Jones-Optimal Partial Evaluation by Specialization-Safe Normalization

MATT BROWN, University of California Los Angeles, USA
JENS PALSBERG, University of California Los Angeles, USA

We present partial evaluation by specialization-safe normalization, a novel partial evaluation technique that is Jones-optimal, that can be self-applied to achieve the Futamura projections and that can be type-checked to ensure it always generates code with the correct type. Jones-optimality is the gold-standard for nontrivial partial evaluation and guarantees that a specializer can remove an entire layer of interpretation. We achieve Jones-optimality by using a novel affine-variable static analysis that directs specialization-safe normalization to always decrease a program's runtime.

We demonstrate the robustness of our approach by showing Jones-optimality in a variety of settings. We have formally proved that our partial evaluator is Jones-optimal for call-by-value reduction, and we have experimentally shown that it is Jones-optimal for call-by-value, normal-order, and memoized normal-order. Each of our experiments tests Jones-optimality with three different self-interpreters.

We implemented our partial evaluator in $F_{\omega}^{\mu i}$, a recent language for typed self-applicable meta-programming. It is the first Jones-optimal and self-applicable partial evaluator whose type guarantees that it always generates type-correct code.

CCS Concepts: • **Software and its engineering → Source code generation**; *Translator writing systems and compiler generators*; Interpreters;

Additional Key Words and Phrases: Partial Evaluation, Jones-Optimality, Futamura Projections, Meta-Programming, Lambda Calculus.

ACM Reference Format:
Matt Brown and Jens Palsberg. 2018. Jones-Optimal Partial Evaluation by Specialization-Safe Normalization. *Proc. ACM Program. Lang.* 2, POPL, Article 14 (January 2018), 53 pages. https://doi.org/10.1145/3158102

1 INTRODUCTION

A partial evaluator implements Kleene's s-m-n theorem. Given a program and specific values for some of its inputs, a partial evaluator generates a version of the program specialized to those inputs. When given the remaining inputs, the specialized program computes the same result as the original when given all the inputs at once. The goal of partial evaluation is to do some of the computation ahead of time, so that the specialized program runs faster than the original. Researchers have developed partial evaluators for Scheme [Bondorf and Danvy 1991], C [Andersen 1992], and many others. Researchers have also shown how to self-apply partial evaluators and generate the Futamura projections [Jones et al. 1985], which can be used to compile programs and generate compilers and compiler-generators. Finally, researchers have developed Jones-optimal partial evaluators, that is, they can specialize away the overhead of an interpreter.

Authors' addresses: Matt Brown, University of California Los Angeles, USA, msb@cs.ucla.edu; Jens Palsberg, University of California Los Angeles, USA, palsberg@ucla.edu.

2475-1421/2018/1-ART14
https://doi.org/10.1145/3158102

What is the type of a partial evaluator? In 1993, Jones, Gomard, and Sestoft [Jones et al. 1993, Section 16.2] suggested the following polymorphic type:

$$\forall \mathtt{A}{:}*.\ \forall \mathtt{B}{:}*.\ \mathtt{Exp}\ (\mathtt{A} \to \mathtt{B}) \to \mathtt{A} \to \mathtt{Exp}\ \mathtt{B}$$

Here, `Exp (A → B)` is the type of a representation of a program with type `A → B`. This technique is called *typed representation*. Implementing a partial evaluator that operates on typed representations guarantees that it always generates well-typed code of the correct type. Jones, Gomard, and Sestoft showed that the above type supports self-application and the Futamura projections. However, the challenge to define a partial evaluator with that type has remained open. Carette, Kiselyov, and Shan [Carette et al. 2009] implemented a partial evaluator with the closely related type that provides the same guarantee:

$$\forall \mathtt{A}{:}*.\ \forall \mathtt{B}{:}*.\ \mathtt{Exp}\ (\mathtt{A} \to \mathtt{B}) \to \mathtt{Exp}\ \mathtt{A} \to \mathtt{Exp}\ \mathtt{B}$$

Their partial evaluator cannot be self-applied and comes with no claim about Jones optimality. In this paper we present a partial evaluator with the latter type, the first self-applicable partial evaluator that operates on typed representations, supports the Futamura projections, and is Jones optimal.

Our starting point is Mogensen's partial evaluator [Mogensen 1995] for the untyped lambda-calculus. He used reduction to β-normal form as the basis for his partial evaluator, a technique we call specialization by normalization. A key to supporting the Futamura projections using specialization by normalization is that the partial evaluator itself must have a β-normal form. Mogensen showed this is possible by using Church-encoding to represent λ-terms.

Our first step is to observe that Mogensen's partial evaluator is not Jones optimal for reduction strategies with *sharing* – where the work of reducing a value bound to a variable is done at most once, either up front or on demand, regardless of how many times that variable is referenced. Such reduction strategies (e.g. call-by-value, call-by-need/lazy) are more commonly used in practice than those without sharing (e.g. call-by-name), because sharing provides significantly better performance. The problem with Mogensen's partial evaluator is that β-normalization is too aggressive for strategies with sharing; it duplicates work that could have been shared, resulting in specialized programs that are slower than the original. Our solution is to define a subset of β-reduction that is *specialization-safe* – that doesn't duplicate work and never causes a slowdown. Specialization-safe reduction relies on a novel affine-variable analysis that is used to identify the β-redexes in that safe subset.

Our partial evaluator modifies Mogensen's specialization by normalization technique to use specialization-safe reduction. Like Mogensen's partial evaluator, ours has a β-normal form, so it supports the Futamura projections. However, ours never causes a slowdown and is Jones optimal.

Jones optimality is defined in terms of a fixed self-interpreter and a fixed reduction strategy. Given the variety of reduction strategies used in practice, and the many ways to write a self-interpreter, there are many different instances of Jones optimality to consider. Our partial evaluator is the first to be Jones optimal for three different self-interpreters combined with three different reduction strategies. Each self-interpreter operates on a different style of representation, and has a different amount of interpretational overhead. The representation techniques we consider are tagless-final higher-order abstract syntax (HOAS), Mogensen-Scott HOAS, and a tagless-final representation with deBruijn indices.

We apply our techniques to untyped λ-calculus and to a typed λ-calculus $F_\omega^{\mu i}$ that supports self-applicable meta-programming. For untyped λ-calculus, we prove that partial evaluation by specialization-safe normalization is Jones optimal for each self-interpreter under call-by-value reduction. For $F_\omega^{\mu i}$, we experimentally verify Jones optimality for each interpreter and each of call-by-value, normal-order, and memoized normal-order reduction, as indicated by the Xs in Table 1.

Reduction	**Tagless-final HOAS**	**Mogensen-Scott HOAS**	**Tagless-final deBruijn**
Call-by-value	X √	X	X
Normal order	X	X	X
Memoized normal order	X	X	X

Table 1. Experimental (X) and formal (√) verification of Jones optimality of our $F_\omega^{\mu i}$ partial evaluator.

The √ indicates that we have also formally proved Jones optimality for the tagless-final HOAS interpreter and call-by-value reduction.

An appendix containing proofs of the theorems stated in this paper is available from our website [Brown and Palsberg 2018]. We also provide our software artifact that includes implementations of $F_\omega^{\mu i}$ and its self-interpreters and partial evaluator, as well as instructions for reproducing our experimental results.

Rest of the paper. In Section 2, we review Mogensen's partial evaluator for untyped λ-calculus and demonstrate that it is not Jones optimal. In Section 3, we introduce our key notion of a specialization-safe reduction relation. In Section 4, we present a specialization-safe reduction that operates on untyped λ-terms with affine variable annotations. In Section 5, we discuss partial evaluation by specialization-safe normalization, and prove it Jones optimal for three self-interpreters. In Section 6, we present our typed self-applicable partial evaluator for $F_\omega^{\mu i}$ and the typed Futamura projections. In Section 7, we discuss our experimental results, and in Section 8 we discuss related work.

2 PARTIAL EVALUATION AND JONES OPTIMALITY

A partial evaluator is a meta-program that specializes another program to some of its inputs. When given the remaining inputs, the specialized program will compute the same result as running the original program on all the inputs. Kleene's s-m-n theorem established that such a specialization process is possible, and since then partial evaluation has been established as a practical technique for program optimization and automatic program generation.

We will define partial evaluation as the specialization of a two-input function to its first input (corresponding to the s-1-1 instance of s-m-n).

