Relating Staged Computation to the Record Calculus

Özyeğin University Technical Report: OZU-COMP-2010-0001

Relating Staged Computation to the Record Calculus

Baris Aktemur
Özyeğin University
baris.aktemur@ozyegin.edu.tr

Wontae Choi
Seoul National University
wtchoi@ropas.snu.ac.kr

Abstract

It has been previously shown that there is a close relation between record calculus and program generation (e.g. Lisp-like quasiquotations): A translation has been defined to convert staged expressions to record calculus expressions, and it has been shown that the call-by-value semantics of the staged and the record calculi are equivalent modulo the translation and admin reductions. In this work, we investigate the relation further. The contributions are twofold: (1) We fine-tune the previously shown relation between the two operational semantics, and obtain more precise results. In particular, we show that only two kinds of admin reductions suffice, and these reductions can be applied exhaustively. (2) We define a reverse translation that converts record calculus expressions back to the staged calculus, allowing us to go back and forth between the two calculi. We believe that these results provide an important step towards reusing already-existing record calculus static analyses to reason about staged expressions.

1 Introduction

Program Generation (PG) is the technique of combining various code fragments to construct a program. PG approaches can be classified into two: those that originate from the idea of partial evaluation and have variable hygiene (e.g. [7, 10, 2]); those that originate from the idea of "code as data" and have unhygienic variable capture (e.g. [6, 9, 11, 5]). In this paper we take the latter approach as our context and use the terms "program generation" and "staged computation" to refer to it, unless stated otherwise.

Recently, a translation that converts program generators to record calculus expressions has been defined, and it has been shown that the operational semantics of program generation is equivalent to the operational semantics of record calculus (i.e. lambda calculus with records) [1]. This semantic relation has led to the result that a record type system can be used as a sound type system for program generation. In fact, such a type system has been shown equal to λ^{open}_{poly} [6]. Furthermore, polymorphic subtyping in record calculus [8] can seamlessly be integrated into the type system, giving us a type system for program generation that is more powerful than existing ones. These results raise a question: *Can we use already existing properties of the record calculus (e.g. data flow analysis, control flow analysis, abstract interpretation) to analyze and reason about staged programs?*

In this paper, to pave the path to answering the question raised above, we elaborate on the previously defined translation and the semantic relation. We provide more precise and stronger results. In particular, there are two contributions:

- Informally, the semantic relation between the record and staged calculi stated in [1] is the following: Let e be a staged program. If e reduces to e' in one small step reduction (using the

staged semantics), the translation of e to the record calculus reduces to the translation of e' in one small step reduction (using the record semantics) followed by a number of so-called "safe reductions" that may happen anywhere in the program regardless of the evaluation order. In this paper we fine-tune the definition of "safe reductions". We show that only two kinds of administrative reductions are needed, and that these reductions can be applied exhaustively without the danger of oversimplifying a term.

- In addition to the translation defined in [1] that converts staged expressions to record calculus expressions, we define in this paper a *reverse translation* to convert record calculus expressions back to staged calculus.

Combining the two results above has the following practical corollary: We can take a staged expression e, translate it to a record expression, evaluate the translation using a record calculus interpreter (and apply the admin reductions as well), translate the result back to the staged calculus and we will have obtained the result of evaluating e in a staged calculus interpreter – without ever implementing that interpreter! Based on the same idea, we think that it is feasible to analyze the translation of e using an analysis defined for record calculus, then translate the analysis results back to the staged calculus. This is a topic for future research.

The paper is organized as follows: In Section 2 we give the formal definition of the staged and the record calculi as the background information. This includes the syntax and the operational semantics, as well as the definition of the translation from staged expressions to record expressions. Section 2 is provided to make this paper self-contained; it does not present new subject. However, there are minor changes (mostly notational) made to the definitions. In Section 3 we state the relation between the two operational semantics. In this relation, "admin" reductions are used. We identify two such reductions. In Section 4 we define a "reverse translation" that converts record expressions back to staged expressions. In Section 5 we take the now-classic exponentiation example and illustrate how the regular and reverse translations work. Finally in Section 6 we conclude.

2 Background

To form a self-contained paper, we give the necessary definitions related to the staged and record calculi. These definitions are not new; they are taken from [1]. (We shall note that the syntax and semantics of the staged and the record calculi in [1] are not brand new. Several definitions in various flavors appeared in previous publications; a survey is out of the scope of this paper. Please see [1] for a sample list and discussion.) There are, however, some differences (mostly notational) between the definitions presented in [1] and given here; the goal of these differences is to improve the presentation and to make the reverse translation easier to define and prove. The theorems and proofs stated in [1] are straightforwardly adapted to take the new changes into account.

2.1 Syntax

The syntax of the staged calculus λ^{gen}_{poly} is given in Figure 1; the syntax of the record calculus λ^{rec}_{poly} is in Figure 2. In addition to plain lambda abstraction and let-binding, the record calculus contains annotated lambda abstraction, fix-point operator, and let-binding. The annotated versions are introduced for the purposes of reverse translation; the annotations are simply ignored in operational semantics and the type system. (A non-annotated fix-point operator is not included in the syntax

$$
\begin{array}{l}
x \in Var \\
c \in Constant \\
\ell \in Location \\
e \in Exp ::= c \mid x \mid e\,e \\
\qquad\qquad \mid \lambda x.e \mid \mathsf{fix}\, f(x).\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \\
\qquad\qquad \mid \langle e \rangle \mid \tilde{}(e) \mid \mathsf{run}(e) \mid \mathsf{lift}(e) \\
\qquad\qquad \mid \ell \mid \mathsf{ref}\ e \mid\ !e \mid e{:=}e
\end{array}
$$

Figure 1: Syntax of λ^{gen}_{poly}.

$$
\begin{array}{ll}
x \in Var & a \in Label = Var \\
\rho \in RVar & w, f \in Name = Var \cup RVar \\
c \in Constant & \ell \in Location \\
e \in RExp ::= c \mid w \mid e\,e & \\
\qquad \mid \lambda w.e \mid \mathsf{let}\, w = e\, \mathsf{in}\, e & \\
\qquad \mid \lambda_x x.e \mid \mathsf{fix}_{f,x}\ f(x).\, e \mid \mathsf{let}_x\ x = e\, \mathsf{in}\, e & \\
\qquad \mid \{\} \mid e\, \mathsf{with}\, \{a = e\} \mid e{\cdot}a & \\
\qquad \mid \ell \mid \mathsf{ref}\ e \mid\ !e \mid e{:=}e &
\end{array}
$$

Figure 2: Syntax of λ^{rec}_{poly}.

because the reverse translation does not produce a non-annotated fix-point operator. Therefore having only the annotated version suffices.)

To reduce the need for extra notation, we do not include the λ^* abstraction that exists in λ^{gen}_{poly}[6]. λ^* works as a gensym operator to avoid variable capture when filling in holes. Extending the reverse translation to include λ^* is not difficult, however, extra annotations would be needed and the proofs would be longer.

In the record calculus syntax we distinguish record variables and non-record variables. This is done for the purposes of type-checking, which we do not cover in this paper. However, we take advantage of this separation to guide the reverse translation. A brief explanation for the reason of this distinction is the following: The translation converts quoted expressions to functions. For example, $\langle 42 \rangle$ becomes $\lambda\rho.42$. If the distinction of variables was not made, the function could be given any type for its input in the record type system. This means that a quoted expression could be treated like an ordinary function after it is translated. However, it should only be treated as a function that takes in an environment (i.e. a record). The syntactic distinction makes it possible to restrict the types given to record variables. Please see [1] for a more detailed discussion.

2.2 Auxiliary Definitions

Definition 2.1. An expression e is a *stage-n expression* iff the nesting level of antiquotations (i.e. $\tilde{}(\cdot)$) that are not matched by quotations (i.e. $\langle\cdot\rangle$) is less than or equal to n. Note this also means that a stage-n expression is also a stage-$(n+1)$ expression. Examples: $\langle\, \tilde{}(c) + 1 \rangle$ is a stage-0 expression; $\langle\, \tilde{}(c\ \tilde{}(d)) + 1 \rangle$ is a stage-1 expression; $\tilde{}(\,\tilde{}(\langle\, \tilde{}(c) \rangle))$ is a stage-2 expression.

Definition 2.2 (Renaming environment)**.** The translation uses a *renaming environment*, which is a record extension expression that is used to associate variables. A renaming environment R is defined as follows.

$$
\begin{array}{l}
y \in Var \\
x \in Label = Var \\
\rho \in RVar \\
R \in RenamingEnv ::= \{\} \mid \rho \mid R\, \mathsf{with}\, \{x = y\}
\end{array}
$$

A renaming environment is also interpreted as a function from variables to expressions, defined as follows:

$$\begin{aligned}(R\,\mathsf{with}\,\{x = y\})(x) &= y\\ (R\,\mathsf{with}\,\{z = y\})(x) &= R(x) \text{ if } x \neq z\\ \rho(x) &= \rho{\cdot}x\\ \{\}(x) &= \textbf{error}\end{aligned}$$

Throughout the paper we assume that the variables mapped to by a renaming environment (e.g. z in $\rho\,\mathsf{with}\,\{x = z\}$) are unique (i.e. they are fresh; they do not occur anywhere else). This property is preserved by the translation.

Notation 2.3. We use the shorter notation $\{a_1 = e_1, a_2 = e_2, \ldots, a_m = e_m\}$ for the expression $\{\}\,\mathsf{with}\,\{a_1 = e_1\}\,\mathsf{with}\,\{a_2 = e_2\}\ldots\,\mathsf{with}\,\{a_m = e_m\}$, and similarly $R\,\mathsf{with}\,\{a_1 = e_1, a_2 = e_2, \ldots, a_m = e_m\}$ for $R\,\mathsf{with}\,\{a_1 = e_1\}\,\mathsf{with}\,\{a_2 = e_2\}\ldots\,\mathsf{with}\,\{a_m = e_m\}$.

Notation 2.4. A list of renaming environments $R_0, \ldots, R_n$ is denoted as $\vec{R}_n$.

Notation 2.5. The function that maps a_i to b_i for $0 \leq i \leq k$ is denoted as $\{a_0 : b_0, \ldots, a_k : b_k\}$, or as $\{\vec{a}_k : \vec{b}_k\}$ for short when the value of k is not important.

Definition 2.6. The function update operator, $\leftarrow$, is defined as follows:

$$(f \leftarrow g)(x) = \begin{cases} g(x), & \text{if } x \in dom(g)\\ f(x), & \text{otherwise}\end{cases}$$

Definition 2.7 (Free variables)**.** The set of free variables of a record expression $\underline{e}$ is denoted as $FV(\underline{e})$. Similarly, the set of stage-0 free variables of a stage-n expression e is denoted as $FV(e)^n$. In both cases, variables are bound by lambda abstractions, let-bindings and fix-point operators in the usual sense.

Definition 2.8 (Substitution)**.** Substituting the free occurrences of w in the record expression $\underline{e}$ with the expression $\underline{e}'$ is denoted as $\underline{e}\{w \backslash \underline{e}'\}$. Similarly, substituting the free occurrences of the stage-0 variable x in the stage-n expression e with the stage-0 expression e' is denoted as $e\{x \backslash e'\}^n$. In both cases, substitution avoids capturing free variables of the substitute.

