DRAFT

Type Checking Program Generators Using the Record Calculus

Baris Aktemur
University of Illinois at Urbana-Champaign
aktemur@illinois.edu

Abstract

Program Generation (PG) is a widely applicable technique that can improve efficiency and modularity of a program as well as programmer productivity. An important challenge in PG is to guarantee that a generator will produce type-safe code. In this paper we address this problem by showing that record calculus can be used to obtain a very powerful type system for program generation. The type system has let-polymorphism and subtyping, and can handle side-effecting expressions.

1 Introduction

Program Generation (PG) is about writing programs that write programs. If a program's structure is so routine that it can be built by an algorithm, it is natural to use PG to manufacture the program because this improves program reusability and performance as well as programmer productivity, while decreasing human error [CE00].

Program generation can be classified into two categories: *PG by partial evaluation* [ACK03, CX03, Dav96, DP96, KKcS08, MTBS99, TN03, YI06] and *PG by program construction* [Baw99, HZS05, HZS07, KCJ03, KYC06, OMY01, PHEK99, Rhi05, ZHS04]. These two approaches to program generation require different mindsets when programming.

PG by partial evaluation is based on the ideas of partial evaluation. This PG style is about *delaying* the execution of some part of a code while regularly evaluating the other parts. The programmer may explicitly annotate the program to indicate which part to delay or not to delay, as opposed to partial evaluation's implicit binding-time analysis [JGS93, NN92]. This kind of PG enjoys the "erasure property" [DP96]: a valid program can be obtained if all the annotations are erased. This means that there can be no unbound variable in a program, even in the delayed fragments (i.e. inside quotations).

PG by program construction is about *building* new programs by composing program fragments. Programmers again explicitly define fragments and how they are combined. There is no erasure property; removing annotations may leave a meaningless program.

An important difference between PG by partial evaluation and PG by program construction is *variable hygiene*. In PG by program construction, free variables in a fragment may get "captured" and bound when the fragment is spliced into a context. Composition of code values is intentionally *unhygienic*. In PG by partial evaluation, this is forbidden. The binding of a variable is statically known, and variables are alpha-converted to avoid capturing; composition of code values is intentionally *hygienic*. For example, the program $\mathsf{let}\, f = \lambda c.\langle \lambda x.x + \tilde{}(c)\rangle\, \mathsf{in}\, \langle \lambda x.\tilde{}(f\langle x\rangle)\rangle$, written in ML-like syntax, yields a value that is alpha-equivalent to $\langle \lambda y.\lambda z.z + y\rangle$ if evaluated in MetaOCaml

[TCLP] — a PG-by-partial-evaluation system. On the other hand, the output is $\langle \lambda x.\lambda x.x + x \rangle$, if evaluated in λ^{open}_{poly} [KYC06] — a PG-by-program-construction system.

The program generation context we assume falls into the "PG by program construction" category.

In this paper we focus on an important challenge of program generation: How can we guarantee that a generator will produce type-safe code? Several program generation type systems investigate the same question [Dav96, DP96, KKcS08, KYC06, MTBS99, OMY01, Rhi05, SGM+03, TN03, YI06]. We show that this problem reduces to the problem of type checking in record calculus, which is well-studied and mature. This allows us to reuse several properties already proved in the record calculus domain, giving us a powerful and sound type system that guarantees type-safety of generated programs.

Major results in this paper include:

- Definition of a translation from a program generation language to the record calculus.
- Showing that evaluating a program generator in the staged operational semantics is equivalent to evaluating its translation in the record operational semantics. This result brings the preservation property to the record type system with respect to the staged semantics *for free*.
- Proving that the record calculus provides a sound type system with respect to the staged operational semantics.
- Showing that the record calculus type system is equal to the λ^{open}_{poly} [KYC06] type system.

We then show that

- the type system can gracefully be extended with subtyping constraints by using already-existing record subtyping definitions from the literature. A staged type system with subtyping constraints, to our knowledge, is new.
- pluggable declarations can be added to the language. Pluggable declarations and subtyping provide a solution to type-checking the "library specialization problem".
- side-effecting expressions such as references can also be handled by an improved version of the translation.

The results elaborated in this paper show that a very powerful staged type system can be obtained by using record calculus properties.

This paper is organized as follows: In Section 2 we give intuition of why record calculus and program generation are closely related. Section 3 informally discusses how a staged type system works and what we expect from it, and motivate the need for subtyping. We formally introduce the program generation language and the record calculus in Sections 4 and 5, respectively. Section 6 gives the definition of the translation from the staged language to the record calculus. Section 7 states the formal relationship between the two calculi. In Sections 8 through 10 we discuss how to extend the languages, translation, and the type system with subtyping, pluggable declarations, and references. We provide a comparison of our contributions to the existing work in Section 11. We conclude the paper in Section 12. Proofs of major lemmas and theorems of this paper are given in the appendix.

2 Using Records for Staged Computing

A quotation defines a program piece that is not executed until it is "run". Consider $\langle 2+3 \rangle$. This expression directly evaluates to the value $\langle 2+3 \rangle$, not $\langle 5 \rangle$. "$2+3$" is executed only when the quoted expression is "run" as in $\mathsf{let}\, x = \langle 2+3 \rangle\, \mathsf{in}\, \mathsf{run}(x)$, which evaluates to 5. This fact brings a question about the relation between quoted expressions and closures. Recall that expressions guarded by lambda abstractions are not executed until the function is applied. We can represent a quoted expression as lambda abstraction, and "run" as function application. $\langle 2+3 \rangle$ can be represented as $\lambda z.2+3$, which directly evaluates to the closure $\lambda z.2+3$ without executing $2+3$. So, we can rewrite $\mathsf{let}\, x = \langle 2+3 \rangle\, \mathsf{in}\, \mathsf{run}(x)$ as $\mathsf{let}\, x = \lambda z.2+3\, \mathsf{in}\, x(0)$, where the application evaluates $2+3$ and results in 5. The name of the function parameter and the function argument are not important in this example. So, we have an indication that there is a close relation between lambda abstractions and quoted expressions, as well as "run" and function application.

Let us now consider a more complicated example which splices a fragment into another one using antiquotation: $\mathsf{let}\, x = \langle 2+3 \rangle\, \mathsf{in}\, \langle 4 + \tilde{}(x) \rangle$. This piece of program evaluates to $\langle 4+2+3 \rangle$. The body of the quoted expression $\langle 2+3 \rangle$ is still not executed, but "extracted out" and spliced into the hole as denoted by the antiquotation. To give a similar effect, we can consider converting the antiquotation to function application: $\mathsf{let}\, x = \lambda z.2+3\, \mathsf{in}\, \lambda w.4 + x(0)$. Because the function application takes place under a lambda abstraction, the expression "$2+3$" is still not executed.

The two examples above were simple in the sense that the quoted expressions were closed; they did not contain any free variables. Consider $\langle y \rangle$. The variable y will obtain a meaning when the expression is spliced into a context that provides a binding for y. For instance, in $\mathsf{let}\, c = \langle y \rangle\, \mathsf{in}\, \langle \mathsf{let}\, y = 2\, \mathsf{in}\, \tilde{}(c) + 3 \rangle$, the variable y is an integer. This binding is provided by the code fragment surrounding the antiquotation. Hence, it makes sense to consider a quoted expression as a lambda abstraction that takes in the bindings of its free variables rather than ignoring the parameter. The "bindings" are nothing but an environment. An occurrence of a variable is then a lookup in the environment. So, using the dot notation $e \cdot \ell$ to access the field ℓ of the record e, we can rewrite $\langle y \rangle$ as $\lambda r.r \cdot y$. Note that quoted expressions also can define and use variables within themselves. These bindings can be considered as updates to the environment. So, an antiquotation becomes a function application that passes the up-to-date environment to the antiquoted fragment. The program above, $\mathsf{let}\, c = \langle y \rangle\, \mathsf{in}\, \langle \mathsf{let}\, y = 2\, \mathsf{in}\, \tilde{}(c) + 3 \rangle$, can be rewritten as $\mathsf{let}\, c = (\lambda r.r \cdot y)\, \mathsf{in}\, \lambda r.\mathsf{let}\, r = r\, \mathsf{with}\, \{y = 2\}\, \mathsf{in}\, c(r) + 3$, where $e\, \mathsf{with}\, \{\ell = e'\}$ is the operation that updates the field ℓ of the record e with e'. The "run" operation is also a function application similar to an antiquotation, but it should pass the empty environment to the fragment, because only complete (i.e. closed) fragments are runnable.

The intuitive closeness between the staged language and the record calculus immediately suggests a systematic translation. In Figure 1, we give a transformation to convert staged expressions to record calculus expressions. The superscript n in the translation denotes the stage. The translation converts a variable to a look-up in the record that stands for the environment of the current stage. Abstractions and let-bindings update the current environment with a new binding. Quoted expressions are converted to functions that take as input a record representing the environment in the next stage. Antiquotations are translated to function applications where the current environment becomes the operand. $\mathsf{run}(\cdot)$ is also converted to a function application, but this time the operand is the empty record. $\mathsf{fix}\, f(x)$ is the fix-point operator for the function f with argument x, and is used for recursion. lift raises a value to the next stage; hence the translation introduces an abstraction.

$$[\![c]\!]^n = c$$
$$[\![x]\!]^n = r_n{\cdot}x$$
$$[\![\lambda x.e]\!]^n = \lambda x.\,\mathsf{let}\, r_n = r_n \,\mathsf{with}\, \{x = x\}\,\mathsf{in}\, [\![e]\!]^n$$
$$[\![\mathsf{fix}\, f(x).\, e]\!]^n = \mathsf{fix}\, f(x).\,\mathsf{let}\, r_n = r_n \,\mathsf{with}\, \{f = f, x = x\}\,\mathsf{in}\, [\![e]\!]^n$$
$$[\![e_1 e_2]\!]^n = [\![e_1]\!]^n [\![e_2]\!]^n$$
$$[\![\mathsf{let}\, x = e_1\,\mathsf{in}\, e_2]\!]^n = \mathsf{let}\, r_n = r_n \,\mathsf{with}\, \{x = [\![e_1]\!]^n\}\,\mathsf{in}\, [\![e_2]\!]^n$$
$$[\![\langle e\rangle]\!]^n = \lambda r_{n+1}.\, [\![e]\!]^{n+1}$$
$$[\![\tilde{}(e)]\!]^{n+1} = [\![e]\!]^n\, r_{n+1}$$
$$[\![\mathsf{run}(e)]\!]^n = [\![e]\!]^n\{\}$$
$$[\![\mathsf{lift}(e)]\!]^n = \lambda r_{n+1}.[\![e]\!]^n$$

Figure 1: A first attempt on a transformation from staged expressions to record calculus expressions. Variable names are also used as record field labels in the target language.

We give this first attempt of a translation; however, we will not use it in the upcoming sections. We will define better versions; the first improvement will provide more useful results in proving formal properties, the second improvement will handle references as well. We initially give this version because it is more intuitive and easier to understand than the other improved versions. Below we give some examples to illustrate the translation process. The reader is encouraged to check that both sides would reduce to equivalent terms when simplified (after substituting r_0 with the empty record in translations). To improve readability of the examples, we assume existence of constructs such as the if-expression, lists, head (hd) and tail (tl) operators, addition, subtraction, etc. It would be straightforward to add these into the language. All the translations take place starting from stage 0 as denoted by the superscript.

$$[\![\lambda c.\langle \mathsf{let}\, x = 5\,\mathsf{in}\, \tilde{}(c)\rangle]\!]^0 = \lambda c.\mathsf{let}\, r_0 = r_0\,\mathsf{with}\,\{c = c\}\,\mathsf{in}\; (\lambda r_1.\mathsf{let}\, r_1 = r_1\,\mathsf{with}\,\{x = 5\}\,\mathsf{in}\, r_0{\cdot}c(r_1))$$

$$[\![\mathsf{let}\, c = \langle x + 8\rangle\,\mathsf{in}\; \mathsf{let}\, g = \langle \lambda x.\tilde{}(c)\rangle\,\mathsf{in}\; (\mathsf{run}(g))(10)]\!]^0 = \mathsf{let}\, r_0 = r_0\,\mathsf{with}\,\{c = \lambda r_1.r_1{\cdot}x + 3\}\,\mathsf{in}\; \mathsf{let}\, r_0 = r_0\,\mathsf{with}\,\{g = \lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1\,\mathsf{with}\,\{x = x\}\,\mathsf{in}\, r_0{\cdot}c(r_1)\}\,\mathsf{in}\; (r_0{\cdot}g(\{\}))(10)$$

The function below is the factorial function and is written completely in stage 0.

$$[\![\mathsf{fix}\, fact(n).\,\mathsf{if}\; n = 0\;\mathsf{then}\; 1\;\mathsf{else}\; fact(n-1) \times n]\!]^0 = \mathsf{fix}\, fact(n).\,\mathsf{let}\, r_0 = r_0\,\mathsf{with}\,\{fact = fact, n = n\}\,\mathsf{in}\; \mathsf{if}\; r_0{\cdot}n = 0\;\mathsf{then}\; 1\;\mathsf{else}\; (r_0{\cdot}fact)(r_0{\cdot}n - 1) \times r_0{\cdot}n$$

The following example, adapted from [CX03], generates a specialized polynomial calculation function for the polynomial $4 + 6x + 2x^2$; specifically $\lambda x.4 + (x \times (6 + x \times (2 + 0)))$. A polynomial is represented as a list of integer values; in this case $[4; 6; 2]$.

$$[\![\mathsf{let}\ poly = \mathsf{fix}\, gen(p).\ \mathsf{if}\ p = \texttt{nil}\ \mathsf{then}\ \langle 0\rangle \quad \mathsf{else}\ \langle\ \tilde{}(\mathsf{lift}(\mathsf{hd}\ p)) + x \times\ \tilde{}(gen(\mathsf{tl}\ p))\rangle\ \mathsf{in}\ \mathsf{run}\langle\lambda x.\ \tilde{}(poly\ [4;6;2])\rangle\ ]\!]^0 =$$

$$\mathsf{let}\ r_0 = r_0\ \mathsf{with}\ \{poly = \mathsf{fix}\, gen(p).\ \mathsf{let}\, r_0 = r_0\ \mathsf{with}\ \{gen = gen, p = p\}\ \mathsf{in}\ \mathsf{if}\ r_0{\cdot}p = \texttt{nil}\ \mathsf{then}\ \lambda r_1.0\ \mathsf{else}\ \lambda r_1.(\lambda r_2.\mathsf{hd}\ (r_0{\cdot}p))(r_1) + r_1{\cdot}x \times (r_0{\cdot}gen(\mathsf{tl}\ r_0{\cdot}p))(r_1)\}\ \mathsf{in}\ (\lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1\ \mathsf{with}\ \{x = x\}\ \mathsf{in}\ (r_0{\cdot}poly\ [4;6;2])r_1)\{\}$$

Using a record look-up allows for distinguishing variables with the same name in different stages. The examples below illustrate such cases. Note how the occurrence of the variable y in stage 0 is separated from the occurrence in stage 1.

$$[\![\lambda y.\langle y + \tilde{}(y)\rangle]\!]^0 = \lambda y.\mathsf{let}\, r_0 = r_0\ \mathsf{with}\ \{y = y\}\ \mathsf{in}\ (\lambda r_1.r_1{\cdot}y + (r_0{\cdot}y)(r_1))$$

$$[\![\lambda y.\langle\lambda y.\ \tilde{}(y)\rangle]\!]^0 = \lambda y.\mathsf{let}\, r_0 = r_0\ \mathsf{with}\ \{y = y\}\ \mathsf{in}\ (\lambda r_1.\lambda y.\mathsf{let}\, r_1 = r_1\ \mathsf{with}\ \{y = y\}\ \mathsf{in}\ (r_0{\cdot}y)(r_1))$$

Being able to translate staged expressions to record calculus expressions brings the question of whether a record type system could be used to type-check staged expressions. This would be desired because record type systems have been studied extensively and have grown mature. The ultimate goal is to use the record type system to decide whether it is safe to execute a staged expression. In particular, we want to be able to say "the staged expression e is type-safe if $[\![e]\!]$ is type-safe." In this paper we show that this is feasible; the record calculus gives a sound type system for staged computation.

3 Type-Checking Program Generators

In this section we give an informal introduction to staged typing. Being one of the state-of-the-art languages for program generation, we take λ^{open}_{poly} [KYC06] as our starting point for a staged type system. We show, by examples, how the type system works, and why an extension with pluggable declarations and subtyping would be desired.

The λ^{open}_{poly} type system gives program fragments a type of the form $\Box(\Gamma \triangleright A)$ with the meaning "the quoted expression will result in a value of type A if it is used in a context that provides the environment Γ." For instance, the fragment $\langle x + 1\rangle$ could be given the type $\Box(\{x : \texttt{int}\} \triangleright \texttt{int})$: in an environment that binds x to `int`, the fragment will result in a value of type `int`. Row variables [Wan91, Rém94] are used as part of environments for flexibility and quantification. For instance, the above fragment can be typed as $\Box(\{x : \texttt{int}\}\rho \triangleright \texttt{int})$, which can be instantiated to types like $\Box(\{x : \texttt{int}, y : \texttt{bool}\} \triangleright \texttt{int})$ or $\Box(\{x : \texttt{int}, z : \texttt{int}, w : \texttt{bool}\} \triangleright \texttt{int})$ allowing for usage in more contexts. The function $\lambda c.\langle \mathsf{let}\, x = 1\ \mathsf{in}\ \tilde{}(c)\rangle$ can be typed as $\Box(\{x : \texttt{int}\}\rho \triangleright \alpha) \to \Box(\rho \triangleright \alpha)$[1] carrying the meaning that the argument of the function has to be a fragment that will receive an environment that maps x to `int`. If the function is applied on $\langle x + y\rangle$, which could be typed to

[1] Technically row variables are kinded based on their domains [Rém94], and this type is really $\Box(\{x : \texttt{int}\}\rho_{\{x\}} \triangleright \alpha) \to \Box(\{x : \theta\}\rho_{\{x\}} \triangleright \alpha)$, where $\rho_{\{x\}}$ means that x is not in the domain of ρ, and θ is a "field variable" that stands for a type or the absence of the binding. For brevity, we follow the notational convention used in [KYC06] and denote $\{x : \theta\}\rho$ simply as ρ, and $\rho_{\{x\}}$ as ρ when the full notation can be inferred from the context.

```
class LinkedList {
  Object value;
  LinkedList next;
  int counter=0;

  void add(Object z) {
    counter++;
    ...
  }
  void reverse() {
    counter++;
    ...
  }
  ...
}
```

```
Code genLib(Code cf, Code ci) {
  return ⟨
    class LinkedList {
      Object value;
      LinkedList next;
      ~(cf)

      void add(Object z) {
        ~(ci)
        ...
      }
      void reverse() {
        ~(ci)
        ...
      }
      ...
    } ⟩;
}
```

Figure 2: Writing a customizable library using program generation.

$\square(\{x : \texttt{int}, y : \texttt{int}\}\rho' \triangleright \texttt{int})$, the application could be given the type $\square(\{y : \texttt{int}\}\rho'' \triangleright \texttt{int})$. This means that the result of the application should be used in a context that provides an integer value bound to y. Note that the condition for x disappears from the type because the fragment itself binds it to an integer.

We use the operator $\mathsf{run}(\cdot)$ for bringing fragments to stage-0 and executing. Only "complete" program fragments can be run. Therefore, λ^{open}_{poly} allows running fragments only if they can be given a type $\square(\varnothing \triangleright ...)$, which means they do not have any unbound variables. Fragments making other assumptions about outer environments are not runnable.

Library Specialization

In [AK09], we gave a comparative study of several techniques addressing the "library specialization" problem. One of the techniques is program generation. We now use this example to motivate two extensions to the staged type system: pluggable declarations and subtyping.

Very briefly, the problem is this: Given a library with several features, how can we exclude unneeded features so that the library becomes lightweight? The main motivation is to make the library free of unnecessary fields so that its memory fingerprint becomes smaller. Program generation allows customizability of the library by making it possible to include/exclude feature-related code fragments in the library. Take the linked-list class in Figure 2 with the "counter" feature that counts the number of operations performed on the list object. To make the counter feature optional, we can write the generator function genLib as shown in the same figure, that takes feature-related code fragments as arguments (cf stands for "counter field", ci stands for "counter increment").

Invoking the genLib function with the arguments ⟨int counter=0;⟩ and ⟨counter++;⟩ would yield a linked-list class with the counter feature included. Passing the empty-code arguments ⟨⟩ and ⟨⟩ produces a linked-list that does not contain the feature, relieving the class from carrying the unneeded field and computation.

Pluggable Declarations

A major motivation of the library specialization problem is to be able to exclude fields. Using program generation to do this requires that the program generation language supports quoting and filling in holes with declarations; we refer to this language feature as "pluggable declarations". In λ^{open}_{poly}, only expressions can be quoted. In Section 9 we discuss how it can be extended with pluggable declarations (the extension is actually a syntactic sugaring that could be expressed using existing program generation facilities and higher-order functions).

Assume that λ^{open}_{poly} provides the ability to quote declarations with the syntax $\langle x = e \rangle$, and quoted declarations can be plugged into let-bindings using the syntax $\mathsf{let}\ `(\cdot)\ \mathsf{in}\ e$. Analogous to a quoted expression, a quoted declaration is given a type $\Diamond(\Gamma_1 \triangleright \Gamma_2)$ with the intuition that "the declaration, when used in a context that provides the environment Γ_1, will output the environment Γ_2." For example, the declaration $\langle x = y + 1 \rangle$ can be typed to $\Diamond(\{y : \mathtt{int}\}\rho \triangleright \{y : \mathtt{int}, x : \mathtt{int}\}\rho)$. The function $\lambda d.\langle \mathsf{let}\ `(d)\ \mathsf{in}\ x + 1 \rangle$ can take the type $\Diamond(\rho_1 \triangleright \{x : \mathtt{int}\}\rho_2) \to \Box(\rho_1 \triangleright \mathtt{int})$ which carries the meaning that the declaration $d$ should let through an environment that binds $x$ to an `int`.

Assume also the existence of tuples and arithmetic operations (both would be straightforward extensions). Then, a simple library generator could be abstracted out as below. Suppose for now that the library class contains only one method, `v` stands for the field of the class, and we return a single function to model the class. Again, `cf` stands for "counter field" and `ci` stands for "counter increment".

$$\begin{array}{l}\mathtt{genLib} = \lambda \mathtt{cf}.\ \lambda \mathtt{ci}.\ \ \langle \mathsf{let}\, \mathtt{v} = 0\, \mathsf{in} \\ \qquad\qquad\qquad\qquad\quad \mathsf{let}\ `(\mathtt{cf})\, \mathsf{in} \\ \qquad\qquad\qquad\qquad\quad (\lambda \mathtt{z}.\ `(\mathtt{ci}) ... \mathtt{v} + \mathtt{z}) \rangle\end{array}$$

We can give the following type to the generator above:

$$\forall \rho_1, \rho_2, \gamma.\ \Diamond(\{\mathtt{v} : \mathtt{int}\}\rho_1 \triangleright \{\mathtt{v} : \mathtt{int}\}\rho_2) \to \Box(\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho_2 \triangleright \gamma) \to \Box(\rho_1 \triangleright \mathtt{int} \to \mathtt{int})$$

The return type of the generator is $\Box(\rho_1 \triangleright \mathtt{int} \to \mathtt{int})$, which names the outermost environment of the returned code fragment as ρ_1. Then, inside the quoted fragment, a binding for `v` is made. Hence, the incoming environment of `cf` is $\{\mathtt{v} : \mathtt{int}\}\rho_1$. Because of the lambda abstraction, a binding for `z` will be added to the environment that comes out of `cf` before it goes into `ci`. Also, this environment has to contain a binding `v:int` because `v` is being used as an integer in the body of the method. The type of `genLib` encapsulates all this information.

The function `genLib` can be applied on appropriate arguments to generate the desired code. One such application[2] is

$$\mathtt{genLib}\ \langle \mathtt{cnt} = \mathsf{ref}\ 0 \rangle\ \langle \mathtt{cnt} := !\mathtt{cnt} + 1 \rangle$$

Substituting ρ_1 with $\varnothing$, ρ_2 with $\{\mathtt{cnt} : \mathtt{int}\ \mathsf{ref}\ \}$, and γ with `int` gives the type $\Diamond(\{\mathtt{v} : \mathtt{int}\} \triangleright \{\mathtt{v} : \mathtt{int}, \mathtt{cnt} : \mathtt{int}\ \mathsf{ref}\ \})$ for the first parameter, and $\Box(\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}, \mathtt{cnt} : \mathtt{int}\ \mathsf{ref}\ \} \triangleright \mathtt{int})$ for the second parameter of the function. Note that actual arguments have these types as well. The result of the application has type $\Box(\varnothing \triangleright \mathtt{int} \to \mathtt{int})$, which is runnable.