Definition 2.1 (Partial Evaluation). `mix` is a partial evaluator if for every program p and input x, there exists a specialized program $\mathsf{p_x}$ such that:

(1) If $\mathtt{mix}\ \overline{\mathsf{p}}\ \overline{\mathsf{x}}$ has a β-normal form, that normal form is $\overline{\mathsf{p_x}}$
(2) $\forall \mathsf{y}.\ \mathsf{p_x}\ \mathsf{y} \equiv_\beta \mathsf{p}\ \mathsf{x}\ \mathsf{y}$

The notation $\overline{\mathsf{p}}$ denotes the representation of p according to some fixed representation scheme. The first condition indicates that the partial evaluator may not be total: specialization may not terminate. When it does terminate, however, it outputs the representation of a specialized program $\mathsf{p_x}$. The second condition is the correctness condition for the specialized program $\mathsf{p_x}$: it must have the same behavior as p x on all inputs y. The definition naturally extends to other numbers of inputs. The inputs to which the program is specialized are often called "static" and the remaining inputs "dynamic". The static inputs are given as representations because they may become part of the output program representation. This is necessary for a λ-calculus without constants [Launchbury 1991].

Futamura projections. A partial evaluator is self-applicable if it can specialize itself. A classical application of a self-applicable partial evaluator is to generate the Futamura projections, which

Description	Definition	Correctness Specification
1. Compile S-program to T	$\texttt{mix}\ \overline{\texttt{int}}\ \overline{\overline{\texttt{s}}} \equiv_\beta \overline{\texttt{t}}$	$\forall \texttt{x}.\ \texttt{t x} \equiv_\beta \texttt{int}\ \overline{\texttt{s}}\ \texttt{x}$
2. Generate an S-to-T compiler	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{int}}} \equiv_\beta \overline{\texttt{comp}}$	$\forall \texttt{s}.\ \texttt{comp}\ \overline{\texttt{s}} \equiv_\beta \texttt{mix}\ \overline{\texttt{int}}\ \overline{\overline{\texttt{s}}}$
3. Generate a compiler-generator	$\texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{mix}}} \equiv_\beta \overline{\texttt{cogen}}$	$\forall \texttt{int}.\ \texttt{cogen}\ \overline{\overline{\texttt{int}}} \equiv_\beta \texttt{mix}\ \overline{\texttt{mix}}\ \overline{\overline{\texttt{int}}}$
4. Self-generation of cogen		$\texttt{cogen}\ \overline{\overline{\texttt{mix}}} \equiv_\beta \overline{\texttt{cogen}}$

Fig. 1. The four Futamura projections for λ-calculus.

$$\texttt{mix}\ \overline{\texttt{f}}\ \overline{\texttt{d}} \equiv_\beta \overline{\texttt{NF}_\beta\texttt{(f d)}}$$

Fig. 2. Mogensen's Specialization by Normalization

use specialization to compile programs and generates compilers and compiler-generators [Futamura 1999]. The four Futamura projections are shown in Figure 1. For each projection we list a correctness specification implied by the correctness of mix. The first Futamura projection compiles programs from a source language S to a target language T by specializing an S-interpreter programmed in T. Given any input to the program, the target program gives the same result as interpreting the source program. The second projection generates a compiler from S-programs to T-programs by specializing the partial evaluator mix to the interpreter. Given any program, the compiler gives the same result as the first projection. The third projection generates a compiler-generator by specializing mix to itself. Given any interpreter, the compiler-generator gives the same result as the second projection. Futamura only discussed these three Futamura projections. Subsequently Glück [Glück 2009] showed that there are more. In particular, the fourth projection demonstates that the compiler-generator cogen can generate itself when applied to mix. This is sometimes called the "mixpoint".

The twice-overlined terms in the Futamura projections are representations of representations. These double-representations arise because mix requires its static input to be represented, as we mentioned above.

Jones-optimality. Jones-optimality is the gold-standard for showing a partial evaluator can perform a significant amount of specialization. The definition of Jones-optimality specializes a self-interpreter as a benchmark for partial evaluation.

Definition 2.2. A term u is a self-interpreter for a representation scheme $\overline{\cdot}$ if for any term p, we have $\texttt{u}\ \overline{\texttt{p}} \equiv_\beta \texttt{p}$.

A partial evaluator is Jones-optimal if it can specialize away all the computational overhead caused by self-interpretation. More precisely, specializing a self-interpreter to a program should generate a program that is no slower than the original program – on *any* input.

The canonical definition of Jones optimality is from Jones, Gomard and Sestoft [Jones et al. 1993, pg. 139]. We restate it for our context of λ-calculus and using our notation:

Definition 2.3 (Jones-optimality). A partial evaluator mix is Jones-optimal with respect to *time* and a self-interpreter u if, for any terms $\texttt{p} : \texttt{A} \to \texttt{B}$ and $\texttt{d} : \texttt{A}$, $\mathit{time}(\texttt{p}'\ \texttt{d}) \leq \mathit{time}(\texttt{p d})$, where $\texttt{mix}\ \overline{\texttt{u}}\ \overline{\overline{\texttt{p}}} \equiv_\beta \overline{\texttt{p}'}$.

Jones optimality is defined in terms of a particular fixed *time* function and self-interpreter u. Since different notions of *time* may be most appropriate for different settings, and because there are different ways of implementing a self-interpreter (and different ways of representing programs) for

$$\frac{}{(\lambda \texttt{x.e})\texttt{v} \to \texttt{e[x:=v]}} \qquad \frac{\texttt{e1} \to \texttt{e1}'}{\texttt{e1 e2} \to \texttt{e1}'\ \texttt{e2}} \qquad \frac{\texttt{e1} \to \texttt{e1}'}{(\lambda \texttt{x.e})\texttt{e1} \to (\lambda \texttt{x.e})\texttt{e1}'}$$

Fig. 3. Call-by-value reduction.

$$\frac{}{\texttt{x} \rhd \texttt{x}}$$

$$\frac{\texttt{e} \rhd \texttt{q}}{\lambda \texttt{x.e} \rhd \texttt{abs}\ (\lambda \texttt{x.q})}$$

$$\frac{\texttt{e1} \rhd \texttt{q1} \qquad \texttt{e2} \rhd \texttt{q2}}{\texttt{e1 e2} \rhd \texttt{app q1 q2}}$$

$$\frac{\texttt{e} \rhd \texttt{q}}{\overline{\texttt{e}} = \lambda \texttt{abs}.\ \lambda \texttt{app}.\ \texttt{q}}$$

$$\texttt{u} = \lambda \texttt{e}.\ \texttt{e}\ (\lambda \texttt{x.x})\ (\lambda \texttt{x.x})$$

Fig. 4. Mogensen's Church-encoded representation and self-interpreter for untyped λ-calculus.

a single language, we can consider multiple notions of Jones optimality. We will always explicitly state which *time* function and self-interpreter is under consideration. In this section, we focus on steps of call-by-value (CBV) reduction to a value as our notion of *time*. Call-by-value reduction is defined in Figure 3.

Definition 2.4. $time_{cbv}(\texttt{e}) = n$ if and only if $\texttt{e} \to^n \texttt{v}$ for some value v.

Mogensen [Mogensen 1995] implemented a self-applicable partial evaluator for untyped λ-calculus by reducing terms to β-normal form, a technique we call "specialization by normalization". With this approach, specializing a program f to an input x returns $\texttt{NF}_\beta\texttt{(f x)}$, the β-normal form of f x. One problem with specialization by normalization is that not all terms have a normal form, which leads to non-termination of the partial evaluator: if f x has no normal form, then specializing f to x will not terminate. In particular, if the self-interpreter u and the partial evaluator mix are not strongly normalizing terms, then the Futamura projections will not terminate. Mogensen solved this problem by using a representation based on Church-encoding, for which a strongly normalizing self-interpreter and normalizer can be defined. The partial evaluator is self-applicable and the Futamura projections terminate.

Mogensen's Church-encoding representation and self-interpreter are shown in Figure 4. A representation is built in two steps: first, the pre-quoter $\rhd$ applies designated variables abs and app throughout the term, at each λ-abstraction and application respectively. We assume abs and app do not occur in the term, which can be ensured by renaming variables. The quoter $\overline{\cdot}$ simply abstracts over abs and app in the pre-quoted term. This formulation of Church encoding produces representations in β-normal form. It is possible to define quotation as a single function (without a pre-quoter), but the representations produced would not be β-normal.