2.3 Operational Semantics

The small-step call-by-value operational semantics of the staged calculus and the record calculus are given in Figure 3 and Figure 4, respectively.

2.4 Translation

The translation that converts staged expressions to record expressions is given in Figure 6. The difference with the definition given in [1] is that the translation of lambda abstractions and let-bindings now involve annotations, and the $\mathsf{run}(\cdot)$ operator is translated to a let-binding instead of a function application. Both modifications are introduced to help the definition of the reverse translation.

The key points of the translation are the following:

$$
\begin{array}{l}
v^n \in Val^n \\
Val^0 \quad ::= c \mid \lambda x.e \mid \mathsf{fix}\, f(x).\, e \mid \langle v^1 \rangle \mid \ell \\
Val^{n+1} ::= c \mid x \mid \lambda x.v^{n+1} \mid \mathsf{fix}\, f(x).\, v^{n+1} \mid v^{n+1} v^{n+1} \\
\qquad\qquad \mid \ell \mid \mathsf{ref}\ v^{n+1} \mid\ !v^{n+1} \mid v^{n+1} {:=} v^{n+1} \\
\qquad\qquad \mid \langle v^{n+2} \rangle \mid \mathsf{lift}(v^{n+1}) \mid \mathsf{run}(v^{n+1}) \mid \mathsf{let}\, x = v^{n+1}\, \mathsf{in}\, v^{n+1} \\
\qquad\qquad \mid \text{`}(v^n) \quad (\text{if } n > 0) \\
\mathcal{S} \in Store = Location \rightharpoonup Val^0
\end{array}
$$

ESABS $$\frac{\mathcal{S}, e \xrightarrow{n+1} \mathcal{S}', e'}{\mathcal{S}, \lambda x.e \xrightarrow{n+1} \mathcal{S}', \lambda x.e'}$$

ESFIX $$\frac{\mathcal{S}, e \xrightarrow{n+1} \mathcal{S}', e'}{\mathcal{S}, \mathsf{fix}\, f(x).\, e \xrightarrow{n+1} \mathcal{S}', \mathsf{fix}\, f(x).\, e'}$$

ESAPP $$\frac{\mathcal{S}, e_1 \xrightarrow{n} \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \xrightarrow{n} \mathcal{S}', e_1' e_2} \qquad \frac{e_1 \in Val^n \quad \mathcal{S}, e_2 \xrightarrow{n} \mathcal{S}', e_2'}{\mathcal{S}, e_1 e_2 \xrightarrow{n} \mathcal{S}', e_1 e_2'} \qquad \frac{e_2 \in Val^0}{\mathcal{S}, (\lambda x.e) e_2 \xrightarrow{0} \mathcal{S}, e\{x \backslash e_2\}^0}$$

$$\frac{e_2 \in Val^0}{\mathcal{S}, (\mathsf{fix}\, f(x).\, e) e_2 \xrightarrow{0} \mathcal{S}, e\{f \backslash \mathsf{fix}\, f(x).\, e\}^0 \{x \backslash e_2\}^0}$$

ESLET $$\frac{\mathcal{S}, e_1 \xrightarrow{n} \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \xrightarrow{n} \mathcal{S}', \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2} \qquad \frac{e_1 \in Val^0}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \xrightarrow{0} \mathcal{S}, e_2\{x \backslash e_1\}^0}$$

$$\frac{e_1 \in Val^{n+1} \quad \mathcal{S}, e_2 \xrightarrow{n+1} \mathcal{S}', e_2'}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \xrightarrow{n+1} \mathcal{S}', \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2'}$$

ESBOX $$\frac{\mathcal{S}, e \xrightarrow{n+1} \mathcal{S}', e'}{\mathcal{S}, \langle e \rangle \xrightarrow{n} \mathcal{S}', \langle e' \rangle}$$

ESUBOX $$\frac{\mathcal{S}, e \xrightarrow{n} \mathcal{S}', e'}{\mathcal{S}, \text{`}(e) \xrightarrow{n+1} \mathcal{S}', \text{`}(e')} \qquad \frac{e \in Val^1}{\mathcal{S}, \text{`}(\langle e \rangle) \xrightarrow{1} \mathcal{S}, e}$$

ESRUN $$\frac{\mathcal{S}, e \xrightarrow{n} \mathcal{S}', e'}{\mathcal{S}, \mathsf{run}(e) \xrightarrow{n} \mathcal{S}', \mathsf{run}(e')} \qquad \frac{e \in Val^1}{\mathcal{S}, \mathsf{run}(\langle e \rangle) \xrightarrow{0} \mathcal{S}, e}$$

ESLIFT $$\frac{\mathcal{S}, e \xrightarrow{n} \mathcal{S}', e'}{\mathcal{S}, \mathsf{lift}(e) \xrightarrow{n} \mathcal{S}', \mathsf{lift}(e')} \qquad \frac{e \in Val^0}{\mathcal{S}, \mathsf{lift}(e) \xrightarrow{0} \mathcal{S}, \langle e \rangle}$$

ESREF $$\frac{\mathcal{S}, e \xrightarrow{n} \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\ e \xrightarrow{n} \mathcal{S}', \mathsf{ref}\ e'} \qquad \frac{e \in Val^0 \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\ e \xrightarrow{0} \mathcal{S} \leftarrow \{\ell : e\}, \ell}$$

ESDEREF $$\frac{\mathcal{S}, e \xrightarrow{n} \mathcal{S}', e'}{\mathcal{S}, !e \xrightarrow{n} \mathcal{S}', !e'} \qquad \frac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \xrightarrow{0} \mathcal{S}, v}$$

ESASGN $$\frac{\mathcal{S}, e_1 \xrightarrow{n} \mathcal{S}', e_1'}{\mathcal{S}, e_1 {:=} e_2 \xrightarrow{n} \mathcal{S}', e_1' {:=} e_2} \qquad \frac{e_1 \in Val^n \quad \mathcal{S}, e_2 \xrightarrow{n} \mathcal{S}', e_2'}{\mathcal{S}, e_1 {:=} e_2 \xrightarrow{n} \mathcal{S}', e_1 {:=} e_2'} \qquad \frac{e_2 \in Val^0}{\mathcal{S}, \ell {:=} e_2 \xrightarrow{0} \mathcal{S} \leftarrow \{\ell : e_2\}, e_2}$$

Figure 3: The operational semantics of λ^{open}_{poly}.

$$v \in RVal ::= c \mid \lambda x.e \mid \mathsf{fix}\, f(x).\, e \mid \{a_i : v_i\}_1^m \mid \ell$$
$$\mathcal{S} \in RStore = Location \rightharpoonup RVal$$

ERAPP
$$\frac{\mathcal{S}, e_1 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1' e_2} \qquad \frac{e_1 \in RVal \quad \mathcal{S}, e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_2'}{\mathcal{S}, e_1 e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1 e_2'} \qquad \frac{e_2 \in RVal}{\mathcal{S}, (\lambda w.e_1) e_2 \xrightarrow{\mathcal{R}} \mathcal{S}, e_1\{w \backslash e_2\}}$$
$$\frac{e_2 \in RVal}{\mathcal{S}, (\mathsf{fix}\, f(x).\, e_1) e_2 \xrightarrow{\mathcal{R}} \mathcal{S}, e_1\{f \backslash \mathsf{fix}\, f(x).\, e_1\}\{x \backslash e_2\}}$$

ERLET
$$\frac{\mathcal{S}, e_1 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', \mathsf{let}\, w = e_1' \,\mathsf{in}\, e_2} \qquad \frac{e_1 \in RVal}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \xrightarrow{\mathcal{R}} \mathcal{S}, e_2\{w \backslash e_1\}}$$

ERUPD
$$\frac{\mathcal{S}, e_1 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1'}{\mathcal{S}, e_1 \,\mathsf{with}\, \{a = e_2\} \xrightarrow{\mathcal{R}} \mathcal{S}', e_1' \,\mathsf{with}\, \{a = e_2\}} \qquad \frac{e_1 \in RVal \quad \mathcal{S}, e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_2'}{\mathcal{S}, e_1 \,\mathsf{with}\, \{a = e_2\} \xrightarrow{\mathcal{R}} \mathcal{S}', e_1 \,\mathsf{with}\, \{a = e_2'\}}$$
$$\frac{e_2 \in RVal}{\mathcal{S}, \{a_j : v_j\}_1^m \,\mathsf{with}\, \{a = e_2\} \xrightarrow{\mathcal{R}} \mathcal{S}, \{a_j : v_j\}_1^m \Leftarrow \{a : e_2\}}$$

ERACC
$$\frac{\mathcal{S}, e \xrightarrow{\mathcal{R}} \mathcal{S}', e'}{\mathcal{S}, e \cdot a \xrightarrow{\mathcal{R}} \mathcal{S}', e' \cdot a} \qquad \mathcal{S}, \{a_j : v_j\}_1^m \cdot a_i \xrightarrow{\mathcal{R}} \mathcal{S}, v_i$$

ERREF
$$\frac{\mathcal{S}, e \xrightarrow{\mathcal{R}} \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\ e \xrightarrow{\mathcal{R}} \mathcal{S}', \mathsf{ref}\ e'} \qquad \frac{e \in RVal \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\ e \xrightarrow{\mathcal{R}} \mathcal{S} \Leftarrow \{\ell : e\}, \ell}$$

ERDEREF
$$\frac{\mathcal{S}, e \xrightarrow{\mathcal{R}} \mathcal{S}', e'}{\mathcal{S}, !e \xrightarrow{\mathcal{R}} \mathcal{S}', !e'} \qquad \frac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \xrightarrow{\mathcal{R}} \mathcal{S}, v}$$

ERASGN
$$\frac{\mathcal{S}, e_1 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1'}{\mathcal{S}, e_1 := e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1' := e_2} \qquad \frac{e_1 \in RVal \quad \mathcal{S}, e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_2'}{\mathcal{S}, e_1 := e_2 \xrightarrow{\mathcal{R}} \mathcal{S}', e_1 := e_2'}$$
$$\frac{e_2 \in RVal}{\mathcal{S}, \ell := e_2 \xrightarrow{\mathcal{R}} \mathcal{S} \Leftarrow \{\ell : e_2\}, e_2}$$

Figure 4: The operational semantics of record calculus with references.

- Converting a quoted expression to a lambda expression, where the input will be a record representing the environment into which the quoted expression is plugged.
- Converting an antiquoted expression to a function application where the argument is a record that represents the accumulated environment.
- Converting a variable to a lookup operation in the environment, if the binding of the variable cannot be resolved.
- Making the evaluation order explicit by taking out the antiquoted expressions occurring inside a quoted expression, and placing them outside the lambda abstraction to which the quoted expression is translated. An illustration is given in Figure 5. The (translations of the) antiquoted expressions are then passed as arguments to the lambda expression that represents the quoted expression. This way, call-by-value semantics of the record calculus ensures that the antiquoted expression will be evaluated before being plugged into a quoted expression.