Another possible application of the generator is $\mathtt{genLib}\ \langle\rangle\ \langle 0 \rangle$, which stands for the case when the feature is excluded. This application results in another runnable code value with the same

[2]This example contains references. We do not include references initially in the formal presentation. References are added later in Section 10.

type $\Box(\varnothing \triangleright \texttt{int} \rightarrow \texttt{int})$. If the generator is applied as $\texttt{genLib}\ \langle\rangle\ \langle \texttt{cnt:=!cnt} + 1\rangle$ the type is $\Box(\{\texttt{cnt} : \texttt{int ref}\ \}\rho \triangleright \texttt{int} \rightarrow \texttt{int})$. Since the input environment of this type is not empty, the type system does not allow evaluation of the code value via $\mathsf{run}(\cdot)$.

Subtyping

We now slightly modify the library generator example to illustrate the need for the second extension: subtyping. Suppose now the library that will be generated contains two base methods, and hence two uses of ci.

$$\begin{aligned}\texttt{genLib} = \lambda\texttt{cf}.\ \lambda\texttt{ci}.\ \ \langle &\mathsf{let}\ \texttt{v} = 0\ \mathsf{in}\\ &\mathsf{let}\ \grave{}(\texttt{cf})\ \mathsf{in}\\ &(\lambda\texttt{z}.\ \grave{}(\texttt{ci})...\texttt{v} + \texttt{z}),\\ &(\lambda\texttt{y}.\ \grave{}(\texttt{ci})...\texttt{y} \times \texttt{v})\rangle\end{aligned}$$

The environment that goes into the first antiquoted ci is $\{\texttt{v} : \texttt{int}, \texttt{z} : \texttt{int}, \texttt{y} : \theta_1\}\rho_2$. However, the incoming environment of the second use is $\{\texttt{v} : \texttt{int}, \texttt{z} : \theta_2, \texttt{y} : \texttt{int}\}\rho_2$. Had we the chance to give a polymorphic type to ci such as $\forall\rho.\Box(\{\texttt{cnt} : \texttt{int ref}\ \}\rho \triangleright \texttt{int})$, we could instantiate appropriately for the two different uses above. However, because ci is an argument of the generator function, it cannot be used polymorphically (unless we enter the dangerous waters of undecidability and use higher-rank polymorphism); every occurrence of ci has to assume the same type. Since the type system has to be conservative because of the possibility that ci may include y or z as a free variable, the derived type for the generator would be

$$\begin{aligned}\forall\rho_1, \rho_2, \gamma.\ \Diamond(\{\texttt{v} : \texttt{int}\}\rho_1 \triangleright \{\texttt{v} : \texttt{int}, \texttt{y} : \texttt{int}, \texttt{z} : \texttt{int}\}\rho_2) &\rightarrow \Box(\{\texttt{v} : \texttt{int}, \texttt{y} : \texttt{int}, \texttt{z} : \texttt{int}\}\rho_2 \triangleright \gamma)\\ &\rightarrow \Box(\rho_1 \triangleright (\texttt{int} \rightarrow \texttt{int}) * (\texttt{int} \rightarrow \texttt{int}))\end{aligned}$$

In this type, the incoming environments of both uses of ci are $\{\texttt{v} : \texttt{int}, \texttt{y} : \texttt{int}, \texttt{z} : \texttt{int}\}\rho_2$, which match exactly the expected environment of ci, as would be required by the type system. However, now an application of the generator that used to be runnable (e.g. $\texttt{genLib}\ \langle\rangle\ \langle 0\rangle$) gets the type $\Box(\{\texttt{y} : \texttt{int}, \texttt{z} : \texttt{int}\}\rho \triangleright ...)$ — not a runnable type.

Subtyping can get around this problem. We do not have to feed a code fragment with the exact environment that it expects. We can provide a richer environment that still satisfies the fragment's expectations. Take $\langle x + 1\rangle$ with the type $\Box(\{x : \texttt{int}\} \triangleright \texttt{int})$. It is safe to use this fragment in the environment $\{x : \texttt{int}\}$, or in $\{x : \texttt{int}, y : \texttt{bool}\}$, or in $\{x : \texttt{int}, w : \texttt{bool}, k : \texttt{int} \rightarrow \texttt{int}\}$. As long as the environment provides $\{x : \texttt{int}\}$ we are fine. This is where subtyping comes into play. Recall that for the above example, the environment that goes into the first antiquoted ci is $\{\texttt{v} : \texttt{int}, \texttt{z} : \texttt{int}, \texttt{y} : \theta_1\}\rho_2$, and the second use is $\{\texttt{v} : \texttt{int}, \texttt{z} : \theta_2, \texttt{y} : \texttt{int}\}\rho_2$. If we give ci the type $\{\texttt{v} : \texttt{int}\}\rho_2$, using the properties

$$\{\texttt{v} : \texttt{int}, \texttt{z} : \texttt{int}, \texttt{y} : \theta_1\}\rho_2 <: \{\texttt{v} : \texttt{int}\}\rho_2$$

$$\{\texttt{v} : \texttt{int}, \texttt{z} : \theta_2, \texttt{y} : \texttt{int}\}\rho_2 <: \{\texttt{v} : \texttt{int}\}\rho_2$$

we can successfully obtain a runnable type as expected. We give in Section 8 more details of subtyping and show how Pottier's subtyping constraints [Pot00] can be used to solve the problem.

$x \in Var$
$c \in Constant$
$e \in Exp ::= c \mid x \mid \lambda x.e \mid \lambda^* x.e \mid \mathsf{fix}\, f(x).\, e \mid e\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e$
$\quad \mid \langle e \rangle \mid \tilde{}(e) \mid \mathsf{run}(e) \mid \mathsf{lift}(e)$

Figure 3: Syntax of λ^{gen}_{poly}.

4 Staged Language

In this section we give the formal definition of the staged language we use. The language is defined based on λ^{open}_{poly}[KYC06] — an ML-like language that supports program generation with freely-open fragments, references, let-polymorphism, and variable hygiene. For the moment we exclude references and open[3], but add a fix-point operator to have recursion, and call this language with core program generation facilities as λ^{gen}_{poly}. The syntax of the language is given in Figure 3. λ^* is the hygienic variable binding; it uniquely renames the bound variable to avoid capturing a variable after a fragment in plugged in the scope of the binding. $\mathsf{lift}(\cdot)$ raises a value to the next stage. Extension of the language with references and pluggable declarations is discussed later.

We use the syntax $\langle\cdot\rangle$ for quotation (box in λ^{open}_{poly}), and $\tilde{}(\cdot)$ for antiquotation (unbox_1 in λ^{open}_{poly}). A quotation denotes a computation in the next stage whereas an antiquotation denotes a computation in the previous stage. There is no multi-stage antiquotation like λ^{open}_{poly}'s unbox_k. This can be achieved by nesting antiquotations k times. We use $\mathsf{run}(\cdot)$, instead of λ^{open}_{poly}'s unbox_0 to evaluate code values.

Definition 4.1. The *depth* of an expression e is the maximum number of nested antiquotations in e that are not enclosed by quotations.

Definition 4.2. An expression e is a *stage-n expression* if the depth of e is less than or equal to n. This also means that a stage-n expression is also a stage-$n+1$ expression.

4.1 Operational Semantics

Despite the large number of different program generation languages, their dynamic semantics for the core program generation constructs are almost the same. Definitions of operational semantics can be found in [CMT04, DP96, KKcS08, KYC06, MTBS99, Rhi05]. A big-step operational semantics of λ^{open}_{poly} is given in [KYC06]. We choose to present a small-step semantics of λ^{gen}_{poly}, adapted from [Rhi05]. The values are given in Figure 4 and the reduction rules in Figure 5. Having added recursion to the language, small-step semantics serves better when reasoning about non-terminating reductions.

Definition 4.3. Staged renaming $[x^n \overset{m}{\mapsto} z]e$ replaces with z the occurrences of the stage-n variable x in the stage-m expression e. Staged renaming is used in operations regarding λ^*. The definition of staged renaming is in [KYC06].

[3] open is a syntax-directed subtyping operator restricted to closed fragments. Our extension with subtyping, discussed later in the paper, subsumes open.

$$
\begin{array}{lll}
& v^n \in Val^n & \\
Val^0 & ::= c \mid \lambda x.e \mid \mathsf{fix}\, f(x).\, e \mid \langle v^1 \rangle & \\
Val^{n+1} & ::= c \mid x \mid \lambda x.v^{n+1} \mid \mathsf{fix}\, f(x).\, v^{n+1} \mid v^{n+1} v^{n+1} & \\
& \quad \mid \langle v^{n+2} \rangle \mid \mathsf{lift}(v^{n+1}) \mid \mathsf{run}(v^{n+1}) \mid \mathsf{let}\, x = v^{n+1}\, \mathsf{in}\, v^{n+1} & \\
& \quad \mid \backprime(v^n) \quad (\text{if } n > 0) &
\end{array}
$$

Figure 4: The definition of values in λ^{gen}_{poly}.

ESABS
$$\frac{e \longrightarrow_{n+1} e'}{\lambda x.e \longrightarrow_{n+1} \lambda x.e'}$$

ESSYM
$$\frac{z \text{ is fresh}}{\lambda^* x.e \longrightarrow_n \lambda z.[x^n \overset{n}{\mapsto} z]e}$$

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

ESAPP
$$\frac{e_1 \longrightarrow_n e_1'}{e_1 e_2 \longrightarrow_n e_1' e_2} \qquad \frac{e_1 \in Val^n \quad e_2 \longrightarrow_n e_2'}{e_1 e_2 \longrightarrow_n e_1 e_2'} \qquad \frac{e_2 \in Val^0}{(\lambda x.e) e_2 \longrightarrow_0 e[x \backslash e_2]^0}$$
$$\frac{e_2 \in Val^0}{(\mathsf{fix}\, f(x).\, e) e_2 \longrightarrow_0 e[f \backslash \mathsf{fix}\, f(x).\, e]^0 [x \backslash e_2]^0}$$

ESLET
$$\frac{e_1 \longrightarrow_n e_1'}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_n \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2} \qquad \frac{e_1 \in Val^0}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_0 e_2[x \backslash e_1]^0}$$
$$\frac{e_1 \in Val^{n+1} \quad e_2 \longrightarrow_{n+1} e_2'}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_{n+1} \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2'}$$

ESBOX
$$\frac{e \longrightarrow_{n+1} e'}{\langle e \rangle \longrightarrow_n \langle e' \rangle}$$

ESUBOX
$$\frac{e \longrightarrow_n e'}{\backprime(e) \longrightarrow_{n+1} \backprime(e')} \qquad \frac{e \in Val^1}{\backprime(\langle e \rangle) \longrightarrow_1 e}$$

ESRUN
$$\frac{e \longrightarrow_n e'}{\mathsf{run}(e) \longrightarrow_n \mathsf{run}(e')} \qquad \frac{e \in Val^1}{\mathsf{run}(\langle e \rangle) \longrightarrow_0 e}$$

ESLIFT
$$\frac{e \longrightarrow_n e'}{\mathsf{lift}(e) \longrightarrow_n \mathsf{lift}(e')} \qquad \frac{e \in Val^0}{\mathsf{lift}(e) \longrightarrow_0 \langle e \rangle}$$

Figure 5: The small-step semantics of λ^{gen}_{poly}.

$$
\begin{array}{rl}
FV^n(c) = \varnothing & \\
FV^0(x) = \{x\} & \\
FV^{n+1}(x) = \varnothing & \\
FV^0(\lambda x.e) = FV^0(e) \setminus \{x\} & \\
FV^{n+1}(\lambda x.e) = FV^{n+1}(e) & \\
FV^n(\lambda^* x.e) = FV^n(\lambda x.e) & \\
FV^n(\langle e \rangle) = FV^{n+1}(e) & \\
FV^{n+1}(\backprime(e)) = FV^n(e) &
\end{array}
\qquad
\begin{array}{rl}
FV^n(e_1 e_2) = FV^n(e_1) \cup FV^n(e_2) & \\
FV^0(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2) = FV^0(e_1) \cup (FV^0(e_2) \setminus \{x\}) & \\
FV^{n+1}(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2) = FV^{n+1}(e_1) \cup FV^{n+1}(e_2) & \\
FV^0(\mathsf{fix}\, f(x).\, e) = FV^0(e) \setminus \{f, x\} & \\
FV^{n+1}(\mathsf{fix}\, f(x).\, e) = FV^{n+1}(e) & \\
FV^n(\mathsf{run}(e)) = FV^n(e) & \\
FV^n(\mathsf{lift}(e)) = FV^n(e) &
\end{array}
$$

Figure 6: Finding the stage-0 free variables of λ^{gen}_{poly} expression.

$$
\begin{array}{rcl}
c[x\backslash e]^n & = & c \\
x[x\backslash e]^0 & = & e \\
y[x\backslash e]^0 & = & y, \text{ if } y \neq x \\
y[x\backslash e]^{n+1} & = & y \\
(\lambda x.e)[x\backslash e']^0 & = & \lambda x.e \\
(\lambda y.e)[x\backslash e']^0 & = & \lambda z.e[y\backslash z]^0[x\backslash e']^0 \\
 & & \quad \text{where } z \text{ is fresh and } y \neq x \\
(\lambda y.e)[x\backslash e']^{n+1} & = & \lambda y.e[x\backslash e']^{n+1} \\
(\lambda^* x.e)[x\backslash e']^0 & = & \lambda^* x.e \\
(\lambda^* y.e)[x\backslash e']^0 & = & \lambda^* z.e[y\backslash z]^0[x\backslash e']^0 \\
 & & \quad \text{where } z \text{ is fresh and } y \neq x \\
(\lambda^* y.e)[x\backslash e']^{n+1} & = & \lambda^* y.e[x\backslash e']^{n+1} \\
(\mathsf{fix}\, f(y).\, e)[x\backslash e']^n & = & \textit{similar to abstraction} \\
(e_1 e_2)[x\backslash e]^n & = & e_1[x\backslash e]^n\ e_2[x\backslash e]^n \\
(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2)[x\backslash e]^0 & = & \mathsf{let}\, x = e_1[x\backslash e]^0\, \mathsf{in}\, e_2 \\
(\mathsf{let}\, y = e_1\, \mathsf{in}\, e_2)[x\backslash e]^0 & = & \mathsf{let}\, z = e_1[x\backslash e]^0\, \mathsf{in}\, e_2[y\backslash z]^0[x\backslash e]^0 \\
 & & \quad \text{where } z \text{ is fresh and } y \neq x \\
(\mathsf{let}\, y = e_1\, \mathsf{in}\, e_2)[x\backslash e]^{n+1} & = & \mathsf{let}\, y = e_1[x\backslash e]^{n+1}\, \mathsf{in}\, e_2[x\backslash e]^{n+1} \\
\langle e \rangle[x\backslash e']^n & = & \langle e[x\backslash e']^{n+1} \rangle \\
\backprime(e)[x\backslash e']^{n+1} & = & \backprime(e[x\backslash e']^n) \\
\mathsf{run}(e)[x\backslash e']^n & = & \mathsf{run}(e[x\backslash e']^n) \\
\mathsf{lift}(e)[x\backslash e']^n & = & \mathsf{lift}(e[x\backslash e']^n)
\end{array}
$$

Figure 7: Staged substitution.

4.2 Auxiliary Definitions

In this section we give definitions of how to find free variables and how to do substitution in a staged expression.

Definition 4.4. The free variables of a staged expression are the free variables at stage 0. The definition is in Figure 6.

Definition 4.5. Substitution in a staged expression replaces variables at stage-0 and is defined in Figure 7.

4.3 Type System

We give the λ^{gen}_{poly} type system, adapted from λ^{open}_{poly}[KYC06]. The definition of types is given in Figure 8; typing rules are in Figure 9. A type can be one of (i) a type variable α, (ii) a constant type ι, (iii) a function type $A \to B$, or (iv) a box-type $\Box(\Gamma \triangleright A)$. Code values type to box-types. A box-type $\Box(\Gamma \triangleright A)$ has the meaning "the fragment will result in a value of type A if it is evaluated in a context that provides the environment Γ." The fields in an environment Γ can be one of (i) a type A, (ii) `Abs`, denoting the absence of the binding for that particular field, or, (iii) a field variable θ. In a judgment $\Delta_0, \ldots, \Delta_n \vdash_S e : A$, the typing environment Δ_i stands for the environment of stage i. Quotations and antiquotations add or remove new typing environments.

For all the typing rules, the difference from λ^{open}_{poly} is that, due to the absence of references, there is no store typing that is being threaded through a proof tree. The TSLET rule, additionally, does not distinguish between expansive and non-expansive expressions. The soundness of this type system with respect to the operational semantics is given in [KYC06].

Definition 4.6. A substitution φ is a partial function from type system variables to types. We extend the definition of a substitution to apply on a compound object in the obvious way.

We assume that all substitutions respect domains of variables. That is, a type variable α is mapped to an $A \in SType$; an environment variable ρ is mapped to a $\Gamma \in SEnv$; and a field variable θ is mapped to an $F \in SField$. Hence, $\varphi A \in SType$ for any A; $\varphi\Gamma \in SEnv$ for any Γ; and $\varphi F \in SField$ for any F.

Definition 4.7 (Instantiation)**.** A type A is an instance of a type scheme $\forall\vec{\psi}.A'$, written $A \prec \forall\vec{\psi}.A'$, if and only if there is a substitution φ with domain $\vec{\psi}$ such that $\varphi A' = A$.

A type scheme σ is more general than a type scheme σ', denoted $\sigma' \prec \sigma$ with a slight abuse of notation, if and only if $A \prec \sigma$ for any $A \prec \sigma'$.

Environment instantiation, overloading $\prec$, is defined as follows:

$$\begin{aligned} \{x_i : F_i\}_1^m \prec \{x_i : \mu_i\}_1^m &\iff F_i \prec \mu_i \text{ for any } i \in [1..m] \\ \{x_i : F_i\}_1^m\rho \prec \{x_i : \mu_i\}_1^m\rho &\iff F_i \prec \mu_i \text{ for any } i \in [1..m] \end{aligned}$$

5 Record Language

Let λ^{rec}_{poly} be a record calculus with the exception that the record and non-record variables are disjoint. Records are mappings from labels to values. The syntax of λ^{rec}_{poly} is given in Figure 10. The record operations are (1) record update via the with operator, (2) accessing a value in a record using the label, and (3) the empty record.

Definition 5.1. We use the shorter notation $\{a_1 = e_1, a_2 = e_2, \ldots, a_m = e_m\}$ for the expression $\{\}$ with $\{a_1 = e_1\}$ with $\{a_2 = e_2\} \ldots$ with $\{a_m = e_m\}$.

5.1 Operational Semantics

We give operational semantics of the record calculus using unrestricted reductions. We will give call-by-value semantics when we introduce references into the language. The reductions are performed according to the following rules.

$$\begin{aligned} (\lambda w.e_1)e_2 &\longrightarrow_\beta e_1[w\backslash e_2] \\ \text{let } w = e_1 \text{ in } e_2 &\longrightarrow_\beta e_2[w\backslash e_1] \\ (e_2 \text{ with } \{a_1 = e_1\})\cdot a_2 &\longrightarrow_\beta e_2\cdot a_2 \text{ if } a_1 \neq a_2 \\ (e_2 \text{ with } \{a = e_1\})\cdot a &\longrightarrow_\beta e_1 \\ e \text{ with } \{a_1 = e_1\} \text{ with } \{a_2 = e_2\} &\longrightarrow_\beta e \text{ with } \{a_2 = e_2\} \text{ with } \{a_1 = e_1\} \text{ if } a_1 \neq a_2 \\ e \text{ with } \{a = e_1\} \text{ with } \{a = e_2\} &\longrightarrow_\beta e \text{ with } \{a = e_2\} \end{aligned}$$

A reduction is a congruence relation. That is, if an expression e_1 is inside a context $C[\ ]$ and $e_1 \longrightarrow_\beta e_2$, then $C[e_1] \longrightarrow_\beta C[e_2]$.

$$
\begin{array}{lcl}
\alpha, \beta & \in & STyVar \\
A, B & \in & SType ::= \alpha \mid \iota \mid A \to B \mid \Box(\Gamma \triangleright A) \\
\theta & \in & SFieldVar \\
F & \in & SField ::= A \mid \texttt{Abs} \mid \theta \\
\rho & \in & SEnvVar \\
\Gamma & \in & SEnv = Var \rightharpoonup SField \\
 & & ::= \{x_i : F_i\}_1^m \mid \{x_i : F_i\}_1^m \rho \\
\psi & \in & SEnvVar \oplus SFieldVar \\
\sigma & \in & STyScheme ::= \forall \psi.\sigma \mid A \\
\mu & \in & SFieldScheme ::= \sigma \mid \texttt{Abs} \\
\Delta & \in & STySchemeEnv = Var \rightharpoonup SFieldScheme \\
 & & ::= \{x_i : \mu_i\}_1^m \mid \{x_i : \mu_i\}_1^m \rho
\end{array}
$$

Figure 8: The definition of types in λ^{gen}_{poly}.

$$
\begin{array}{ll}
\textsf{TSCON} & \Delta_0, \ldots, \Delta_n \vdash_S c : \iota \\
\textsf{TSVAR} & \dfrac{A \prec \Delta_n(x)}{\Delta_0, \ldots, \Delta_n \vdash_S x : A} \\
\textsf{TSABS} & \dfrac{\Delta_0, \ldots, \Delta_n \twoheadleftarrow \{x : A\} \vdash_S e : B}{\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e : A \to B} \\
\textsf{TSSYM} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S \lambda z.[x^n \overset{n}{\mapsto} z]e : A \quad z \text{ is fresh}}{\Delta_0, \ldots, \Delta_n \vdash_S \lambda^* x.e : A} \\
\textsf{TSFIX} & \dfrac{\Delta_0, \ldots, \Delta_n \twoheadleftarrow \{f : A \to B, x : A\} \vdash_S e : B}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{fix}\, f(x).\, e : A \to B} \\
\textsf{TSAPP} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \to B \quad \Delta_0, \ldots, \Delta_n \vdash_S e_2 : A}{\Delta_0, \ldots, \Delta_n \vdash_S e_1 e_2 : B} \\
\textsf{TSLET} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \quad \Delta_0, \ldots, \Delta_n \twoheadleftarrow \{x : \mathrm{GEN}_A(\Delta_0, \cdots, \Delta_n)\} \vdash_S e_2 : B}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B} \\
\textsf{TSBOX} & \dfrac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e : A}{\Delta_0, \ldots, \Delta_n \vdash_S \langle e \rangle : \Box(\Gamma \triangleright A)} \\
\textsf{TSUNBOX} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S e : \Box(\Gamma \triangleright A) \quad \Gamma \prec \Delta_{n+1}}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_S \,`(e) : A} \\
\textsf{TSRUN} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S e : \Box(\varnothing \triangleright A)}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{run}(e) : A} \\
\textsf{TSLIFT} & \dfrac{\Delta_0, \ldots, \Delta_n \vdash_S e : A}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{lift}(e) : \Box(\Gamma \triangleright A)}
\end{array}
$$

Figure 9: The λ^{open}_{poly}[KYC06] type system rules adapted for λ^{gen}_{poly}– a language with core program generation facilities, recursion, and no references.

$$
\begin{array}{l}
x \in Var \\
a \in Label = Var \\
r \in RVar \\
w, f \in Name = Var \cup RVar \\
c \in Constant \\
e \in RExp ::= c \mid w \mid \lambda w.e \mid \mathsf{fix}\, f(x).\, e \mid e\, e \mid \mathsf{let}\, w = e\, \mathsf{in}\, e \\
\qquad\qquad \mid \{\} \mid e\, \mathsf{with}\, \{a = e\} \mid e{\cdot}a
\end{array}
$$

Figure 10: Record calculus syntax.

$$
\begin{array}{lcl}
\alpha, \beta & \in & RLegTyVar \\
A, B & \in & RLegType ::= \alpha \mid \iota \mid T \to A \\
T & \in & RType ::= A \mid \Gamma \\
\theta & \in & RFieldVar \\
F & \in & RField ::= A \mid \texttt{Abs} \mid \theta \\
\rho & \in & RRecVar \\
\Gamma & \in & RRec = Label \rightharpoonup RField \\
 & & ::= \{a_i : F_i\}_1^m \mid \{a_i : F_i\}_1^m \rho \\
\psi & \in & RTyVar \oplus RRecVar \oplus RFieldVar \\
\sigma & \in & RTyScheme ::= \forall \psi.\sigma \mid T \\
\Delta & \in & RTySchemeEnv = Name \rightharpoonup RTyScheme
\end{array}
$$

Figure 11: The definition of types in the record calculus.

