{apprand}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{apprand}, e)}$$

TYINMOVE

$$\frac{p(e) = t_{\texttt{in}} \to t_{\texttt{out}}}{\texttt{tyin}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{typein}, e)}$$

TYOUTMOVE

$$\frac{p(e) = t_{\texttt{in}} \to t_{\texttt{out}}}{\texttt{tyout}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{typeout}, e)}$$

UPMOVE

$$\frac{q \in \{\texttt{lambod}, \texttt{lamty}, \texttt{apprator}, \texttt{apprand}, \texttt{tyin}, \texttt{tyout}\}}{\texttt{up}(p\ q, e) \overset{!}{\rightsquigarrow} (p,e)}$$

$\boxed{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2}$

ACTION

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e) \in \mathcal{E}_2 \qquad a(p,e) \rightsquigarrow \mathcal{E}_3}{s\ a(p,e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \cup \mathcal{E}_3}$$

DOUBLEACTION

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e_1) \in \mathcal{E}_2 \qquad (p,e_2) \in \mathcal{E}_2 \qquad a(p,e_1,e_2) \rightsquigarrow \mathcal{E}_3}{s\ a(p,e_1,e_2)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \cup \mathcal{E}_3}$$

ACTIONMUTATE

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e) \in \mathcal{E}_2 \qquad a(p,e) \overset{!}{\rightsquigarrow} (p_{\texttt{new}}, e_{\texttt{new}})}{s\ a(p,e)(\mathcal{E}) \rightsquigarrow (\mathcal{E}_2 - \{(p,e)\}) \cup \{(p_{\texttt{new}}, e_{\texttt{new}})\}}$$

ACTIONREFLEXIVE

$$(\mathcal{E}) \overset{*}{\rightsquigarrow} \mathcal{E}$$

Figure 6. Definitions of editing actions.

the binding context which they appear. With this proof in hand, we show by induction on the number of applications in a term how the terms from the unzipping may be combined to form the target term.

The destruction lemma is straightforward, as the action to replace a term with `unit` is sufficient to accomplish all it requires.

The proof strategy appears readily generalizable to more powerful type systems. In fact, the construction of terms will likely be less constrained, as the polymorphism of type systems such as System F [6, 17] and various dependent calculi will liberalize the constraints imposed on editing actions (Section 6.1).

5. Programming With Editing Actions

Functional languages lend themselves to two general programming strategems. In *top-down* programming, a programmer begins writing the top-level structure of a program, referencing not-yet-implemented functionality by means of identifiers which will later be bound to an implementation. In *bottom-up* programming, a programmer begins by writing small pieces of functionality, and composes them into larger pieces until the top-level program emerges. Our system supports both of these approaches, without locking the programmer into one or the other.

Top-down programming is supported primarily by the λ action. Upon encountering the need for a new piece of functionality, the programmer λ-abstracts over the structure he has written so far, and alters the type of the λ binding to be the type of the component required. The programmer can later implement this component and apply the top-level structure to it. Of course, an action to β-reduce or inline would be of great utility here (see Section 8).

Bottom-up programming is supported primarily by the apply action. Once a programmer has implemented some components, some combinator (often function composition) must be applied in order to compose them. Alternately, once an intermediate value is obtained, a function is applied to obtain the final or next intermediate value.

6. Discussion

6.1 Generalization to Other Typed λ-Calculi

The simply-typed lambda calculus is quite restrictive and does not admit many interesting programs. It is instructive to consider the application of this technique to polymorphic calculi. In particular, the operation of the type-constraint operation $\mathtt{c}(p, e)$ to the operand of an application will change significantly. In the simply-typed lambda calculus, the operand of an application is constrained to a single type, namely, the input type of the operator.

In a polymorphic calculus, the set of acceptable types expands to any type which is compatible with the input type of the operator. Further, the acceptable types of operators expands to any type with whose input type the type of the operand is compatible. This supports the intuitive expectation that a more powerful and flexible calculi will be more flexible to edit under this system as well.

6.2 Additional Actions

The set of actions described here is theoretically complete, but several more desirable actions immediately spring to mind. For instance, the top-down programming approach would benefit greatly from an operation to inline or β-reduce an application. Refactoring of programs would benefit from operations to re-order bindings and applications.