A translation of staged expressions to record calculus was previously discussed by Kameyama, Kiselyov and Shan [4]. However, a formal definition was not given. Instead, a translation to System F with tuples is provided. A translation that makes the order of evaluation explicit by taking out

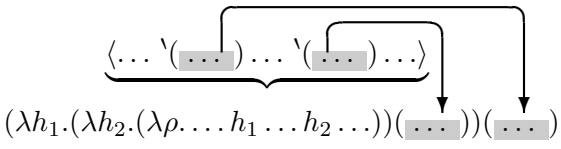

Figure 5: Illustration of the translation for a fragment with two holes.

antiquoted expressions from inside quoted expressions was previously given by Davies and Pfenning [3]. The translation in Figure 6 follows the same principles of Davies and Pfenning's translation; in addition, it performs conversion to record expressions.

The translation replaces an antiquotation with a function application. The operator of the application is a freshly generated variable, say h, that stands for the quoted expression that will fill the hole; the operand is the accumulated environment. The translation converts the quotation surrounding the antiquotation to a lambda abstraction, which receives the value for the freshly generated variable h. For example, $\langle 1 + \text{‵}(\langle 2 \rangle) \rangle$ is translated to $(\lambda h.\lambda\rho.1 + h\ \rho)(\lambda\rho.2)$. If there are two holes inside a quotation, the translation contains one more lambda abstraction. For example, $\langle 1 + \text{‵}(\langle 2 \rangle) + \text{‵}(\langle 3 \rangle) \rangle$ is translated to $(\lambda h.(\lambda h'.(\lambda\rho.1 + h\ \rho + h'\ \rho))(\lambda\rho.3))(\lambda\rho.2)$. Note that the order of evaluation stays the same. To do this conversion, each hole during the translation corresponds to a context in the form $(\lambda h.[\,])e$, where h is the freshly generated variable that replaces the hole, and e is the plug. Contexts can be nested in the case of multiple holes. For example, in the latter example above, the context is $(\lambda h.(\lambda h'.[\,])(\lambda\rho.3))(\lambda\rho.2)$. The formal definition of contexts is below.

Definition 2.9 (Context)**.** A context is defined as follows:

$$K \in \mathit{Context} \quad ::= \quad (\lambda h.[\,])e \mid (\lambda h.K)e$$

$K[e]$ denotes the expression where the hole inside the context K has been replaced with the expression e. A list of contexts $K_i :: K_{i+1} :: \ldots :: K_j$ is denoted as κ_i^j. If the length of the list is not relevant, we simply use κ. Note that κ may stand for `nil`, the empty context list, in which case $j < i$. Throughout the paper we assume that the variable h abstracted by a context is unique. Because the translation freshly generates h, this property is preserved.

The definition of the translation is in Figure 6. The result of translating an expression is a tuple, whose first element is the transformed expression. The second is a list of contexts, where each context corresponds to an antiquoted expression at a certain stage. The stages go "deeper" (i.e. the stage number decreases) as we go from left to right in the context list. For example, assuming the environment at stage 1 is represented by ρ_1, stage 2 is represented by ρ_2, and y is a bound variable, the translation of $\text{‵}(x) + \text{‵}(\text{‵}(y))$ at stage 2 yields $(h_1\ \rho_2 + h_2\ \rho_2,\ (\lambda h_1.(\lambda h_2.[\,])(h_3\ \rho_1))(\rho_1 x) :: (\lambda h_3.[\,])(y))$.

The result of a translation at any level can be packed into a single record calculus expression by filling in the contexts in order. This is done by the *Close* operation.

Definition 2.10. $\mathit{Close} : (\mathit{RExp} \times \mathit{Context}\ \text{list}) \to \mathit{RExp}$ is defined as below.

$$\begin{aligned} \mathit{Close}(e, \texttt{nil}) &= e \\ \mathit{Close}(e, \kappa_i^j) &= K_j[K_{j-1}[\cdots K_{i+1}[K_i[e]]\cdots]] \end{aligned}$$

Note that the following statements hold.

$$[\![\_]\!]_{\_} : (Exp \times RenamingEnv \text{ list}) \rightharpoonup (RExp \times Context \text{ list})$$

$$[\![c]\!]_{\vec{R}_n} = (c, \texttt{nil}) \qquad [\![x]\!]_{\vec{R}_n} = (R_n(x), \texttt{nil})$$

$$\frac{[\![e]\!]_{(\vec{R}_{n-1}, R_n \text{ with } \{x=z\})} = (\underline{e}, \kappa) \quad z \in Var \text{ is fresh}}{[\![\lambda x.e]\!]_{\vec{R}_n} = (\lambda_x z.\underline{e}, \kappa)}$$

$$\frac{[\![e]\!]_{(\vec{R}_{n-1}, R_n \text{ with } \{f=g, x=z\})} = (\underline{e}, \kappa) \quad g, z \in Var \text{ are fresh}}{[\![\mathsf{fix}\, f(x).\, e]\!]_{\vec{R}_n} = (\mathsf{fix}_{f,x}\ g(z).\, \underline{e}, \kappa)}$$

$$\frac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad [\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')}{[\![e_1 e_2]\!]_{\vec{R}_n} = (\underline{e}_1 \underline{e}_2, zip(\kappa, \kappa'))}$$

$$\frac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad [\![e_2]\!]_{(\vec{R}_{n-1}, R_n \text{ with } \{x=z\})} = (\underline{e}_2, \kappa') \quad z \in Var \text{ is fresh}}{[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!]_{\vec{R}_n} = (\mathsf{let}_x\ z = \underline{e}_1 \,\mathsf{in}\, \underline{e}_2, zip(\kappa, \kappa'))}$$

$$\frac{[\![e]\!]_{\vec{R}_n, \rho} = (\underline{e}, K :: \kappa) \quad \rho \in RVar \text{ is fresh}}{[\![\langle e \rangle]\!]_{\vec{R}_n} = (K[\lambda\rho.\underline{e}], \kappa)} \qquad \frac{[\![e]\!]_{\vec{R}_n, \rho} = (\underline{e}, \texttt{nil}) \quad \rho \in RVar \text{ is fresh}}{[\![\langle e \rangle]\!]_{\vec{R}_n} = (\lambda\rho.\underline{e}, \texttt{nil})}$$

$$\frac{[\![e]\!]_{\vec{R}_n} = (\underline{e}, \kappa) \quad h \in Var \text{ is fresh}}{[\![\backprime(e)]\!]_{\vec{R}_n, R_{n+1}} = (h(R_{n+1}), ((\lambda h.[\,])\underline{e}) :: \kappa)}$$

$$\frac{[\![e]\!]_{\vec{R}_n} = (\underline{e}, \kappa) \quad h \in Var \text{ is fresh}}{[\![\mathsf{run}(e)]\!]_{\vec{R}_n} = (\mathsf{let}\, h = \underline{e} \,\mathsf{in}\, h\,\{\}, \kappa)}$$

$$\frac{[\![e]\!]_{\vec{R}_n} = (\underline{e}, \kappa) \quad h \in Var \text{ and } \rho \in RVar \text{ are fresh}}{[\![\mathsf{lift}(e)]\!]_{\vec{R}_n} = (\mathsf{let}\, h = \underline{e} \,\mathsf{in}\, \lambda\rho.h, \kappa)}$$

$$[\![\ell]\!]_{\vec{R}_n} = (\ell, \texttt{nil}) \qquad \frac{[\![e]\!]_{\vec{R}_n} = (\underline{e}, \kappa)}{[\![\mathsf{ref}\ e]\!]_{\vec{R}_n} = (\mathsf{ref}\ \underline{e}, \kappa)}$$

$$\frac{[\![e]\!]_{\vec{R}_n} = (\underline{e}, \kappa)}{[\![!e]\!]_{\vec{R}_n} = (!\underline{e}, \kappa)} \qquad \frac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad [\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')}{[\![e_1 := e_2]\!]_{\vec{R}_n} = (\underline{e}_1 := \underline{e}_2, zip(\kappa, \kappa'))}$$

The *zip* function is defined below, where :: is the separator in a list:

$$\begin{aligned} zip(K :: \kappa, K' :: \kappa') &= K[K'] :: zip(\kappa, \kappa') \\ zip(\texttt{nil}, \kappa) &= \kappa \\ zip(\kappa, \texttt{nil}) &= \kappa \end{aligned}$$

Figure 6: Transformation modified to handle expressions with side-effects. The order of evaluation is preserved with this translation.

- $Close(e, K :: \kappa) = Close(K[e],\ \kappa)$
- $Close(e, \kappa :: K) = K[Close(e,\ \kappa)]$

Finally, the translation for stores is defined.

Definition 2.11 (Store translation)**.** Let $\mathcal{S} = \{\vec{\ell_k} : \vec{v_k}\}$ be a λ^{open}_{poly} store. Its translation to a λ^{rec}_{poly} store is defined as follows:

$$[\![\{\vec{\ell_k} : \vec{v_k}\}]\!] = \{\vec{\ell_k} : \vec{u_k}\} \text{ where } (u_i, \texttt{nil}) = [\![v_i]\!]_{\{\}} \text{ for all } i \text{ s.t. } 0 \leq i \leq k$$

Note that all the values in $\mathcal{S}$ are stage-0 values, and the second item of the result of translating a stage-0 value is always `nil`.

3 Relation Between the Two Operational Semantics

In this section we state the relation between the two calculi. For this purpose, we first define "admin" reductions in the record calculus. In [1], we had defined a collection of "safe" reductions that are guaranteed not to modify the store (i.e. they are side-effect-free). That definition is broader than needed; we fine-tune the reductions here.

Definition 3.1 (Admin reductions)**.** An admin reduction from expression e to the expression e', denoted $e \xrightarrow{\mathcal{A}} e'$, is the congruence closure of the rules below.

$$\begin{array}{rcl} (\lambda\rho.e)R & \xrightarrow{\mathcal{A}} & e\{\rho \backslash R\} \text{ where } R \in RenamingEnv \\ R{\cdot}y & \xrightarrow{\mathcal{A}} & R(y) \end{array}$$

The Kleene closure of admin reductions is denoted as $\xrightarrow{\mathcal{A}^*}$. Note that the expression an admin reduction simplifies (i.e. a renaming environment) does not have side effects. Therefore it is safe to perform admin reductions in the presence of side effects.

Notation 3.2. An admin reduction is straightforwardly extended to include stores. That is, $e \xrightarrow{\mathcal{A}} e'$ iff $\mathcal{S}, e \xrightarrow{\mathcal{A}} \mathcal{S}, e'$ for any store $\mathcal{S}$.

We define the following notation to express translations and reductions using a uniform "arrow" notation.

Notation 3.3. We use the notation $\mathcal{S}, e \xrightarrow{[\![\cdot]\!]_{\vec{R}_n}} \underline{\mathcal{S}}, \underline{e}$ iff $Close([\![e]\!]_{\vec{R}_n}) = \underline{e}$ and $[\![\mathcal{S}]\!] = \underline{\mathcal{S}}$.

Notation 3.4. The semicolon notation is used to denote the composition of reductions. We write $e \xrightarrow{A;B} e'$ iff there is an e'' such that $e \xrightarrow{A} e''$ and $e'' \xrightarrow{B} e'$. Note that the semicolon operator is associative.