5.2 Auxiliary Definitions

Definition 5.2. The free variables of a record calculus expression are defined as follows.

$$
\begin{array}{r}
FV(c) = \varnothing \\
FV(w) = \{w\} \\
FV(\lambda w.e) = FV(e) \setminus \{w\} \\
FV(\mathsf{fix}\, f(x).\, e) = FV(e) \setminus \{f, x\} \\
FV(e_1 e_2) = FV(e_1) \cup FV(e_2)
\end{array}
\qquad
\begin{array}{r}
FV(\mathsf{let}\, w = e_1\, \mathsf{in}\, e_2) = FV(e_1) \cup (FV(e_2) \setminus \{w\}) \\
FV(\{\}) = \varnothing \\
FV(e_1\, \mathsf{with}\, \{a = e_2\}) = FV(e_1) \cup FV(e_2) \\
FV(e{\cdot}a) = FV(e)
\end{array}
$$

5.3 Type System

The definition of types and other objects is in Figure 11; typing rules are in Figure 12. This record type system is not completely standard. One can notice that (1) we distinguish between record variables and non-record variables, (2) the grammar of types does not allow construction of certain types that would normally be allowed in a standard record calculus; in particular, we want to avoid having types of the form $T \to \Gamma$. These restrictions are needed to make the type system sound with respect to the staged semantics. The essence of the problem comes from the fact that a quoted expression is translated to a function. Let us now illustrate with examples how these restrictions help:

- **Why should record variables be separated from non-record variables?**
  Consider the expression $\langle 42 \rangle\, 2$. This is ill-typed because a quoted expression is being used as

$$\textsf{TRCON}\quad \Delta \vdash_R c : \iota$$

$$\textsf{TRVAR}\quad \frac{A \prec \Delta(x) \qquad \Gamma \prec \Delta(r)}{\Delta \vdash_R x : A \qquad \Delta \vdash_R r : \Gamma}$$

$$\textsf{TRABS}\quad \frac{\Delta \twoheadleftarrow \{x : A\} \vdash_R e : B}{\Delta \vdash_R \lambda x.e : A \to B} \qquad \frac{\Delta \twoheadleftarrow \{r : \Gamma\} \vdash_R e : B}{\Delta \vdash_R \lambda r.e : \Gamma \to B}$$

$$\textsf{TRFIX}\quad \frac{\Delta \twoheadleftarrow \{f : A \to B, x : A\} \vdash_R e : B}{\Delta \vdash_R \mathsf{fix}\, f(x).\, e : A \to B}$$

$$\textsf{TRAPP}\quad \frac{\Delta \vdash_R e_1 : T \to B \quad \Delta \vdash_R e_2 : T}{\Delta \vdash_R e_1 e_2 : B}$$

$$\textsf{TRLET}\quad \frac{\Delta \vdash_R e_1 : A \qquad \Delta \twoheadleftarrow \{x : \mathrm{GEN}_A(\Delta)\} \vdash_R e_2 : B}{\Delta \vdash_R \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B} \qquad \frac{\Delta \vdash_R e_1 : \Gamma \qquad \Delta \twoheadleftarrow \{r : \mathrm{GEN}_\Gamma(\Delta)\} \vdash_R e_2 : B}{\Delta \vdash_R \mathsf{let}\, r = e_1 \,\mathsf{in}\, e_2 : B}$$

$$\textsf{TRACC}\quad \frac{\Delta \vdash_R e : \Gamma \quad \Gamma(a) = A}{\Delta \vdash_R e{\cdot}a : A}$$

$$\textsf{TREMPTY}\quad \Delta \vdash_R \{\} : \varnothing$$

$$\textsf{TRUPD}\quad \frac{\Delta \vdash_R e_1 : \Gamma \quad \Delta \vdash_R e_2 : A}{\Delta \vdash_R e_1 \,\mathsf{with}\, \{a = e_2\} : \Gamma \twoheadleftarrow \{a : A\}}$$

Figure 12: The type system of the record calculus.

a function. The translation of this expression at stage 0 is $(\lambda r_1.42)2$. If we do not distinguish record variables, the lambda abstraction in the translation can be given the type $\texttt{int} \to \texttt{int}$, meaning that $(\lambda r_1.42)2$ would pass the type-checker. This is certainly not wanted. Restricting record variables to record types prevents this kind of failure.

For similar reasons, non-record variables should be restricted to non-record types. An ill-typed staged expression whose translation would otherwise pass the record type system is $(\lambda x.\langle 42\rangle\, x)$. Assigning a record type to the variable x would yield a valid type if non-record variables are not restricted to non-record types.

- **Why should types in the form $T \to \Gamma$ not be allowed?**
  Restricting non-record variables to non-record types is not sufficient. A quoted expression is translated to a function, but we want to apply this function only when filling in a hole or running the expression. Other applications should not be allowed. If types of the form $T \to \Gamma$ could be constructed, we could create an expression which results in the type Γ, and could feed this type to the lambda abstraction that represents the quoted expression. Consider the expression $(\lambda x.\lambda y.\langle 42\rangle\, (x\ y))$. Assigning the type $\texttt{int} \to \varnothing$ to x and the type $\texttt{int}$ to y would yield a valid type for the translation of this expression, which is ill-typed in the staged semantics.

The record type system enjoys the following (standard) definitions and lemmas.

Definition 5.3.

$$\mathrm{GEN}_T(\Delta) = \forall\vec{\psi}.T \text{ where } \vec{\psi} = FV(T) \setminus FV(\Delta)$$

Definition 5.4. A substitution φ is a partial function from type system variables to types. We extend the definition of a substitution to apply on a compound object in the obvious way.

We assume that all substitutions respect domains of variables. That is, a type variable α is mapped to an $A \in \mathit{RLegType}$; a record variable ρ is mapped to a $\Gamma \in \mathit{RRec}$; and a field variable θ is mapped to an $F \in \mathit{RField}$. Hence, $\varphi A \in \mathit{RLegType}$ for any A; $\varphi\Gamma \in \mathit{RRec}$ for any Γ; and $\varphi F \in \mathit{RField}$ for any F.

Definition 5.5 (Instantiation)**.** A type T is an instance of a type scheme $\forall\vec{\psi}.T'$, written $T \prec \forall\vec{\psi}.T'$, if and only if there is a substitution φ with domain $\vec{\psi}$ such that $\varphi T' = T$.

A type scheme σ is more general than a type scheme σ', denoted $\sigma' \prec \sigma$ with a slight abuse of the notation, if and only if $T \prec \sigma$ for any $T \prec \sigma'$.

The record calculus satisfies the standard lemmas such as Weakening/Strengthening, Substitution, Generalization, Preservation, and Progress.

6 Transformation

In this section we provide the definition of a translation, $[\![\cdot]\!]_{R_0,\ldots,R_n}$, from λ^{gen}_{poly} expressions to λ^{rec}_{poly}. This is an improved version of the initial translation given in Figure 1. We do not use that original translation because the improved version provides a more useful result about the relation between staged and record operational semantics as well as making some of the proofs less complicated. Consider the expression $\langle\lambda x.x + y\rangle$. This would be translated by the first translation to $(\lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1 \,\mathsf{with}\, \{x = x\}\, \mathsf{in}\, r_1{\cdot}x + r_1{\cdot}y)$. Obviously, we could as well translate the expression to $(\lambda r_1.\lambda x.x + r_1 \cdot y)$, which gives the same meaning. We do not need to look up a variable in a record if the variable already exists in the scope. The new translation does that. However, we need to be careful about variables with the same name that occur in different stages, because when the quotations are removed, a higher-stage binding may capture a lower stage variable. Take the expression $(\lambda y.\langle\lambda y.\tilde{}(y) + y\rangle)$. It is wrong to simply translate it to $(\lambda y.\lambda r_1.\lambda y.y(r_1) + y)$. It should rather be translated to $(\lambda z.\lambda r_1.\lambda w.z(r_1) + w)$, which preserves the meaning. For this purpose, we use "renaming environments", denoted by the subscript $R_0, \ldots, R_n$, where R_i carries the variables bound so far at stage i. It maps them to fresh names so that we can replace a variable avoiding any unintentional capture.

Definition 6.1 (Renaming environment)**.** A renaming environment R is defined as follows.

$$R \in \mathit{RenamingEnv} ::= \{\} \mid r \mid R\,\mathsf{with}\,\{x = y\}$$

A renaming environment defines a function from variables to record expressions:

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

The domain of a renaming environment is the set of variables for which there are explicit mappings:

$$\begin{aligned}
\mathit{dom}(R\,\mathsf{with}\,\{x = y\}) &= \mathit{dom}(R) \cup \{x\} \\
\mathit{dom}(r) &= \{\} \\
\mathit{dom}(\{\}) &= \{\}
\end{aligned}$$

Throughout this paper, we assume that free variables in a renaming environment are unique. That is, for any renaming environment sequence $R_0, \ldots, R_n$ in $[\![e]\!]_{R_0,\ldots,R_n}$ we have

(i) $z \notin FV(R'_i) \cup FV^n(e)$ for any $R_i = R'_i\,\mathsf{with}\,\{x = z\}$

$[\![c]\!]_{R_0,\ldots,R_n} = c$

$[\![x]\!]_{R_0,\ldots,R_n} = R_n(x)$

$[\![\lambda x.e]\!]_{R_0,\ldots,R_n} = \lambda z.\, [\![e]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}}$ where z is fresh

$[\![\lambda^* x.e]\!]_{R_0,\ldots,R_n} = [\![\lambda z.[x^n \overset{n}{\mapsto} z]e]\!]_{R_0,\ldots,R_n}$ where z is fresh

$[\![\mathsf{fix}\, f(x).\, e]\!]_{R_0,\ldots,R_n} = \mathsf{fix}\, g(z).\, [\![e]\!]_{R_0,\ldots,R_n \text{ with } \{f=g,x=z\}}$ where g, z are fresh

$[\![e_1 e_2]\!]_{R_0,\ldots,R_n} = [\![e_1]\!]_{R_0,\ldots,R_n} [\![e_2]\!]_{R_0,\ldots,R_n}$

$[\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!]_{R_0,\ldots,R_n} = \mathsf{let}\, z = [\![e_1]\!]_{R_0,\ldots,R_n}\, \mathsf{in}\, [\![e_2]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}}$ where z is fresh

$[\![\langle e \rangle]\!]_{R_0,\ldots,R_n} = \lambda r.\, [\![e]\!]_{R_0,\ldots,R_n,r}$ where r is fresh

$[\![\backprime(e)]\!]_{R_0,\ldots,R_n,R_{n+1}} = ([\![e]\!]_{R_0,\ldots,R_n})\, R_{n+1}$

$[\![\mathsf{run}(e)]\!]_{R_0,\ldots,R_n} = ([\![e]\!]_{R_0,\ldots,R_n})\{\}$

$[\![\mathsf{lift}(e)]\!]_{R_0,\ldots,R_n} = \lambda r.\, [\![e]\!]_{R_0,\ldots,R_n}$ where r is fresh

Figure 13: Transformation from λ^{gen}_{poly} expressions to λ^{rec}_{poly}.

(ii) $FV(R_i) \cap FV(R_j) = \varnothing$ if $i \neq j$

Note that these conditions are preserved by the transformation. Also, in order to reduce notational clutter, we assume that the record variable in R_n is r_n.

6.1 Type Transformation

The translation in Figure 14 converts objects in the λ^{gen}_{poly} type system to objects in the record calculus.

Lemma 6.2 (Type translation is well-defined)**.** *Let A be a λ^{gen}_{poly} type. Then $[\![A]\!] \in RLegType$. Similarly, $[\![\Gamma]\!] \in RRec$, for any Γ. Furthermore, for any $B' \in RLegType$, there exists a unique λ^{gen}_{poly} type B such that $[\![B]\!] = B'$. Similarly, for any $\Gamma' \in RType$, there exists a unique Γ such that $[\![\Gamma]\!] = \Gamma'$. Therefore, $[\![\cdot]\!]$ for types is reversible (i.e. is a bijection).*

Proof. Straightforward. □

7 Relation Between Staged Programming and Record Calculus

In this section we provide formal properties about the relation between staged computation and the record calculus. We show that the record type system can be used as a sound type system for staged programming, and this type system is as powerful as the λ^{open}_{poly} [KYC06] type system. Although our focus is on typing, we first begin with the theorem which states that evaluating a staged expression in the staged semantics is equivalent to evaluating the expression's translation in the record semantics. In addition to showing the close relation between staged computation and record calculus, this theorem's real value comes into play when proving that the record type system preserves types with respect to staged semantics. The relation between the two operational semantics gives the preservation property *for free*.

$$core(\texttt{Abs}) = \texttt{Abs}$$

$$core(\forall\vec{\psi}.A) = A$$

$$[\![\iota]\!] = \iota$$

$$[\![\psi]\!] = \psi$$

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$$

$$[\![\Box(\Gamma \triangleright A)]\!] = [\![\Gamma]\!] \to [\![A]\!]$$

$$[\![\texttt{Abs}]\!] = \texttt{Abs}$$

$$[\![\forall\psi.\sigma]\!] = \forall\psi.[\![\sigma]\!]$$

$$[\![\{x_1 : \mu_1, \ldots, x_m : \mu_m\}\rho]\!] = \forall\vec{\psi}.\{x_1 : [\![core(\mu_1)]\!], \ldots, x_m : [\![core(\mu_m)]\!]\}\rho$$
where $\vec{\psi} = BV(\mu_1) \cup \ldots \cup BV(\mu_2)$, and $BV(\mu_1) \ldots BV(\mu_2)$ are distinct from each other and free variables.

$$\begin{aligned}[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} =& \{r_0 : [\![\Delta_0]\!], \ldots, r_n : [\![\Delta_n]\!]\} \twoheadleftarrow \\ & \{z : [\![\Delta_0(x)]\!] \mid x \in dom(R_0) \wedge z = R_0(x)\} \twoheadleftarrow \ldots \twoheadleftarrow \\ & \{z : [\![\Delta_n(x)]\!] \mid x \in dom(R_n) \wedge z = R_n(x)\}\end{aligned}$$

Figure 14: Translating λ^{gen}_{poly} types to record calculus types.

Theorem 7.1 (Operational Equivalence)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $e_1 \longrightarrow_n e_2$, then $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. □

We now show that the record type system can be used as a sound type system for the staged language. We follow the standard approach of splitting the soundness into two: preservation and progress. Preservation comes for free as a result of Theorem 7.1. Progress is explicitly proved.

Later in this section we also prove that the record calculus forms a type system equal to λ^{open}_{poly}. This result suffices to prove soundness of the record type system with respect to staged semantics (because $\lambda^{open}_{poly}$is proven to be sound [KYC06]). However, we prefer to prove soundness via preservation and progress because this would be the approach to take if a variant of a record type system is used for which there is no equal staged type system known. And in fact this is exactly the case for the type system with subtyping (see Section 8).

Theorem 7.2 (Preservation)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e) = \varnothing$ (i.e. e_1 does not have any stage-0 free variables). If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$ and $e_1 \longrightarrow_n e_2$, then $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$.*

Proof. By Theorem 7.1 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. By the Preservation property of the record type system with respect to the record semantics, we have $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$. □

Lemma 7.3. *Let e be a stage-n λ^{gen}_{poly} expression. If $\Delta \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : T$, then $T \in RLegType$.*

Proof. By a straightforward case analysis. □

Theorem 7.4 (Progress)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e_1. Lemma 7.3 forms a key part in the proof. □

Theorem 7.5 (Soundness)**.** *Let e_1 be a stage-0 λ^{gen}_{poly} expression. If $\varnothing \vdash_R [\![e_1]\!]_{\{\}} : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $e_1 \longrightarrow^*_0 e_2$ and $\varnothing \vdash_R [\![e_2]\!]_{\{\}} : A$.*

Proof. Follows from Theorems 7.2 and 7.4. □

We finally show that the record calculus provides a type system that is the same as λ^{open}_{poly} [KYC06]. This result is important because it shows the power of the type system we obtain via record calculus. Proving only soundness is not sufficient for usefulness — a type system that rejects everything is also sound.

Theorem 7.6. *Let e be a stage-n λ^{gen}_{poly} program. Then*

$$\Delta_0, \ldots, \Delta_n \vdash_S e : A \iff [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$$

Proof. By structural induction on e. □

8 Extending λ^{gen}_{poly} with Subtyping

In Section 3 we showed using the library specialization problem why subtyping is needed. In this section we give more details about the need and use of subtyping. We then discuss the existing literature that provides a record type system with subtyping. We finally relate the subtyped record calculus to staged semantics. To distinguish subtyping between record fields from subtyping between types, we use the notation of Rémy [RÝ94] in this section. Existing bindings are denoted as $\texttt{Pre}\ A$, whereas absence is still `Abs`.

8.1 Power of Subtyping

A simplification of the library specialization example from Section 3 — assuming the existence of tuples — is the function

$$\lambda c.\langle \mathsf{let}\, x = 1\, \mathsf{in}\, \text{`}(c), \mathsf{let}\, y = 1\, \mathsf{in}\, \text{`}(c)\rangle$$

Let us call this function G. The best type that λ^{open}_{poly} can give to G is

$$\square(\{x : \texttt{Pre int}, y : \texttt{Pre int}\}\rho \triangleright \alpha) \to \square(\{x : \texttt{Pre int}, y : \texttt{Pre}, \texttt{int}\}\rho \triangleright \alpha * \alpha)$$

Even though this type is sound, it is not satisfactory. Let us now examine through two examples why this type does not suffice and what the ideal type would look like.

- Suppose G is applied on $\langle 0\rangle$. The result of the application would be $\langle \mathsf{let}\, x = 1\, \mathsf{in}\, 0, \mathsf{let}\, y = 1\, \mathsf{in}\, 0\rangle$. This fragment does not require any variables from outside; hence it is runnable. Its type ideally would be $\square(\rho \triangleright \texttt{int} * \texttt{int})$. However, the λ^{open}_{poly} type for the application is $\square(\{x : \texttt{Pre int}, y : \texttt{Pre int}\}\rho \triangleright \texttt{int} * \texttt{int})$, which makes unnecessary requirements for x and y, and does not allow us to `run`() the value.

- Suppose G is applied on $\langle x{+}1\rangle$. The result of the application would be $\langle \mathsf{let}\, x = 1\, \mathsf{in}\, x{+}1, \mathsf{let}\, y = 1\, \mathsf{in}\, x+1\rangle$. This fragment requires x to come as an integer value from outside, but imposes no requirements for y. Hence its type ideally would be $\Box(\{x : \mathtt{Pre\ int}\}\rho \triangleright \mathtt{int} * \mathtt{int})$. However, the λ^{open}_{poly} type for the application is again $\Box(\{x : \mathtt{Pre\ int}, y : \mathtt{Pre\ int}\}\rho \triangleright \mathtt{int} * \mathtt{int})$ which makes an unnecessary requirement for y similar to the case above.

In summary, we want the type system to not result in code types that put unnecessary requirement on the outer environment. λ^{open}_{poly} does not satisfy this. The technical reason is that c is a parameter of a function and in the Hindley/Milner style let-polymorphism function parameters cannot be given polymorphic types because type checking and inference then becomes undecidable [Wel94, Jim96]. Otherwise we could give a more general type to c and instantiate it accordingly for the two different uses. Because we cannot use a polymorphic type, different uses of the variable have to have the exact same type, resulting in unneeded requirements. This is where subtyping becomes very handy: We can loosen the condition that different uses must have the same type. Suppose c has the type $\Box(\Gamma \triangleright \mathtt{int})$. As long as the context of an antiquotation of c provides the contents of Γ, we are fine; there is no harm in providing c with a richer environment than it needs. Suppose also that the outer environment of the fragment $\langle \mathsf{let}\, x = 1\, \mathsf{in}\, \text{`}(c), \mathsf{let}\, y = 1\, \mathsf{in}\, \text{`}(c)\rangle$ is Γ'. The environment that goes into the first antiquotation of c is $\Gamma' \hookleftarrow \{x : \mathtt{Pre\ int}\}$ and the second is $\Gamma' \hookleftarrow \{y : \mathtt{Pre\ int}\}$. We want these environment to "satisfy" Γ. That is, we want

$$\Gamma' \hookleftarrow \{x : \mathtt{Pre\ int}\} <: \Gamma \text{ and } \Gamma' \hookleftarrow \{y : \mathtt{Pre\ int}\} <: \Gamma$$

Recall that the environments are nothing but records. Therefore this relation is simply record subtyping [Pie02]: Γ_1 is a subtype of Γ_2 if $\Gamma_1(z)$ is a subtype of $\Gamma_2(z)$ for all $z \in dom(\Gamma_2)$ (thus, $\mathtt{Pre}\ A <: \mathtt{Abs}$). Because x and y are critical variables, let us write Γ' as $\{x : \theta_1, y : \theta_2\}\rho$ where the field variable θ_i means either absence of the binding or presence of a type. Then, we want Γ to be the least upper bound (lub) of $\{x : \mathtt{Pre\ int}, y : \theta_2\}\rho$ and $\{x : \theta_1, y : \mathtt{Pre\ int}\}\rho$ in the record subtyping lattice. This lub value is $\{x : \theta_1, y : \theta_2\}\rho$ with the condition that $\mathtt{Pre\ int} <: \theta_1$ and $\mathtt{Pre\ int} <: \theta_2$. So, we can give G the type

$$\Box(\{x : \theta_1, y : \theta_2\}\rho \triangleright \alpha) \to \Box(\{x : \theta_1, y : \theta_2\}\rho \triangleright \alpha * \alpha) \text{ where } \mathtt{Pre\ int} <: \theta_1 \text{ and } \mathtt{Pre\ int} <: \theta_2$$

Let us now check if this type can fulfill our needs.

- Suppose G is applied on $\langle 0\rangle$. The operand has no requirements for x or y. Therefore we can set $\theta_1 = \mathtt{Abs}$ and $\theta_2 = \mathtt{Abs}$. This satisfies the constraints of the type because $\mathtt{Pre\ int} <: \mathtt{Abs}$ and results in $\Box(\{x : \mathtt{Abs}, y : \mathtt{Abs}\}\rho \triangleright \mathtt{int} * \mathtt{int})$: a runnable type (simply instantiate ρ with $\varnothing$ and note that `Abs` stands for the absence of the binding). This is a desired type as mentioned before.

- Suppose G is applied on $\langle x+1\rangle$. The operand requires x to come from the outer environment as an integer and has no requirements for y. Therefore we can set $\theta_1 = \mathtt{Pre\ int}$ and $\theta_2 = \mathtt{Abs}$. This satisfies the constraints of the type because $\mathtt{Pre\ int} <: \mathtt{Pre\ int}$ and $\mathtt{Pre\ int} <: \mathtt{Abs}$, resulting in $\Box(\{x : \mathtt{Pre\ int}, y : \mathtt{Abs}\}\rho \triangleright \mathtt{int} * \mathtt{int})$. Again, a desired type for the application.

To check that the type is sound, suppose we apply G on a fragment that requires y to be a boolean value such as $\langle y\, \&\, \mathtt{false}\rangle$. This would set θ_2 to $\mathtt{Pre\ bool}$. The application would result in the fragment $\langle \mathsf{let}\, x = 1\, \mathsf{in}\, y\, \&\, \mathtt{false}, \mathsf{let}\, y = 1\, \mathsf{in}\, y\, \&\, \mathtt{false}\rangle$ which clearly is type-incorrect and should be rejected. With the substitution $\theta_2 = \mathtt{Pre\ bool}$, the constraint $\mathtt{Pre\ int} <: \theta_2$ fails because `int` is not a subtype of `bool`. Hence the type system would reject the application as expected.

8.2 Subtyped Record Calculus

Using a type with subtyping constraints works very well for our purposes. The question is, does there exist any type system that could give us such types? The answer is, fortunately, yes. Odersky, Sulzmann and Wehr define HM(X) [OSW99] which is a Hindler/Milner-style type system parameterized on a constraint system X. Given a constraint system that satisfies the requirements, HM(X) can provide a type system with a principal type inference algorithm for free. Pottier defines SRC, a constraint system that combines subtyping, records, and row variables [Pot00]. SRC is a sound constraint system in the style of [OSW99], and yields the type system HM(SRC).