There are two ways of introducing additional actions. Actions can be added to the initial set of actions, which provides more definitional power but requires re-proving the soundness theorem. Alternately, actions can be composed to form new actions. For instance, it is plausible to imagine a composite action which λ-abstracts a term, gives the binding the appropriate type, and applies the term to the bound variable.

6.3 Implementation and User Interface

One advantage of defining editing actions directly on terms is that the the set of actions does not constrain the user interface. Text input and editing is awkward at best on touch-centric and mobile devices, game systems, and accessible interfaces. We expect the approach described here to work well on these platforms, as both movement through the program and alterations to the program are done at the granularity of subterms, rather than characters in program strings.

We are in the process of implementing these editing actions for the simply-typed lambda calculus. Our initial implementation will target Javascript for local in-browser editing of STLC terms. As we extend this work to more polymorphic calculi, we intend to implement the extensions as well. Further, we intend to perform human-computer interaction studies to ascertain the best possible user interface for this approach to term editing on multiple platforms.

One interesting aspect of the user interface is the attachment and presentation of term metadata. Such metadata might be names for bindings (which are semantically formulated as De Bruijn indices), comments, library documentation, and version history. We expect the kind and presentation of this metadata to be of great import in the experience of programmers using our proposed system.

7. Related Work

Graphical programming languages by their nature must eschew text-editing actions as the primary means of creating programs, in favor of actions on graphical elements and structures. The Scratch [10] programming language is a graphical and imperative programming language in which programs control sprites in a virtual arena. The Kodu programming language [9] is a small graphical language, running on the XBox game console and intended for children. Kodu allows users to create games by composing tiles, which are graphical representations of concurrent actors. YinYang [12] is another tile-based concurrent graphical language, which adds the ability to define new tiles and is targeted at touch devices, and intended for more general software development.

Structural editors have a long history, going back at least as far as MENTOR [5] and the Cornell Program Synthesizer [18]. A structural editor is an editor which provides actions on the syntax of a language, instead of or in addition to text-editing actions. Many modern structural editors (such as ParEdit [3] and Structured Haskell Mode [4]) take the second route, extending existing text editors with structural editing actions (for s-expressions and Haskell, respectively.) The Lamdu [8] programming environment uses a structured editor and abstract representations of programs as the basis of an integrated development environment for a Haskell-like language.

Theorem proving systems such as Isabelle [15], Coq [19], and ACL2 [1] provide actions called *tactics* for constructing proof terms. However, these actions generally do not support deconstructing or refactoring the proof terms, and the particular term produced is generally considered irrelevant so long as its existence is demonstrated.

The ability to construct terms by automated theorem proving has proven useful for programming in the dependently-typed language Idris [20]. In particular, automated theorem proving is often used to construct a term of a desired type from a similar term of a different type. Since our soundness theorem asserts that for any starting set of well-typed terms, we can only further reach well-typed terms, it is plausible to consider introducing well-typed terms from other sources, such as automated theorem provers, so long as these terms are typechecked beforehand.

8. Conclusions and Further Work

We have described a set of editing actions on terms in the simply-typed lambda calculus. Further, we have sketched proofs that this set of actions is *sound* (beginning with well-typed terms, only well-typed terms may be derived) and *complete* (any term may be reached from any other term. We have described how this set of actions may be used to produce programs in both top-down and bottom-up style. We have shown that this approach has promise for extension to more powerful typed λ-calculi.

There are several lines of further work open from this point. The simply-typed lambda calculus is, of course, not the most powerful or flexible language. Thus, we intend to generalize this approach to editing to more polymorphic (and eventually, dependent) calculi.

We are working to implement the actions described here, with the intent to eventually re-implement this system in itself. The implementation of the rules is straightfoward, but the question of the appropriate user interface opens up a cross-disciplinary line of inquiry between programming languages and human-computer interaction. Further, while the set of rules described here is certainly sufficient to derive any term from any other term, it is not at all clear that it is convenient or efficient to program with. We believe that a cross-disciplinary inquiry between the fields of human-computer interaction and programming languages will provide insight into the additional actions necessary for a pleasant and productive programming experience.