Mogensen's self-interpreter u is instantiates both abs and app to identity functions, resulting in a term that is β-equivalent to e. The self-interpreter u is β-normal. While Mogensen's normalizer is significantly more complex than his self-interpreter, it is also strongly normalizing.

$$
\begin{aligned}
\mathtt{n}_i &= \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}^i\ \mathtt{z}\\
\mathtt{succ} &= \lambda \mathtt{n}.\lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})\\
\\
\mathtt{plus} &= \lambda \mathtt{n}.\lambda \mathtt{m}.\mathtt{n}\ \mathtt{succ}\ \mathtt{m}\\
\mathtt{times} &= \lambda \mathtt{n}.\lambda \mathtt{m}.\mathtt{n}\ (\mathtt{plus}\ \mathtt{m})\ \mathtt{n}_0\\
\mathtt{square} &= \lambda \mathtt{n}.\mathtt{times}\ \mathtt{n}\ \mathtt{n}\\
\\
\mathtt{f} &= \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{times}\ \mathtt{x}\ (\mathtt{square}\ \mathtt{y})\\
\mathtt{g} &= \mathtt{f}\ \mathtt{n}_2
\end{aligned}
$$

Fig. 5. An implementation of $g(x) = 2x^2$ on Church numerals

To summarize, Mogensen made three key insights: 1) a self-evaluator that evaluates to normal form can be used as the basis for a partial evaluator (specialization by normalization), 2) specialization by normalization supports the Futamura projections if the self-interpreter and partial evaluator themselves are strongly normalizing, and 3) a representation based on Church-encoding supports a strongly normalizing self-interpreter and specialization by normalization partial evaluator.

Specialization by normalization is not optimal for $time_{cbv}$. For example, consider the λ-term g in Figure 5 that computes the polynomial $g(x) = 2x^2$ on Church numerals. The specialization of u to $\overline{\mathtt{g}}$ yields a term $\mathtt{u}_\mathtt{g}$ that is slower than g on some inputs. For example, $\mathtt{u}_\mathtt{g}\ \mathtt{n}_3$ evaluates in 89 steps, while $\mathtt{g}\ \mathtt{n}_3$ evaluates in 82 steps. The problem is not limited to compilation; specialization in general can cause slowdown. The specialization of the λ-term f in Figure 5 to the church numeral $\mathtt{n}_2$ evaluates in more steps than $\mathtt{f}\ \mathtt{n}_2$.

The reason for this slowdown is that β-normalization contracts some β-redexes that cause duplicated work in the residual code – in particular, redexes of the form $(\lambda \mathtt{x.e1})\mathtt{e2}$ where e2 is not a value. Reducing such a redex can introduce multiple new copies of e2 into the term, each of which may need to be evaluated separately. For example, consider the specialization of f to $\mathtt{n}_2$, which reduces in one step to $\lambda \mathtt{y}.\ \mathtt{times}\ \mathtt{n}_2\ (\mathtt{square}\ \mathtt{y})$, and then in a few more steps to $\lambda \mathtt{y}.\ \mathtt{plus}\ (\mathtt{square}\ \mathtt{y})\ (\mathtt{plus}\ (\mathtt{square}\ \mathtt{y})\ \mathtt{n}_0)$. The specializer will continue reducing, but we can see the problem already at this point: we now have two copies of square y instead of one. We can't compute either copy at specialization-time, because we don't know the value of y until run-time. When we run the specialized program with the input $\mathtt{n}_2$ for y, will have to evaluate $(\mathtt{square}\ \mathtt{n}_2)$ twice.

Returning to our first example, specializing u to g, we see that since $\mathtt{u}\ \overline{\mathtt{g}} \equiv_\beta \mathtt{g} = \mathtt{f}\ \mathtt{n}_2$, the β-normal form of $\mathtt{u}\ \overline{\mathtt{g}}$ is the same as the β-normal form of $\mathtt{f}\ \mathtt{n}_2$. The result is the same as for the specialization of f to $\mathtt{n}_2$, and we get a slowdown for the same reason.

3 SPECIALIZATION SAFETY

In this section, we consider how Mogensen's specialization-by-normalization approach can be modified so that it can support the Futamura projections and also be Jones optimal. Jones optimality has two requirements: first, the partial evaluator must be strong enough to specialize away the work of a self-interpreter; second, it must not duplicate work or otherwise cause a slowdown.

Our goal is to identify a reduction relation that can be used used as the basis for a Jones-optimal specialization-by-normalization partial evaluator. Our main requirements are as follows: first, the reduction relation should be specialization-safe, meaning it never causes a slowdown at run-time;

and second, it should be strong enough to remove all the work of a self-interpreter. We now formally state these two requirements.

Definition 3.1. A reduction relation $\rightsquigarrow$ is specialization-safe for *time* if $\mathsf{e1} \rightsquigarrow^* \mathsf{e2}$ implies $time(\mathsf{e2}\ \mathsf{x}) \leq time(\mathsf{e1}\ \mathsf{x})$ for all x.

Definition 3.2. For a given self-interpreter u for a representation function $\widehat{\cdot}$, a reduction relation $\rightsquigarrow$ is u-strong if $\mathsf{u}\ \widehat{\mathsf{p}} \rightsquigarrow^* \mathsf{p}$ for all p.

We also require that the reduction relation be confluent and have unique normal forms, so that we can implement a specialization-by-normalization partial evaluator for it. Finally, in order to support the Futamura projections, we need the implementation of the partial evaluator itself to have a normal form.

Theorem 3.3. *If $\rightsquigarrow$ is confluent, specialization-safe for time, and* u*-strong, then for any terms* p *and* d, $time(\mathsf{p}'\ \mathsf{d}) \leq time(\mathsf{p}\ \mathsf{d})$, *where* $\mathsf{p}' = \mathsf{NF}_{\rightsquigarrow}(\mathsf{u}\ \widehat{\mathsf{p}})$.

Proof. From that $\rightsquigarrow$ is confluent and u-strong, we have $\mathsf{p}' = \mathsf{NF}_{\rightsquigarrow}(\mathsf{u}\ \widehat{\mathsf{p}}) = \mathsf{NF}(\mathsf{p})$, so $\mathsf{p} \rightsquigarrow^* \mathsf{p}'$. From that $\rightsquigarrow$ is specialization-safe for *time*, we have $time(\mathsf{p}'\ \mathsf{d}) \leq time(\mathsf{p}\ \mathsf{d})$. □

In the remainder of this section, we examine a variety of β-redexes and determine if they can be included in a specialization-safe reduction relation. In the next section we introduce affine variable annotations that help identify those specialization-safe β-redexes and define a specialization-safe subset of β-reduction that operates on annotated terms.

Intuitively, whether a redex $(\lambda\mathsf{x}.\mathsf{e})\mathsf{a}$ is specialization-safe depends upon two things: the potential cost of evaluating a at run-time, and the number of times the value of a might be needed.

Consider for example the term $\lambda\mathsf{f}.(\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x}\ \mathsf{x})\ \mathsf{n}_2$, which contains a single redex $(\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x}\ \mathsf{x})\ \mathsf{n}_2$. Since there are two occurrences of x, the value of n_2 will be needed twice. However, n_2 is itself a value, so the cost of evaluating it is 0. Therefore, this redex is specialization-safe – contracting does not duplicate any work.

Consider instead the term $\lambda\mathsf{f}.(\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{square}\ \mathsf{n}_2)$, which contains the redex $(\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{square}\ \mathsf{n}_2)$. This time, the argument $(\mathsf{square}\ \mathsf{n}_2)$ is not a value, so contracting this redex would duplicate work: in the reduct $\mathsf{f}\ (\mathsf{square}\ \mathsf{n}_2)\ (\mathsf{square}\ \mathsf{n}_2)$, we have to evaluate $(\mathsf{square}\ \mathsf{n}_2)$ twice. However, $(\mathsf{square}\ \mathsf{n}_2)$ can be reduced to a value at specialization time, so we can reduce the redex $(\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{square}\ \mathsf{n}_2)$ to a specialization-safe redex and then contract it.

Next, consider the term $\lambda\mathsf{f}.(\lambda\mathsf{x}.\mathsf{times}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{f}\ \mathsf{n}_2)$ and its redex $(\lambda\mathsf{x}.\mathsf{times}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{f}\ \mathsf{n}_2)$. This time, specialization cannot reduce the argument $\mathsf{f}\ \mathsf{n}_2$ to a value, because f is unknown. If we leave the redex uncontracted, then $\mathsf{f}\ \mathsf{n}_2$ will be evaluated once at run-time, and the redex will be contracted afterwards. If we contract the redex, the resulting term $\mathsf{times}\ (\mathsf{f}\ \mathsf{n}_2)\ (\mathsf{f}\ \mathsf{n}_2)$ duplicates $\mathsf{f}\ \mathsf{n}_2$ so it will need to be evaluated twice at run-time, potentially causing a slowdown. Therefore, this redex is not specialization-safe.