Taking advantage of the close relation between staged computation and record calculus, we can use HM(SRC) to type-check staged expressions after translating them to the record calculus. The translation is the same.

SRC is a very powerful constraint system; it is more powerful than we need. It provides *conditional constraints* that handle the tricky record concatenation problem. We do not need record concatenation. Handling record extension suffices in our context. So we ignore conditional constraints. SRC is parameterized on a *ground signature*. Pottier defines a sample ground signature in [Pot00] and uses the resulting system to obtain a type system that can type-check accurate pattern matching, record concatenation and first-class messages using a single framework. We use the same ground signature Pottier defines. Below are the modifications we need to make to the definition of types in Figure 11. This definition is then fed into SRC to obtain a type system with record subtyping and row variables. It is straightforward to check that these definitions preserve the properties of the ground signature.

$$\begin{array}{rcl} T & \in & RType ::= \ldots \mid \top \mid \bot \\ F & \in & RField ::= \ldots \mid \texttt{bot} \end{array}$$

The modifications are straightforward. Pottier requires types to form a lattice where the smallest and greatest elements are nullary. Therefore we add $\top$ and a $\bot$ to the types, and `bot` to the definition of fields. The ordering between types is standard and the same as in [Pot00]; the left-hand-side of a function type is contravariant, its right-hand-side and record types are covariant. Fields are ordered as $\texttt{bot} < \texttt{Pre}\,A < \texttt{Abs}$. Note that this ordering is simpler than Pottier's, where `Abs` and $\texttt{Pre}\,A$ are incomparable and have a common upper value, $\texttt{Either}\ A$. Pottier uses that ordering again to handle the record concatenation problem. Because we do not need concatenation, a simple chain ordering suffices.

HM(X) assumes core ML as the syntax of its language. New syntax can be treated as function applications where the functions are kept in a pervasive environment. In HM(SRC), record operations have the following types.

$$\begin{array}{rcl} \{\} & : & \{\} \\ \_\cdot a & : & \forall\alpha,\rho.\{a : \texttt{Pre}\,\alpha\}\rho \to \alpha \\ \_\,\mathsf{with}\,\{a = \_\} & : & \forall\theta,\alpha,\rho.\{a : \theta\}\rho \to \alpha \to \{a : \texttt{Pre}\,\alpha\}\rho \end{array}$$

In the record language we distinguished record variables from regular variables in order to keep the type system sound with respect to staged semantics. We need to do the same in HM(SRC). We do not elaborate this issue since it is straightforward.

A question arises for the requirement of a lattice in the definition of types. Suppose f is a function with two applications: $f(1)$ and $f(\texttt{true})$. Let the input type of f be α. Type inference

collects for α the constraints `int` $<: \alpha$ and `bool` $<: \alpha$, which give $\alpha = \top$ assuming a standard flat lattice, and the type is accepted by the type system. However, this could be considered as an ill-typed situation by many type systems. Even though Pottier requires types to form a lattice, this is not a requirement for the soundness of the constraint system, but for the correctness of constraint simplification algorithms which improve readability of constraints that are attached to types. There are simplification algorithms that do not impose a lattice structure but are less efficient than Pottier's [Fre97, Reh98], as well as other record subtype systems that do not assume lattices [EST95].

Nanevski [Nan02] defines a staged language where the type of a fragment contains the free variables of the fragment, called the "support set". A subtyping rule is defined, making it possible to use a code fragment in a context that provides more variables than required. In other words, subtyping loosens the support set of a code value. While this idea is very useful, it does not provide a general solution to the subtyping problem we cover here, because a support set contains only the names of the variables; no type information is stored. Hence, we can only reason about existence or absence of a variable in the support set. On the other hand, Pottier's subtyping constraints give the ability to keep the types related to variables and also the subtyping relations between these types. Therefore, Pottier's system subsumes Nanevski's definition of subtyping in our context.

8.3 Staged Semantics and Subtyped Record Calculus

Subtyping is about being able to replace a value of some type with another value of subtype without sacrificing safety. We mentioned above that it is OK to supply a "richer" environment to a code fragment than the environment it expects. Suppose the code fragment's type is $\Box(\Gamma \triangleright A)$ and it is provided with the environment Γ'. Then, what we really want is that the $\Box(\Gamma \triangleright A)$ behave like $\Box(\Gamma' \triangleright A)$. That is

$$\Box(\Gamma \triangleright A) <: \Box(\Gamma' \triangleright A)$$

Because Γ' is richer than Γ, we have $\Gamma' <: \Gamma$. This suggests that the subtyping relation for the environment component of a $\Box$-type is *contravariant*. With a similar reasoning, it is easy to find that the relation for the type part is *covariant*. So we have

$$\frac{\Gamma' <: \Gamma \quad A <: A'}{\Box(\Gamma \triangleright A) <: \Box(\Gamma' \triangleright A')}$$

Recall that $\Box$-types are translated to function types. That is, $[\![\Box(\Gamma \triangleright A)]\!] = [\![\Gamma]\!] \to [\![A]\!]$, where the left-hand-side type is contravariant and the right-hand-side is covariant [Pie02]. Therefore, subtyping relations are preserved by the translation.

We can again use the record calculus type system to type-check staged expressions. The following state related properties. We first state that the record type system with subtyping is sound with respect to staged semantics. This is done via Preservation and Progress again.

Theorem 8.1 (Preservation)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e) = \varnothing$ (i.e. e_1 does not have any stage-0 free variables). If $e_1 \longrightarrow_n e_2$ and $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$ is typable in HM(SRC), then $[\![e_2]\!]_{\{\},R_1,\ldots,R_n}$ is also typable in HM(SRC) to the same type under same assumptions.*

Proof. By Theorem 7.1 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. Because HM(SRC) is a sound type system, it has the Preservation property. Therefore, $[\![e_2]\!]_{\{\},R_1,\ldots,R_n}$ can be given the same type of $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$. □

Theorem 8.2 (Progress)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression. If $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$ is typable in HM(SRC), then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. Similar to Theorem 7.4, by reverse reasoning about the structure of types that a result of the translation can get. □

Theorem 8.3 (Soundness)**.** *HM(SRC) is a sound type system with respect to staged semantics.*

Proof. By Theorems 8.1 and 8.2. □

The theorem below says that anything typable in the λ^{open}_{poly} type system is also typable in the record calculus with subtyping. As illustrated by the library specialization example, subtyped record calculus can type more expressions.

Theorem 8.4. *Let e be a stage-n λ^{gen}_{poly} program. If $\Delta_0, \ldots, \Delta_n \vdash_S e : A$ then in HM(SRC), $[\![e]\!]_{R_0,\ldots,R_n}$ is typable to $[\![A]\!]$ with no constraints under the environment $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n}$.*

Proof. By Theorem 7.6, we have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$. Because record calculus with subtyping subsumes record calculus without subtyping, the same judgment holds in HM(SRC), too. □

We do not give a definition for a standalone staged type system that has subtyping. We leave it as future work.

9 Extending λ^{gen}_{poly} with Pluggable Declarations

In this section we discuss how we can extend the staged language with pluggable declarations. We list the necessary additions to the syntax, and static and dynamic semantics. We show that the extension retains soundness of the staged type system. We then show pluggable declarations are syntactic sugaring. We finally discuss how the translation into the record calculus is affected.

Necessary extensions to the syntax and semantics are shown in Figure 15. We refer to this language as λ^{decl}_{poly}.

9.1 Soundness of the λ^{decl}_{poly} Type System

We now show that the new staged type system that handles pluggable declarations is sound. The original staged type system, λ^{gen}_{poly}, was already proven sound in [KYC06]. We show that the Preservation and Progress are still valid.

Theorem 9.1 (Preservation)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_1 : A$ and $e_1 \longrightarrow_n e_2$, then $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_2 : A$.*

Proof. By structural induction on e_1. □

Theorem 9.2 (Progress)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_1 : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e_1. □

Syntax

$$Exp ::= \ldots \mid \langle\rangle \mid \langle x = e\rangle \mid \mathsf{let}\ \backprime(e)\ \mathsf{in}\ e$$

Values

$$\begin{array}{lll} Val^0 & ::= \ldots \mid \langle\rangle \mid \langle x = v^1\rangle & \\ Val^{n+1} & ::= \ldots \mid \langle\rangle \mid \langle x = v^{n+2}\rangle & \\ & \quad \mid \mathsf{let}\ \backprime(v^n)\ \mathsf{in}\ v^{n+1} & (\text{if } n > 0) \end{array}$$

Operational Rules

$$\textsf{ESDEC} \quad \frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

$$\textsf{ESLET2} \quad \frac{e_1 \longrightarrow_n e_1'}{\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_1')\ \mathsf{in}\ e_2}$$

$$\frac{e_1 \in Val^n \quad e_2 \longrightarrow_{n+1} e_2'}{\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2'}$$

$$\frac{e_1 \in Val^1 \quad e_2 \in Val^1}{\mathsf{let}\ \backprime(\langle x = e_1\rangle)\ \mathsf{in}\ e_2 \longrightarrow_1 \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}$$

$$\frac{e_2 \in Val^1}{\mathsf{let}\ \backprime(\langle\rangle)\ \mathsf{in}\ e_2 \longrightarrow_1 e_2}$$

Types

$$SType ::= \ldots \mid \Diamond(\Gamma \triangleright \Gamma')$$

Typing Rules

$$\textsf{TSEDEC} \quad \Delta_0, \ldots, \Delta_n \vdash_P \langle\rangle : \Diamond(\Gamma \triangleright \Gamma)$$

$$\textsf{TSDEC} \quad \frac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \lessdot \{x : A\})}$$

$$\textsf{TSLET2} \quad \frac{\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_{n+1} \quad \Delta_0, \ldots, \Delta_n, \Gamma' \vdash_P e_2 : A}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2 : A}$$

Other typing rules are copied from λ^{gen}_{poly}.

Other Definitions

$$\begin{array}{rcl} FV^n(\langle x = e\rangle) & = & FV^{n+1}(e) \\ FV^{n+1}(\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2) & = & FV^n(e_1) \cup FV^{n+1}(e_2) \\ \langle y = e\rangle[x \backslash e']^n & = & \langle y = e[x \backslash e']^n\rangle \\ (\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2)[x \backslash e']^{n+1} & = & \mathsf{let}\ \backprime(e_1[x \backslash e']^n)\ \mathsf{in}\ e_2[x \backslash e']^{n+1} \\ \langle x = e\rangle[y^m \overset{n}{\mapsto} z] & = & \langle x = e[y^m \overset{n+1}{\mapsto} z]\rangle \\ (\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2)[y^m \overset{n+1}{\mapsto} z] & = & \mathsf{let}\ \backprime(e_1[y^m \overset{n}{\mapsto} z])\ \mathsf{in}\ e_2[y^m \overset{n+1}{\mapsto} z] \end{array}$$

Figure 15: Extending λ^{gen}_{poly} with pluggable declarations. We refer to the resulting language as λ^{decl}_{poly}.

9.2 Pluggable Declarations are Syntactic Sugar

We now show that pluggable declarations are syntactic sugaring; what we can express using them can already be expressed using the existing quotation/antiquotation mechanism.

First, define the following desugaring function, $\delta(\cdot)$, from λ^{decl}_{poly} expressions to λ^{gen}_{poly} expressions. (Cases not shown simply recurse into subexpressions.)

$$\begin{aligned}
\delta(\langle\rangle) &= \lambda y.\langle \backprime(y)\rangle \\
\delta(\langle x = e\rangle) &= (\lambda v.\lambda y.\langle \mathsf{let}\, x = \backprime(v)\, \mathsf{in}\, \backprime(y)\rangle)\ \langle\delta(e)\rangle \\
\delta(\mathsf{let}\ \backprime(e_1)\ \mathsf{in}\ e_2) &= \backprime(\delta(e_1)\ \langle\delta(e_2)\rangle)
\end{aligned}$$

Note that the desugaring function for the quoted declaration $\langle x = e\rangle$ produces a function application where $\langle\delta(e)\rangle$ is the operand. Not placing it inside the λ-abstraction allows for any antiquotations to be evaluated, which would not be possible under the abstraction at stage-0.

Also define the desugaring function below from λ^{decl}_{poly} types to λ^{gen}_{poly} types. Cases not shown simply recurse into sub-components.

$$\delta(\Diamond(\Gamma_1 \triangleright \Gamma_2)) = \Box(\delta(\Gamma_2) \triangleright A) \to \Box(\delta(\Gamma_1) \triangleright A) \text{ for any type } A.$$

The following theorem states that desugaring preserves operational semantics; evaluation of an expression with pluggable declarations is equivalent to evaluating its desugared version.

Theorem 9.3. *Let e_1 be a λ^{decl}_{poly} expressions such that $e_1 \longrightarrow_n e_2$. Then $\delta(e_1) \longrightarrow^*_n \delta(e_2)$.*

In addition to operational semantics, we also show that desugaring preserves typing. The theorem below states that anything typable in λ^{decl}_{poly} is also typable in λ^{gen}_{poly} after desugaring.

Theorem 9.4. $\Delta_0, \ldots, \Delta_n \vdash_P e : A \implies \delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e) : \delta(A)$

Proof. By structural induction on e. □

Note that although the desugaring function $\delta(\cdot)$ is surjective, it is not injective. This prevents the desugaring from being reversible. This is the reason why the relation shown in Theorem 9.4 is one-directional ($\implies$) instead of an if-and-only-if ($\iff$) relation.

The properties shown in Theorems 9.3 and 9.4 mean that anything expressible using pluggable declarations is also expressible using core program generation facilities (i.e. quotable expressions). In other words, pluggable declarations do not bring extra expressive power. Despite this fact, they are useful because they eliminate the need to use higher-order functions which may make a program hard to understand and manipulate for programmers.

9.3 Translation into Record Calculus

The fact that a declaration is syntactically not an expression but that it becomes an expression when quoted brings a problem in typing. We first explain the problem, then elaborate a solution.

Analogous to a quoted expression, the immediate idea is to represent a quoted declaration as a function that takes in an environment as its input. As the output, the function produces another environment. More concretely, the quoted declaration $\langle x = e\rangle$ at stage n would be translated to the function $\lambda r.r\,\mathsf{with}\,\{x = e'\}$ where e' is the translation of e. This function would have to be

given a type of the form $\Gamma_1 \to \Gamma_2$. However, this type cannot be constructed from the definition in Figure 11 given for the record type system. Adding this type to the grammar of types introduces the problems mentioned in Section 5.3. To overcome this problem, we can translate quoted declarations to higher order functions just like desugaring. For instance, the declaration $\langle x = 1 \rangle$ would first be converted to $\lambda y.\langle \mathsf{let}\, x = 1\, \mathsf{in}\, \backprime(y)\rangle$ and then translated to record calculus. However, this translation brings another problem. Misuses of declarations as functions cannot be detected; e.g. $\langle x = 1\rangle\langle 0\rangle$ would pass the type checker. To overcome this problem, we add to the record language a special variable and constant κ that has type κ. This constant and its type do not exist in the staged language. We then extend the definition of record types as follows:

$$T \quad \in \quad RType ::= \ldots \mid \kappa$$

The translation function is then extended with the following definitions.

$[\![\langle\rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\, y(r)$

$[\![\langle x = e\rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\, \mathsf{let}\, z = [\![e]\!]_{R_0,\ldots,R_n,r}\, \mathsf{in}\, y(r\, \mathsf{with}\, \{x = z\})$ where r, y, z are fresh.

$[\![\mathsf{let}\, \backprime(e_1)\, \mathsf{in}\, e_2]\!]_{R_0,\ldots,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\, (\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\, R_{n+1}$ where r is fresh.

We now show that the record calculus provides a sound type system with respect to λ^{decl}_{poly} operational semantics. Following the same approach we did for λ^{gen}_{poly}, we first state the theorem about the relation between two operational semantics, which provides the preservation theorem for free. This is followed by the progress theorem.

Theorem 9.5 (Operational Equivalence)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $e_1 \longrightarrow_n e_2$, then $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. □

Theorem 9.6 (Preservation)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$ and $e_1 \longrightarrow_n e_2$, then $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$.*

Proof. By the same proof method we used in Theorem 7.2: By Theorem 9.5 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. By the Preservation property of the record type system with respect to the record semantics, we have $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$. □

Theorem 9.7 (Progress)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e. □

Theorem 9.8 (Soundness)**.** *Let e_1 be a stage-0 λ^{decl}_{poly} expression. If $\varnothing \vdash_R [\![e_1]\!]_{\{\}} : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $e_1 \longrightarrow^*_0 e_2$ and $\varnothing \vdash_R [\![e_2]\!]_{\{\}} : A$.*

Proof. Follows from Theorems 9.6 and 9.7. □

We finally show that using the record type system to type-check λ^{decl}_{poly} expression yields a type system that is as powerful as the λ^{decl}_{poly} type system. We first extend the definition of type translation:

$$[\![\Diamond(\Gamma_1 \triangleright \Gamma_2)]\!] = \kappa \to ([\![\Gamma_2]\!] \to B) \to ([\![\Gamma_1]\!] \to B) \text{ for any } B$$

Note that the first half of Lemma 6.2 still holds. That is, for any λ^{decl}_{poly} type A, $[\![A]\!] \in RLegType$. However, the backwards direction, which says that for any $A' \in RLegType$ there exists a λ^{gen}_{poly} type A such that $[\![A]\!] = A'$, is no longer valid due to the extension of the type system with κ. Because of this fact, the relation between λ^{decl}_{poly} type system and record type system is no longer bi-directional (i.e. not an iff relation). The relation we have now says that anything typable in λ^{decl}_{poly} is also typable in the record calculus. This property essentially means that record calculus provides a type system as powerful as λ^{decl}_{poly}, and is much more important than the other direction.

Theorem 9.9. *Let e be a stage-n λ^{decl}_{poly} program. Then*

$$\Delta_0, \ldots, \Delta_n \vdash_P e : A \implies [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$$

Proof. By structural induction on e. □

In conclusion, the record calculus provides a sound type system that is as powerful as λ^{decl}_{poly}.

10 Extending λ^{gen}_{poly} with References

The translation that converts λ^{gen}_{poly} expressions to record calculus expressions does not preserve the order of evaluation. In the staged semantics holes in a quoted fragment are evaluated first. However, the translation converts a quoted expression to a lambda abstraction which would immediately evaluate to a closure, giving the behavior that the holes would be evaluated only when the quoted expression is "run". Because we did not have any side-effects in the language (and because non-termination is ignored by the type system due to undecidability), this difference in the order of evaluation did not matter. The translation, however, would be problematic in the presence of side effects. Consider the stage-0 expression $\langle x + \backprime(\mathsf{ref}\ 0; \langle 1 \rangle)\rangle$. Its transformation would be $\lambda r_1.(r_1 \cdot x + (\mathsf{ref}\ 0; \lambda r.1) r_1)$. Executing the staged expression evaluates the hole, which results in a new memory allocation. On the other hand, its translation is a lambda-abstraction and immediately evaluates to a closure without expanding the memory.

In this section we first add references to the record calculus and the staged language. We then modify the translation to preserve the order of execution and show several formal properties. We conclude with a discussion of how to handle pluggable declarations in the presence of references.

10.1 Adding References to the Staged and Record Calculi

The following syntax is added to the staged language. The resulting language is the same as λ^{open}_{poly} except open. The same syntax is added to the record calculus as well.

$$e \in Exp ::= \ldots \mid \ell \mid \mathsf{ref}\ e \mid\ !e \mid e{:=}e$$
$$\ell \in Location$$

The operational semantics of λ^{gen}_{poly} is extended with references as shown in Figure 16. The definitions of FV, FV^n and substitution are extended straightforwardly. A similar extension to the record calculus is in Figure 17.

An extension is also made to the staged type system as shown in Figure 18. This extension requires a new reference type and a store typing to be added to the judgments.

$$Val^0 ::= \ldots \mid \ell$$
$$Val^{n+1} ::= \ldots \mid \ell \mid \mathsf{ref}\ v^{n+1} \mid\ !v^{n+1} \mid\ v^{n+1}{:=}v^{n+1}$$
$$\mathcal{S} \in Store = Location \rightharpoonup Val^0$$

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

ESSYM
$$\frac{z \text{ is fresh}}{\mathcal{S}, \lambda^* x.e \longrightarrow_n \mathcal{S}, \lambda z.[x^n \overset{n}{\mapsto} z]e}$$

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

ESAPP
$$\frac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \longrightarrow_n \mathcal{S}', e_1' e_2} \quad \frac{e_1 \in Val^n \quad \mathcal{S}, e_2 \longrightarrow_n \mathcal{S}', e_2'}{\mathcal{S}, e_1 e_2 \longrightarrow_n \mathcal{S}', e_1 e_2'} \quad \frac{e_2 \in Val^0}{\mathcal{S}, (\lambda x.e) e_2 \longrightarrow_0 \mathcal{S}, e[x \backslash e_2]^0}$$
$$\frac{e_2 \in Val^0}{\mathcal{S}, (\mathsf{fix}\, f(x).\, e) e_2 \longrightarrow_0 \mathcal{S}, e[f \backslash \mathsf{fix}\, f(x).\, e]^0 [x \backslash e_2]^0}$$

ESLET
$$\frac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_n \mathcal{S}', \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2} \quad \frac{e_1 \in Val^0}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_0 \mathcal{S}, e_2[x \backslash e_1]^0}$$
$$\frac{e_1 \in Val^{n+1} \quad \mathcal{S}, e_2 \longrightarrow_{n+1} \mathcal{S}', e_2'}{\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_{n+1} \mathcal{S}', \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2'}$$

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

ESUBOX
$$\frac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, `(e) \longrightarrow_{n+1} \mathcal{S}', `(e')} \quad \frac{e \in Val^1}{\mathcal{S}, `(\langle e \rangle) \longrightarrow_1 \mathcal{S}, e}$$

ESRUN
$$\frac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{run}(e) \longrightarrow_n \mathcal{S}', \mathsf{run}(e')} \quad \frac{e \in Val^1}{\mathcal{S}, \mathsf{run}(\langle e \rangle) \longrightarrow_0 \mathcal{S}, e}$$

ESLIFT
$$\frac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{lift}(e) \longrightarrow_n \mathcal{S}', \mathsf{lift}(e')} \quad \frac{e \in Val^0}{\mathcal{S}, \mathsf{lift}(e) \longrightarrow_0 \mathcal{S}, \langle e \rangle}$$

ESREF
$$\frac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_n \mathcal{S}', \mathsf{ref}\ e'} \quad \frac{e \in Val^0 \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_0 \mathcal{S} \nleftarrow \{\ell : e\}, \ell}$$

ESDEREF
$$\frac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, !e \longrightarrow_n \mathcal{S}', !e'} \quad \frac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \longrightarrow_0 \mathcal{S}, v}$$

ESASGN
$$\frac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_n \mathcal{S}', e_1'{:=}e_2} \quad \frac{e_1 \in Val^n \quad \mathcal{S}, e_2 \longrightarrow_n \mathcal{S}', e_2'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_n \mathcal{S}', e_1{:=}e_2'} \quad \frac{e_2 \in Val^0}{\mathcal{S}, \ell{:=}e_2 \longrightarrow_0 \mathcal{S} \nleftarrow \{\ell : e_2\}, e_2}$$

Figure 16: The operational semantics of λ^{open}_{poly} with references.

$$A \in SType ::= \ldots \mid A\ \mathsf{ref}$$
$$\Sigma \in SStoreTyping = Location \rightharpoonup SType$$

The store typing is used to look up the types of the locations occurring free (see the TSLOC rule). Let-bindings now have to take memory expansion into account when generalizing types. This is done by the expansive^n predicate in [KYC06], which is an adaptation of Wright's original definition [Wri95].