Acknowledgments

Michael Vitousek assisted with the proof strategies for this paper. Tim Zakian provided many helpful comments and assisted with LaTeX formatting.

References

[1] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph. Academic Press, New York, 1979. ISBN 0-12-122950-5. URL `http://www.cs.utexas.edu/users/boyer/acl.pdf`.

[2] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S095679681300018X`.

[3] T. Campbell. ParEdit. URL `http://mumble.net/~campbell/emacs/paredit.el`.

[4] C. Done. Structured Haskell mode. https://github.com/chrisdone/structured-haskell-mode, 2014.

[5] V. Donzeau-Gouge, G. Huet, B. Lang, G. Kahn, et al. Programming environments based on structured editors: The MENTOR experience. 1980.

[6] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supŕieur*. PhD thesis, Université Paris 7, 1972.

[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.01. `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/`, September 2013.

[8] E. Lotem and Y. Chuchem. Lambdu. URL `https://peaker.github.io/lamdu`.

[9] M. B. MacLaurin. The design of kodu: A tiny visual programming language for children on the xbox 360. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 241–246, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. . URL `http://doi.acm.org/10.1145/1926385.1926413`.

[10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010. ISSN 1946-6226. . URL `http://doi.acm.org/10.1145/1868358.1868363`.

[11] S. Marlow, editor. *Haskell 2010 Language Report*. 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`.

[12] S. McDirmid. Coding at the speed of touch. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 61–76, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. . URL `http://doi.acm.org/10.1145/2048237.2048246`.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised*. MIT Press, 1997. ISBN 9780262631815. URL `http://books.google.com/books?id=e0PhKfbj-p8C`.

[14] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[15] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989. ISSN 0168-7433. . URL `http://dx.doi.org/10.1007/BF00248324`.

[16] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachussetts, 2002. ISBN 978-0-262-16209-8. URL `http://www.cis.upenn.edu/~bcpierce/tapl/`.

[17] J. Reynolds. Towards a theory of type structure. *Colloque sur la Programmation*, pages 408–425, April 1974.

[18] T. Teitelbaum and T. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sept. 1981. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/358746.358755`.

[19] The Coq Development Team. *Coq Reference Manual*. 2012. URL `http://coq.inria.fr/distrib/current/refman/index.html`.

[20] The Idris Community. *Programming in Idris, A Tutorial*. 2014. URL `http://eb.host.cs.st-andrews.ac.uk/writings/idris-tutorial.pdf`.

A. Proofs

A.1 Definitions

Very often we care that a particular term is in a state, without caring what path is associated with it. The $\texttt{has}(\mathcal{E}, e)$ predicate captures this notion:

Definition 2.

$$\texttt{has}(\mathcal{E}, e) \equiv \exists p.(p, e) \in \mathcal{E}$$

The $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$ predicate says that for all terms in a state $\mathcal{E}_1$, $\mathcal{E}_2$ has that term.

Definition 3.

$$\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2) \equiv \forall (p, e) \in \mathcal{E}_1.has(\mathcal{E}_2, e)$$

A.2 Proof of Theorem 1

Proof. (Proof in progress) □

A.3 Proof of Theorem 2

In order to prove Theorem 2, we shall require another function on terms, and two lemmas. The function $\mathtt{u}(e)$ (defined in Figure 7) splits the term tree into sequences of lambda bindings, essentially breaking it apart at application sites. Lemma 1 states that for any goal term e, the set of terms $\mathtt{u}(e)$ (with associated paths) can be derived from the unit term. Lemma 2 states that a goal term e can be derived from the set of terms $\mathtt{u}(e)$.

Lemma 1.

$$\vdash e : t \Rightarrow \exists s, \mathcal{E}.s(\{(\texttt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge \{(\texttt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$$

Proof. By induction on $|\mathtt{u}(e)|$

Case 1: Base case: $|\mathtt{u}(e)| = 0$

(a) $|\mathtt{u}(e)|$ is never 0, so this case holds vacuously.

$$\mathtt{u}(\bullet) = \{\bullet\}$$
$$\mathtt{u}(v) = \{v\}$$
$$\mathtt{u}(\lambda : t.e) = \{\lambda : t.e_u | e_u \in \mathtt{u}(e)\}$$
$$\mathtt{u}(e_1\ e_2) = \mathtt{u}(e_1) \cup \mathtt{u}(e_2)$$

Figure 7. Definition of the term splitting function.