The following is a key theorem that gives the semantic relation between the two calculi.

Theorem 3.5. *Let e_1 be a stage-n λ^{open}_{poly} expression with no free variables, such that $[\![e_1]\!]_{\vec{R}_n}$ is defined. Also let the values in the store $\mathcal{S}_1$ have no free variables. If $\mathcal{S}_1, e_1 \xrightarrow{n} \mathcal{S}_2, e_2$, then $\mathcal{S}_1, e_1 \xrightarrow{[\![\cdot]\!]_{\vec{R}_n}; \mathcal{R}; \mathcal{A}^*} [\![\mathcal{S}_2]\!], Close([\![e_2]\!]_{\vec{R}_n})$. This is illustrated by the diagram below.*

$$\begin{array}{ccc} \mathcal{S}_1, e_1 & \xrightarrow{\quad n \quad} & \mathcal{S}_2, e_2 \\ {\scriptstyle [\![\cdot]\!]_{\vec{R}_n}}\downarrow & & \downarrow{\scriptstyle [\![\cdot]\!]_{\vec{R}_n}} \\ [\![\mathcal{S}_1]\!], \underline{e}_1 & \xrightarrow{\mathcal{R}; \mathcal{A}^*} & [\![\mathcal{S}_2]\!], \underline{e}_2 \end{array}$$

Proof. Given in the Appendix. □

Finally, the following lemma states that there are no opportunities for admin reductions in the result of a translation. This means that the admin reductions performed in Theorem 3.5 above are *exhaustive*. In other words, by applying admin reductions exhaustively, we will reach $Close([\![e_2]\!]_{\vec{R}_n})$ without worrying about over-simplification.

Lemma 3.6. *Let e be a λ^{open}_{poly} expression such that $[\![e]\!]_{\vec{R}_n}$ is defined. There does <u>not</u> exist an e' such that $Close([\![e]\!]_{\vec{R}_n}) \xrightarrow{\mathcal{A}} e'$.*

Proof. By a straightforward structural induction. □

This completes the operational relation between the staged and the record calculi. Next, we discuss how to convert record calculus expressions to staged expressions.

4 Reverse Translation

In this section we define a reverse translation that transforms record expressions back to the staged calculus. The definition is given in Figure 7.

For the reverse translation, we interpret contexts as functions:

Definition 4.1. Let κ be a list of contexts. $\overline{\kappa}$ denotes a function from variables to expressions defined as follows.

$$\begin{array}{rcl} \overline{\texttt{nil}} & = & \varnothing \\ \overline{(\lambda h.[\,])e :: \kappa'} & = & \{h : e\} \cup \overline{\kappa'} \\ \overline{(\lambda h.K)e :: \kappa'} & = & \{h : e\} \cup \overline{K :: \kappa'} \end{array}$$

Theorem 4.2 states a key result of this paper: we can perform reverse translation on the translation of an expression to obtain the original expression.

Theorem 4.2. *Let the result of $[\![e]\!]_{\vec{R}_n}$ be $(\underline{e}, \kappa)$. Then $[\![\underline{e}, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = e$ for any $\mathcal{H} \supseteq \overline{\kappa}$.*

Proof. By induction on the structure of e. We show representative cases below. Other cases either follow straightforward from the induction hypothesis or are very similar to a given case.

- Case $e = x$.
  There are three subcases as investigated below. In these subcases, recall that $\overline{\texttt{nil}} = \varnothing$ and any $\mathcal{H} \supseteq \varnothing$.

$\mathcal{H} \in Var \rightharpoonup RExp$

$[\![\_, \_]\!]^{-1}_{\_} : (RExp \times (Var \rightharpoonup RExp) \times RenamingEnv \text{ list}) \rightharpoonup Exp$

$$[\![c, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = c$$
$$[\![x, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = R_n^{-1}(x)$$
$$[\![\rho{\cdot}x, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = x$$
$$[\![\lambda_x z.e, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \lambda x.[\![e, \mathcal{H}]\!]^{-1}_{(\vec{R}_{n-1}, R_n \text{ with } \{x=z\})}$$
$$[\![\mathsf{fix}_{f,x} g(z).\, e, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \mathsf{fix}\, f(x).\, [\![e, \mathcal{H}]\!]^{-1}_{(\vec{R}_{n-1}, R_n \text{ with } \{f=g, x=z\})}$$
$$[\![e_1 e_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = [\![e_1, \mathcal{H}]\!]^{-1}_{\vec{R}_n}\ [\![e_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n}$$
where $\nexists e'$ such that $e_1 = \lambda h.e'$, and $e_2 \notin RenamingEnv$.
$$[\![\mathsf{let}_x\ z = e_1 \text{ in } e_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \mathsf{let}\, x = [\![e_1, \mathcal{H}]\!]^{-1}_{\vec{R}_n} \text{ in } [\![e_2, \mathcal{H}]\!]^{-1}_{(\vec{R}_{n-1}, R_n \text{ with } \{x=z\})}$$
$$[\![(\lambda h.e)e', \mathcal{H}]\!]^{-1}_{\vec{R}_n} = [\![e, (\mathcal{H} \cup \{h : e'\})]\!]^{-1}_{\vec{R}_n}$$
$$[\![(\lambda \rho.e), \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \langle [\![e, \mathcal{H}]\!]^{-1}_{(\vec{R}_n, \rho)} \rangle$$
$$[\![h\ R, \mathcal{H}]\!]^{-1}_{(\vec{R}_n, R_{n+1})} = \text{`}([\![\mathcal{H}(h), \mathcal{H}]\!]^{-1}_{\vec{R}_n})$$
$$[\![\mathsf{let}\, h = e \text{ in } h\, \{\}, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \mathsf{run}([\![e, \mathcal{H}]\!]^{-1}_{\vec{R}_n})$$
$$[\![\mathsf{let}\, h = e \text{ in } \lambda \rho.h, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \mathsf{lift}([\![e, \mathcal{H}]\!]^{-1}_{\vec{R}_n})$$
$$[\![\ell, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \ell$$
$$[\![\mathsf{ref}\ e, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = \mathsf{ref}\ [\![e, \mathcal{H}]\!]^{-1}_{\vec{R}_n}$$
$$[\![!e, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = ![\![e, \mathcal{H}]\!]^{-1}_{\vec{R}_n}$$
$$[\![e_1 {:=} e_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = [\![e_1, \mathcal{H}]\!]^{-1}_{\vec{R}_n} {:=} [\![e_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n}$$

Figure 7: Transformation of record expressions to the staged calculus.

1. $R_n(x)$ is undefined (error case): Not possible. Otherwise $[\![e]\!]_{\vec{R}_n}$ would not be defined.
2. $R_n(x) = z$: This means $R_n^{-1}(z) = x$. Then, $[\![e]\!]_{\vec{R}_n} = (z, \texttt{nil})$, and by definition $[\![z, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = R_n^{-1}(z) = x$ for any $\mathcal{H}$.
3. $R_n(x) = \rho{\cdot}x$: Then, $[\![e]\!]_{\vec{R}_n} = (\rho{\cdot}x, \texttt{nil})$, and by definition $[\![\rho{\cdot}x, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = x$ for any $\mathcal{H}$.

- Case $e = e_1 e_2$.
We have
$$\frac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad [\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')}{[\![e_1 e_2]\!]_{\vec{R}_n} = (\underline{e}_1 \underline{e}_2, zip(\kappa, \kappa'))}$$
Using the inductive hypothesis and the fact that $\overline{zip(\kappa, \kappa')} \supseteq \overline{\kappa}$ and $\overline{zip(\kappa, \kappa')} \supseteq \overline{\kappa'}$, we have
$$[\![\underline{e}_1, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = e_1$$
$$[\![\underline{e}_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = e_2$$
for any $\mathcal{H} \supseteq \overline{zip(\kappa, \kappa')}$. Finally, by the definition of reverse translation
$$[\![\underline{e}_1 \underline{e}_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = [\![\underline{e}_1, \mathcal{H}]\!]^{-1}_{\vec{R}_n} [\![\underline{e}_2, \mathcal{H}]\!]^{-1}_{\vec{R}_n} = e_1 e_2$$

- Case $e = \langle e' \rangle$.
We have,
$$\frac{[\![e']\!]_{\vec{R}_n, \rho} = (\underline{e}, K :: \kappa) \quad \rho \in RVar \text{ is fresh}}{[\![\langle e' \rangle]\!]_{\vec{R}_n} = (K[\lambda\rho.\underline{e}], \kappa)}$$
Let $\mathcal{H}$ be any function such that $\mathcal{H} \supseteq \overline{\kappa}$. Note that we have $\overline{K} \cup \mathcal{H} \supseteq \overline{K} \cup \overline{\kappa} = \overline{K :: \kappa}$. Let $K = (\lambda h_1. \cdots (\lambda h_j.[\,]) e_j \cdots) e_1$ for some j. Then,
$$\begin{aligned}
[\![K[\lambda\rho.\underline{e}], \mathcal{H}]\!]^{-1}_{\vec{R}_n} &= [\![(\lambda h_1. \cdots (\lambda h_j.\lambda\rho.\underline{e}) e_j \cdots) e_1,\ \mathcal{H}]\!]^{-1}_{\vec{R}_n} \\
&= [\![(\lambda h_2. \cdots (\lambda h_j.\lambda\rho.\underline{e}) e_j \cdots) e_2,\ \{h_1 : e_1\} \cup \mathcal{H}]\!]^{-1}_{\vec{R}_n} \\
&\vdots \\
&= [\![\lambda\rho.\underline{e},\ \underbrace{\{h_1 : e_1, \ldots, h_j : e_j\}}_{\overline{K}} \cup \mathcal{H}]\!]^{-1}_{\vec{R}_n} \\
&= \langle [\![\underline{e},\ \overline{K} \cup \mathcal{H}]\!]^{-1}_{(\vec{R}_n, \rho)} \rangle \\
&= \langle e' \rangle \qquad \textit{by I.H.}
\end{aligned}$$

- Case $e = \backprime(e_0)$.
We have
$$\frac{[\![e_0]\!]_{\vec{R}_n} = (\underline{e}, \kappa) \quad h \in Var \text{ is fresh}}{[\![\backprime(e_0)]\!]_{\vec{R}_n, R_{n+1}} = (h\ R_{n+1}, (\lambda h.[\,])\underline{e} :: \kappa)}$$

Let $\mathcal{H}$ be any function such that $\mathcal{H} \supseteq \{h : \underline{e}\} \cup \overline{\kappa}$. Also note that we trivially have $\mathcal{H} \supseteq \overline{\kappa}$. So,

$$\begin{aligned}
[\![h\ R_{n+1}, \mathcal{H}]\!]^{-1}_{\vec{R}_n, R_{n+1}} &= \text{`}([\![\mathcal{H}(h), \mathcal{H}]\!]^{-1}_{\vec{R}_n}) \\
&= \text{`}([\![\underline{e}, \mathcal{H}]\!]^{-1}_{\vec{R}_n}) \\
&= \text{`}(e_0) \qquad\qquad \textit{by I.H.}
\end{aligned}$$

□

The following is a direct corollary of Theorem 4.2.