DRAFT

$$v \in RVal ::= \ldots \mid \ell$$
$$\mathcal{S} \in RStore = Location \rightharpoonup RVal$$

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

ERLET
$$\frac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \longrightarrow_R \mathcal{S}', \mathsf{let}\, w = e_1' \,\mathsf{in}\, e_2} \quad \frac{e_1 \in RVal}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \longrightarrow_R \mathcal{S}, e_2[w \backslash e_1]}$$

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

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

ERREF
$$\frac{\mathcal{S}, e \longrightarrow_R \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_R \mathcal{S}', \mathsf{ref}\ e'} \quad \frac{e \in RVal \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_R \mathcal{S} \lessdot \{\ell : e\}, \ell}$$

ERDEREF
$$\frac{\mathcal{S}, e \longrightarrow_R \mathcal{S}', e'}{\mathcal{S}, !e \longrightarrow_R \mathcal{S}', !e'} \quad \frac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \longrightarrow_R \mathcal{S}, v}$$

ERASGN
$$\frac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, e_1 := e_2 \longrightarrow_R \mathcal{S}', e_1' := e_2} \quad \frac{e_1 \in RVal \quad \mathcal{S}, e_2 \longrightarrow_R \mathcal{S}', e_2'}{\mathcal{S}, e_1 := e_2 \longrightarrow_R \mathcal{S}', e_1 := e_2'}$$
$$\frac{e_2 \in RVal}{\mathcal{S}, \ell := e_2 \longrightarrow_R \mathcal{S} \lessdot \{\ell : e_2\}, e_2}$$

Figure 17: The operational semantics of record calculus with references.

Definition 10.1. expansive$^n(e)$ is as defined in [KYC06], except the following cases:

$$\begin{aligned} \text{expansive}^n(\lambda x.e) &= \text{false} \\ \text{expansive}^n(\lambda^* x.e) &= \text{false} \\ \text{expansive}^n(\mathsf{fix}\, f(x).\, e) &= \text{false} \end{aligned}$$

This definition still preserves *demotion-closedness* because only stage-1 values can be demoted to stage-0, and stage-1 values do not contain holes not filled in yet. In other words, any hole that possibly exists under the abstraction has to be filled in before the lambda abstraction can be demoted to stage-0, making the abstraction non-expansive at any stage.

The record calculus type system is extended analogously as in Figure 19. The definition of possibly memory-expanding expressions is also given. In the typing rules omitted in Figure 19, the store typing is simply threaded through a proof tree.

Below we define safe-β-reductions: reductions that are guaranteed to not modify the store. This is used in the main theorem which states that translating a staged expression and then evaluating it in the record semantics produces the same side-effects as evaluation using the staged semantics.

Definition 10.2 (Side-effect freedom)**.** An expression e is said to be "side-effect-free", denoted as $SEF(e)$, if it is guaranteed not to change the store when evaluated. The formal definition is as follows:

$$\textsf{TSREF}\quad \frac{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e : A}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S \textsf{ref}\ e : A\ \textsf{ref}} \qquad \textsf{TSDEREF}\quad \frac{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e : A\ \textsf{ref}}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S !e : A}$$

$$\textsf{TSASGN}\quad \frac{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e_1 : A\ \textsf{ref} \quad \Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e_2 : A}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e_1{:=}e_2 : A} \qquad \textsf{TSLOC}\quad \frac{\Sigma(\ell) = A}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S \ell : A\ \textsf{ref}}$$

$$\textsf{TSLETIMP}\quad \frac{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e_1 : A \quad \text{expansive}^n(e_1) \quad \Sigma;\Delta_0,\ldots,\Delta_n \mathbin{\textsf{<+}} \{x : A\} \vdash_S e_2 : B}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S \textsf{let}\, x = e_1\, \textsf{in}\, e_2 : B}$$

$$\textsf{TSLETAPP}\quad \frac{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S e_1 : A \quad \neg\text{expansive}^n(e_1) \quad \Sigma;\Delta_0,\ldots,\Delta_n \mathbin{\textsf{<+}} \{x : \text{GEN}_A(\Sigma,\Delta_0,\ldots,\Delta_n)\} \vdash_S e_2 : B}{\Sigma;\Delta_0,\ldots,\Delta_n \vdash_S \textsf{let}\, x = e_1\, \textsf{in}\, e_2 : B}$$

Figure 18: The λ^{open}_{poly} typing rules to handle references. Other rules are the same as before except propagating the store typing.

$$\textsf{TRREF}\quad \frac{\Sigma;\Delta \vdash_R e : A}{\Sigma;\Delta \vdash_R \textsf{ref}\ e : A\ \textsf{ref}}$$

$$\textsf{TRDEREF}\quad \frac{\Sigma;\Delta \vdash_R e : A\ \textsf{ref}}{\Sigma;\Delta \vdash_R !e : A}$$

$$\textsf{TRASGN}\quad \frac{\Sigma;\Delta \vdash_R e_1 : A\ \textsf{ref} \quad \Sigma;\Delta \vdash_R e_2 : A}{\Sigma;\Delta \vdash_R e_1{:=}e_2 : A}$$

$$\textsf{TRLOC}\quad \frac{\Sigma(\ell) = A}{\Sigma;\Delta \vdash_R \ell : A\ \textsf{ref}}$$

$$\textsf{TRLETIMP}\quad \frac{\Sigma;\Delta \vdash_R e_1 : A \quad \text{expansive}(e_1) \quad \Sigma;\Delta \mathbin{\textsf{<+}} \{x : A\} \vdash_R e_2 : B}{\Sigma;\Delta \vdash_R \textsf{let}\, x = e_1\, \textsf{in}\, e_2 : B}$$

$$\textsf{TRLETAPP}\quad \frac{\Sigma;\Delta \vdash_R e_1 : A \quad \neg\text{expansive}(e_1) \quad \Sigma;\Delta \mathbin{\textsf{<+}} \{x : \text{GEN}_A(\Sigma,\Delta)\} \vdash_R e_2 : B}{\Sigma;\Delta \vdash_R \textsf{let}\, x = e_1\, \textsf{in}\, e_2 : B}$$

$$\begin{aligned}
&\text{expansive}(c) = \text{false}\\
&\text{expansive}(w) = \text{false}\\
&\text{expansive}(\lambda w.e) = \text{false}\\
&\text{expansive}(\textsf{fix}\, f(w).\, e) = \text{false}\\
&\text{expansive}(e_1 e_2) = \text{true}\\
&\text{expansive}(\textsf{let}\, x = e_1\, \textsf{in}\, e_2) =\\
&\qquad \text{expansive}(e_1) \vee \text{expansive}(e_2)\\
&\text{expansive}(e{\cdot}w) = \text{expansive}(e)\\
&\text{expansive}(\{\}) = \text{false}\\
&\text{expansive}(e_1\ \textsf{with}\ \{w = e_2\}) =\\
&\qquad \text{expansive}(e_1) \vee \text{expansive}(e_2)\\
&\text{expansive}(\ell) = \text{false}\\
&\text{expansive}(\textsf{ref}\ e) = \text{true}\\
&\text{expansive}(!e) = \text{expansive}(e)\\
&\text{expansive}(e_1{:=}e_2) =\\
&\qquad \text{expansive}(e_1) \vee \text{expansive}(e_2)
\end{aligned}$$

Figure 19: The new typing rules to handle references in the record calculus. These are standard [Har94, Wri95].

$$\begin{aligned}
SEF(c) &= \text{true}\\
SEF(w) &= \text{true}\\
SEF(\lambda w.e) &= \text{true}\\
SEF(\textsf{fix}\, f(x).\, e) &= \text{true}\\
SEF(e_1 e_2) &= \text{false}\\
SEF(\textsf{let}\, w = e_1\, \textsf{in}\, e_2) &= SEF(e_1) \wedge SEF(e_2)\\
SEF(e_1\ \textsf{with}\ \{a = e_2\}) &= SEF(e_1) \wedge SEF(e_2)
\end{aligned}$$

$$\begin{aligned}
SEF(e{\cdot}a) &= SEF(e)\\
SEF(\{\}) &= \text{true}\\
SEF(\ell) &= \text{true}\\
SEF(\textsf{ref}\ e) &= \text{false}\\
SEF(!e) &= SEF(e)\\
SEF(e_1{:=}e_2) &= \text{false}
\end{aligned}$$

Definition 10.3 (Safe β-reduction)**.** The following are defined to be safe β-reductions.

$$\begin{array}{rcl}
(\lambda w.e_1)e_2 & \longrightarrow_{|\beta|} & e_1[w\backslash e_2] \text{ if } SEF(e_2) \\
\mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 & \longrightarrow_{|\beta|} & e_2[w\backslash e_1] \text{ if } SEF(e_1) \\
(e_2 \,\mathsf{with}\, \{a_1 = e_1\})\cdot a_2 & \longrightarrow_{|\beta|} & e_2\cdot a_2 \text{ if } a_1 \neq a_2 \text{ and } SEF(e_1) \\
(e_2 \,\mathsf{with}\, \{a = e_1\})\cdot a & \longrightarrow_{|\beta|} & e_1 \text{ if } SEF(e_2) \\
e \,\mathsf{with}\, \{a_1 = e_1\} \,\mathsf{with}\, \{a_2 = e_2\} & \longrightarrow_{|\beta|} & e \,\mathsf{with}\, \{a_2 = e_2\} \,\mathsf{with}\, \{a_1 = e_1\} \\
 & & \text{if } a_1 \neq a_2, SEF(e_1) \text{ and } SEF(e_2) \\
e \,\mathsf{with}\, \{a = e_1\} \,\mathsf{with}\, \{a = e_2\} & \longrightarrow_{|\beta|} & e \,\mathsf{with}\, \{a = e_2\} \text{ if } SEF(e_1)
\end{array}$$

10.2 Accounting for References in the Translation

We present a new version of the translation in Figure 20 that converts hole-filling into function application where holes become arguments. The example we gave at the beginning of the section, $\langle x + \text{`}(\mathsf{ref}\ 0; \langle 1\rangle)\rangle$, for instance, translates to $(\lambda h.\lambda r_1.r_1\cdot x + h(r_1))(\mathsf{ref}\ 0; \lambda r_1.1)$. Call-by-value semantics ensures that holes are evaluated before being filled in, preserving the order of evaluation. Below is the classical exponentiation example, written using a reference. Instead of threading the exponent value through recursive calls, we keep it as a global variable and decrement before each recursion. Even though the translation renames variables, not to harm readability of the code, we do not rename them unless they are accessed from a record. The given code generates the function $(\lambda x.x \times x \times x \times 1)$ which takes the cube of its argument. The translation returns the function

$$\lambda y.(\lambda r.r\cdot x \times (\lambda r.r\cdot x \times (\lambda r.r\cdot x \times (\lambda r.1)r)r)r)\{x = y\}$$

which does the same thing through record operations.

$$\begin{array}{l}
[\![\mathsf{let}\, n = \mathsf{ref}\ 0\,\mathsf{in} \\
\ \mathsf{let}\ pow = \mathsf{fix}\, gen().\, \mathsf{if}\ !n = 0\ \mathsf{then}\ \langle 1\rangle\ \ \mathsf{else}\ \langle x \times \text{`}(n{:=}!n - 1; gen())\rangle \\
\ \mathsf{in}\ n{:=}3; \mathsf{run}\, \langle \lambda x.\text{`}(pow())\rangle]\!]^0 =
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\, n = \mathsf{ref}\ 0\,\mathsf{in} \\
\mathsf{let}\ pow = \mathsf{fix}\, gen().\, \mathsf{if}\ !n = 0\ \mathsf{then}\ (\lambda r.1)\ \ \mathsf{else}\ (\lambda\pi.\lambda r.r\cdot x \times \pi(r))(n{:=}!n - 1; gen()) \\
\mathsf{in}\ n{:=}3; (\lambda\pi.\lambda r.\lambda y.\pi(r \,\mathsf{with}\, \{x = y\}))(pow())\{\}
\end{array}$$

The power function below is yet another version that counts the number of multiplications generated. It is adapted from [KKcS08, §6].

$$\begin{array}{l}
[\![\mathsf{let}\, cnt = \mathsf{ref}\ 0\,\mathsf{in} \\
\ \mathsf{let}\ pow = \mathsf{fix}\, gen(n).\, \lambda x.\mathsf{if}\ n = 0\ \mathsf{then}\ \langle 1\rangle \\
\qquad\qquad\qquad\qquad \mathsf{else}\ cnt{:=}!cnt + 1;\ \langle\, \text{`}(x) \times \text{`}(gen\ n\ x)\rangle \\
\ \mathsf{in}\ \mathsf{run}\, \langle \lambda x.\text{`}(pow\ 3\ \langle x\rangle)\rangle]\!]^0 =
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\, cnt = \mathsf{ref}\ 0\,\mathsf{in} \\
\mathsf{let}\ pow = \mathsf{fix}\, gen(n).\, \lambda x.\mathsf{if}\ n = 0\ \mathsf{then}\ (\lambda r.1) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ cnt{:=}!cnt + 1;\ (\lambda\pi_1.\lambda\pi_2.\lambda r.\pi_1(r) \times \pi_2(r))\ x\ (gen\ n\ x) \\
\mathsf{in}\ (\lambda\pi.\lambda r.\lambda y.\pi(r \,\mathsf{with}\, \{x = y\}))(pow\ 3\ (\lambda r.r\cdot x))\{\}
\end{array}$$

$[\![c]\!]_{R_0,\ldots,R_n} = (c, \texttt{nil})$

$[\![x]\!]_{R_0,\ldots,R_n} = (R_n(x), \texttt{nil})$

$[\![\lambda x.e]\!]_{R_0,\ldots,R_n} = (\lambda z.e_0, H)$
where $[\![e]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} = (e_0, H)$, and z is fresh.

$[\![\lambda^* x.e]\!]_{R_0,\ldots,R_n} = [\![\lambda z.[x^n \overset{n}{\mapsto} z]e]\!]_{R_0,\ldots,R_n}$, where z is fresh

$[\![\mathsf{fix}\, f(x).\, e]\!]_{R_0,\ldots,R_n} = (\mathsf{fix}\, g(z).\, e_0, H)$
where $[\![e]\!]_{R_0,\ldots,R_n \text{ with } \{f=g,x=z\}} = (e_0, H)$, and g, z are fresh.

$[\![e_1 e_2]\!]_{R_0,\ldots,R_n} = (e_1' e_2', zip(H_1, H_2))$
where $[\![e_1]\!]_{R_0,\ldots,R_n} = (e_1', H_1)$ and $[\![e_2]\!]_{R_0,\ldots,R_n} = (e_2', H_2)$.

$[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!]_{R_0,\ldots,R_n} = (\mathsf{let}\, z = e_1' \,\mathsf{in}\, e_2', zip(H_1, H_2))$
where $[\![e_1]\!]_{R_0,\ldots,R_n} = (e_1', H_1)$ and $[\![e_2]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} = (e_2', H_2)$, and z is fresh.

$[\![\langle e \rangle]\!]_{R_0,\ldots,R_n} = ((\lambda \vec{\pi}.\lambda r.e_0)\vec{e_p}, H)$
where $[\![e]\!]_{R_0,\ldots,R_n,r} = (e_0, \{(\vec{\pi}, \vec{e_p})\} :: H)$ and r is fresh.

$[\![\,\backprime(e)]\!]_{R_0,\ldots,R_n,R_{n+1}} = (\pi(R_{n+1}), \{(\pi, e_0)\} :: H)$
where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$ and π is fresh.

$[\![\mathsf{run}(e)]\!]_{R_0,\ldots,R_n} = (e_0\{\}, H)$ where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$.

$[\![\mathsf{lift}(e)]\!]_{R_0,\ldots,R_n} = (\mathsf{let}\, \pi = e_0 \,\mathsf{in}\, \lambda r.\pi, H)$ where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$, and π is fresh.

$[\![\ell]\!]_{R_0,\ldots,R_n} = (\ell, \texttt{nil})$

$[\![\mathsf{ref}\ e]\!]_{R_0,\ldots,R_n} = (\mathsf{ref}\ e_0, H)$ where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$.

$[\![!e]\!]_{R_0,\ldots,R_n} = (!e_0, H)$ where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$.

$[\![e_1 := e_2]\!]_{R_0,\ldots,R_n} = (e_1' := e_2', zip(H_1, H_2))$
where $[\![e_1]\!]_{R_0,\ldots,R_n} = (e_1', H_1)$ and $[\![e_2]\!]_{R_0,\ldots,R_n} = (e_2', H_2)$.

Figure 20: Transformation modified to handle expressions with side-effects.

The example below produces a specialized version of vector multiplication. For this example we assume that there is a built-in function nth that, given i and a list ℓ, returns the i^{th} element of ℓ. This is a two-level specialization. The first level produces a generator specialized for a fixed length; the second level specializes the code further for the values kept in a vector. For instance, $\mathsf{run}(prod\ 2)$ gives

$$\lambda v.\langle \mathsf{nth}\ `(\mathsf{lift}(2))\ w \times `(\mathsf{lift}(\mathsf{nth}\ 2\ v)) + \mathsf{nth}\ `(\mathsf{lift}(1))\ w \times `(\mathsf{lift}(\mathsf{nth}\ 1\ v)) + `(\langle 0\rangle)\rangle$$

and $\langle \lambda w.`(\mathsf{run}(prod\ 2)[5;7])\rangle$ is $\langle \lambda w.(\mathsf{nth}\ 2\ w) \times 5 + (\mathsf{nth}\ 1\ w) \times 7 + 0\rangle$. We can now run this code value and apply it to a vector of length 2, such as $[2;3]$.

$$\begin{array}{l}
[\![\mathsf{let}\ prod = \\
\quad \mathsf{let}\ aux = \mathsf{fix}\, gen(n). \\
\qquad \mathsf{if}\ n = 0\ \mathsf{then}\ \langle\langle 0\rangle\rangle \\
\qquad \mathsf{else}\ \langle\langle \mathsf{nth}\ `(\mathsf{lift}(`(\mathsf{lift}(n))))\ w \times `(\mathsf{lift}(\mathsf{nth}\ `(\mathsf{lift}(n))\ v)) + `(`(gen(n-1)))\rangle\rangle \\
\quad \mathsf{in}\ \lambda n.\langle \lambda v.`(aux\ n)\rangle \\
\mathsf{in}\ (\mathsf{run}\ \langle \lambda w.`(\mathsf{run}(prod\ 2)[5;7])\rangle)[2;3]]\!]^0 =
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\ prod = \\
\quad \mathsf{let}\ aux = \mathsf{fix}\, gen(n). \\
\qquad \mathsf{if}\ n = 0\ \mathsf{then}\ \lambda r_2.\lambda r_1.0 \\
\qquad \mathsf{else}\ (\lambda\pi_4.\lambda\pi_5.\lambda\pi_6.\lambda r_2.\ (\lambda\pi_1.\lambda\pi_2.\lambda\pi_3.\lambda r_1.(\mathsf{nth}\ (\pi_1 r_1)\ (r_1 \cdot w)) \times \pi_2(r_2) + \pi_3(r_2)) \\
\qquad\qquad\qquad\qquad (\mathsf{let}\,\pi = \pi_4(r_2)\,\mathsf{in}\,\lambda r.\pi) \\
\qquad\qquad\qquad\qquad (\mathsf{let}\,\pi = \mathsf{nth}\ (\pi_5\, r_2)\ (r_2 \cdot v)\,\mathsf{in}\,\lambda r.\pi) \\
\qquad\qquad\qquad\qquad (\pi_6(r_2))) \\
\qquad\qquad (\mathsf{let}\,\pi = n\,\mathsf{in}\,\lambda r.\pi) \\
\qquad\qquad (\mathsf{let}\,\pi = n\,\mathsf{in}\,\lambda r.\pi) \\
\qquad\qquad (gen(n-1)) \\
\quad \mathsf{in}\ \lambda n.(\lambda\pi_7.\lambda r.\lambda v'.\pi_7(r\ \mathsf{with}\ \{v = v'\}))(aux\ n) \\
\mathsf{in}\ (\lambda\pi_8.\lambda r.\lambda w'.\pi_8(r\ \mathsf{with}\ \{w = w'\}))((prod\ 2)\{\}[5;7])\{\}[2;3]
\end{array}$$

The translation uses the function zip, defined below where :: is the cons operation:

$$\begin{array}{rcl}
zip(h_1 :: H_1, h_1 :: H_2) & = & (h_1 \cup h_2) :: zip(H_1, H_2) \\
zip(\mathtt{nil}, H_2) & = & H_2 \\
zip(H_1, \mathtt{nil}) & = & H_1
\end{array}$$

The value that is returned by the transformation of an expression e now has the form of a pair: $(e_0,\ \{(\pi_i, e_i)\}_1^m :: \ldots :: \{(\pi_i, e_i)\}_1^p)$. Here, e_0 is the actual result of the transformation where each hole not enclosed by a quotation has been replaced by a unique variable π. The second item in the return value, a list of variable and expression sets, contains these unique hole-filler variables accompanied with the corresponding antiquoted expression. A set in the returned list corresponds to a specific stage. The stage gets closer to 0 as we move from left to right in the list. The transformation of a quotation retrieves the variable-expression pairs from the top of the list, and uses them to "fill" in the holes via function application. For the example given above, $[\![`(\mathsf{ref}\ 0; \langle 1\rangle)]\!]_{\{\},r_1}$ gives $(\pi(r_1), [\{(\pi, \mathsf{ref}\ 0; \lambda r.1)\}])$. Based on this value, $[\![\langle x + `(\mathsf{ref}\ 0; \langle 1\rangle)\rangle]\!]_{\{\}}$ returns $((\lambda\pi.\lambda r_1.r_1 \cdot x + \pi(r_1))(\mathsf{ref}\ 0; \lambda r.1), \mathtt{nil})$. Note that the number of sets returned is equal to the depth of the transformed expression:

Lemma 10.4. *Let e be a stage-n program, and $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$. The length of H is equal to the depth of e, giving $n \geq$ length of H.*

Proof. By a straightforward induction on the structure of e. Note that length of $zip(H_1, H_2)$ is equal to max(length of H_1, length of H_2). The only expression that adds a new item to H is quotation, and the only expression that removes an item from H is antiquotation. □

Translation of types and judgments stays the same. There are two extensions to be made. The first one converts a staged store typing to a record store typing:

$$[\![\{\ell_i : A\}]\!] = \{\ell_i : [\![A]\!]\}$$

The second extension translates a store:

Definition 10.5 (Store translation)**.** Let $\mathcal{S} = \{\ell_i : v_i\}$ be a λ^{open}_{poly} store. Its translation to a λ^{rec}_{poly} store is defined as follows:

$$[\![\{\ell_i : v_i\}]\!] = \{\ell_i : v'_i\} \text{ where } (v'_i, \texttt{nil}) = [\![v_i]\!]_{\{\}}$$

Note that all the values in $\mathcal{S}$ are stage-0 values, and the second item of the result of translating a stage-0 is guaranteed to always be `nil` by Lemma 10.4.

10.3 Relating the Staged and Record Calculi

We now show that the translation preserves the order of evaluation and the record calculus still provides a sound type system with respect to staged semantics with side-effects. The properties related to the record calculus are still valid with the extension made. We do not repeat them. We first give auxiliary definitions and lemmas. The *Close* operation below packs the result of a translation into a single function application. We use the notation $\{(\vec{\pi}, \vec{e})\}$ for the set $\{(\pi_i, e_i)\}^m_1$.

Definition 10.6. Let $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$. *Close* is defined as

$$Close([\![e]\!]_{R_0,\ldots,R_n}) = (\lambda\vec{\pi_m}.(\cdots((\lambda\vec{\pi_1}.e_0)\vec{e_1})\cdots))\vec{e_m}$$

Below is the theorem stating that record operational semantics together with the translation is equivalent to staged operational semantics. The crucial property expressed by this theorem is that translation from the staged language into the record calculus preserves the order of evaluation, and hence the side effects of an expression.

Theorem 10.7. *Let e_1 be a stage-n λ^{open}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$, then $[\![\mathcal{S}]\!], Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) \longrightarrow_R [\![\mathcal{S}']\!], e'_2$ such that $e'_2 \longrightarrow^*_{|\beta|} Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. □

The following definition states the consistency between stores and store typings.

Definition 10.8 (Well typed stores)**.** A store $\mathcal{S}$ is well typed with respect to a store typing Σ, denoted $\Sigma \models \mathcal{S}$, if and only if $dom(\mathcal{S}) = dom(\Sigma)$, and $\Sigma; \varnothing \vdash_R \mathcal{S}(\ell) : \Sigma(\ell)$ for any $\ell \in dom(\mathcal{S})$.