Next, consider $\lambda\mathsf{f}.(\lambda\mathsf{y}.\mathsf{square}\ \mathsf{y})\ (\mathsf{f}\ \mathsf{n}_2)$, which is β-equivalent to the previous example, and its redex $(\lambda\mathsf{y}.\mathsf{square}\ \mathsf{y})\ (\mathsf{f}\ \mathsf{n}_2)$. As before, specialization cannot reduce $\mathsf{f}\ \mathsf{n}_2$ to a value. But in this case the value of y is needed only once, so this redex is specialization-safe and can be contracted to $\mathsf{square}\ (\mathsf{f}\ \mathsf{n}_2)$ without risking a slowdown. The resulting term $\lambda\mathsf{f}.\mathsf{square}\ (\mathsf{f}\ \mathsf{n}_2) = \lambda\mathsf{f}.(\lambda\mathsf{x}.\mathsf{times}\ \mathsf{x}\ \mathsf{x})\ (\mathsf{f}\ \mathsf{n}_2)$ recovers the previous example.

Next, consider the term $(\lambda\mathsf{y}.\mathsf{square}\ \mathsf{y}) = \lambda\mathsf{y}.(\lambda\mathsf{x}.\mathsf{times}\ \mathsf{x}\ \mathsf{x})\ \mathsf{y}$ and its redex $(\lambda\mathsf{x}.\mathsf{times}\ \mathsf{x}\ \mathsf{x})\ \mathsf{y}$. Here, the argument y is a variable and so not a value, and neither can specialization reduce it to a value. However, since call-by-value always reduces arguments to values before β-contracting, we

$$\begin{aligned}\text{(values) } \mathtt{v} &:= \lambda\mathtt{x.e} \mid \lambda\mathtt{x}^\circ\mathtt{.e}\\ \text{(terms) } \mathtt{e} &:= \mathtt{x} \mid \mathtt{x}^\circ \mid \mathtt{e1\ e2} \mid \lambda\mathtt{x.e} \mid \lambda\mathtt{x}^\circ\mathtt{.e}\end{aligned}$$

Fig. 6. Untyped λ-calculus with affine-variable annotations

$$\text{CBV-1}\ \frac{}{(\lambda\mathtt{x.e})\mathtt{v} \to \mathtt{e[x:=v]}} \qquad \text{CBV-4}\ \frac{}{(\lambda\mathtt{x}^\circ\mathtt{.e})\mathtt{v} \to \mathtt{e[x}^\circ\mathtt{:=v]}}$$

$$\text{CBV-2}\ \frac{\mathtt{e1} \to \mathtt{e1}'}{\mathtt{e1\ e2} \to \mathtt{e1}'\ \mathtt{e2}} \qquad \text{CBV-5}\ \frac{\mathtt{e1} \to \mathtt{e1}'}{(\lambda\mathtt{x}^\circ\mathtt{.e})\mathtt{e1} \to (\lambda\mathtt{x}^\circ\mathtt{.e})\mathtt{e1}'}$$

$$\text{CBV-3}\ \frac{\mathtt{e1} \to \mathtt{e1}'}{(\lambda\mathtt{x.e})\mathtt{e1} \to (\lambda\mathtt{x.e})\mathtt{e1}'}$$

Fig. 7. Call-by-value reduction on untyped λ-terms with affine variable annotations.

know that at run-time, y will be substituted with a value. Therefore this redex is also specialization-safe, and $(\lambda\mathtt{y.square\ y})$ can be reduced to $(\lambda\mathtt{y.times\ y\ y}) \equiv_\alpha \mathtt{square}$. This example shows that η-contraction is specialization-safe.

Finally, consider $\lambda\mathtt{f.}\lambda\mathtt{g.g\ (times\ (f\ n_2))}$, and its redex $\mathtt{times\ (f\ n_2)} = (\lambda\mathtt{n.}\lambda\mathtt{m.n\ (plus\ m)\ n_0)\ (f\ n_2)}$. Since $\mathtt{(f\ n_2)}$ cannot be reduced to a value at specialization time, it will have a nonzero cost at run-time. However, since n occurs only once in the definition of times, it is tempting to conclude that this redex is specialization-safe. This conclusion is wrong. Since n occurs under the λ-abstraction $\lambda\mathtt{m.\ n\ (plus\ m)\ n_0}$, the value of n will be needed every time that abstraction is applied. Since g is unknown, we cannot predict how many times the λ-abstraction will be applied and n will be needed. If we contract the redex, then we have to recompute $\mathtt{f\ n_2}$ every time g is applied. This could cause a lot of duplicated work.

4 AFFINE VARIABLES AND SPECIALIZATION-SAFE REDUCTION

The examples in the previous section demonstrate that full β-reduction is not specialization-safe: some β-contractions at specialization-time can cause a slowdown at run-time. In this section, we define *specialization-safe reduction*, a subset of full β-reduction that is indeed specialization-safe: Theorem 4.5 states that no sequence of specialization-safe reductions can result in a term that evaluates in more call-by-value steps than the original.

We achieve specialization safety by only contracting β-redexes that cannot cause a slowdown. We use affine-variable annotations to indicate which variables are guaranteed to be referenced at most once at run-time. Affine variables are similar to linear variables, except that linear variables are guaranteed to be referenced *exactly* once.

Figure 6 defines the untyped λ-calculus (ULC) with *affine* variables. Affine variables are annotated like $\mathtt{x}^\circ$, while unlimited variables are unannotated. The values are λ-abstractions as usual, and they may abstract over affine or unlimited variables. Throughout this section, all λ-terms can have annotations.

Figure 7 defines call-by-value reduction on annotated λ-terms. Call-by-value reduction ignores annotations – the rules CBV-1 and CBV-3 are identical except for the annotations, as are CBV-4 and CBV-5. We overload the arrow $\to$ that denotes call-by-value reduction for unannotated terms. We will make clear which kind of term is under consideration at all times. We define $time_{cbv}$ for annotated terms similarly to the definition for unannotated terms.

Definition 4.1. For an annotated term e, $time_{cbv}(\mathtt{e}) = n$ if and only if $\mathtt{e} \to^n \mathtt{v}$ for some value v.

(unlimited contexts) $\Gamma = \langle\rangle \mid \Gamma, x$
(affine contexts) $\Sigma = \langle\rangle \mid x^\circ$

$$\langle\rangle \uplus x = x \qquad x \uplus \langle\rangle = x \qquad \langle\rangle \uplus \langle\rangle = \langle\rangle$$

$$\frac{x \in \Gamma}{\Gamma; \Sigma \vdash x}$$

$$\frac{}{\Gamma; x^\circ \vdash x^\circ}$$

$$\frac{\Gamma; \Sigma_1 \vdash e_1 \qquad \Gamma; \Sigma_2 \vdash e_2}{\Gamma; \Sigma \vdash e_1\ e_2}\ \Sigma = \Sigma_1 \uplus \Sigma_2$$

$$\frac{(\Gamma, x); \langle\rangle \vdash e}{\Gamma; \Sigma \vdash \lambda x.e}$$

$$\frac{\Gamma; x^\circ \vdash e}{\Gamma; \Sigma \vdash \lambda x^\circ.e}$$

Fig. 8. Well-formedness rules for affine-variable annotations.

$$\text{SSR-1}\ \frac{}{(\lambda x.e)v \rightarrow_s e[x:=v]} \qquad \text{SSR-5}\ \frac{}{(\lambda x.e)y \rightarrow_s e[x:=y]}$$

$$\text{SSR-2}\ \frac{e1 \rightarrow_s e1'}{e1\ e2 \rightarrow_s e1'\ e2} \qquad \text{SSR-6}\ \frac{e \rightarrow_s e'}{(\lambda x.e) \rightarrow_s (\lambda x.e')}$$

$$\text{SSR-3}\ \frac{e1 \rightarrow_s e1'}{e\ e1 \rightarrow_s e\ e1'} \qquad \text{SSR-7}\ \frac{e \rightarrow_s e'}{(\lambda x^\circ.e) \rightarrow_s (\lambda x^\circ.e')}$$

$$\text{SSR-4}\ \frac{}{(\lambda x^\circ.e)e' \rightarrow_s e[x^\circ:=e']}$$

Fig. 9. Specialization-Safe Reduction

Figure 8 defines a simple set of rules for checking affine-variable annotations. We track variables in two contexts: a context Γ that contains unlimited variables, and a context Σ that contains affine variables. Σ is allowed to contain at most one variable at a time. The rule for unlimited variables checks for the variable in the unlimited context, and the one for affine variables checks the affine context. For an application $e_1\ e_2$, we check e_1 and e_2 using the same unlimited context, but disjoint affine contexts. This ensures that only e_1 or e_2 may reference the current affine variable. We always check the body of an unlimited λ-abstraction using an empty affine context. We check the body of an affine λ-abstraction $(\lambda x^\circ.e)$ using the affine context containing only x°. This means that affine variables may never occur under a nested λ abstraction.