Case 2: : $|u(e)| > 0$.

(a) WLOG, pick some $e' \in u(e)$. Then $\vdash e : t \Rightarrow \exists s', \mathcal{E}'.s'(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E}' \wedge \{(\mathtt{top}, e_u) | e_u \in u(e) - \{e'\}\}$

(b) Assume $\vdash e : t$.

(c) By (2b), Lemma 4, and modus ponens: $\mathtt{keeps}(\{(\mathtt{top}, \bullet)\}, \mathcal{E}')$. By the definition of `keeps` (Definition 3) and of `has` (Definition 2), $\exists p'.(p', \bullet) \in \mathcal{E}'$.

(d) (Proof in progress.)

□

Lemma 2.

$\vdash e : t \Rightarrow \forall \mathcal{E}_1 \supseteq \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\}.\exists s, \mathcal{E}_2.(s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$

Proof. (Proof in progress) □

With these lemmas, the proof of Theorem 2 is simple:

Proof. Assume $\vdash e : t$. Then by Lemma 1 and modus ponens, $\exists s, \mathcal{E}.s(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$. By Lemma 2 and modus ponens, $\forall \mathcal{E}_1 \supseteq \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\}.\exists s', \mathcal{E}_2.(s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$. Since $\{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$, let $\mathcal{E}_1 = \mathcal{E}$. Then $\exists s', \mathcal{E}_2.(s'(\mathcal{E}) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$. By Lemma 6 and modus ponens, $s\ s'(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E}_2$. Then $s\ s'$ and $\mathcal{E}_2$ are the witnesses for the existential in the theorem.

□

A.4 Utility Lemmas

Lemma 3.

$$s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge \mathcal{E}_1 \subseteq \mathcal{E}_3 \Rightarrow \exists \mathcal{E}_4.(\mathcal{E}_2 \subseteq \mathcal{E}_4 \wedge s(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_4)$$

Proof. By induction on s:

- $s = \epsilon$:
  1. The only rule for the empty sequence is ACTIONREFLEXIVE.
  2. So $\mathcal{E}_1 = \mathcal{E}_2$.
  3. $\mathcal{E}_1 \subseteq \mathcal{E}_3$ by the assumption of the lemma.
  4. Then by substitution using (2) in (3), $\mathcal{E}_2 \subseteq \mathcal{E}_3$.
  5. $\mathcal{E}_3 \overset{*}{\rightsquigarrow} \mathcal{E}_3$ by ACTIONREFLEXIVE.
  6. So if we let $\mathcal{E}_4 = \mathcal{E}_3$, then we have a witness to the existential $\exists \mathcal{E}_4.(\mathcal{E}_2 \subseteq \mathcal{E}_4 \wedge s(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_4)$.
- $s = s'\ a(p, e)$:

  By cases on the derivation of $s'\ a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$:
  - ACTION:
    1. By the use of the ACTION rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p, e) \in \mathcal{E}_{2s'}$.
       (c) $a(p, e) \rightsquigarrow \mathcal{E}_a$.
       (d) $\mathcal{E}_2 = \mathcal{E}_{2s'} \cup \mathcal{E}_a$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), and (2), we have $(p, e) \in \mathcal{E}_{4s'}$.
    4. By the ACTION rule, (1c), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    5. By (2) and equational reasoning, we have that $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a$.
    6. By (1d), (5). and substitution, we have that $\mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    7. By the conjunction of (5) and (6), we have $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a \wedge \mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    8. By (7), we see that $\mathcal{E}_{4s'} \cup \mathcal{E}_a$ is a witness to the existential in our conclusion.
  - ACTIONMUTATE:
    1. By the use of the ACTIONMUTATE rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p_1, e_1) \in \mathcal{E}_{2s'}$.
       (c) $a(p_1, e_1) \overset{!}{\rightsquigarrow} (p_2, e_2)$.
       (d) $\mathcal{E}_2 = (\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), and (2), we have $(p_1, e_1) \in \mathcal{E}_{4s'}$.
    4. By the ACTIONMUTATE rule, (1c), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    5. By (2) and equational reasoning, we have that $((\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    6. By (1d), (5). and substitution, we have that $\mathcal{E}_2 \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    7. By the conjunction of (5) and (6), we have $((\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \wedge \mathcal{E}_2 \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    8. By (7), we see that $((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$ is a witness to the existential in our conclusion.
- $s = s\ a(p, e_1, e_2)$:

  By cases on the derivation of $s'\ a(p, e_1, e_2)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$:
  - DOUBLEACTION:
    1. By the use of the DOUBLEACTION rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p, e_1) \in \mathcal{E}_{2s'}$.
       (c) $(p, e_2) \in \mathcal{E}_{2s'}$.
       (d) $a(p, e_1, e_2) \rightsquigarrow \mathcal{E}_a$.
       (e) $\mathcal{E}_2 = \mathcal{E}_{2s'} \cup \mathcal{E}_a$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), (1c), and (2), we have $(p, e_1) \in \mathcal{E}_{4s'} \wedge (p, e_2) \in \mathcal{E}_4 s'$.
    4. By the DOUBLEACTION rule, (1d), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    5. By (2) and equational reasoning, we have that $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a$.
    6. By (1e), (5). and substitution, we have that $\mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.

7. By the conjunction of (5) and (6), we have $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a \wedge \mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
8. By (7), we see that $\mathcal{E}_{4s'} \cup \mathcal{E}_a$ is a witness to the existential in our conclusion.

□

Lemma 4.

$$s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \rightarrow \texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$$

Proof. By induction on s.

Case 1: Base case: $s = \epsilon$.
- (a) Assume $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$.
- (b) By ACTIONREFLEXIVE rule in the derivation of $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ (1a), $\mathcal{E}_1 = \mathcal{E}_2$.
- (c) By (1b), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$ holds trivially.

Case 2: $s = s'a(p, e)$.
- (a) Inductive hypothesis: $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2' \Rightarrow \texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2')$
- (b) Assume:
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
- (c) By (2a) and the definition of `keeps` (Definition 3), and the definition of `has` (Definition 2): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2' \Rightarrow \forall (p, e) \in \mathcal{E}_1 \exists p'.(p', e) \in \mathcal{E}_2'$
- (d) The derivation of $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ (2(b)i) is either ACTION or ACTIONMUTATE. By cases:
  - i. ACTION:
    - A. By the premises of the ACTION rule in the derivation (2(d)i): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2'$.
    - B. By (2(d)iA), (2a), and modus ponens: $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2'$
    - C. By the conclusion of the ACTION rule: $\mathcal{E}_2 = \mathcal{E}_2' \cup \mathcal{E}_3$
    - D. By (2(d)iC), $s_2' \subseteq s_2$.
    - E. By (2(d)iD), $\forall (p, e) \in \mathcal{E'}_2 . (p, e) \in \mathcal{E}_2$.
    - F. By (2(d)iD) and (2(d)iA), $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2$.
    - G. By (2(d)iF), the definition of `keeps` (Definition 3), and the definition of `has` (Definition 2), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$.
  - ii. ACTIONMUTATE:
    - A. By the premises of the ACTIONMUTATE rule in the derivation (2(d)ii): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2'$.
    - B. By (2(d)iiA), (2a), and modus ponens: $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2'$
    - C. By the premises of the ACTIONMUTATE rule in the derivation (2(d)ii): $a(p, e) \overset{!}{\rightsquigarrow} \mathcal{E}_3$.
    - D. By Lemma 5, (2(d)iiC), and modus ponens: $\texttt{has}(\mathcal{E}_3, e)$.
    - E. By the conclusion of the ACTIONMUTATE rule in the derivation (2(d)ii), $\mathcal{E}_2 = (\mathcal{E}_2' - \{(p, e)\}) \cup \mathcal{E}_3$
    - F. Observing that $\texttt{has}(\mathcal{E}, e) \Rightarrow \texttt{has}(\mathcal{E}' \cup \mathcal{E}, e)$, and by (2(d)iiB), (2(d)iiD) and (2(d)iiE): $\forall (p, e) \in \mathcal{E}_1 . \texttt{has}(\mathcal{E}_2, e)$ and thus, by the definition of `keeps` (Definition 3), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$.