Theorem 10.9 (Preservation)**.** *Let e_1 be a stage-n λ^{open}_{poly} expression. If*

$$\Sigma; \Delta \vdash_R Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) : A \text{ and } \mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$$

such that $\Sigma \models [\![\mathcal{S}]\!]$, then for some $\Sigma' \supseteq \Sigma$ we have

$$\Sigma'; \Delta \vdash_R Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n}) : A \text{ and } \Sigma' \models [\![\mathcal{S}']\!]$$

Proof. By Theorem 10.7 we have $[\![\mathcal{S}]\!], Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) \longrightarrow_R [\![\mathcal{S}']\!], e_2'$ such that $e_2' \longrightarrow^*_{|\beta|} Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$. By the preservation property of the record calculus, there exists Σ'' such that $\Sigma'' \supseteq \Sigma$ and $\Sigma'' \models [\![\mathcal{S}']\!]$ giving the judgment $\Sigma''; \Delta \vdash_R e_2' : A$. Using the preservation property of the record calculus again, and the fact that $e_2' \longrightarrow^*_{|\beta|} Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$ does not modify the store, we get $\Sigma''; \Delta \vdash_R Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n}) : A$. □

Theorem 10.10 (Progress)**.** *Let e_1 be a stage-n λ^{open}_{poly} expression. If $\Sigma; \Delta \vdash_R Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) : A$, then either $e_1 \in Val^n$, or for any store $\mathcal{S}$ such that $\Sigma \models \mathcal{S}$, there exist e_2 and $\mathcal{S}'$ such that $\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$.*

Proof. By structural induction on e_1. □

Theorem 10.11 (Soundness)**.** *Let e_1 be a stage-0 λ^{open}_{poly} expression and $[\![e_1]\!]_{\{\}} = (e_0, \texttt{nil})$. If $\varnothing; \varnothing \vdash_R e_0 : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $[\![e_2]\!]_{\{\}} = (e_0', \texttt{nil})$ and $\varnothing, e_1 \longrightarrow^*_0 \mathcal{S}, e_2$ and $\Sigma; \varnothing \vdash_R e_0' : A$ where $\Sigma \models \mathcal{S}$.*

Proof. Follows from Theorems 10.9 and 10.10. □

We have the following relation between expansion tests of record calculus and staged typing:

Lemma 10.12. *For any stage-n expression e,* $\text{expansive}^n(e) \iff \text{expansive}(e_0)$ *where* $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$.

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

Theorem 7.6, which stated that the record type system combined with the translation is equal to the λ^{open}_{poly} type system, is now stated as follows:

Theorem 10.13. *Let e be a staged program.*

$$\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e : A \iff [\![\Sigma]\!]; [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R Close([\![e]\!]_{R_0,\ldots,R_n}) : [\![A]\!]$$

Proof. By induction on the structure of e. □

Remark 10.14. The definition of expansive^n in [KYC06] is unnecessarily conservative for abstractions. We modified the definition in our work. Without this modification, we would not be able to get the $\iff$ relation in Theorem 10.13, but would only have $\implies$. This is because one can find an expression e such that the original definition of expansive$^n(e)$ from [KYC06] is true, but expansive(e_0) is false, where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$. An example that can be type-checked using the record type system after translation, but that is rejected by λ^{open}_{poly} due to the conservative definition of expansive^n, is below:

$$\langle \mathsf{let}\, id = (\lambda x.\mathsf{let}\, t = \text{`}(\langle 0 \rangle)\, \mathsf{in}\, x)\, \mathsf{in}\, id(1), id(\texttt{true}) \rangle$$

10.4 Handling Pluggable Declarations in the Presence of References

In Section 9 we extended the staged language with pluggable declarations. We also gave a translation to the record calculus. In the presence of references, the translation has to be modified in the same way we did for the core language. The new definition of the translation is given below.

$$[\![\langle\rangle]\!]_{R_0,\ldots,R_n} = (\lambda\kappa.\lambda y.\lambda r.\, y(r),\, \texttt{nil})$$

$$[\![\langle x = e\rangle]\!]_{R_0,\ldots,R_n} = ((\lambda\vec{\pi}.\lambda\kappa.\lambda y.\lambda r.\mathsf{let}\, z = e_0\, \mathsf{in}\, y(r\, \mathsf{with}\, \{x = z\}))\vec{e_p}, H)$$
where $[\![e]\!]_{R_0,\ldots,R_n,r} = (e_0, \{(\vec{\pi}, \vec{e_p})\} :: H)$, and r, z are fresh.

$$[\![\mathsf{let}\ \backprime(e_1)\, \mathsf{in}\, e_2]\!]_{R_0,\ldots,R_n,R_{n+1}} = (\pi\, \kappa\, (\lambda r.e_0')R_{n+1},\ \ zip(\{(\pi, e_0)\} :: H,\, H'))$$
where $[\![e_1]\!]_{R_0,\ldots,R_n} = (e_0, H)$, π, r are fresh, and $[\![e_2]\!]_{R_0,\ldots,R_n,r} = (e_0', H')$

A similar modification to the desugaring function is also needed. Because this change is along the same lines of the change made to the translation function, we omit it.

11 Related Work

We now compare the papers that are closest to our work.

Translating a staged language to record calculus, as motivated in Section 2, has previously been proposed by Kameyama, Kiselyov and Shan [KKcS08]. They translate $\lambda^{\alpha}_{1\nu}$, a two-stage version of Taha and Nielsen's λ^{α} [TN03], to System F [Gir72, Rey74] with tuples and higher order polymorphism. Our work differs from [KKcS08] as follows.

- Our translation is not restricted to two-stage program generation; it is multi-staged.
- Targeting "PG by program construction", our source language allows freely-open fragments, as opposed to the "PG by partial evaluation" approach in [KKcS08] which rejects fragments containing free variables that are not in the scope of an outer binding.
- The translation of $\lambda^{\alpha}_{1\nu}$ is guided by type and environment classifier annotations. Neither the source nor the target language in our translation contains type annotations. The target language, record calculus, already has a principal type inference algorithm defined. We simply use this algorithm to infer types.
- We provide a proof of the equivalence of the dynamic semantics of staged computation and record calculus. For a similar relation, [KKcS08] gives a conjecture.
- We have let-polymorphism (i.e. rank-1 polymorphism) as in λ^{open}_{poly}. We do not allow higher order polymorphism as in $\lambda^{\alpha}_{1\nu}$. This prevents polymorphic types to live across stages. The following program is typable in MetaML-like typing (e.g. in $\lambda^{\alpha}_{1\nu}$) but not in our system.

$$\langle \mathsf{let}\, f = \lambda x.x\, \mathsf{in}\, \backprime(\langle f(1), f(\texttt{true})\rangle)\rangle$$

Chen and Xi [CX03] give a translation to convert fragments to first-order abstract syntax expressions. They represent program variables using deBruijn indices. Their target language is second-order lambda calculus with recursion. One advantage is that they can translate first-order abstract syntax (without holes) back to regular syntax (with quotations). A problem in their

system is "at level $k > 0$, a bound variable merely represents a deBruijn index and a binding may vanish or occur 'unexpectedly'" [CX03]. An example that illustrates this problem in the existence of references is given by Kim, Yi and Calcagno [KYC06, §6.4]. In [CX03], polymorphism is restricted to stage 0 only.

Kim, Yi and Calcagno define a language called λ^{open}_{poly} that allows program generation using freely-open code fragments [KYC06]. The language combines many features such as references, variable hygiene to avoid unintended capture as well as intentional variable capturing, and let-polymorphism together with core program generation facilities of quotation, antiquotation, and "run" (to execute quoted fragments). A sound type system and a principal type inference algorithm are provided. We took λ^{open}_{poly} as a starting point for a staged language.

Nanevski [Nan02] takes the approach of relaxing Davies' notion of closed code [DP96, Dav96] to allow manipulation of open code together with a sound "run" construct. He introduces a new semantic category, *names*, that stands for the free variables in a code piece. Free variables become part of a fragment's type as the "support set" — the variables that the code piece depends on. Only code values that have empty support sets can be "run"; other code values can only be used to fill in holes. Similar to row-polymorphism, there exists support set polymorphism. Pattern matching for code values is introduced. This makes it possible to do computation based on the structure of a fragment. An example is given that performs β-reduction inside quotations. An important feature of the type system is subtyping: the support set of a code value can be loosened to allow its use in different contexts. Because no type information is kept for free variables inside a support set, only the existence or absence of a variable can be expressed. In our work, we use subtyping constraints as defined by Pottier [Pot00]. Subtyping constraints subsume Nanevski's notion of subtyping and provide more expressibility. In particular, they successfully address the subtyping requirement revealed by the library specialization problem. The definition of subtyping in [Nan02] does not suffice for this problem. To our knowledge, a staged type system with subtyping constraints is new.

In addition to these differences, we added *pluggable declarations* to our staged language and showed that pluggable declarations are a syntactic sugaring that helps us avoid higher order functions.

12 Conclusions

Guaranteeing that a program generator produces type-safe programs has received extensive interest in the literature. We have tackled the same problem in the context of PG by program construction. We have shown that the problem reduces to type-checking in record calculus, which is an extensively studied area. This allows us to apply formal properties of the record calculus to program generation. An example is subtyping; we have discussed how a staged type system can be enriched with subtyping constraints. This yields a powerful type system that can successfully address an important application of program generation: the library specialization problem. The close relation between the record and staged calculi exists in the presence of side-effecting expressions as well. We have also shown how to extend the language and the type system with pluggable declarations. We believe that all these extensions are orthogonal; it is possible to combine them without any fundamental problems. None of the extensions, nor the core program generation language, requires the programmer to write any type annotations; existing type-inference algorithms can be used to infer the types. These features form a nice package as a program generation system.

Acknowledgments

The author would like to thank Sam Kamin, Elsa Gunter, and Peter Sestoft for their extremely useful feedback on this paper, and Chung-chieh Shan for pointing out that higher-order functions can be used instead of pluggable declarations.

References

[ACK03] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Codebricks: code fragments as building blocks. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 66–74. ACM Press, 2003.

[AK09] Baris Aktemur and Sam Kamin. Writing customizable libraries - a comparative study. In *The 24th Annual ACM Symposium on Applied Computing (SAC)*, Honolulu, HI, USA, 2009.

[Baw99] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-based Program Manipulation*, pages 4–12, 1999.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[CMT04] C Calcagno, E Moggi, and W Taha. Ml-like inference for classifiers. *Programming Languages and Systems*, 2986:79–93, 2004.

[CX03] 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.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 184, Washington, DC, USA, 1996. IEEE Computer Society.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270, New York, NY, USA, 1996. ACM.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1. Elsevier, 1995.

[Fre97] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 265–277, London, UK, 1997. Springer-Verlag.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris VII, 1972.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. In R. Glueck and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326, Tallinn, Estonia, September 2005. Springer.

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424. Springer-Verlag, 2007.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–53, St. Petersburg Beach, Florida, United States, 1996.

[KCJ03] 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.

[KKcS08] 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.

[KYC06] 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.

[MTBS99] 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.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 206–217, New York, NY, USA, 2002. ACM.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, New York, NY, USA, 1992.

[OMY01] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. Dynjava: Type safe dynamic code generation in java. In *The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*, March 2001.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[PHEK99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[Ré94] Didier Rémy. Type inference for records in natural extension of ml. *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 67–95, 1994.

[Reh98] Jacop Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, 1998.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

[Rhi05] 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.

[SGM+03] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.

[TCLP] Walid Taha, Cristiano Calcagno, Xavier Leroy, and Ed Pizzi. Metaocaml. `http://www.metaocaml.org/`.

[TN03] 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.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *Logic in Computer Science*, pages 176–185, 1994.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[YI06] Yosihiro Yuse and Atsushi Igarashi. A modal type system for multi-level generating extensions with persistent code. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 201–212, New York, NY, USA, 2006. ACM.

[ZHS04] 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

A.1 Record Language

The record calculus λ^{rec}_{poly} satisfies the following standard lemmas.

Lemma A.1 (Weakening/Strengthening)**.** *If $\Delta(w) = \Delta'(w)$ for all $w \in FV(e)$, then $\Delta \vdash_R e : T$ iff $\Delta' \vdash_R e : T$.*

Lemma A.2 (Substitution)**.** *If $\Delta \vdash_R e_2 : T$ and $\Delta \nleftarrow \{w : \forall\vec{\psi}.T\} \vdash_R e_1 : T'$ where $\vec{\psi} \cap FV(\Delta) = \varnothing$, then $\Delta \vdash_R e_1[w \backslash e_2] : T'$.*

Lemma A.3. *If $\Delta \vdash_R e : T$, then $\varphi\Delta \vdash_R e : \varphi T$ for any substitution φ.*

Lemma A.4 (Generalization)**.** *Let $\Delta :: \{w : \sigma'\} \vdash_R e : T$ and $\sigma' \prec \sigma$. Then $\Delta :: \{w : \sigma\} \vdash_R e : A$.*

A.2 Transformation

Definition A.5. inst [KYC06] creates a monotype environment Γ from the polytype environment Δ, such that $\Gamma \prec \Delta$, by instantiating the contained polytypes using distinct renaming substitutions.

Lemma A.6. *We have the following properties:*

- $\Gamma \prec \Delta \iff [\![\Gamma]\!] \prec [\![\Delta]\!]$
- $\mathrm{GEN}_{\mathsf{inst}(\Delta)}(\Delta) = \Delta$
- $\mathrm{GEN}_\Gamma(\Delta) \prec \Delta$ *if* $\Gamma \prec \Delta$.

Lemma A.7. *Let e be a stage-n program and $m \geq n$. Then $FV([\![e]\!]_{R_0,\ldots,R_m}) \subseteq \bigcup_{i=m-n}^{m} FV(R_i)$.*

Proof. By a straightforward structural induction on e. □

Lemma A.8. *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. □

Corollary A.9. *Let e be a stage-0 expression with no free variables. Then $[\![e]\!]_{\{\}} = [\![e]\!]_{R_0}$ for any R_0.*

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_1 be a stage-n and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. Then*

$$[\![e_1]\!]_{R_0,R_1,\ldots,R_n}[z\backslash[\![e_2]\!]_{\{\}}] = [\![e_1[x\backslash e_2]^n]\!]_{R_0,R_1,\ldots,R_n}$$

where $R_0(x) = z$.

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

- Case $e_1 = y$, $n > 0$: Because of our assumption on the renaming environments, $R_n(y) \neq z$. Hence, $[\![y]\!]_{R_0 \text{ with } \{x=z\},R_1,\ldots,R_n}[z\backslash[\![e_2]\!]_{\{\}}] = R_n(y)[z\backslash[\![e_2]\!]_{\{\}}] = R_n(y)$. And we have $[\![y[x\backslash e_2]^n]\!]_{R_0,R_1,\ldots,R_n} = [\![y]\!]_{R_0,R_1,\ldots,R_n} = R_n(y)$.
- Case $e_1 = x$, $n = 0$: We have $[\![x]\!]_{R_0 \text{ with } \{x=z\}}[z\backslash[\![e_2]\!]_{\{\}}] = z[z\backslash[\![e_2]\!]_{\{\}}] = [\![e_2]\!]_{\{\}}$. We also have $[\![x[x\backslash e_2]^0]\!]_{R_0} = [\![e_2]\!]_{R_0}$. By Lemma A.8, $[\![e_2]\!]_{\{\}} = [\![e_2]\!]_{R_0}$.
- Case $e_1 = y$, $n = 0$, and $y \neq x$: Trivial. □

A.3 Relation Between Staged Programming and Record Calculus

Lemma A.12. *$R{\cdot}x \longrightarrow^*_\beta R(x)$ for any R.*

Proof. By structural induction on R.

- Case $R = \{\}$: We have $\{\}{\cdot}x \longrightarrow_\beta \textbf{error}$. Also $\{\}(x) = \textbf{error}$ by definition.
- Case $R = r$: Trivial.
- Case $R = R' \text{ with } \{y = z\}$: If $x = y$, then $R' \text{ with } \{x = z\}{\cdot}x \longrightarrow_\beta z$ and $(R' \text{ with } \{x = z\})(x) = z$.

  If $x \neq y$, then $R' \text{ with } \{y = z\}{\cdot}x \longrightarrow_\beta R'{\cdot}x$ and $(R' \text{ with } \{y = z\})(x) = R'(x)$. By I.H. we have $R'{\cdot}x \longrightarrow^*_\beta R'(x)$. □

Lemma A.13. *$(R(x))[r\backslash R'] \longrightarrow^*_\beta (R[r\backslash R'])(x)$ for any R, R'.*

Proof. By structural induction on R.

- Case $R = \{\}$: We have $\{\}(x)[r\backslash R'] = \textbf{error}$ and $(\{\}[r\backslash R'])(x) = \textbf{error}$ by definition.
- Case $R = r'$: If $r = r'$, then $(r(x))[r\backslash R'] = R'{\cdot}x$ and $(r[r\backslash R'])(x) = R'(x)$. By Lemma A.12, $R'{\cdot}x \longrightarrow^*_\beta R'(x)$.

  If $r \neq r'$, then $(r'(x))[r\backslash R'] = r'{\cdot}x$ and $(r'[r\backslash R'])(x) = r'{\cdot}x$.
- Case $R = R_1 \text{ with } \{y = z\}$: If $x = y$, then $((R_1 \text{ with } \{x = z\})(x))[r\backslash R'] = z$ and $((R_1 \text{ with } \{x = z\})[r\backslash R'])(x) = z$.

  If $x \neq y$, then $((R_1 \text{ with } \{y = z\})(x))[r\backslash R'] = (R_1(x))[r\backslash R']$ and $((R_1 \text{ with } \{y = z\})[r\backslash R'])(x) = (R_1[r\backslash R'])(x)$. By I.H. we have $(R_1(x))[r\backslash R'] \longrightarrow^*_\beta (R_1[r\backslash R'])(x)$. □

Lemma A.14. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$[\![e]\!]_{R_0,\ldots,R_n}[r_m \backslash R_m] \longrightarrow^*_\beta [\![e]\!]_{R_0[r_m\backslash R_m],\ldots,R_n[r_m\backslash R_m]}$$

Proof. By structural induction on e. In the VAR case we use Lemma A.13. In the BOX case we use the fact that the newly introduced environment variable R_{n+1} is fresh. Other cases easily follow from the I.H. □

Proof of Theorem 7.1. By induction on the structure of e_1, based on the last applied reduction rule. The proof mostly follows from the I.H. We only show interesting cases here.

- Case APP(3). We have

$$\frac{e' \in Val^0}{(\lambda x.e)e' \longrightarrow_0 e[x\backslash e']^0}$$

  So,

$$\begin{aligned}
&[\![(\lambda x.e)e']\!]_{\{\}} = (\lambda z.[\![e]\!]_{\{x=z\}})[\![e']\!]_{\{\}} && \text{where } z \text{ is fresh}\\
&\longrightarrow_\beta [\![e]\!]_{\{x=z\}}[z\backslash [\![e']\!]_{\{\}}]\\
&= [\![e[x\backslash e']^0]\!]_{\{\}} && \text{by Lemma A.11}
\end{aligned}$$

- Case UBOX(2). We have

$$\frac{e \in Val^1}{\backprime(\langle e\rangle) \longrightarrow_1 e}$$

  Note that

$$\begin{aligned}
&[\![\backprime(\langle e\rangle)]\!]_{\{\},R_1} = [\![\langle e\rangle]\!]_{\{\}}R_1\\
&= (\lambda r_1.[\![e]\!]_{\{\},r_1})R_1 && \text{where } r_1 \text{ is fresh}\\
&\longrightarrow_\beta ([\![e]\!]_{\{\},r_1})[r_1\backslash R_1]\\
&\longrightarrow^*_\beta [\![e]\!]_{\{\},R_1} && \text{by Lemma A.14}
\end{aligned}$$

- Case RUN(2). We have

$$\frac{e \in Val^1}{\mathsf{run}(\langle e\rangle) \longrightarrow_0 e}$$

  So,

$$\begin{aligned}
&[\![\mathsf{run}(\langle e\rangle)]\!]_{\{\}} = ([\![\langle e\rangle]\!]_{\{\}})\{\}\\
&= (\lambda r_1.[\![e]\!]_{\{\},r_1})\{\} && \text{where } r_1 \text{ is fresh}\\
&\longrightarrow_\beta [\![e]\!]_{\{\},r_1}[r_1\backslash\{\}]\\
&\longrightarrow^*_\beta [\![e]\!]_{\{\},\{\}} && \text{by Lemma A.14}\\
&= [\![e]\!]_{\{\}} && \text{by Lemma A.10}
\end{aligned}$$

- Case LIFT(2). We have

$$\frac{e \in Val^0}{\mathsf{lift}(e) \longrightarrow_0 \langle e \rangle}$$

So,

$$[\![\mathsf{lift}(e)]\!]_{\{\}} = (\lambda r_1.[\![e]\!]_{\{\}}) \qquad \text{where } r_1 \text{ is fresh}$$

Since $FV^0(e) = \varnothing$, $[\![e]\!]_{\{\}} = [\![e]\!]_{r_1}$ by Lemma A.8. Hence;

$$\begin{aligned} &= (\lambda r_1.[\![e]\!]_{r_1}) \\ &= (\lambda r_1.[\![e]\!]_{\{\},r_1}) \qquad \text{by Lemma A.10} \\ &= [\![\langle e \rangle]\!]_{\{\}} \end{aligned}$$

□

Proof of Lemma 7.4. By structural induction on e. The proof mostly follows from the I.H. In UBOX, APP and RUN cases we use Lemma 7.3 and do reverse reasoning from types to expressions. We show the APP and UBOX cases.

- Case $e = e_1 e_2$ at stage n. We have, using Lemma 7.3,

$$\frac{\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A \to B \quad \Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A}{\Delta \vdash_R [\![e_1 e_2]\!]_{\{\},R_1,\ldots,R_n} : B}$$

By I.H we have two subcases:

1. $\exists e_1'$ such that $e_1 \longrightarrow_n e_1'$. In this case, $e_1 e_2 \longrightarrow_n e_1' e_2$.
2. $e_1 \in Val^n$. By I.H. we have two subcases:
   (a) $\exists e_2'$ such that $e_2 \longrightarrow_n e_2'$. In this case, $e_1 e_2 \longrightarrow_n e_1 e_2'$.
   (b) $e_2 \in Val^n$. We again have two subcases:
      i. $n > 0$: In this case $e_1 e_2 \in Val^n$.
      ii. $n = 0$: Because $[\![e_1]\!]_{\{\}}$ has the function type $A \to B$ and e_1 is a value at stage-0, e_1 must be either $\lambda x.e_3$ or $\mathsf{fix}\, f(x).\, e_3$, for some e_3. Therefore we have either $(\lambda x.e_3)e_2 \longrightarrow_0 e_3[x \backslash e_2]^0$ or $(\mathsf{fix}\, f(x).\, e_3)e_2 \longrightarrow_0 e_3[f \backslash \mathsf{fix}\, f(x).\, e_3]^0[x \backslash e_2]^0$.

- Case $e = \text{`}(e_1)$ at stage $n+1$. Note that $[\![\text{`}(e_1)]\!]_{\{\},R_1,\ldots,R_{n+1}} = ([\![e_1]\!]_{\{\},R_1,\ldots,R_n})R_{n+1}$. We have

$$\Delta \vdash_R ([\![e_1]\!]_{\{\},R_1,\ldots,R_n})R_{n+1} : A$$

Because the record expression R_{n+1} can only be given record types, we must have

$$\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : \Gamma \to A$$

for some Γ. By I.H. we have two subcases:

1. $\exists e_1'$ such that $e_1 \longrightarrow_n e_1'$. In this case, $\text{`}(e_1) \longrightarrow_{n+1} \text{`}(e_1')$.

2. $e_1 \in Val^n$. We have two subcases:
   (a) $n > 0$: In this case, $\backprime(e_1) \in Val^{n+1}$.
   (b) $n = 0$: Recall that $e_1 \in Val^0$ and it types to $\Gamma \to A$. The only stage-0 value whose translation can have such a type is $\langle e' \rangle$ for some $e' \in Val^1$. Hence, $\backprime(e_1) = \backprime(\langle e' \rangle)$, and by ESUBOX, we have $\backprime(\langle e' \rangle) \longrightarrow_1 e'$ □

Lemma A.15. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$\Delta :: \{r_n : [\![\Delta_n]\!]\} \vdash_R [\![e]\!]_{R_0,\ldots,R_{n-1},R_n} : A$$

if and only if

$$\Delta :: \{r_n : [\![\Delta_n \twoheadleftarrow \{x : \sigma\}]\!]\} \vdash_R [\![e]\!]_{R_0,\ldots,R_{n-1},R_n} : A$$

where $x \in dom(R_n)$.