Figure 9 defines specialization-safe reduction on annotated terms. The rule SSR-1 is ordinary call-by-value β-reduction. The rules SSR-2 and SSR-3 allow a step to occur in either sub-term of an application, like in full β-reduction. Call-by-value reduction would restrict SSR-3 to when e is a value. The rule SSR-4 is the key rule enabled by the affine-variable annotations. It allows a β-reduction when the argument is not a value, as long as the λ-abstraction binds an affine variable. The rule SSR-5 allows a β-reduction when the argument is an unlimited variable, as long as the λ-abstraction is also unlimited. Note that we do not allow β-reduction when the argument is an affine variable and the λ-abstraction is unlimited, because that could duplicate the affine variable. If the λ-abstraction is affine, SSR-4 applies for any argument. The last two rules, SSR-6 and SSR-7, allow reduction under λ-abstractions.

We now apply specialization-safe reduction to the examples from the previous section. First was the term $\lambda f.(\lambda x.f\ x\ x)\ n_2$. The variable x cannot be annotated as affine, because it occurs twice. Neither can f be annotated, because it occurs under the abstraction over x. However, since n_2 is a value, SSR-1 applies, so this is a redex for specialization-safe reduction.

The next example was λ`f.(`λ`x.f x x) (square n`$_2$`)`. Again, `x` and `f` are not affine, but `square n`$_2$ can be reduced to a value, after which SSR-1 will apply.

The next example was λ`f.(`λ`x.times x x) (f n`$_2$`)`, which has the β-redex `(`λ`x.times x x) (f n`$_2$`)`. Again `x` is not affine and `(f n`$_2$`)` is not a value, so this is not a specialization-safe redex. Also, since `(f n`$_2$`)` is not reducible to a value, the β-redex can't be reduced to a specialization-safe redex.

The next example was λ`f.(`λ`y.square y) (f n`$_2$`)`, which can be annotated λ`f`$^\circ$`.(`λ`y`$^\circ$`.square y`$^\circ$`) (f`$^\circ$` n`$_2$`)`. Now consider the β-redex `(`λ`y`$^\circ$`.square y`$^\circ$`) (f`$^\circ$` n`$_2$`)`. The argument `(f`$^\circ$` n`$_2$`)` is not value and can't be reduced to one. However, since `y`$^\circ$ is affine, this is still a specialization-safe redex, reducible using rule SSR-4.

Next consider λ`y.(`λ`x.times x x) y`. The variable `y` is affine, so we can annotate the term λ`y`$^\circ$`.(`λ`x.times x x) y`$^\circ$. In this case, specialization-safe reduction does not allow the β-redex `(`λ`x.times x x) y`$^\circ$ to be reduced, because the reduct `times y`$^\circ$` y`$^\circ$ would not be well-formed. We could instead leave `y` unannotated, and the term is still well-formed. Then `(`λ`x.times x x) y` can be reduced by rule SSR-5 because `y` is unlimited, and the reduct `(`λ`y.times y y)` is well-formed. This demonstrates two important points: a term can have multiple annotations, and different annotations can lead to different specialization-safe reduction behavior. Our partial evaluator always inserts the maximum number of annotations allowed by the rules, as described in the next section.

The last example was λ`f.`λ`g.g (times (f n`$_2$`))`, and its redex `times (f n`$_2$`) = (`λ`n.`λ`m.n (plus m) n`$_0$`) (f n`$_2$`)`. The variable `n` cannot be annotated as affine because it occurs on the λ-abstraction for `m`. Also, the argument `(f n`$_2$`)` is not a value, so this is not reducible by specialization-safe reduction.

The following theorem states that well-formedness is preserved by specialization-safe reduction.

Theorem 4.2. *If* $\Gamma;\Sigma \vdash \mathsf{e}$ *and* $\mathsf{e} \rightarrow_s \mathsf{e}'$, *then* $\Gamma;\Sigma \vdash \mathsf{e}'$.

A term `e` is in specialization-safe normal form (or is specialization-safe-normal) if there is no e' such that $\mathsf{e} \rightarrow_s \mathsf{e}'$. All β-normal forms are also specialization-safe-normal, but a specialization-safe normal form may contain β-redexes (specifically, specialization-unsafe redexes). Specialization-safe reduction is confluent, and so has unique normal forms. Our proof of confluence is based on Nipkow's lecture notes on λ-calculus [Nipkow 2012]. We define $\mathsf{NF}_s(\mathsf{e}) = \mathsf{e}'$ if and only if $\mathsf{e} \rightarrow_s^* \mathsf{e}'$ and e' is specialization-safe-normal.

Theorem 4.3. *Specialization-safe reduction is confluent.*

Corollary 4.4. *Specialization-safe normal forms are unique.*

The following theorem states that specialization-safe reduction cannot cause a slowdown: no sequence of specialization-safe reductions can result in a term that evaluates in more more call-by-value steps than the original.

Theorem 4.5. *If* $\langle\rangle;\langle\rangle \vdash \mathsf{e}$ *and* $\mathsf{e} \rightarrow_s^* \mathsf{e}'$ *and* $\mathsf{e} \rightarrow^i \mathsf{v}$, *then there exists an* $i' \leq i$ *and a value* v' *such that* $\mathsf{e}' \rightarrow^{i'} \mathsf{v}'$, *and* $\mathsf{v} \rightarrow_s^* \mathsf{v}'$.

Theorem 4.5 is illustrated in Figure 10. The solid arrows correspond to the premises of the theorem, and the dashed arrows correspond the conclusion. After specialization-safe reduction, we may end up with a value v' that is different than what we'd get with call-by-value reduction alone. This is possible because specialization-safe reduction allows reduction under λ-abstractions. The two values are equivalent and converge with some additional specialization-safe reduction.

Specialization-safety follows from Theorem 4.5:

Corollary 4.6 (Specialization-safety). *If* $\langle\rangle;\langle\rangle \vdash \mathsf{p}$ *and* $\mathsf{e} \rightarrow_s^* \mathsf{e}'$, *then for all* `x`, $time_{cbv}(\mathsf{e}\ \mathsf{x}) = i$ *implies* $time_{cbv}(\mathsf{e}'\ \mathsf{x}) \leq i$.

$$\begin{tikzcd}
 & \mathtt{e'} \arrow[rr, dashed, "i'"] & & \mathtt{v'} & i' \leq i \\
\mathtt{e} \arrow[ur, "*_s"] \arrow[rr, "i"'] & & \mathtt{v} \arrow[ur, dashed, "*_s"'] & &
\end{tikzcd}$$

Fig. 10. Illustration of Theorem 4.5.

$$\begin{tikzcd}
\mathtt{e1'} \arrow[r, "*_s"] & \mathtt{e2'} \arrow[d, "erase"] \\
\mathtt{e1} \arrow[u, "annotate"] \arrow[r, Rightarrow, "s"'] & \mathtt{e2}
\end{tikzcd}$$

Fig. 11. Specialization-safe reduction for unannotated terms.

Proof. $time_{cbv}(\mathtt{e\ x}) = i$ implies that $\mathtt{e\ x} \rightarrow^{i} \mathtt{v}$ for some $\mathtt{v}$. Since $\mathtt{e} \rightarrow_s^* \mathtt{e'}$, we have $\mathtt{e\ x} \rightarrow_s^* \mathtt{e'\ x}$. By Theorem 4.5, there exists an $i' \leq i$ and a value $\mathtt{v'}$ such that $\mathtt{e'\ x} \rightarrow^{i'} \mathtt{v'}$ and $\mathtt{v} \rightarrow_s^* \mathtt{v'}$. Therefore, $time_{cbv}(\mathtt{e'\ x}) = i' \leq i = time_{cbv}(\mathtt{e\ x})$. □

5 PARTIAL EVALUATION BY SPECIALIZATION-SAFE NORMALIZATION

The specification for a partial evaluator based on specialization-safe normalization is as follows:

$$\mathtt{mix}\ \overline{\mathtt{p}}\ \overline{\mathtt{d}} \equiv_\beta \overline{erase(\mathsf{NF}_s(annotate(\mathtt{p\ d})))}$$

It operates on unannotated terms, but first annotates them with the maximum number of affine annotations allowed by the rules. There is a unique maximally annotated term for each unannotated term, and the annotation process is straightforward. A bound variable can be annotated as affine if 1) the number of occurrences is at most one, and 2) the number of occurrences inside inner lambda abstractions is zero. After reducing to specialization-safe normal form it erases the annotations. Since *annotate*, NF_s, and *erase* are all well-defined functions, there is a unique specialized program for each program and static input.