Corollary 4.3. *Let e be a λ^{gen}_{poly} expression and $[\![e]\!]_{\vec{R}_n} = (e', \kappa)$. The following hold:*

- $[\![e', \overline{\kappa}]\!]^{-1}_{\vec{R}_n} = e$
- $[\![Close([\![e]\!]_{\vec{R}_n}), \varnothing]\!]^{-1}_{\vec{R}_n} = e$

Finally, we state that, instead of evaluating a staged expression using the staged semantics, we can first translate it to the record calculus, then evaluate using record semantics and simplify via exhaustive admin reductions, and finally translate back to staged calculus to obtain the same result. For this purpose, we define reverse translation for stores, and also an arrow notation for the reverse translation for better illustration of the process. The statement is given in Corollary 4.6.

Definition 4.4 (Reverse store translation)**.** Let $\mathcal{S} = \{\vec{\ell_k} : \vec{v}_k\}$ be a λ^{rec}_{poly} store. Its translation to a λ^{open}_{poly} store is defined as follows:

$$[\![\{\vec{\ell_k} : \vec{v}_k\}]\!]^{-1} = \{\vec{\ell_k} : \vec{u}_k\} \text{ where } u_i = [\![v_i, \varnothing]\!]^{-1}_{\{\}} \text{ for all } i \text{ s.t. } 0 \le i \le k.$$

Notation 4.5. We use the notation $\underline{\mathcal{S}}, \underline{e} \xrightarrow{[\![\cdot]\!]^{-1}_{\vec{R}_n}} \mathcal{S}, e$ iff $[\![\underline{e}, \varnothing]\!]^{-1}_{\vec{R}_n} = e$ and $[\![\underline{\mathcal{S}}]\!]^{-1} = \mathcal{S}$.

Corollary 4.6. *Let e_1 be a stage-n λ^{open}_{poly} expression with no free variables, such that $[\![e_1]\!]_{\vec{R}_n}$ is defined and Also let the values in the store $\mathcal{S}_1$ have no free variables. $\mathcal{S}_1, e_1 \xrightarrow{n} \mathcal{S}_2, e_2$. Then, $\mathcal{S}_1, e_1 \xrightarrow{[\![\cdot]\!]_{\vec{R}_n}\,;\,\mathcal{R}\,;\,\mathcal{A}^*\,;\,[\![\cdot]\!]^{-1}_{\vec{R}_n}} \mathcal{S}_2, e_2$, where the admin reductions are performed exhaustively. The relation is illustrated by the following diagram.*

$$\begin{array}{ccc}
\mathcal{S}_1, e_1 & \xrightarrow{\ n\ } & \mathcal{S}_2, e_2 \\
{\scriptstyle [\![\cdot]\!]_{\vec{R}_n}}\downarrow & & \uparrow{\scriptstyle [\![\cdot]\!]^{-1}_{\vec{R}_n}} \\
[\![\mathcal{S}_1]\!], \underline{e}_1 & \xrightarrow{\ \mathcal{R}\,;\,\mathcal{A}^*\ } & [\![\mathcal{S}_2]\!], \underline{e}_2
\end{array}$$

5 Exponentiation Example

Let us now see the translation in action. Below is the program that generates an exponentiation function specialized for a fixed power value. This is a now-classic example that can be found in many papers related to staged computation. In this example, the evaluation returns the function $\lambda x.x \times x \times x \times 1$. We assume that our languages have been extended with if-expressions and arithmetic operations.

$$[\![\mathsf{let}\ pow = \mathsf{fix}\, gen(n).\, \mathsf{if}\ n = 0\ \mathsf{then}\ \langle 1\rangle \quad \mathsf{else}\ \langle x \times \backprime(gen(n-1))\rangle \quad \mathsf{in}\ \mathsf{run}\langle \lambda x.\backprime(pow\ 3)\rangle\ ]\!]_{\{\}}$$

The result of the translation to the record calculus is below. To allow easier comparison, we do not rename variables and we use annotations only for λx.

$$\mathsf{let}\ pow = \mathsf{fix}\, gen(n).\, \mathsf{if}\ n = 0\ \mathsf{then}\ \lambda\rho.1 \quad \mathsf{else}\ (\lambda h.(\lambda\rho.\ \rho{\cdot}x \times h\ \rho))(gen(n-1)) \quad \mathsf{in}\ \mathsf{let}\, h' = (\lambda h.\lambda\rho.\lambda_x x.h\ (\rho\,\mathsf{with}\,\{x = x\}))(pow\ 3)\,\mathsf{in}\, h'\ \{\}$$

Let us call the staged version G and its translation $\underline{G}$. There are many small steps of reduction in the evaluation of both. We take a look at two interesting sections (we ignore the store since there are no side effects):

- Take the step when $pow\ 3$ in G reduces to $\langle x \times \backprime(\langle x \times \backprime(\langle x \times \backprime(\langle 1\rangle)\rangle)\rangle)\rangle$. In three more steps of reduction (ESUBOX), we reach $\langle x \times x \times x \times 1\rangle$:

$$\begin{aligned}\langle x \times \backprime(\langle x \times \backprime(\langle x \times \backprime(\langle 1\rangle)\rangle)\rangle)\rangle &\xrightarrow{0} \langle x \times \backprime(\langle x \times \backprime(\langle x \times 1\rangle)\rangle)\rangle \\ &\xrightarrow{0} \langle x \times \backprime(\langle x \times x \times 1\rangle)\rangle \\ &\xrightarrow{0} \langle x \times x \times x \times 1\rangle\end{aligned}$$

  In $\underline{G}$, the same subexpression, $pow\ 3$ will have reduced to

$$(\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.1)))$$

  Note that this expression and $\langle x \times \backprime(\langle x \times \backprime(\langle x \times \backprime(\langle 1\rangle)\rangle)\rangle)\rangle$ translate to each other. Now, taking small step reductions and admin reductions in the record calculus:

$$\begin{aligned}&(\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.1))) \\ &\quad\xrightarrow{\mathcal{R}} (\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times (\lambda\rho.1)\ \rho)) \\ &\quad\xrightarrow{\mathcal{A}} (\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times 1)) \\ &\quad\xrightarrow{\mathcal{R}} (\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times (\lambda\rho.\rho{\cdot}x \times 1)\ \rho) \\ &\quad\xrightarrow{\mathcal{A}} (\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times 1) \\ &\quad\xrightarrow{\mathcal{R}} (\lambda\rho.\rho{\cdot}x \times (\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times 1)\ \rho) \\ &\quad\xrightarrow{\mathcal{A}} (\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times \rho{\cdot}x \times 1)\end{aligned}$$

  Note that the result of each admin reduction is translatable to the corresponding term in the staged reduction (and vice-versa):

$$\begin{aligned}&(\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)((\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times 1)) \xrightarrow{[\![\cdot]\!]^{-1}_{\{\}}} \langle x \times \backprime(\langle x \times \backprime(\langle x \times 1\rangle)\rangle)\rangle \\ &(\lambda h.\lambda\rho.\rho{\cdot}x \times h\ \rho)(\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times 1) \xrightarrow{[\![\cdot]\!]^{-1}_{\{\}}} \langle x \times \backprime(\langle x \times x \times 1\rangle)\rangle \\ &(\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times \rho{\cdot}x \times 1) \xrightarrow{[\![\cdot]\!]^{-1}_{\{\}}} \langle x \times x \times x \times 1\rangle\end{aligned}$$

- As the second case, let us continue with the evaluation of run in G:

$$\begin{aligned}\mathsf{run}(\lambda x.\,\backprime(\langle x \times x \times x \times 1\rangle)) &\xrightarrow{0} \mathsf{run}(\lambda x.x \times x \times x \times 1)\\ &\xrightarrow{0} \lambda x.x \times x \times x \times 1\end{aligned}$$

  The corresponding steps of the translation are

$$\begin{aligned}&\mathsf{let}\, h' = (\lambda h.\lambda\rho.\lambda_x x.\ h\ (\rho\,\mathsf{with}\,\{x = x\}))(\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times \rho{\cdot}x \times 1)\,\mathsf{in}\, h'\ \{\}\\ &\xrightarrow{\mathcal{R}} \mathsf{let}\, h' = (\lambda\rho.\lambda_x x.\ (\lambda\rho.\rho{\cdot}x \times \rho{\cdot}x \times \rho{\cdot}x \times 1)\ (\rho\,\mathsf{with}\,\{x = x\}))\,\mathsf{in}\, h'\ \{\}\\ &\xrightarrow{\mathcal{A}} \mathsf{let}\, h' = (\lambda\rho.\lambda_x x.\ (\rho\,\mathsf{with}\,\{x = x\}){\cdot}x \times (\rho\,\mathsf{with}\,\{x = x\}){\cdot}x \times (\rho\,\mathsf{with}\,\{x = x\}){\cdot}x \times 1)\,\mathsf{in}\, h'\ \{\}\\ &\xrightarrow{\mathcal{A}^*} \mathsf{let}\, h' = (\lambda\rho.\lambda_x x.\ x \times x \times x \times 1)\,\mathsf{in}\, h'\ \{\}\\ &\xrightarrow{\mathcal{R}} (\lambda\rho.\lambda_x x.\ x \times x \times x \times 1)\{\}\\ &\xrightarrow{\mathcal{A}} (\lambda_x x.\ x \times x \times x \times 1)\end{aligned}$$

  Note once more that the corresponding terms are translatable to each other.

6 Conclusion

We have identified two kinds of admin reductions to simplify a record expression. We have shown that record semantics, together with the admin reductions, is equivalent to staged semantics when we translate staged expressions to record expressions. We have also shown that the admin reductions can be applied exhaustively without worrying about over-simplification. Finally, we have given a reverse translation to convert record expressions back to staged expressions.

References

[1] Baris Aktemur. *Improving Efficiency and Safety of Program Generation.* PhD thesis, University of Illinois at Urbana-Champaign, 2009.

[2] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 275–286, New York, NY, USA, 2003. ACM.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[4] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Closing the stage: from staged code to typed closures. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 147–157, New York, NY, USA, 2008. ACM.

[5] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for java and its applications. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 48–56. IEEE Computer Society, 2003.

[6] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 257–268. ACM Press, 2006.

[7] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized metaml: Simpler, and more expressive. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[8] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[9] Morten Rhiger. First-class open and closed code fragments. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, pages 127–144, Tallinn, Estonia, 2005.

[10] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2003. ACM.

[11] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating aspectj programs with meta-aspectj. In G. Karsai and E. Visser, editors, *Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–18, Vancouver, Canada, October 2004. Springer.