Proof. This is the "weakening" lemma adapted to translation and records. Assume $R_n(x) = z$. Then, because of the translation, any occurrence of x at level n will be replaced with z and its type is grabbed from Δ — it is independent from r_n's type. Any variable $y \notin dom(R_n)$ will be translated to $r_n \cdot y$, and can still be given the same type because $[\![\Delta_n]\!](y) = [\![\Delta_n \twoheadleftarrow \{x : \sigma\}]\!](y)$. □

Proof of Lemma 7.6. By structural induction on e.

- Case $e = c$.

  Trivial.

- Case $e = x$, ($\Longrightarrow$). We have $\Delta_0, \ldots, \Delta_n \vdash_S x : A$ with the premise $A \prec \Delta_n(x)$. Note that $[\![x]\!]_{R_0,\ldots,R_n} = R_n(x)$. We have two subcases.

  (i) Case $R_n(x) = z$ for some z: By the definition of type translation,

  $$([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z) = [\![\Delta_n(x)]\!]$$

  Using the fact that $[\![A]\!] \prec [\![\Delta_n(x)]\!]$, we get

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R z : [\![A]\!]$$

  (ii) Case $R_n(x) = r_n \cdot x$: By the definition of type translation, $([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(r_n) = [\![\Delta_n]\!]$. Using the fact that $A \prec \Delta_n(x)$, it is easy to construct a Γ such that $\Gamma(x) = [\![A]\!]$ and $\Gamma \prec [\![\Delta_n]\!]$. Hence, $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n : \Gamma$, which gives

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n \cdot x : [\![A]\!]$$

- Case $e = x$, ($\Longleftarrow$). Note that $[\![x]\!]_{R_0,\ldots,R_n} = R_n(x)$. We have two subcases.

  (i) Case $R_n(x) = z$ for some z: We have

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R z : [\![A]\!]$$

  with the premise $[\![A]\!] \prec ([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z)$. By the definition of type translation, $([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z) = [\![\Delta_n(x)]\!]$, hence $[\![A]\!] \prec [\![\Delta_n(x)]\!]$. Therefore, $A \prec \Delta_n(x)$, which gives $\Delta_0, \ldots, \Delta_n \vdash_S x : A$.

(ii) Case $R_n(x) = r_n \cdot x$: We have

$$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n \cdot x : [\![A]\!]$$

with the premises $\Gamma \prec [\![\Delta_n]\!]$ and $\Gamma(x) = [\![A]\!]$. These two premises imply $A \prec \Delta_n(x)$, which gives $\Delta_0, \ldots, \Delta_n \vdash_S x : A$.

- Case $e = \lambda x.e'$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda z.\, [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}}$ where z is fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e' : A \to B$ with the premise
  2. $\quad \Delta_0, \ldots, \Delta_n \leftarrowtail \{x : A\} \vdash_S e' : B$.
  3. $[\![\Delta_0, \ldots, \Delta_n \leftarrowtail \{x : A\}]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![B]\!]$ by I.H. and (2).
  4. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \leftarrowtail \{z : [\![A]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![B]\!]$ by (3) and Lemma A.15.
  5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![A]\!] \to [\![B]\!]$ by (4) and TRABS.

- Case $e = \lambda x.e'$, ($\Longleftarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda z.\, [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}}$ where z is fresh.

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![C]\!]$
  2. By TRABS, we have $[\![C]\!] = A' \to B'$ for some A', B'. By Lemma 6.2, there exist A, B such that $[\![A]\!] = A'$ and $[\![B]\!] = B'$. Therefore $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![A]\!] \to [\![B]\!]$
  3. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \leftarrowtail \{z : [\![A]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![B]\!]$ as a premise of by (2).
  4. $[\![\Delta_0, \ldots, \Delta_n \leftarrowtail \{x : A\}]\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \text{ with } \{x=z\}} : [\![B]\!]$ by (3) and Lemma A.15.
  5. $\Delta_0, \ldots, \Delta_n \leftarrowtail \{x : A\} \vdash_S e' : B$ by I.H. and (4).
  6. $\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e' : A \to B$ (5) and TSABS.

- Case $e = \lambda^* x.e'$ is very similar to the abstraction case.

- Case $e = \mathsf{fix}\, f(x).\, e'$ is very similar to the abstraction case.

- Case $e = e_1 e_2$, ($\Longrightarrow$). Easily follows from the I.H.

- Case $e = e_1 e_2$, ($\Longleftarrow$).

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1 e_2]\!]_{R_0,\ldots,R_n} : [\![A]\!]$ with the premises
  2. $\quad [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : T \to [\![A]\!]$and
  3. $\quad [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_2]\!]_{R_0,\ldots,R_n} : T$ for some T
  4. $T = B' \in \mathit{RLegType}$ by Lemma 7.3
  5. $B' = [\![B]\!]$ for some B by Lemma 6.2
  6. $\Delta_0, \ldots, \Delta_n \vdash_S e_2 : B$ by (3), (5), and I.H.
  7. $\Delta_0, \ldots, \Delta_n \vdash_S e_1 : B \to A$ by (2), (5), and I.H.
  8. $\Delta_0, \ldots, \Delta_n \vdash_S e_1 e_2 : A$ by (6), (7), and TSAPP.

- Case $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ uses the same principles in the abstraction and application cases together with the fact that type translation does not alter bound/unbound type variables.

- Case $e = \langle e' \rangle$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda r_{n+1}.\, [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}}$ where r_{n+1} is fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_S \langle e' \rangle : \Box(\Gamma \triangleright A)$ with the premise
  2. $\quad \Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e' : A$.
  3. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$ by I.H. and (2).
  4. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} = [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow \{r_{n+1} : [\![\Gamma]\!]\}$ by the definition of type translation
  5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow \{r_{n+1} : [\![\Gamma]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$ by (3) and (4).
  6. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r_{n+1}.[\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![\Gamma]\!] \to [\![A]\!]$ by (5) and TRABS.

- Case $e = \langle e' \rangle$, ($\Longleftarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda r_{n+1}.\, [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}}$ where r_{n+1} is fresh.

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r_{n+1}.[\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$
  2. $[\![A]\!] = \Gamma' \to B'$ for some Γ' and B'. by TRABS
  3. $[\![\Gamma]\!] = \Gamma'$ and $[\![B]\!] = B'$ for some Γ and B. by Lemma 6.2
  4. (1) has the premise $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow \{r_{n+1} : [\![\Gamma]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![B]\!]$ by TRABS
  5. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} = [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow \{r_{n+1} : [\![\Gamma]\!]\}$ by the definition of type translation
  6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![B]\!]$ by (4) and (5).
  7. $\Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e' : B$ by I.H. and (6).
  8. $\Delta_0, \ldots, \Delta_n \vdash_S \langle e' \rangle : \Box(\Gamma \triangleright B)$ by (7) and TSBOX.

- Case $e = \backprime(e_1)$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})R_{n+1}$.

  1. We have $\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_S \backprime(e_1) : A$ with the premises
  2. $\quad \Delta_0, \ldots, \Delta_n \vdash_S e_1 : \Box(\Gamma \triangleright A)$ and
  3. $\quad \Gamma \prec \Delta_{n+1}$.
  4. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : [\![\Gamma]\!] \to [\![A]\!]$ by I.H. and (2).
  5. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : [\![\Gamma]\!] \to [\![A]\!]$ by (4) and Lemma A.1.
  6. Without loss of generality, assume $R_{n+1} = r_{n+1}\,\mathsf{with}\,\{x = z\}$. We have
  7. $\quad ([\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}})(r_{n+1}) = [\![\Delta_{n+1}]\!]$
  8. From (3) we have $[\![\Gamma]\!] \prec [\![\Delta_{n+1}]\!]$.
  9. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R r_{n+1} : [\![\Gamma]\!]$ by (7), (8), and TRVAR.
  10. $([\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}})(z) = [\![\Delta_{n+1}(x)]\!] = [\![\Delta_{n+1}]\!](x)$ by definition
  11. From (8) we have $[\![\Gamma]\!](x) \prec [\![\Delta_{n+1}]\!](x)$.
  12. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R z : [\![\Gamma]\!](x)$ by (10), (11), and TRVAR.
  13. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma]\!] \Leftarrow \{x : [\![\Gamma]\!](x)\}$ by (9), (12), and TRUPD.

14. $[\![\Gamma]\!] \Leftarrow\!\!+ \{x : [\![\Gamma]\!](x)\} = [\![\Gamma]\!]$
15. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma]\!]$ by (13) and (14).
16. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R ([\![e_1]\!]_{R_0,\ldots,R_n}) R_{n+1} : [\![A]\!]$ by (5), (15) and TRAPP.

- Case $e = \backprime(e_1)$, ($\Longleftarrow$). This case applies the ($\Longrightarrow$) case in the backwards direction with the additional use of Lemma 6.2 and the fact that $\Gamma \prec \Delta$ and $A \prec \Delta(x)$ imply $\Gamma \Leftarrow\!\!+ \{x : A\} \prec \Delta$.

- Case $e = \mathsf{run}(e')$.

  Follows easily from the I.H.

- Case $e = \mathsf{lift}(e')$.

  Follows easily from the I.H. and Lemma A.1. □

A.4 Extension with Pluggable Declarations

Proof of Theorem 9.1. By structural induction on e_1, based on the last applied reduction. We only show the cases for the new syntax.

- Case $e_1 = \langle\rangle$. Not possible because $\langle\rangle \in Val^n$.

- Case $e_1 = \langle x = e \rangle$. We have

$$\frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

  and

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \Leftarrow\!\!+ \{x : A\})}$$

  By I.H., we have $\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e' : A$, which gives, by TSDEC, that

$$\varnothing, \Delta_1 \ldots, \Delta_n \vdash_P \langle x = e'\rangle : \Diamond(\Gamma \triangleright \Gamma \Leftarrow\!\!+ \{x : A\})$$

- Case $e_1 = \mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4$. The two cases when we have

$$\frac{e_3 \longrightarrow_n e_3'}{\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_3')\ \mathsf{in}\ e_4}$$

  and

$$\frac{e_3 \in Val^n \quad e_4 \longrightarrow_{n+1} e_4'}{\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4'}$$

  easily follow from the I.H.

- Case $e_1 = \mathsf{let}\ \backprime(\langle x = e_3\rangle)\ \mathsf{in}\ e_4$ and

$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ \backprime(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

  We have

$$\frac{\varnothing \vdash_P \langle x = e_3\rangle : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_1 \quad \varnothing, \Gamma' \vdash_P e_4 : A}{\varnothing, \Delta_1 \vdash_P \mathsf{let}\ \backprime(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 : A}$$

By TSDEC, for some B, we must have $\varnothing, \Gamma \vdash_P e_3 : B$ and $\Gamma' = \Gamma \twoheadleftarrow \{x : B\}$.

By the Generalization Lemma (from [KYC06]) and the premise $\Gamma \prec \Delta_1$, we have $\varnothing, \Delta_1 \vdash_P e_3 : B$.

Again by the Generalization Lemma and the fact that $\Gamma \twoheadleftarrow \{x : B\} \prec \Delta_1 \twoheadleftarrow \{x : \mathrm{GEN}_B(\varnothing, \Delta_1)\}$, we have $\varnothing, \Delta_1 \twoheadleftarrow \{x : \mathrm{GEN}_B(\varnothing, \Delta_1)\} \vdash_P e_4 : A$.

Finally, by TSLET, we obtain $\varnothing, \Delta_1 \vdash_P \mathsf{let}\, x = e_3 \,\mathsf{in}\, e_4 : A$.

- Case $e_1 = \mathsf{let}\ `(\langle\rangle)\,\mathsf{in}\, e$ is straightforward. □

Proof of Theorem 9.2. By structural induction on e_1. We only show the cases for the new syntax.

- Case $e_1 = \langle\rangle$. $\langle\rangle \in Val^n$.
- Case $e_1 = \langle x = e\rangle$. We have

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \twoheadleftarrow \{x : A\})}$$

  By I.H. we either have $e \in Val^{n+1}$, which means $\langle x = e\rangle \in Val^n$, or we have e' such that $e \longrightarrow_{n+1} e'$, which means $\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle$.

- Case $e_1 = \mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4$. We have

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_3 : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_{n+1} \quad \varnothing, \Delta_1, \ldots, \Delta_n, \Gamma' \vdash_P e_4 : A}{\varnothing, \Delta_1, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4 : A}$$

  By I.H. we either have $e_3 \longrightarrow_n e_3'$, which means $\mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4 \longrightarrow_{n+1} \mathsf{let}\ `(e_3')\,\mathsf{in}\, e_4$. Or we have $e_3 \in Val^n$. In this case, by I.H. we have two subcases:

  - $e_4 \longrightarrow_{n+1} e_4'$, which means $\mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4 \longrightarrow_{n+1} \mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4'$.
  - $e_4 \in Val^{n+1}$. We again have two subcases. If $n > 0$, we have $\mathsf{let}\ `(e_3)\,\mathsf{in}\, e_4 \in Val^{n+1}$. If $n = 0$, we first recall that $e_3 \in Val^0$ and that e_3 types to $\Diamond(\Gamma \triangleright \Gamma')$. The only stage-0 value that can be given such a type is either $\langle x = e_5\rangle$ for some $e_5 \in Val^1$, which by ESLET2 gives $\mathsf{let}\ `(\langle x = e_5\rangle)\,\mathsf{in}\, e_4 \longrightarrow_1 \mathsf{let}\, x = e_5 \,\mathsf{in}\, e_4$; or $\langle\rangle$, which again by ESLET2 gives $\mathsf{let}\ `(\langle\rangle)\,\mathsf{in}\, e_4 \longrightarrow_1 e_4$. □

Lemma A.16. *Let e be a λ^{decl}_{poly} expression such that $e \in Val^{n+1}$. Then $\delta(e)$ is a λ^{gen}_{poly} expression such that $\delta(e) \in Val^{n+1}$.*

Proof. By a straightforward structural induction on e. □

Proof of Theorem 9.3. By structural induction on e_1, based on the last applied reduction. The proof mostly follows from the I.H. We show the most interesting cases here.

- We have

$$\frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

  By I.H., $\delta(e) \longrightarrow^*_{n+1} \delta(e')$. Hence,

$$\begin{aligned}\delta(\langle x = e\rangle) &= (\lambda v.\lambda y.\langle \mathsf{let}\, x = `(v)\,\mathsf{in}\, `(y)\rangle)\ \langle\delta(e)\rangle \\ &\longrightarrow^*_n (\lambda v.\lambda y.\langle \mathsf{let}\, x = `(v)\,\mathsf{in}\, `(y)\rangle)\ \langle\delta(e')\rangle \\ &= \delta(\langle x = e'\rangle)\end{aligned}$$

- We have

$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ \grave{}(\langle x = e_3 \rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

Note that

$$\delta(\mathsf{let}\ \grave{}(\langle x = e_3 \rangle)\ \mathsf{in}\ e_4) \quad = \quad \grave{}(((\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y) \rangle)\ \langle \delta(e_3) \rangle)\ \langle \delta(e_4) \rangle)$$

By Lemma A.16, $\delta(e_3), \delta(e_4) \in Val^1$. Hence,

$$\begin{aligned}
&\grave{}(((\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y) \rangle)\ \langle \delta(e_3) \rangle)\ \langle \delta(e_4) \rangle) \\
&\longrightarrow_1 \grave{}((\lambda y.\langle \mathsf{let}\ x = \grave{}(\langle \delta(e_3) \rangle)\ \mathsf{in}\ \grave{}(y) \rangle)\ \langle \delta(e_4) \rangle) \\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \grave{}(\langle \delta(e_3) \rangle)\ \mathsf{in}\ \grave{}(\langle \delta(e_4) \rangle) \rangle) \\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \grave{}(\langle \delta(e_4) \rangle) \rangle) \\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \delta(e_4) \rangle) \\
&\longrightarrow_1 \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \delta(e_4) \\
&= \delta(\mathsf{let}\ x = e_3\ \mathsf{in}\ e_4)
\end{aligned}$$

□

Proof of Theorem 9.4. By structural induction on e. The most interesting cases are below.

- We have

$$\frac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e_1 : A}{\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e_1 \rangle : \Diamond(\Gamma \triangleright \Gamma \twoheadleftarrow \{x : A\})}$$

Note that

$$\begin{aligned}
\delta(\langle x = e_1 \rangle) \quad &= \quad (\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y) \rangle)\ \langle \delta(e_1) \rangle \\
\delta(\Diamond(\Gamma \triangleright \Gamma \twoheadleftarrow \{x : A\})) \quad &= \quad \Box(\delta(\Gamma \twoheadleftarrow \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B) \text{ for any } B
\end{aligned}$$

We now proceed as follows:

1. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Gamma) \vdash_S \delta(e_1) : \delta(A)$ by I.H.
2. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \langle \delta(e_1) \rangle : \Box(\delta(\Gamma) \triangleright \delta(A))$ by (1) and TSBOX
3. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S (\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y) \rangle) : \Box(\delta(\Gamma) \triangleright \delta(A)) \to \Box(\delta(\Gamma \twoheadleftarrow \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B)$ by a series of typing rules. Note that this judgment can be derived for any B.
4. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S (\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y) \rangle)\ \langle \delta(e_1) \rangle : \Box(\delta(\Gamma \twoheadleftarrow \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B)$ by (2), (3), and TSAPP

- We have

$$\frac{\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_{n+1} \quad \Delta_0, \ldots, \Delta_n, \Gamma' \vdash_P e_2 : A}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 : A}$$

Note that

$$\delta(\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2) \quad = \quad \grave{}(\delta(e_1)\ \langle \delta(e_2) \rangle)$$

We now proceed as follows:

1. $\delta(\Diamond(\Gamma \triangleright \Gamma')) = \Box(\delta(\Gamma') \triangleright \delta(A)) \to \Box(\delta(\Gamma) \triangleright \delta(A))$ by the definition of $\delta(\cdot)$
2. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e_1) : \Box(\delta(\Gamma') \triangleright \delta(A)) \to \Box(\delta(\Gamma) \triangleright \delta(A))$ by (1) and I.H.
3. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Gamma') \vdash_S \delta(e_2) : \delta(A)$ by I.H.
4. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \langle \delta(e_2) \rangle : \Box(\delta(\Gamma') \triangleright \delta(A))$ by (3) and TSBOX
5. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e_1)\ \langle \delta(e_2) \rangle : \Box(\delta(\Gamma) \triangleright \delta(A))$ by (2), (4) and TSAPP
6. $\delta(\Gamma) \prec \delta(\Delta_{n+1})$ by the premise of the assumption
7. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Delta_{n+1}) \vdash_S \ `(\delta(e_1)\ \langle \delta(e_2) \rangle) : \delta(A)$ by (5), (6) and TSUBOX

□

Note that the extension with pluggable declarations to the translation preserves the Lemmata A.7 and 7.3.

Proof of Theorem 9.5. By induction on the structure of e_1, based on the last applied reduction rule. This proof is an extension of Theorem 7.1 with the pluggable declaration syntax. The cases mostly follow from the I.H. We only show the most interesting case. Note that the extension with pluggable declarations preserves Lemmata A.8, A.10, A.11, and A.14, which are used in the proof of Theorem 7.1 (and here in this proof, too).

- Case LET2(3). We have

$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ `(\langle x = e_3 \rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

Note that

$$\begin{aligned}
&[\![\mathsf{let}\ `(\langle x = e_3 \rangle)\ \mathsf{in}\ e_4]\!]_{\{\},R_1} \\
&= ([\![\langle x = e_3 \rangle]\!]_{\{\}})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})\,R_1 \\
&= ([\![\langle x = e_3 \rangle]\!]_{\{\}})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})\,R_1 \\
&= (\lambda\kappa.\lambda y.\lambda r.\,\mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}))\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})R_1 \\
&\longrightarrow_\beta (\lambda y.\lambda r.\,\mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}))(\lambda r.[\![e_4]\!]_{\{\},r})R_1 \\
&\longrightarrow_\beta (\lambda r.\,\mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ (\lambda r.[\![e_4]\!]_{\{\},r})(r\ \mathsf{with}\ \{x = z\}))R_1 \\
&\longrightarrow_\beta (\mathsf{let}\ z = [\![e_3]\!]_{\{\},r}[r \backslash R_1]\ \mathsf{in}\ (\lambda r.[\![e_4]\!]_{\{\},r})(R_1\ \mathsf{with}\ \{x = z\})) \\
&\longrightarrow_\beta \mathsf{let}\ z = [\![e_3]\!]_{\{\},r}[r \backslash R_1]\ \mathsf{in}\ [\![e_4]\!]_{\{\},r}[r \backslash R_1\ \mathsf{with}\ \{x = z\}] \\
&\longrightarrow_\beta^* \mathsf{let}\ z = [\![e_3]\!]_{\{\},R_1}\ \mathsf{in}\ [\![e_4]\!]_{\{\},R_1\ \mathsf{with}\ \{x=z\}} \qquad \text{by Lemma A.14} \\
&= [\![\mathsf{let}\ x = e_3\ \mathsf{in}\ e_4]\!]_{\{\},R_1}
\end{aligned}$$

□

Proof of Theorem 9.7. By structural induction on e_1. The proof is the same as Theorem 7.4, except it is extended for the new syntax for pluggable declarations. The proof for the new cases mostly follow easily from the I.H. We only show the most interesting case here.

Let e_1 be the stage-$n+1$ expression $\mathsf{let}\ `(e_3)\ \mathsf{in}\ e_4$. We have

$$\Delta \vdash_R [\![\mathsf{let}\ `(e_3)\ \mathsf{in}\ e_4]\!]_{\{\},R_1,\ldots,R_{n+1}} : A$$

Note that

$$[\![\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4]\!]_{\{\},R_1,\ldots,R_{n+1}} = ([\![e_3]\!]_{\{\},R_1,\ldots,R_n})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},R_1,\ldots,R_n,r})\,R_{n+1}$$

As the (sub)premises, we must have

$$\Delta \vdash_R R_{n+1} : \Gamma$$
$$\Delta \vdash_R (\lambda r.[\![e_4]\!]_{\{\},R_1,\ldots,R_n,r}) : \Gamma' \to B$$
$$\Delta \vdash_R [\![e_3]\!]_{\{\},R_1,\ldots,R_n} : \kappa \to (\Gamma' \to B) \to \Gamma \to A$$

for some Γ, Γ', and B. As the premise of the second judgment above, we also must have

$$\Delta \lessdot \{r : \Gamma'\} \vdash_R [\![e_4]\!]_{\{\},R_1,\ldots,R_n,r} : B$$

By I.H. we have two subcases:

- $\exists e_3'$ such that $e_3 \longrightarrow_n e_3'$. In this case, $\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_3')\ \mathsf{in}\ e_4$.
- $e_3 \in Val^n$. In this case, by I.H., we have two subcases:
  - $\exists e_4'$ such that $e_4 \longrightarrow_{n+1} e_4'$. In this case, $\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4'$.
  - $e_4 \in Val^n$. We again have two subcases:
    (i) $n > 0$: In this case, $\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 \in Val^{n+1}$.
    (ii) $n = 0$: Recall that $e_3 \in Val^0$ and its translation types to $\kappa \to (\Gamma' \to B) \to \Gamma \to A$. The only stage-0 value whose translation can have such a type is $\langle x = e'' \rangle$ for some x and $e'' \in Val^1$. Hence, $\mathsf{let}\ \backprime(e_3)\ \mathsf{in}\ e_4 = \mathsf{let}\ \backprime(\langle x = e'' \rangle)\ \mathsf{in}\ e_4$, and by ESLET2, we have
    $$\mathsf{let}\ \backprime(\langle x = e'' \rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e''\ \mathsf{in}\ e_4$$ □

Proof of Theorem 9.9. By structural induction on e. The proof is the same as Theorem 7.6, except being extended for the new syntax for pluggable declarations. The proof for the new cases mostly follow easily from the I.H. We show the two interesting cases here. Note that the extension with pluggable declarations preserves Lemma A.15, which is used in the proof of Theorem 7.6 (and here in this proof, too).