For unannotated terms $\mathtt{e1}$ and $\mathtt{e2}$, we write $\mathtt{e1} \Longrightarrow_s \mathtt{e2}$ to mean the maximally annotated version of $\mathtt{e1}$ reduces in some number of steps to an annotated version of $\mathtt{e2}$. This is depicted in Figure 11.

Since call-by-value reduction on annotated terms ignores affine-variable annotations, an annotated term evaluates in the same amount of time as the corresponding unannotated term:

Lemma 5.1. *For any annotated term* $\mathtt{e}$, $time_{cbv}(e) = time_{cbv}(erase(e))$.

Proof. Straightforward. □

The following Lemma combines Corollary 4.6 and Lemma 5.1 to bridge the gap between unannotated and annotated λ-terms.

Lemma 5.2. *For any self-interpreter* $\mathtt{u}$ *for a representation function* $\widehat{\cdot}$, *and for any term* $\mathtt{p}$, *if* $\mathtt{u}\ \widehat{\mathtt{p}} \Rightarrow_s \mathtt{p}$, *then* $\mathtt{mix}$ *is Jones-optimal for* $\mathtt{u}$ *and* $time_{cbv}$.

$$\frac{}{\widehat{\mathtt{x}} = \lambda\mathtt{var}.\lambda\mathtt{abs}.\lambda\mathtt{app}.\mathtt{var}\ \mathtt{x}}$$

$$\frac{\widehat{\mathtt{e}} = \mathtt{q}}{\widehat{\lambda\mathtt{x}.\mathtt{e}} = \lambda\mathtt{var}.\lambda\mathtt{abs}.\lambda\mathtt{app}.\mathtt{abs}\ (\lambda\mathtt{x}.\mathtt{q})}$$

$$\frac{\widehat{\mathtt{e1}} = \mathtt{q1} \qquad \widehat{\mathtt{e2}} = \mathtt{q2}}{\widehat{\mathtt{e1}\ \mathtt{e2}} = \lambda\mathtt{var}.\lambda\mathtt{abs}.\lambda\mathtt{app}.\mathtt{app}\ \mathtt{q1}\ \mathtt{q2}}$$

```
fixf  = (λx. f (λy. x x y)) (λx. f (λy. x x y))
fix   = λf. fixf

msint = fix (λu. λe. e (λx.x) (λf.λx.u (f ((λy.x)x))) (λf.λx.u f (u x)))
```

Fig. 12. Mogensen-Scott encoding and self-interpreter.

We prove that a partial evaluator based on specialization-safe normalization is strong enough to specialize away the work of three different self-interpreters. Therefore it is also Jones-optimal for each interpreter and $time_{cbv}$. The first self-interpreter is the one by Mogensen shown in Figure 4. It operates on the same Church-encoded representation $\overline{\cdot}$ that `mix` does. The second uses Mogensen-Scott encoding, another encoding of λ-terms by Mogensen [Mogensen 1992] that is based on Scott's encoding of natural numbers. The third is a tagless-final encoding that uses deBruijn indices to represent variables, based on Kiselyov's representation of STLC in Haskell [Kiselyov 2012]. Whenever `mix` specializes a self-interpreter other than Mogensen's, we have two representations involved: `mix` itself always operates on representations produced by $\overline{\cdot}$.

The following two theorems prove that a partial evaluator based on specialization-safe normalization is Jones-optimal for Mogensen's Church-encoding self-interpreter (Figure 4) and $time_{cbv}$.

LEMMA 5.3. *For any closed term* $\mathtt{e}$, $\mathtt{u}\ \overline{\mathtt{e}} \Longrightarrow_s \mathtt{e}$.

THEOREM 5.4. `mix` *is Jones optimal for* $time_{cbv}$ *and Mogensen's self-interpreter.*

PROOF. Follows from Lemma 5.2 and Lemma 5.3. □

Mogensen-Scott encoding. Quotation for a Mogensen-Scott-encoded representation is defined in Figure 12. It's a higher-order abstract syntax (HOAS) representation like Mogensen's Church-encoded representation. The key difference between the two representations is that while Church-encoding defines a fold over the syntax of the term, Mogensen-Scott-encoding defines pattern matching on the term. The two representations are isomorphic, but each encoding technique is well-suited to particular applications. Church-encoding is good for folds, while Mogensen-Scott is preferable for operations that aren't easily programmed as folds. Programming a fold like our self-interpreter takes a bit more work on a Mogensen-Scott representation. In particular, we have to use a fixpoint combinator to traverse the term. Still, this extra work can be done using only specialization-safe reduction, so our partial evaluator is Jones-optimal for the Mogensen-Scott interpreter as well.

LEMMA 5.5. *For any term* $\mathtt{e}$, $\mathtt{msint}\ \widehat{\mathtt{e}} \Longrightarrow_s \mathtt{e}$.

THEOREM 5.6. `mix` *is Jones-optimal for* $time_{cbv}$ *and* `msint`.

PROOF. Follows from Lemma 5.2 and Lemma 5.5. □

$$\frac{}{\Gamma,\mathtt{x} \vdash \mathtt{x} \blacktriangleright \mathtt{fst}} \qquad \frac{\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{e}}{\Gamma,\mathtt{y} \vdash \mathtt{x} \blacktriangleright (\lambda \mathtt{p}.\ \mathtt{e}\ (\mathtt{snd}\ \mathtt{p}))}$$

$$\frac{\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{e}}{\Gamma \vdash \mathtt{x} \rhd \mathtt{var}\ \mathtt{e}} \qquad \frac{\Gamma,\mathtt{x} \vdash \mathtt{e} \rhd \mathtt{q}}{\Gamma \vdash (\lambda \mathtt{x}.\mathtt{e}) \rhd \mathtt{abs}\ \mathtt{q}} \qquad \frac{\Gamma \vdash \mathtt{e1} \rhd \mathtt{q1} \qquad \Gamma \vdash \mathtt{e2} \rhd \mathtt{q2}}{\Gamma \vdash \mathtt{e1}\ \mathtt{e2} \rhd \mathtt{app}\ \mathtt{q1}\ \mathtt{q2}} \qquad \frac{\langle\rangle \vdash \mathtt{e} \rhd \mathtt{q}}{\widehat{\mathtt{e}} = \lambda \mathtt{var}.\ \lambda \mathtt{abs}.\ \lambda \mathtt{app}.\ \mathtt{q}}$$

```
dbVar = (λf.λe. f e)
dbAbs = (λb.λe.λx.b (λf. f x e))
dbApp = (λa.λb.λe.a e (b e))
dbint = (λq. q dbVar dbAbs dbApp (λx.x))
```

Fig. 13. Tagless-final encoding with deBruijn indices.

deBruijn indices. Figure 13 shows a tagless-final representation that uses deBruijn indices to represent variables, unlike the two previous HOAS representations. It's based on Kiselyov's representation of simply-typed λ-calculus in Haskell [Kiselyov 2012], which uses nested pairs to represent environments, and composed projection functions to represent deBruijn indices. For example, the deBruijn index 0 is represented as the `fst`, index 1 is represented as `λp. fst (snd p)`, and so on. The quoter uses a function $\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{e}$ to construct the projection function for variables. The self-interpreter for this representation is quite different than the ones for either the tagless-final or the Mogensen-Scott HOAS, but our partial evaluator is Jones optimal for it as well.

Lemma 5.7. *For any closed term* `e`, $\mathtt{dbint}\ \widehat{\mathtt{e}} \Longrightarrow_s \mathtt{e}$.

Theorem 5.8. `mix` *is Jones-optimal for* $time_{cbv}$ *and* `dbint`.