A Proofs

Definition A.1 (Admin reduction of contexts)**.**

$$\begin{array}{rcl} e \xrightarrow{\mathcal{A}} e' & \iff & (\lambda h.[\,])e \xrightarrow{\mathcal{A}} (\lambda h.[\,])e' \\ e \xrightarrow{\mathcal{A}} e' & \iff & (\lambda h.K[\,])e \xrightarrow{\mathcal{A}} (\lambda h.K[\,])e' \\ K \xrightarrow{\mathcal{A}} K' & \iff & (\lambda h.K[\,])e \xrightarrow{\mathcal{A}} (\lambda h.K'[\,])e \end{array}$$

Definition A.2 (Admin reduction of context lists)**.**

$$\begin{array}{rcl} \texttt{nil} \xrightarrow{\mathcal{A}} \texttt{nil} & & \\ K :: \kappa \xrightarrow{\mathcal{A}} K' :: \kappa & \iff & K \xrightarrow{\mathcal{A}} K' \\ K :: \kappa \xrightarrow{\mathcal{A}} K :: \kappa' & \iff & \kappa \xrightarrow{\mathcal{A}} \kappa' \end{array}$$

Notation A.3 (Substitution in contexts)**.**

$$\begin{array}{rcl} ((\lambda h.[\,])e)\{h'\backslash e'\} & = & (\lambda h.[\,])(e\{h'\backslash e'\}) \\ ((\lambda h.K[\,])e)\{h'\backslash e'\} & = & (\lambda h.K\{h'\backslash e'\}[\,])(e\{h'\backslash e'\}) \text{ if } h \neq h' \end{array}$$

Proof of Theorem 3.5. By a case analysis. Let $[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa)$ and $[\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')$.

- Case 1. $n = 0$:
  By Theorem A.4, $\kappa = \kappa' = \texttt{nil}$ and $[\![\mathcal{S}_1]\!], \underline{e}_1 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_2$. Note that $Close([\![e_1]\!]_{R_0}) = \underline{e}_1$ and $Close([\![e_2]\!]_{R_0}) = \underline{e}_2$. Hence, $[\![\mathcal{S}_1]\!], Close([\![e_1]\!]_{R_0}) \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], Close([\![e_1]\!]_{R_0})$.
- Case 2. $n > 0$:
  Let $\kappa = K_1 :: \ldots :: K_n$. We now have four subcases.
  - For some h, K and $\underline{e}_3$, we have $K_n = (\lambda h.K[\,])\underline{e}_3$ such that $\underline{e}_3 \in RVal$:
    1. $\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ — by Theorem A.4
    2. $\mathcal{S}_1 = \mathcal{S}_2$ — by Theorem A.4
    3. $(K_1 :: \ldots :: K_{n-1} :: K)\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ — by Theorem A.4
    4. Let $\kappa' = K'_1 :: \ldots :: K'_n$.
    5. $Close(\underline{e}_1, \kappa) = (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\underline{e}_3$
    6. $Close(\underline{e}_2, \kappa') = (K'_n[K'_{n-1}[\cdots K'_1[\underline{e}_2]\cdots]])\underline{e}_3$
    7. $[\![\mathcal{S}_1]\!], (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\underline{e}_3 \xrightarrow{\mathcal{R}} [\![\mathcal{S}_1]\!], (K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\{h\backslash\underline{e}_3\}$ — by ERAPP, because $\underline{e}_3 \in RVal$
    8. $(K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} (K'_n[K'_{n-1}[\cdots K'_1[\underline{e}_2]\cdots]])\underline{e}_3$ — by (1), (2) and because h is distinct
    9. $[\![\mathcal{S}_1]\!], Close(\underline{e}_1, \kappa) \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], Close(\underline{e}_2, \kappa')$ — by (5), (7), (8), (6), (2)
  - For some h and $\underline{e}_3$, we have $K_n = (\lambda h.[\,])\underline{e}_3$ such that $\underline{e}_3 \in RVal$:
    Similar to the case above.

– For some h, K and $\underline{e}_3$, we have $K_n = (\lambda h.K[\,])\underline{e}_3$ such that $\underline{e}_3 \notin RVal$:

1. $\exists \underline{e}_4$ such that $[\![\mathcal{S}_1]\!], \underline{e}_3 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_4$ by Theorem A.4
2. $\kappa' = K_1 :: \ldots :: K_{n-1} :: (\lambda h.K[\,])\underline{e}_4$ by Theorem A.4
3. $\underline{e}_1 = \underline{e}_2$ by Theorem A.4
4. $Close(\underline{e}_1, \kappa) = (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\underline{e}_3$
5. $Close(\underline{e}_2, \kappa') = (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_2]\cdots]])\underline{e}_4$
6. $[\![\mathcal{S}_1]\!], (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\underline{e}_3 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], (\lambda h.K[K_{n-1}[\cdots K_1[\underline{e}_1]\cdots]])\underline{e}_4$ by (1)
7. $[\![\mathcal{S}_1]\!], Close(\underline{e}_1, \kappa) \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], Close(\underline{e}_2, \kappa')$ by (4), (6), (3), (5)

– For some h and $\underline{e}_3$, we have $K_n = (\lambda h.[\,])\underline{e}_3$ such that $\underline{e}_3 \notin RVal$:
Similar to the case above. □

Theorem A.4. *Let e_1 be a stage-n λ^{open}_{poly} expression with no free variables, the values in the store $\mathcal{S}_1$ have no free variables, and $\mathcal{S}_1, e_1 \xrightarrow{n} \mathcal{S}_2, e_2$. Let $[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa)$ and $[\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')$. Then*

- *The length of κ is n.*
- *If $n = 0$ then $[\![\mathcal{S}_1]\!], \underline{e}_1 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_2$ and $\kappa' = \texttt{nil}$.*
- *If $n > 0$ then $\exists \kappa'', K''$ such that $\kappa = \kappa'' :: K''$ and*
  - *if $K'' = (\lambda h.K[\,])\underline{e}_3$ for some h, K and $\underline{e}_3$ such that $\underline{e}_3 \in RVal$, then*
    - *$\underline{e}_1\{h \backslash \underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ and*
    - *$(\kappa'' :: K)\{h \backslash \underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ and*
    - *$\mathcal{S}_1 = \mathcal{S}_2$.*
  - *if $K'' = (\lambda h.[\,])\underline{e}_3$ for some h and $\underline{e}_3$ such that $\underline{e}_3 \in RVal$, then*
    - *$\underline{e}_1\{h \backslash \underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ and*
    - *$(\kappa'')\{h \backslash \underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ and*
    - *$\mathcal{S}_1 = \mathcal{S}_2$.*
  - *if $K'' = (\lambda h.K[\,])\underline{e}_3$ for some h, K and $\underline{e}_3$ such that $\underline{e}_3 \notin RVal$, then $\exists \underline{e}_4$ such that*
    - *$[\![\mathcal{S}_1]\!], \underline{e}_3 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_4$ and*
    - *$\kappa' = \kappa'' :: (\lambda h.K[\,])\underline{e}_4$ and*
    - *$\underline{e}_1 = \underline{e}_2$.*
  - *if $K'' = (\lambda h.[\,])\underline{e}_3$ for some h and $\underline{e}_3$ such that $\underline{e}_3 \notin RVal$, then $\exists \underline{e}_4$ such that*
    - *$[\![\mathcal{S}_1]\!], \underline{e}_3 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_4$ and*
    - *$\kappa' = \kappa'' :: (\lambda h.[\,])\underline{e}_4$ and*
    - *$\underline{e}_1 = \underline{e}_2$.*

Proof. By induction on the structure of e_1, based on the last applied reduction. We only show interesting cases; other cases are either straightforward or very similar to a given case.

- Case ESAPP(1): We have

$$\frac{\mathcal{S}, e_1 \xrightarrow{n} \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \xrightarrow{n} \mathcal{S}', e_1' e_2} \quad \text{and} \quad \frac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad [\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')}{[\![e_1 e_2]\!]_{\vec{R}_n} = (\underline{e}_1 \underline{e}_2, zip(\kappa, \kappa'))}$$

  This case follows from the I.H. The following facts are used:

  - If $\kappa \xrightarrow{\mathcal{A}^*} \kappa_1$ and $\kappa' \xrightarrow{\mathcal{A}^*} \kappa_2$, then $zip(\kappa, \kappa') \xrightarrow{\mathcal{A}^*} zip(\kappa_1, \kappa_2)$.
  - The outermost context of the rightmost context in κ is also the outermost context of the rightmost context in $zip(\kappa, \kappa')$.
  - Note that $length(\kappa) = depth(e_1)$ and $length(\kappa') = depth(e_2)$. Also, $depth(e_1 e_2) = max(depth(e_1), depth(e_2))$ by definition, and $length(zip(\kappa, \kappa')) = max(length(\kappa), length(\kappa'))$. Hence, $length(zip(\kappa, \kappa')) = depth(e_1 e_2)$.

- Case ESAPP(3): We have

$$\frac{e_2 \in Val^0}{\mathcal{S}, (\lambda x.e_1) e_2 \xrightarrow{0} \mathcal{S}, e_1\{x \backslash e_2\}^0} \quad \text{and} \quad \frac{\dfrac{[\![e_1]\!]_{R_0 \text{ with } \{x=z\}} = (\underline{e}_1, \texttt{nil}) \quad z \text{ is fresh}}{[\![\lambda x.e_1]\!]_{R_0} = (\lambda_x z.\underline{e}_1,\ \texttt{nil})} \quad [\![e_2]\!]_{R_0} = (\underline{e}_2,\ \texttt{nil})}{[\![(\lambda x.e_1) e_2]\!]_{R_0} = ((\lambda_x z.\underline{e}_1)\underline{e}_2,\ \texttt{nil})}$$

  Also, let $[\![e_1\{x \backslash e_2\}^0]\!]_{R_0} = (\underline{e}', \texttt{nil})$. Because $e_2 \in Val^0$, we have $\underline{e}_2 \in RVal$ by Lemma A.11. Then, at the record semantics side

$$[\![\mathcal{S}]\!], (\lambda_x z.\underline{e}_1)\underline{e}_2 \xrightarrow{\mathcal{R}} [\![\mathcal{S}]\!], \underline{e}_1\{z \backslash \underline{e}_2\}$$

  Note that $Close([\![e_1]\!]_{R_0 \text{ with } \{x=z\}}) = \underline{e}_1$ and $Close([\![e_2]\!]_{R_0}) = \underline{e}_2$. So,

$$\begin{array}{rcll} \underline{e}_1\{z \backslash \underline{e}_2\} & = & Close([\![e_1]\!]_{R_0 \text{ with } \{x=z\}})\{z \backslash Close([\![e_2]\!]_{R_0})\} & \\ & = & Close([\![e_1]\!]_{R_0 \text{ with } \{x=z\}})\{z \backslash Close([\![e_2]\!]_{\{\}})\} & \text{by Lemma A.7} \\ & = & Close([\![e_1\{x \backslash e_2\}^0]\!]_{R_0 \text{ with } \{x=z\}}) & \text{by Lemma A.6} \\ & = & Close([\![e_1\{x \backslash e_2\}^0]\!]_{R_0}) & \text{by Lemma A.7} \\ & = & \underline{e}' & \end{array}$$

  Hence, $[\![\mathcal{S}]\!], (\lambda_x z.\underline{e}_1)\underline{e}_2 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}]\!], \underline{e}'$.

- Case ESBOX: We have $\dfrac{\mathcal{S}_1, e_1 \xrightarrow{n+1} \mathcal{S}_2, e_2}{\mathcal{S}_1, \langle e_1 \rangle \xrightarrow{n} \mathcal{S}_2, \langle e_2 \rangle}$. Because e_1 takes a reduction, its depth is $n+1$; otherwise it would be a value and no reduction could be taken. So, for the translation we have

$$\frac{[\![e_1]\!]_{\vec{R}_n, \rho} = (\underline{e}_1, K_0 :: \kappa) \quad \rho \text{ is fresh}}{[\![\langle e_1 \rangle]\!]_{\vec{R}_n} = (K_0[\lambda \rho.\underline{e}_1], \kappa)}$$

  Also let $[\![e_2]\!] = (\underline{e}_2, \kappa')$.