- Case $e = \langle x = e' \rangle$ at stage n. Note that

  $$[\![\langle x = e' \rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\,\mathsf{let}\ z = [\![e']\!]_{R_0,\ldots,R_n,r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\})$$

  where r, y, z are fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e' \rangle : \Diamond(\Gamma \triangleright \Gamma \lessdot \{x : A\})$
  2. Note that $[\![\Diamond(\Gamma \triangleright \Gamma \lessdot \{x : A\})]\!] = \kappa \to ([\![\Gamma]\!] \lessdot \{x : [\![A]\!]\} \to B) \to ([\![\Gamma]\!] \to B)$ for any B.
  3. $\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e' : A$ as a premise of (1)
  4. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r} : [\![A]\!]$ by I.H. and (3)
  5. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \vdash_R r : [\![\Gamma]\!]$ by TRVAR
  6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \lessdot \{z : \mathrm{GEN}_{[\![A]\!]}(\ldots)\} \vdash_R (r\ \mathsf{with}\ \{x = z\}) : [\![\Gamma]\!] \lessdot \{x : [\![A]\!]\}$ by (4), (5), and TRUPD

7. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \leftarrow\!\!+ \{z : \mathrm{GEN}_{[\![A]\!]}(\ldots)\} \vdash_R$
$\lambda\kappa.\lambda y.\lambda r.\, \mathsf{let}\ z = [\![e']\!]_{R_0,\ldots,R_n,r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}) :$
$\kappa \to ([\![\Gamma]\!] \leftarrow\!\!+ \{x : [\![A]\!]\} \to B) \to ([\![\Gamma]\!] \to B)$
by (6), TRLET, and multiple applications of TRABS

- Case $e = \mathsf{let}\ `(e_1)\ \mathsf{in}\ e_2$ at stage $n+1$. Note that

$$[\![\mathsf{let}\ `(e_1)\ \mathsf{in}\ e_2]\!]_{R_0,\ldots,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\,(\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\,R_{n+1}$$

where r is fresh.

1. We have $\Delta_0, \ldots, \Delta_{n+1} \vdash_P \mathsf{let}\ `(e_1)\ \mathsf{in}\ e_2 : A$
2. $\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma_1 \triangleright \Gamma_2)$ for some Γ_1, Γ_2, as a premise of (1)
3. $\Gamma_1 \prec \Delta_{n+1}$ as a premise of (1)
4. $\Delta_0, \ldots, \Delta_n, \Gamma_2 \vdash_P e_2 : A$ as a premise of (1)
5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : \kappa \to ([\![\Gamma_2]\!] \to [\![A]\!]) \to [\![\Gamma_1]\!] \to [\![A]\!]$ by I.H. and (2)
6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma_2]\!]_{R_0,\ldots,R_n,r} \vdash_R [\![e_2]\!]_{R_0,\ldots,R_n,r} : [\![A]\!]$ by I.H. and (4)
7. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r} : [\![\Gamma_2]\!] \to [\![A]\!]$ by (6) and TRABS
8. $[\![\Gamma_1]\!] \prec [\![\Delta_{n+1}]\!]$ by (3) and Lemma A.6
9. $[\![\Delta_0, \ldots, \Delta_{n+1}]\!]_{R_0,\ldots,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma_1]\!]$
by (8) and multiple TRVAR (see Theorem 7.6, case $e = `(e_1)$, ($\Longrightarrow$), items (6) through (15) for a similar case)
10. $[\![\Delta_0, \ldots, \Delta_{n+1}]\!]_{R_0,\ldots,R_{n+1}} \vdash_R ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\,(\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\,R_{n+1} : [\![A]\!]$
by (5), (7), (9) and TRAPP

□

A.5 Extension with References

Lemma A.17. *If $\Sigma' \supseteq \Sigma$, then $\Sigma; \Delta \vdash_R e : A \implies \Sigma'; \Delta \vdash_R e : A$.*

Proof. This is a standard lemma. □

Lemma A.18. *Let $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$ for some stage-n expression e. Then,*

$$FV(e_0) \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$
$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$
$$\vdots$$
$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1})\{\vec{\pi_m}\}$$
$$FV(\vec{e_m}) \subseteq FV(R_{n-m})$$

Proof. By structural induction on e. We show the quotation and antiquotation cases. Other cases are straightforward from the I.H.

- Case $e = \langle e' \rangle$, stage n.

1. Suppose $[\![e']\!]_{R_0,\ldots,R_n,r} = (e'', \{(\vec{\pi_0}, \vec{e_0})\} :: \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
2. By I.H.
$$FV(e'') \subseteq \{r\} \cup \{\vec{\pi_0}\}$$
$$FV(\vec{e_0}) \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$
$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$
$$\vdots$$
$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1}) \cup \{\vec{\pi_m}\}$$
$$FV(\vec{e_m}) \subseteq FV(R_{n-m}\}$$
3. By definition, $[\![\langle e' \rangle]\!]_{R_0,\ldots,R_n} = ((\lambda\vec{\pi_0}.\lambda r.\, e'')\vec{e_0}, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
4. $FV((\lambda\vec{\pi_0}.\lambda r.\, e'')\vec{e_0}) \subseteq (\{r\} \cup \{\vec{\pi_0}\} \setminus \{\vec{\pi_0}, r\}) \cup FV(R_n) \cup \{\vec{\pi_1}\} = FV(R_n) \cup \{\vec{\pi_1}\}$
5. Properties for $FV(\vec{e_1}), \ldots, FV(\vec{e}_m)$ are immediate from the I.H.

- Case $e = \backprime(e')$, stage $n+1$.

1. Suppose $[\![e']\!]_{R_0,\ldots,R_n} = (e'', \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
2. By I.H.
$$FV(e'') \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$
$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$
$$FV(\vec{e_2}) \subseteq FV(R_{n-2}) \cup \{\vec{\pi_3}\}$$
$$\vdots$$
$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1}) \cup \{\vec{\pi_m}\}$$
$$FV(\vec{e_m}) \subseteq FV(R_{n-m}\}$$
3. By definition, with a fresh π,
$[\![\backprime(e')]\!]_{R_0,\ldots,R_n,R_{n+1}} = (\pi(R_{n+1}), \{(\pi, e'')\} :: \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
4. $FV(\pi(R_{n+1})) \subseteq FV(R_{n+1}) \cup \{\pi\}$
5. Properties for $FV(e''), FV(\vec{e_1}), \ldots, FV(\vec{e}_m)$ are immediate from the I.H. □

Lemma A.19. *Let e be a stage-n λ^{gen}_{poly} expression with $FV(e) = \{x_1, \ldots, x_m\}$. Then,*

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

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

Proof. This is an adaptation of Lemma A.8 for the improved translation and *Close*. □

Lemma A.20. *Let e be a λ^{gen}_{poly} expression such that $e \in Val^{n+1}$. Then*

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

Proof. This is an adaptation of Lemma A.10 for the improved translation and *Close*. □

Lemma A.21. *Let e_1 be a stage-n and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. Then*

$$Close([\![e_1]\!]_{R_0,R_1,\ldots,R_n})[z \backslash Close([\![e_2]\!]_{\{\}})] = Close([\![e_1[x \backslash e_2]^n]\!]_{R_0,R_1,\ldots,R_n})$$

where $R_0(x) = z$.

Proof. This is an adaptation of Lemma A.11 for the improved translation and *Close*. □

Lemma A.22. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$Close([\![e]\!]_{R_0,\ldots,R_n})[r_m \backslash R_m] \longrightarrow^*_{|\beta|} Close([\![e]\!]_{R_0[r_m \backslash R_m],\ldots,R_n[r_m \backslash R_m]})$$

Proof. This is an adaptation of Lemma A.14 for the improved translation and *Close*. By structural induction on e. □

Proof of Theorem 10.7. By induction on the structure of e_1, based on the last applied reduction. We only show interesting cases.

- Case ESABS: $\mathcal{S}, \lambda x.e \longrightarrow_{n+1} \mathcal{S}', \lambda x.e'$ with the premise $\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'$. Without loss of generality, assume

$$[\![e]\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$

$$[\![e']\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}} = (e'_0, [\{\pi'_1, e'_1\}, \ldots, \{\pi'_q, e'_q\}])$$

  So,

$$Close([\![e]\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}}) = (\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_p$$

$$Close([\![e']\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.e'_0)e'_1)\cdots)e'_q$$

  and therefore

$$Close([\![\lambda x.e]\!]_{\{\},R_1,\ldots,R_{n+1}}) = (\lambda\pi_p.\cdots((\lambda\pi_1.\lambda z.e_0)e_1)\cdots)e_p$$

$$Close([\![\lambda x.e']\!]_{\{\},R_1,\ldots,R_{n+1}}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.\lambda z.e'_0)e'_1)\cdots)e'_q$$

  By I.H. we have

$$[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}}) \longrightarrow_R [\![\mathcal{S}']\!], e'' \tag{1}$$

  such that $e'' \longrightarrow^*_{|\beta|} Close([\![e']\!]_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}})$.

  Recall that in staged semantics, evaluation occurs only at stage-0, or at stage-1 as a hole fill-in. Because of this, it must be that $p = n+1$ (otherwise $\lambda x.e$ would be a stage-$n+1$ value that cannot take a step of evaluation) and there are only two possibilities:

  1. A staged-0 reduction happens as part of e, meaning the premise of judgment (1) above is $[\![\mathcal{S}]\!], e_p \longrightarrow_R [\![\mathcal{S}']\!], \overline{e}_p$.
     This makes e'' equal to $(\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)\overline{e}_p$, giving

$$(\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)\overline{e}_p \longrightarrow^*_{|\beta|} ((\lambda\pi'_q.\cdots((\lambda\pi'_1.e'_0)e'_1)\cdots)e'_q)$$

     Let $C[\cdot]$ be the context $(\lambda\pi_p.\cdots((\lambda\pi_1.[\,])e_1)\cdots)\overline{e}_p$, and $C'[\cdot]$ be the context $(\lambda\pi'_q.\cdots((\lambda\pi'_1.[\,])e'_1)\cdots)e'_q$. Then the two terms above are, respectively, $C[e_0]$ and $C'[e'_0]$;

and $C[e_0] \longrightarrow^*_{|\beta|} C'[e'_0]$. The reductions included can be outside of e_0 in the context $C[\cdot]$, or directly inside e_0:

In the former case, the context would change and become, say, $C_1[\cdot]$, and some substitutions[4] may be performed on e_0. Let us represent the effect of these substitutions as S. The term we obtain is then $C_1[Se_0]$.

In the latter case, the context would have no change at all, but only e_0 would reduce to another term, say, $\overline{e}_0$. So the term we finally obtain is $C_1[S\overline{e}_0]$, giving $C_1[S\overline{e}_0] = C'[e'_0]$. So, $C_1[\cdot] = C'[\cdot]$ and $S\overline{e}_0 = e'_0$.

When it comes to $Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}})$, because of the premise, we have

$$\llbracket \mathcal{S} \rrbracket, Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) \longrightarrow_R \llbracket \mathcal{S}' \rrbracket, C[K[e_0]]$$

where $K[\cdot]$ is the context $\lambda z.[\,]$. Also,

$$Close(\llbracket \lambda x.e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) = C'[K[e'_0]]$$

Applying the same safe reductions for $C[\cdot]$, and e_0 above, we obtain $C[K[e_0]] \longrightarrow^*_{|\beta|} C_1[S(K[\overline{e_0}])]$. Note that the context $K[\cdot]$ binds the fresh variable z, but this variable does not exist free in $C[\cdot]$. Hence, the substitution S does not contain it, and we can safely say that $S(K[\overline{e_0}]) = K[S\overline{e_0}]$. Using the equalities above, we obtain $C_1[K[S\overline{e_0}]] = C'[K[e'_0]]$, which means that $C[K[e_0]] \longrightarrow^*_{|\beta|} Close(\llbracket \lambda x.e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}})$.

2. No stage-0 evaluation occurs, but a stage-1 hole gets filled in. This means e_p is a value and $\mathcal{S} = \mathcal{S}'$, because filling in a hole does not alter the store. So, by TRAPP, we have

$$\begin{array}{l} \llbracket \mathcal{S} \rrbracket, Close(\llbracket e \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}}) \\ \quad \longrightarrow_R \quad \llbracket \mathcal{S} \rrbracket, ((\lambda \pi_{p-1}.\cdots((\lambda \pi_1.e_0)e_1)\cdots)e_{p-1})[\pi_p \backslash e_p] \end{array}$$

and

$$\begin{array}{l} \llbracket \mathcal{S} \rrbracket, Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) \\ \quad \longrightarrow_R \quad \llbracket \mathcal{S} \rrbracket, ((\lambda \pi_{p-1}.\cdots((\lambda \pi_1.\lambda z.e_0)e_1)\cdots)e_{p-1})[\pi_p \backslash e_p] \end{array}$$

Note that by I.H.

$$((\lambda \pi_{p-1}.\cdots((\lambda \pi_1.e_0)e_1)\cdots)e_{p-1})[\pi_p \backslash e_p] \longrightarrow^*_{|\beta|} (\lambda \pi'_q.\cdots((\lambda \pi'_1.e'_0)e'_1)\cdots)e'_q$$

We now need to show that

$$((\lambda \pi_{p-1}.\cdots((\lambda \pi_1.\lambda z.e_0)e_1)\cdots)e_{p-1})[\pi_p \backslash e_p] \longrightarrow^*_{|\beta|} (\lambda \pi'_q.\cdots((\lambda \pi'_1.\lambda z.e'_0)e'_1)\cdots)e'_q$$

which can be done by reasoning about the contexts the same way we did above for the first case.

- Cases ESSYM, ESFIX, ESAPP(1), ESAPP(2), ESLET(1), ESLET(2), ESRUN(1), ESLIFT(1), ESREF(1), ESDEREF(1), ESASGN(1), and ESASGN(2) require using the I.H. the same way as in the ESABS case.

[4] This would be the case, for instance, of expanding a function application or a let-expression.

- Case ESAPP(3): $\mathcal{S}, (\lambda x.e_1)e_2 \longrightarrow_0 \mathcal{S}, e_1[x\backslash e_2]^0$ with the premise $e_2 \in Val^0$. Note that

$$Close([\![(\lambda x.e_1)e_2]\!]_{\{\}}) = (\lambda z.e_0)([\![e_0']\!])$$

where $[\![e_1]\!]_{\{x=z\}} = (e_0, \texttt{nil})$ and $[\![e_2]\!]_{\{\}} = (e_0', \texttt{nil})$. Because $e_2 \in Val^0$, we have $e_0' \in RVal$. Hence, $SEF(e_0')$. Then, at the record semantics side we have

$$[\![\mathcal{S}]\!], (\lambda z.e_0)(e_0') \longrightarrow_R [\![\mathcal{S}]\!], e_0[z\backslash e_0']$$

Note that $e_0[z\backslash e_0'] = Close([\![e_1]\!]_{\{x=z\}})[z\backslash Close([\![e_2]\!]_{\{\}})]$, which is equal to $Close([\![e_1[x\backslash e_2]^0]\!]_{\{x=z\}})$ by A.21, and $Close([\![e_1[x\backslash e_2]^0]\!]_{\{x=z\}}) = Close([\![e_1[x\backslash e_2]^0]\!]_{\{\}})$ by Lemma A.19.

- Case ESAPP(4): $\mathcal{S}, (\mathsf{fix}\, f(x).\, e_1)e_2 \longrightarrow_0 \mathcal{S}, e_1[f\backslash \mathsf{fix}\, f(x).\, e_2]^0[x\backslash e_2]^0$ with the premise $e_2 \in Val^0$. This is a case that is very similar to ESAPP(3) above.

- Case ESLET(3): $\mathcal{S}, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_0 \mathcal{S}, e_2[x\backslash e_1]^0$ with the premise $e_1 \in Val^0$. This is a case that is very similar to ESAPP(3).

- Case ESBOX: $\mathcal{S}, \langle e\rangle \longrightarrow_n \mathcal{S}', \langle e'\rangle$ with the premise $\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'$. Without loss of generality, assume

$$[\![e]\!]_{\{\},R_1,\ldots,R_n,r} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$
$$[\![e']\!]_{\{\},R_1,\ldots,R_n,r} = (e_0', [\{\pi_1', e_1'\}, \ldots, \{\pi_q', e_q'\}])$$

So,

$$Close([\![e]\!]_{\{\},R_1,\ldots,R_n,r}) = (\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_p$$
$$Close([\![e']\!]_{\{\},R_1,\ldots,R_n,r}) = (\lambda\pi_q'.\cdots((\lambda\pi_1'.e_0')e_1')\cdots)e_q'$$

Hence,

$$Close([\![\langle e\rangle]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_p.\cdots((\lambda\pi_1.\lambda r.e_0)e_1)\cdots)e_p$$
$$Close([\![\langle e'\rangle]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_q'.\cdots((\lambda\pi_1'.\lambda r.e_0')e_1')\cdots)e_q'$$

By I.H. we have

$$[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_n,r}) \longrightarrow_R [\![\mathcal{S}']\!], e''$$

such that $e'' \longrightarrow^*_{|\beta|} Close([\![e']\!]_{\{\},R_1,\ldots,R_n,r})$. And the rest of the proof for this case goes in the same style of the ESABS case using the I.H.

- Case ESUBOX(1): $\mathcal{S}, \backprime(e) \longrightarrow_{n+1} \mathcal{S}', \backprime(e')$ with the premise $\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'$. Without loss of generality, assume

$$[\![e]\!]_{\{\},R_1,\ldots,R_n} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$
$$[\![e']\!]_{\{\},R_1,\ldots,R_n} = (e_0', [\{\pi_1', e_1'\}, \ldots, \{\pi_q', e_q'\}])$$

So,

$$Close([\![e]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_p$$
$$Close([\![e']\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_q'.\cdots((\lambda\pi_1'.e_0')e_1')\cdots)e_q'$$

Hence,

$$Close([\![\backprime(e)]\!]_{\{\},R_1,\ldots,R_n,R_{n+1}}) = (\lambda\pi_p.\cdots((\lambda\pi_1.((\lambda\pi_0.\pi_0(R_{n+1}))e_0))e_1)\cdots)e_p$$

$$Close([\![\,\backprime(e')]\!]_{\{\},R_1,\ldots,R_n,R_{n+1}}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.((\lambda\pi'_0.\pi'_0(R_{n+1}))e'_0))e'_1)\cdots)e'_q$$

By I.H. we have

$$[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_n}) \longrightarrow_R [\![\mathcal{S}']\!], e''$$

such that $e'' \longrightarrow^*_{|\beta|} Close([\![e']\!]_{\{\},R_1,\ldots,R_n})$. And the rest of the proof for this case goes in the same style of the the ESABS case using the I.H.

- Case ESUBOX(2): $\mathcal{S}, \backprime(\langle e\rangle) \longrightarrow_1 \mathcal{S}, e$ with the premise $e \in Val^1$. Because of the premise, we have $[\![e]\!]_{\{\},r} = (e_0, \texttt{nil})$.
  Therefore, $[\![\langle e\rangle]\!]_{\{\}} = (\lambda r.e_0, \texttt{nil})$ and $Close([\![\backprime(\langle e\rangle)]\!]_{\{\},R_1}) = (\lambda\pi_0.\pi_0 R_1)(\lambda r.e_0)$. By ERAPP, we have

  $$[\![\mathcal{S}]\!], (\lambda\pi_0.\pi_0 R_1)(\lambda r.e_0) \longrightarrow_R [\![\mathcal{S}]\!], (\lambda r.e_0)R_1$$

  Note that $(\lambda r.e_0)R_1 \longrightarrow_{|\beta|} e_0[r\backslash R_1]$. Using the fact that $e_0 = Close([\![e]\!]_{\{\},r})$, we have $Close([\![e]\!]_{\{\},r})[r\backslash R_1] \longrightarrow^*_{|\beta|} Close([\![e]\!]_{\{\},R_1})$ by Lemma A.22.

- Case ESRUN(2): $\mathcal{S}, \mathsf{run}(\langle e\rangle) \longrightarrow_0 \mathcal{S}, e$ with the premise $e \in Val^1$. Because of the premise, we have $[\![e]\!]_{\{\},r} = (e_0, \texttt{nil})$.
  Therefore, $[\![\langle e\rangle]\!]_{\{\}} = (\lambda r.e_0, \texttt{nil})$ and $Close([\![\mathsf{run}(\langle e\rangle)]\!]_{\{\}}) = (\lambda r.e_0)\{\}$. By ERAPP, we have

  $$[\![\mathcal{S}]\!], (\lambda r.e_0)\{\} \longrightarrow_R [\![\mathcal{S}]\!], e_0[r\backslash\{\}]$$

  Using the fact that $e_0 = Close([\![e]\!]_{\{\},r})$, we have $Close([\![e]\!]_{\{\},r})[r\backslash\{\}] \longrightarrow^*_{|\beta|} Close([\![e]\!]_{\{\},\{\}})$ by Lemma A.22. And finally $Close([\![e]\!]_{\{\},\{\}}) = Close([\![e]\!]_{\{\}})$ by Lemma A.20.

- Case ESLIFT(2): $\mathcal{S}, \mathsf{lift}(e) \longrightarrow_0 \mathcal{S}, \langle e\rangle$ with the premise $e \in Val^0$. Because of the premise, we have $[\![e]\!]_{\{\}} = (e_0, \texttt{nil})$. Therefore, $Close([\![\mathsf{lift}(e)]\!]_{\{\}}) = \mathsf{let}\,\pi = e_0\,\mathsf{in}\,\lambda r.\pi$. Since $e \in Val^0$, we have $e_0 \in RVal$. By ERLET we have

  $$[\![\mathcal{S}]\!], (\mathsf{let}\,\pi = e_0\,\mathsf{in}\,\lambda r.\pi) \longrightarrow_R [\![\mathcal{S}]\!], \lambda r.e_0$$

  Using the fact that $e_0 = Close([\![e]\!]_{\{\}})$, we have $\lambda r.e_0 = Close([\![e]\!]_{\{\}})$. By Lemma A.19, $Close([\![e]\!]_{\{\}}) = Close([\![e]\!]_r)$. And by Lemma A.20, we get $Close([\![e]\!]_r) = Close([\![e]\!]_{\{\},r}) = (e_0, \texttt{nil})$. Hence, $Close([\![\langle e\rangle]\!]_{\{\}}) = \lambda r.e_0$.

- Case ESREF(2): $\mathcal{S}, \mathsf{ref}\ e \longrightarrow_0 \mathcal{S}\text{↞}\{\ell : e\}, \ell$ with the premise $e \in Val^0$ and $\ell \notin dom(\mathcal{S})$.

  Note that, because $e \in Val^0$, we have $[\![e]\!]_{\{\}} = (e_0, \texttt{nil})$ and $e_0 \in RVal$. Therefore, $Close([\![\mathsf{ref}\ e]\!]_{\{\}}) = \mathsf{ref}\ e_0$, and by ERREF

  $$[\![\mathcal{S}]\!], \mathsf{ref}\ e_0 \longrightarrow_R [\![\mathcal{S}]\!]\text{↞}\{\ell : e_0\}, \ell$$

  Trivially, $\ell \longrightarrow^*_{|\beta|} \ell$, and $[\![\mathcal{S}\text{↞}\{\ell : e\}]\!] = [\![\mathcal{S}]\!]\text{↞}\{\ell : e_0\}$ because $Close([\![e]\!]_{\{\}}) = e_0$.

- Case ESDEREF(2): $\mathcal{S}, !\ell \longrightarrow_0 \mathcal{S}, v$ with the premise $\mathcal{S}(\ell) = v$.

  Note that $Close([\![!\ell]\!]_{\{\}}) = !\ell$ and $Close([\![v]\!]_{\{\}}) = (v_0, \texttt{nil})$ and $[\![\mathcal{S}]\!](\ell) = v_0$. Hence, by ERDEREF

  $$[\![\mathcal{S}]\!], !\ell \longrightarrow_R [\![\mathcal{S}]\!], v_0$$

- Case ESASGN(3): $\mathcal{S}, \ell{:=}e_2 \longrightarrow_0 \mathcal{S}\!\!\leftarrow\!\!\{\ell : e_2\}, e_2$ with the premise $e_2 \in Val^0$.

  Note that, because $e_2 \in Val^0$, we have $[\![e_2]\!]_{\{\}} = (e_0, \texttt{nil})$ and $e_0 \in RVal$. Therefore, $Close([\![\ell{:=}e_2]\!]_{\{\}}) = \ell{:=}e_0$, and by ERREF

  $$[\![\mathcal{S}]\!], \ell{:=}e_0 \longrightarrow_R [\![\mathcal{S}]\!]\!\!\leftarrow\!\!\{\ell : e_0\}, e_0$$

  Recall that $e_0 = Close([\![e_2]\!]_{\{\}})$. □

Proof of Theorem 10.10. By structural induction on e_1. This proof is similar to Theorem 7.4 with additional use of the fact that the reduction does not alter the unmatched holes inside expressions if the stage is greater than 0, and that there are no unmatched holes if the stage is 0. □

Proof of Theorem 10.13. By induction on the structure of e. The proof frequently uses Lemma A.1 based on the fact obtained from Lemma A.18. □