Proof. Follows from Lemma 5.2 and Lemma 5.7. □

Discussion. We now demonstate why each of the three specialization-safe β-reductions — rules SSR-1, SSR-4, and SSR-5 — are needed to achieve Jones-optimality for our three self-interpreters.

First, consider how the Church-encoding interpreter (Figure 4) operates on the representation of `(λa. a a)`. We show the affine variable annotations for clarity.

$$\begin{aligned}
&\mathtt{u}\ \overline{(\lambda \mathtt{a}.\ \mathtt{a}\ \mathtt{a})}\\
&= (\lambda \mathtt{e}^\circ.\ \mathtt{e}^\circ\ (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ)\ (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ))\ (\lambda \mathtt{abs}.\ \lambda \mathtt{app}.\ \mathtt{abs}\ (\lambda \mathtt{a}.\ \mathtt{app}\ \mathtt{a}\ \mathtt{a}))\\
&\to_s (\lambda \mathtt{abs}.\ \lambda \mathtt{app}.\ \mathtt{abs}\ (\lambda \mathtt{a}.\ \mathtt{app}\ \mathtt{a}\ \mathtt{a}))\ (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ)\ (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ)\\
&\to_s^2 (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ)\ (\lambda \mathtt{a}.\ (\lambda \mathtt{x}^\circ.\mathtt{x}^\circ)\ \mathtt{a}\ \mathtt{a})\\
&\to_s^2 (\lambda \mathtt{a}.\ \mathtt{a}\ \mathtt{a})
\end{aligned}$$

The first step is derived by SSR-4, since `e` is affine. The next two steps are derived by by SSR-1 because `abs` and `app` are not affine and their parameters are values. The last two steps are derived by by SSR-4 because the two `x`s are affine.

To demonstrate the need for SSR-5, consider how the Mogensen-Scott interpreter (Figure 12) operates on the representation of `(λa. a a)`:

$$\begin{aligned} & \texttt{msint}\ (\widehat{\lambda\texttt{a}.\ \texttt{a a}}) \\ \to_s^* & \ \lambda\texttt{x}.\ \texttt{msint}\ ((\lambda\texttt{a}.\ \widehat{\texttt{a a}})\ ((\lambda\texttt{y}^\circ.\texttt{x})\texttt{x})) \\ \to_s & \ \lambda\texttt{x}.\ \texttt{msint}\ ((\lambda\texttt{a}.\ \widehat{\texttt{a a}})\ \texttt{x}) \\ \to_s & \ \lambda\texttt{x}.\ \texttt{msint}\ (\widehat{\texttt{x x}}) \\ \to_s^* & \ \lambda\texttt{x}.\ \texttt{x x} \end{aligned}$$

The first sequence of steps does not need SSR-5. The term $((\lambda\texttt{y}^\circ.\texttt{x})\texttt{x})$ ensures that x is unlimited, and reduces x in the next step, which is derived by SSR-4 because y is affine. The next step is not reducible by SSR-1 because x is a variable (and variables are not values), and is also not reducible by SSR-4 because a is not affine. However, it is reducible by SSR-5 because we forced x to be unlimited. This step allows the interpreter to proceed and eventually return the original term (up to renaming).

6 TYPED SELF-APPLICABLE PARTIAL EVALUATION FOR $F_\omega^{\mu i}$

We implement a typed version of our partial evaluator for $F_\omega^{\mu i}$, a Turing-complete typed λ-calculus with support for typed self-representation. The language is defined in Figure 14. It extends System F_ω with iso-recursive types and intensional type functions, which together are useful for typed meta-programming. It is type-safe and type checking is decidable. For more details, we refer the interested reader to Brown and Palsberg [Brown and Palsberg 2017].

Our self-representation type Exp is defined in Figure 15. It is a tagless-final style representation – intuitively, a typed version of Mogensen's Church-encoded representation. It is also Parametric Higher-Order Abstract Syntax (PHOAS) [Chlipala 2008; Washburn and Weirich 2003]. In particular, the type PExp is parametric in V, which determines the type of free variables in a representation. Intuitively, PExp can be understood as the type of representations that may contain free variables. The type Exp quantifies over V, which ensures that the representation is closed. The self-representation of $F_\omega^{\mu i}$ presented by Brown and Palsberg was a typed Mogensen-Scott representation – intuitively, a Generalized Algebraic Data Type (GADT) encoded using λ-terms. We instead use the tagless-final style because it supports a normalizing self-interpreter and a normalizing self-applicable partial evaluator, allowing us to generate the Futamura projections.

Quotation for our $F_\omega^{\mu i}$ representation is defined in Figure 16. The quotation function $\overline{\cdot}$ is defined only on closed terms, and depends on a pre-quotation function $\triangleright$ from type derivations to terms. In the judgment $\Gamma \vdash \texttt{e} : \texttt{T} \triangleright \texttt{q}$, the input is the type derivation $\Gamma \vdash \texttt{e} : \texttt{T}$, and the output is a term q. We call q the pre-representation of e. In addition to the cases for variables, λ-abstractions and applications, we have cases for type abstraction and application and the folding and unfolding of iso-recursive types. The case of type abstractions and type applications use the IsAll T proof terms, which prove that the type T is a quantified type. They also use utility functions $\texttt{stripAll}_\texttt{K}$, $\texttt{underAll}_{\texttt{X,K,T}}$, and $\texttt{inst}_{\texttt{X,K,T,S}}$, which are useful to meta-programs for operating on type abstractions and applications. These IsAll proofs and utility functions are part of Brown and Palsberg's extensional representation of polymorphism.

We prove two theorems about our representation: first, every closed and well-typed term has a representation that is closed and well-typed, and the type of a term determines the type of its representation. Ill-typed terms cannot be represented. Second, representations are β-normal forms.

Theorem 6.1. *If* $\langle\rangle \vdash \texttt{e} : \texttt{T}$, *then* $\langle\rangle \vdash \overline{\texttt{e}} : \texttt{Exp T}$.

Theorem 6.2. *If* $\langle\rangle \vdash \texttt{e} : \texttt{T}$, *then* $\overline{\texttt{e}}$ *is* β*-normal.*

$$
\begin{array}{rlllll}
\text{(kinds)} & \mathsf{K} ::= \ * & \mid \mathsf{K_1} \to \mathsf{K_2} & & & \\
\text{(types)} & \mathsf{T} ::= \ \mathsf{X} & \mid \mathsf{T_1} \to \mathsf{T_2} & \mid \forall\mathsf{X{:}K.T} & \mid \lambda\mathsf{X{:}K.T} \mid \mathsf{T_1\ T_2} \mid & \mid \mathsf{Typecase} \\
\text{(terms)} & \mathsf{e} ::= \ \mathsf{x} & \mid \lambda\mathsf{x{:}T.e} & \mid \mathsf{e_1\ e_2} & \mid \Lambda\mathsf{X{:}K.e} \mid \mathsf{e\ T} & \mid \mathsf{fold\ T_1\ T_2\ e} \mid \mathsf{unfold\ T_1\ T_2\ e} \\
\text{(environments)} & \Gamma ::= \ \langle\rangle & \mid \Gamma,(\mathsf{x{:}T}) & \mid \Gamma,(\mathsf{X{:}K}) & & \\
 & & & & & \\
\text{(normal form terms)} & \mathsf{v} ::= \mathsf{n} \mid (\lambda\mathsf{x{:}T.v}) & \mid (\Lambda\mathsf{X{:}K.v}) & \mid \mathsf{fold\ T_1\ T_2\ v} & & \\
\text{(neutral terms)} & \mathsf{n} ::= \mathsf{x} \mid \mathsf{n\ v} & \mid \mathsf{n\ T} & \mid \mathsf{unfold\ T_1\ T_2\ n} & &
\end{array}
$$

Grammar

$$\frac{(\mathsf{X{:}K}) \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{K}}$$

$$\frac{\Gamma \vdash \mathsf{T_1} : * \qquad \Gamma \vdash \mathsf{T_2} : *}{\Gamma \vdash \mathsf{T_1} \to \mathsf{T_2} : *} \qquad \frac{\Gamma,(\mathsf{X{:}K}) \vdash \mathsf{T} : *}{\Gamma \vdash (\forall\mathsf{X{:}K.T}) : *}$$

$$\frac{\Gamma,(\mathsf{X{:}K_1}) \vdash \mathsf{T} : \mathsf{K_2}}{\Gamma \vdash (\lambda\mathsf{X{:}K_1.T}) : \mathsf{K_1} \to \mathsf{K_2}} \qquad \frac{\Gamma \vdash \mathsf{T_1} : \mathsf{K_2} \to \mathsf{K} \qquad \Gamma \vdash \mathsf{T_2} : \mathsf{K_2}}{\Gamma \vdash \mathsf{T_1\ T_2} : \mathsf{K}}$$

$$\Gamma \vdash \mu : ((* \to *) \to * \to *) \to * \to *$$

$$
\begin{array}{ll}
\Gamma \vdash \mathsf{Typecase} : & (* \to * \to *) \to \\
& (* \to *) \to (* \to *) \to \\
& (((* \to *) \to * \to *) \to * \to *) \to \\
& *
\end{array}
$$