  First, note that, by I.H., $length(K_0 :: \kappa) = n + 1$. Therefore, $length(\kappa) = n$.

  Case 1. $n = 0$:
  In this case $\kappa = \texttt{nil}$.

- Case 1.1: $K_0 = (\lambda h.K[\ ])\underline{e}_3$ and $\underline{e}_3 \in RVal$.
  1. $\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ by I.H.
  2. $K\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ by I.H.
  3. $\mathcal{S}_1 = \mathcal{S}_2$ by I.H.
  4. Let $\kappa' = K'$.
  5. $[\![\langle e_2\rangle]\!]_{R_0} = (K'[\lambda\rho.\underline{e}_2], \texttt{nil})$ by (4)
  6. $K_0[\lambda\rho.\underline{e}_1] = (\lambda h.K[\lambda\rho.\underline{e}_1])\underline{e}_3$
  7. $[\![\mathcal{S}_1]\!], (\lambda h.K[\lambda\rho.\underline{e}_1])\underline{e}_3 \xrightarrow{\mathcal{R}} [\![\mathcal{S}_1]\!], (K[\lambda\rho.\underline{e}_1])\{h\backslash\underline{e}_3\}$ because $\underline{e}_3 \in RVal$
  8. $(K[\lambda\rho.\underline{e}_1])\{h\backslash\underline{e}_3\} = K\{h\backslash\underline{e}_3\}[\lambda\rho.\underline{e}_1\{h\backslash\underline{e}_3\}]$ because h is distinct
  9. $K\{h\backslash\underline{e}_3\}[\lambda\rho.\underline{e}_1\{h\backslash\underline{e}_3\}] \xrightarrow{\mathcal{A}^*} K'[\lambda\rho.\underline{e}_2]$ by (1), (2), (4)
  10. $[\![\mathcal{S}_1]\!], K_0[\lambda\rho.\underline{e}_1] \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_1]\!], K'[\lambda\rho.\underline{e}_2]$ by (6), (7), (8), (9)
  11. $[\![\mathcal{S}_1]\!], K_0[\lambda\rho.\underline{e}_1] \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], K'[\lambda\rho.\underline{e}_2]$ by (3) and (10)
- Case 1.2: $K_0 = (\lambda h.[\ ])\underline{e}_3$ and $\underline{e}_3 \in RVal$.
  1. $\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ by I.H.
  2. $\texttt{nil} \xrightarrow{\mathcal{A}^*} \kappa'$, hence $\kappa' = \texttt{nil}$ by I.H.
  3. $\mathcal{S}_1 = \mathcal{S}_2$ by I.H.
  4. $[\![\langle e_2\rangle]\!]_{R_0} = (\lambda\rho.\underline{e}_2, \texttt{nil})$ by (2)
  5. $K_0[\lambda\rho.\underline{e}_1] = (\lambda h.\lambda\rho.\underline{e}_1)\underline{e}_3$
  6. $[\![\mathcal{S}_1]\!], (\lambda h.\lambda\rho.\underline{e}_1)\underline{e}_3 \xrightarrow{\mathcal{R}} [\![\mathcal{S}_1]\!], (\lambda\rho.\underline{e}_1)\{h\backslash\underline{e}_3\}$ because $\underline{e}_3 \in RVal$
  7. $(\lambda\rho.\underline{e}_1)\{h\backslash\underline{e}_3\} = \lambda\rho.\underline{e}_1\{h\backslash\underline{e}_3\}$ because h is distinct
  8. $\lambda\rho.\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \lambda\rho.\underline{e}_2$ by (1)
  9. $[\![\mathcal{S}_1]\!], K_0[\lambda\rho.\underline{e}_1] \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_1]\!], \lambda\rho.\underline{e}_2$ by (5), (6), (7), (8)
  10. $[\![\mathcal{S}_1]\!], K_0[\lambda\rho.\underline{e}_1] \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \lambda\rho.\underline{e}_2$ by (3) and (9)
- Case 1.3: $K_0 = (\lambda h.K[\ ])\underline{e}_3$ and $\underline{e}_3 \notin RVal$.
  Straightforward from the I.H.
- Case 1.4: $K_0 = (\lambda h.[\ ])\underline{e}_3$ and $\underline{e}_3 \notin RVal$.
  Straightforward from the I.H.

Case 2. $n > 0$:
Let $\kappa = K_1 :: \ldots :: K_n$.

- Case 2.1: $K_n = (\lambda h.K[\ ])\underline{e}_3$ and $\underline{e}_3 \in RVal$.
  1. $\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ by I.H.
  2. $(K_0 :: K_1 :: \ldots :: K_{n-1} :: K)\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ by I.H.
  3. $\mathcal{S}_1 = \mathcal{S}_2$ by I.H.
  4. Let $\kappa' = (K'_0 :: K'_1 :: \ldots :: K'_{n-1} :: K'_n)$.
  5. Note that $[\![\langle e_1\rangle]\!]_{\vec{R}_n} = (K_0[\lambda\rho.\underline{e}_1], K_1 :: \ldots :: K_n)$
  6. and $[\![\langle e_2\rangle]\!]_{\vec{R}_n} = (K'_0[\lambda\rho.\underline{e}_2], K'_1 :: \ldots :: K'_n)$

7. We want to show that
   i. $(K_0[\lambda\rho.\underline{e}_1])\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} K_0'[\lambda\rho.\underline{e}_2]$ and
   ii. $(K_1 :: \ldots :: K_{n-1} :: K)\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} (K_1' :: \ldots :: K_n')$ and
   iii. $\mathcal{S}_1 = \mathcal{S}_2$
8. Item (i) is straightforward from (1), (2) and the fact that h is distinct. Item (ii) is straightforward from (2). Item (iii) is trivial from (3).

- Case 2.2: $K_n = (\lambda h.[\,])\underline{e}_3$ and $\underline{e}_3 \in RVal$.
  1. $\underline{e}_1\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \underline{e}_2$ by I.H.
  2. $(K_0 :: K_1 :: \ldots :: K_{n-1})\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} \kappa'$ by I.H.
  3. $\mathcal{S}_1 = \mathcal{S}_2$ by I.H.
  4. Let $\kappa' = (K_0' :: K_1' :: \ldots :: K_{n-1}')$.
  5. Note that $[\![\langle e_1\rangle]\!]_{\vec{R}_n} = (K_0[\lambda\rho.\underline{e}_1], K_1 :: \ldots :: K_n)$
  6. and $[\![\langle e_2\rangle]\!]_{\vec{R}_n} = (K_0'[\lambda\rho.\underline{e}_2], K_1' :: \ldots :: K_{n-1}')$
  7. We want to show that
     i. $(K_0[\lambda\rho.\underline{e}_1])\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} K_0'[\lambda\rho.\underline{e}_2]$ and
     ii. $(K_1 :: \ldots :: K_{n-1})\{h\backslash\underline{e}_3\} \xrightarrow{\mathcal{A}^*} (K_1' :: \ldots :: K_{n-1}')$ and
     iii. $\mathcal{S}_1 = \mathcal{S}_2$
  8. Item (i) is straightforward from (1), (2) and the fact that h is distinct. Item (ii) is straightforward from (2). Item (iii) is trivial from (3).
- Case 2.3: $K_n = (\lambda h.K[\,])\underline{e}_3$ and $\underline{e}_3 \notin RVal$.
  Straightforward from the I.H.
- Case 2.4: $K_n = (\lambda h.[\,])\underline{e}_3$ and $\underline{e}_3 \notin RVal$.
  Straightforward from the I.H.

- Case ESUBOX(1): We have $\dfrac{\mathcal{S}_1, e_1 \xrightarrow{n} \mathcal{S}_2, e_2}{\mathcal{S}_1, \text{`}(e_1) \xrightarrow{n+1} \mathcal{S}_2, \text{`}(e_2)}$ and $\dfrac{[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, \kappa) \quad h \text{ is fresh}}{[\![\text{`}(e_1)]\!]_{\vec{R}_n, R_{n+1}} = (h\, R_{n+1}, (\lambda h.[\,])\underline{e}_1 :: \kappa)}$

  Let $[\![e_2]\!]_{\vec{R}_n} = (\underline{e}_2, \kappa')$. Then, $[\![\text{`}(e_2)]\!]_{\vec{R}_n, R_{n+1}} = (h\, R_{n+1}, (\lambda h.[\,])\underline{e}_2 :: \kappa')$.

  First, note that by I.H. $length(\kappa) = n$. Therefore $length((\lambda h.[\,])\underline{e}_1 :: \kappa) = n + 1$.

  Case 1. $n = 0$:
  In this case $\kappa = \kappa' = \texttt{nil}$.
  1. $[\![\mathcal{S}_1]\!], \underline{e}_1 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_2$ by I.H.
  2. Because e_1 takes a step of evaluation, $e_1 \notin Val^0$. Hence, $\underline{e}_1 \notin RVal$. by Lemma A.11
  3. What we need to show is $\exists \underline{e}_4$ such that
     (a) $[\![\mathcal{S}_1]\!], \underline{e}_1 \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}_2]\!], \underline{e}_4$
     (b) $(\lambda h.[\,])\underline{e}_2 = (\lambda h.[\,])\underline{e}_4$
     (c) $h\, R_{n+1} = h\, R_{n+1}$
  4. Take $\underline{e}_4 = \underline{e}_2$. Then, item (i) is satisfied by (1); item (ii) and (iii) are satisfied trivially.

Case 2. $n > 0$:
This case follows straightforward from the I.H.

- Case ESUBOX(2): We have $\dfrac{e \in Val^1}{\mathcal{S},\, \backprime(\langle e \rangle) \xrightarrow{1} \mathcal{S}, e}$. Because e is a value, for fresh ρ and h, the translation is

$$\dfrac{\dfrac{[\![e]\!]_{R_0,\rho} = (\underline{e}, \mathtt{nil})}{[\![\langle e \rangle]\!]_{R_0} = (\lambda\rho.\underline{e}, \mathtt{nil})}}{[\![\backprime(\langle e \rangle)]\!]_{R_0,R_1} = (h\, R_1,\ (\lambda h.[\,])(\lambda\rho.\underline{e}))}$$

  First, note that $length((\lambda h.[\,])(\lambda\rho.\underline{e})) = 1$.
  Let $[\![e]\!]_{R_0,R_1} = (\underline{e}', \mathtt{nil})$. Because $\lambda\rho.\underline{e} \in RVal$, what we need to show are

  i. $(h\, R_1)\{h \backslash (\lambda\rho.\underline{e})\} \xrightarrow{\mathcal{A}^*} \underline{e}'$
  ii. $\mathtt{nil} \xrightarrow{\mathcal{A}^*} \mathtt{nil}$
  iii. $\mathcal{S} = \mathcal{S}$

  Items (ii) and (iii) are trivial. Now we show that item (i) holds:

$$\begin{array}{rcl} (h\, R_1)\{h \backslash (\lambda\rho.\underline{e})\} & = & (\lambda\rho.\underline{e})R_1 \\ & \xrightarrow{\mathcal{A}} & \underline{e}\{\rho \backslash R_1\} \end{array}$$

  Using the fact that $Close([\![e]\!]_{R_0,\rho}) = \underline{e}$, we obtain $(\lambda\rho.\underline{e})R_1 \xrightarrow{\mathcal{A}} Close([\![e]\!]_{R_0,\rho})\{\rho \backslash R_1\}$. By Lemma A.8, $Close([\![e]\!]_{R_0,\rho})\{\rho \backslash R_1\} \xrightarrow{\mathcal{A}^*} Close([\![e]\!]_{R_0,R_1})$. Note that $Close([\![e]\!]_{R_0,R_1}) = \underline{e}'$. Therefore, $(h\, R_1)\{h \backslash (\lambda\rho.\underline{e})\} \xrightarrow{\mathcal{A}^*} \underline{e}'$.