□

Lemma 5.

$$a(p, e) \overset{!}{\rightsquigarrow} (p', e') \Rightarrow \texttt{has}(\mathcal{E}, e)$$

Proof. 1. Assume $a(p, e) \overset{!}{\rightsquigarrow} \mathcal{E}$.

2. By cases on the derivation of $a(p, e) \overset{!}{\rightsquigarrow} (p, e)$ (1):
- Case (a): LAMTYMOVE
  - i. By the conclusion of the derivation of LAMTYMOVE (2(a)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (b): LAMBODMOVE
  - i. By the conclusion of the derivation of LAMBODMOVE (2(b)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (c): APPRATORMOVE
  - i. By the conclusion of the derivation of APPRATORMOVE (2(c)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (d): APPRANDMOVE
  - i. By the conclusion of the derivation of APPRANDMOVE (2(d)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (e): TYINMOVE
  - i. By the conclusion of the derivation of TYINMOVE (2(e)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (f): TYOUTMOVE
  - i. By the conclusion of the derivation of TYOUTMOVE (2(f)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (g): UPMOVE
  - i. By the conclusion of the derivation of UPMOVE (2(g)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.

□

Lemma 6.

$$s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3 \Rightarrow s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$$

Proof. By induction on s_2.

Case 1: Base case: $s_2 = \epsilon$.
- (a) Assume
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
  - ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
- (b) Since $s_2 = \epsilon$, the derivation of $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$ (1(a)ii) is ACTIONREFLEXIVE, and $\mathcal{E}_2 = \mathcal{E}_3$.
- (c) Since $s_2 = \epsilon$, $s_1 s_2 = s_1$.
- (d) Substituting (1c) and (1d) into (1(a)i), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$

Case 2: $s_2 = s_2' a(p, e)$.
- (a) Inductive hypothesis: $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \Rightarrow s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
- (b) Assume
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
  - ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
- (c) The derivation of $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$ (2(b)ii) is either ACTION or ACTIONMUTATE. By cases:
  - i. ACTION:
    - A. By the ACTION rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$

B. By the ACTION rule in the derivation: $(p, e) \in \mathcal{E}_3'$
C. By the ACTION rule in the derivation: $a(p, e) \rightsquigarrow \mathcal{E}_4$
D. By the ACTION rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
E. By (2(b)i), (2(c)iA), (2a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
F. By (2(c)iE), (2(c)iB), (2(c)iC) and the ACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
G. By substition of (2) and (2(c)iD) into (2(c)iF), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

ii. ACTIONMUTATE:
A. By the ACTIONMUTATE rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
B. By the ACTIONMUTATE rule in the derivation: $(p, e) \in \mathcal{E}_3'$
C. By the ACTIONMUTATE rule in the derivation: $a(p, e) \rightsquigarrow \mathcal{E}_4$
D. By the ACTIONMUTATE rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
E. By (2(b)i), (2(c)iiA), (2a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
F. By (2(c)iiE), (2(c)iiB), (2(c)iiC) and the ACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
G. By substition of (2) and (2(c)iiD) into (2(c)iiF), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

Case 3: $s_2 = s_2' a(p, e_1, e_2)$.
(a) Inductive hypothesis: $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \Rightarrow s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
(b) Assume
i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
(c) By the DOUBLEACTION rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
(d) By the DOUBLEACTION rule in the derivation: $(p, e_1) \in \mathcal{E}_3'$
(e) By the DOUBLEACTION rule in the derivation: $(p, e_2) \in \mathcal{E}_3'$
(f) By the DOUBLEACTION rule in the derivation: $a(p, e_1, e_2) \rightsquigarrow \mathcal{E}_4$
(g) By the DOUBLEACTION rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
(h) By (3(b)i), (3c), (3a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
(i) By (3h), (3d), (3e), (3f) and the DOUBLEACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
(j) By substition of (3) and (3g) into (3i), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

□