- Case ESRUN(2): We have $\dfrac{e \in Val^1}{\mathcal{S}, \mathsf{run}(\langle e \rangle) \xrightarrow{0} \mathcal{S}, e}$. Because e is a value, for fresh ρ and h, the translation is

$$\dfrac{\dfrac{[\![e]\!]_{R_0,\rho} = (\underline{e}, \mathtt{nil})}{[\![\langle e \rangle]\!]_{R_0} = (\lambda\rho.\underline{e}, \mathtt{nil})}}{[\![\mathsf{run}(\langle e \rangle)]\!]_{R_0} = (\mathsf{let}\, h = (\lambda\rho.\underline{e})\, \mathsf{in}\, h\, \{\},\ \mathtt{nil})}$$

  First, note that $length(\mathtt{nil}) = 0$.
  Let $[\![e]\!]_{R_0} = (\underline{e}', \mathtt{nil})$. What we need to show is

$$[\![\mathcal{S}]\!], \mathsf{let}\, h = (\lambda\rho.\underline{e})\, \mathsf{in}\, h\, \{\} \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}]\!], \underline{e}'$$

  By ERLET and an admin reduction we get

$$[\![\mathcal{S}]\!], \mathsf{let}\, h = (\lambda\rho.\underline{e})\, \mathsf{in}\, h\, \{\} \xrightarrow{\mathcal{R}} [\![\mathcal{S}]\!], (\lambda\rho.\underline{e})\{\} \xrightarrow{\mathcal{A}} [\![\mathcal{S}]\!], \underline{e}\{\rho \backslash \{\}\}$$

Using the fact that $Close([\![e]\!]_{R_0,\rho}) = \underline{e}$,

$$\begin{array}{rcll} \underline{e}\{\rho\backslash\{\}\} & = & Close([\![e]\!]_{R_0,\rho})\{\rho\backslash\{\}\} & \\ & \xrightarrow{\mathcal{A}^*} & Close([\![e]\!]_{R_0,\{\}}) & \text{by Lemma A.8} \\ & = & Close([\![e]\!]_{\{\},\{\}}) & \text{by Lemma A.7} \\ & = & Close([\![e]\!]_{\{\}}) & \text{by Lemma A.10} \\ & = & Close([\![e]\!]_{R_0}) & \text{by Lemma A.7} \\ & = & \underline{e}' & \end{array}$$

- Case ESLIFT(2): We have $\dfrac{e \in Val^0}{\mathcal{S}, \mathsf{lift}(e) \xrightarrow{0} \mathcal{S}, \langle e \rangle}$. Because e is a value, for fresh ρ and h, the translation is

$$\frac{[\![e]\!]_{R_0} = (\underline{e}, \texttt{nil})}{[\![\mathsf{lift}(e)]\!]_{R_0} = (\mathsf{let}\, h = \underline{e}\, \mathsf{in}\, \lambda\rho.h,\ \texttt{nil})}$$

  First, note that $length(\texttt{nil}) = 0$.

  Also note that

$$\begin{array}{rcll} [\![e]\!]_{R_0,\rho} & = & [\![e]\!]_{\{\},\rho} & \text{by Lemma A.7} \\ & = & [\![e]\!]_{\rho} & \text{by Lemma A.10} \\ & = & [\![e]\!]_{R_0} & \text{by Lemma A.7} \\ & = & (\underline{e}, \texttt{nil}) & \end{array}$$

  Thus, as the translation of $\langle e \rangle$, we have

$$\frac{[\![e]\!]_{R_0,\rho} = (\underline{e}, \texttt{nil})}{[\![\langle e \rangle]\!]_{R_0} = (\lambda\rho.\underline{e}, \texttt{nil})}$$

  What we need to show is

$$[\![\mathcal{S}]\!], \mathsf{let}\, h = \underline{e}\, \mathsf{in}\, \lambda\rho.h \xrightarrow{\mathcal{R};\mathcal{A}^*} [\![\mathcal{S}]\!], \lambda\rho.\underline{e}$$

  which is immediate by ERLET. □

Lemma A.5. *Let e_1 be a stage-n λ^{gen}_{poly} expression and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. Let $[\![e_1]\!]_{\vec{R}_n} = (\underline{e}_1, K_1 :: \ldots :: K_p)$ and $[\![e_2]\!]_{\{\}} = (\underline{e}_2, \texttt{nil})$. Note that $p \leq n$. Assume $R_0(x) = z$. Then*

- *If $n = 0$ then $[\![e_1\{x\backslash e_2\}^0]\!]_{R_0} = (\underline{e}_1\{z\backslash\underline{e}_2\},\ \texttt{nil})$.*
- *If $n > 0$ and $p < n$ then $[\![e_1\{x\backslash e_2\}^n]\!]_{\vec{R}_n} = (\underline{e}_1,\ K_1 :: \ldots :: K_p)$ and $z \notin FV(\underline{e}_1)$.*
- *If $n > 0$ and $p = n$ then $[\![e_1\{x\backslash e_2\}^n]\!]_{\vec{R}_n} = (\underline{e}_1,\ K_1 :: \ldots :: K_{p-1} :: K'_p)$ such that $K'_p = (K_p[\cdot])\{z\backslash\underline{e}_2\}$ and $z \notin FV(\underline{e}_1)$.*

Proof. By induction on the structure of e_1. We only show the interesting variable case below. Other cases follow easily from the I.H.

- Case $e_1 = x$, $n = 0$: Note that $[\![x]\!]_{R_0} = (R_0(x), \texttt{nil}) = (z, \texttt{nil})$. Then,

$$\begin{array}{rcll} [\![x\{x\backslash e_2\}^0]\!]_{R_0} & = & [\![e_2]\!]_{R_0} & \\ & = & [\![e_2]\!]_{\{\}} & \text{by Lemma A.7} \\ & = & (\underline{e}_2, \texttt{nil}) & \\ & = & (z\{z\backslash \underline{e}_2\}, \texttt{nil}) & \end{array}$$

□

Lemma A.6. *Let e_1 be a stage-n and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. If $R_0(x) = z$, then*

$$Close([\![e_1]\!]_{\vec{R}_n})\{z\backslash\, Close([\![e_2]\!]_{\{\}})\} = Close([\![e_1\{x\backslash e_2\}^n]\!]_{\vec{R}_n})$$

Proof. Follows from Lemma A.5. □

Lemma A.7. *Let e be a stage-n λ^{gen}_{poly} expression with $FV(e) = \{x_1, \ldots, x_m\}$. Then,*

$$[\![e]\!]_{R_0,R_1,\ldots,R_n} = [\![e]\!]_{R'_0,R_1,\ldots,R_n}$$

if $R_0(x_i) = R'_0(x_i)$ for any $i \in \{1..m\}$.

Proof. By a straightforward structural induction on e. □

Lemma A.8. *Let e be a stage-n λ^{gen}_{poly} expression, $[\![e]\!]_{R_0,\ldots,R_n} = (e_1, \kappa)$ and $[\![e]\!]_{R_0\{\rho_m\backslash R_m\},\ldots,R_n\{\rho_m\backslash R_m\}} = (e'_1, \kappa')$. Then*

$$Close([\![e]\!]_{R_0,\ldots,R_n})\{\rho_m\backslash R_m\} \xrightarrow{\mathcal{A}^*} Close([\![e]\!]_{R_0\{\rho_m\backslash R_m\},\ldots,R_n\{\rho_m\backslash R_m\}})$$

and

$$e_1\{\rho_m\backslash R_m\} \xrightarrow{\mathcal{A}^*} e'_1$$

Proof. By structural induction on e. In the VAR case we use Lemma A.9. In the BOX case we use the fact that the newly introduced environment variable ρ_{n+1} is fresh. Other cases easily follow from the I.H. □

Lemma A.9. $(R(x))\{\rho\backslash R'\} \xrightarrow{\mathcal{A}^*} (R\{\rho\backslash R'\})(x)$ *for any renaming environments R, R'.*

Proof. By structural induction on R.

- Case $R = \{\}$: We have $\{\}(x)\{\rho\backslash R'\} = \textbf{error}$ and $(\{\}\{\rho\backslash R'\})(x) = \textbf{error}$ by definition.
- Case $R = \rho'$: If $\rho = \rho'$, then $(\rho(x))\{\rho\backslash R'\} = R' \cdot x$ and $(\rho\{\rho\backslash R'\})(x) = R'(x)$. By the definition of admin reductions, $R' \cdot x \xrightarrow{\mathcal{A}} R'(x)$. If $\rho \neq \rho'$, then $(\rho'(x))\{\rho\backslash R'\} = \rho' \cdot x$ and $(\rho'\{\rho\backslash R'\})(x) = \rho' \cdot x$.
- Case $R = R_1 \,\mathsf{with}\, \{y = z\}$: If $x = y$, then $((R_1 \,\mathsf{with}\, \{x = z\})(x))\{\rho\backslash R'\} = z$ and $((R_1 \,\mathsf{with}\, \{x = z\})\{\rho\backslash R'\})(x) = z$.

  If $x \neq y$, then $((R_1 \,\mathsf{with}\, \{y = z\})(x))\{\rho\backslash R'\} = (R_1(x))\{\rho\backslash R'\}$ and $((R_1 \,\mathsf{with}\, \{y = z\})\{\rho\backslash R'\})(x) = (R_1\{\rho\backslash R'\})(x)$. By I.H. we have $(R_1(x))\{\rho\backslash R'\} \xrightarrow{\mathcal{A}^*} (R_1\{\rho\backslash R'\})(x)$. □

Lemma A.10. *Let e be a λ^{gen}_{poly} expression such that $e \in Val^{n+1}$. Then*

$$[\![e]\!]_{\{\},R_1,\ldots,R_{n+1}} = [\![e]\!]_{R_1,\ldots,R_{n+1}}$$

Proof. By a straightforward induction on the structure of e. □

Lemma A.11. *Let $[\![e]\!]_{R_0} = (\underline{e}, \kappa)$. Then, $e \in Val^0 \iff \underline{e} \in RVal$, and $e \notin Val^0 \iff \underline{e} \notin RVal$.*

Proof. By a straightforward case analysis. □