Type Formation

$$\frac{(\mathsf{x{:}T}) \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}}$$

$$\frac{\Gamma \vdash \mathsf{T_1} : * \qquad \Gamma,(\mathsf{x{:}T_1}) \vdash \mathsf{e} : \mathsf{T_2}}{\Gamma \vdash (\lambda\mathsf{x{:}T_1.e}) : \mathsf{T_1} \to \mathsf{T_2}}$$

$$\frac{\Gamma \vdash \mathsf{e_1} : \mathsf{T_2} \to \mathsf{T} \qquad \Gamma \vdash \mathsf{e_2} : \mathsf{T_2}}{\Gamma \vdash \mathsf{e_1\ e_2} : \mathsf{T}}$$

$$\frac{\Gamma,(\mathsf{X{:}K}) \vdash \mathsf{e} : \mathsf{T}}{\Gamma \vdash (\Lambda\mathsf{X{:}K.e}) : (\forall\mathsf{X{:}K.T})}$$

$$\frac{\Gamma \vdash \mathsf{e} : (\forall\mathsf{X{:}K.T_1}) \qquad \Gamma \vdash \mathsf{T_2} : \mathsf{K}}{\Gamma \vdash \mathsf{e} : \mathsf{T_1}[\mathsf{X}{:=}\mathsf{T_2}]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathsf{T} : * \\ \Gamma \vdash \mathsf{e} : \mathsf{F}\ (\mu\ \mathsf{F})\ \mathsf{T}\end{array}}{\Gamma \vdash \mathsf{fold\ F\ T\ e} : \ \mathsf{F\ T}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{F} : (* \to *) \to * \to * \qquad \Gamma \vdash \mathsf{T} : * \\ \Gamma \vdash \mathsf{e} : \mu\ \mathsf{F\ T}\end{array}}{\Gamma \vdash \mathsf{unfold\ F\ T\ e} : \mathsf{F}\ (\mu\ \mathsf{F})\ \mathsf{T}}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{T_1} \qquad \mathsf{T_1} \equiv \mathsf{T_2} \qquad \Gamma \vdash \mathsf{T_2} : *}{\Gamma \vdash \mathsf{e} : \mathsf{T_2}}$$

Term Formation

$$\mathsf{T} \equiv \mathsf{T} \qquad \frac{\mathsf{T1} \equiv \mathsf{T2}}{\mathsf{T2} \equiv \mathsf{T1}} \qquad \frac{\mathsf{T1} \equiv \mathsf{T2} \qquad \mathsf{T2} \equiv \mathsf{T3}}{\mathsf{T1} \equiv \mathsf{T3}}$$

$$\frac{\mathsf{T1} \equiv \mathsf{T1'} \qquad \mathsf{T2} \equiv \mathsf{T2'}}{\mathsf{T1} \to \mathsf{T2} \equiv \mathsf{T1'} \to \mathsf{T2'}} \qquad \frac{\mathsf{T} \equiv \mathsf{T'}}{(\forall\mathsf{X{:}K.T}) \equiv (\forall\mathsf{X{:}K.T'})}$$

$$\frac{\mathsf{T} \equiv \mathsf{T'}}{(\lambda\mathsf{X{:}K.T}) \equiv (\lambda\mathsf{X{:}K.T'})} \qquad \frac{\mathsf{T1} \equiv \mathsf{T1'} \qquad \mathsf{T2} \equiv \mathsf{T2'}}{\mathsf{T1\ T2} \equiv \mathsf{T1'\ T2'}}$$

$$
\begin{array}{c}
(\lambda\mathsf{X{:}K.T1})\ \mathsf{T2} \equiv (\mathsf{T1}[\mathsf{X} := \mathsf{T2}]) \\
(\forall\mathsf{X{:}K.T2}) \equiv (\forall\mathsf{X'{:}K.T2}[\mathsf{X} := \mathsf{X'}]) \\
(\lambda\mathsf{X{:}K.T}) \equiv (\lambda\mathsf{X'{:}K.T}[\mathsf{X} := \mathsf{X'}])
\end{array}
$$

$$
\begin{array}{c}
\mathsf{Typecase\ F1\ F2\ F3\ F4\ (T1 \to T2)} \equiv \mathsf{F1\ T1\ T2} \\
\mathsf{Typecase\ F1\ F2\ F3\ F4}\ (\mu\ \mathsf{T1\ T2}) \equiv \mathsf{F4\ T1\ T2}
\end{array}
$$

$$\frac{\mathsf{X} \notin \mathsf{FV(F3)}}{\mathsf{Typecase\ F1\ F2\ F3\ F4}\ (\forall\mathsf{X{:}K.T}) \equiv \mathsf{F2}\ (\forall\mathsf{X{:}K.\ F3\ T})}$$

Type Equivalence

$$
\begin{array}{c}
(\lambda\mathsf{x{:}T.e})\ \mathsf{e_1} \longrightarrow \mathsf{e}[\mathsf{x} := \mathsf{e_1}] \\
(\Lambda\mathsf{X{:}K.e})\ \mathsf{T} \longrightarrow \mathsf{e}[\mathsf{X} := \mathsf{T}] \\
\mathsf{unfold\ F\ T\ (fold\ F'\ T'\ e)} \longrightarrow \mathsf{e}
\end{array}
$$

$$\frac{\mathsf{e_1} \longrightarrow \mathsf{e_2}}{\begin{array}{c}\mathsf{e_1\ e_3} \longrightarrow \mathsf{e_2\ e_3} \\ \mathsf{e_3\ e_1} \longrightarrow \mathsf{e_3\ e_2} \\ \mathsf{e_1\ T} \longrightarrow \mathsf{e_2\ T} \\ (\lambda\mathsf{x{:}T.e_1}) \longrightarrow (\lambda\mathsf{x{:}T.e_2}) \\ (\Lambda\mathsf{X{:}K.e_1}) \longrightarrow (\Lambda\mathsf{X{:}K.e_2}) \\ \mathsf{fold\ F\ T\ e_1} \longrightarrow \mathsf{fold\ F\ T\ e_2} \\ \mathsf{unfold\ F\ T\ e_1} \longrightarrow \mathsf{unfold\ F\ T\ e_2}\end{array}}$$

Reduction

Fig. 14. Definition of $F_\omega^{\mu i}$

The self-interpreter unquote for the tagless-final HOAS representation is shown in Figure 17. We prove that our partial evaluator is Jones-optimal for unquote and a *time*(−) function based on call-by-value type-erasure semantics.

Definition 6.1 (Type erasure). The type erasure of an $F_\omega^{\mu i}$ term e is defined recursively as follows:

```
decl PExp : (∗ → ∗) → ∗ → ∗ =
  λV : ∗ → ∗. λA:∗.
  (∀S:∗. ∀T:∗. (V S → V T) → V (S → T)) →
  (∀A:∗. ∀B:∗. V (A → B) → V A → V B) →
  (∀A:∗. IsAll A → StripAll A → UnderAll A → All Id V A → V A) →
  (∀A:∗. ∀B:∗. IsAll A → Inst A B → V A → V B) →
  (∀F : (∗ → ∗) → ∗ → ∗. ∀B : ∗. V (F (μ F) B) → V (μ F B)) →
  (∀F : (∗ → ∗) → ∗ → ∗. ∀B : ∗. V (μ F B) → V (F (μ F) B)) →
  V A

decl Exp : ∗ → ∗ = λA:∗. ∀V:∗ → ∗. PExp V A

decl Abs  V = ∀A1:∗. ∀A2:∗. (V A1 → V A2) → V (A1 → A2)
decl App  V = ∀A:∗. ∀B:∗. V (A → B) → V A → V B
decl TAbs V =
  ∀A:∗. IsAll A → StripAll A → UnderAll A → All Id V A → V A
decl TApp V = ∀A:∗. ∀B:∗. IsAll A → Inst A B → V A → V B
decl Fold V =
  ∀F : (∗→∗) → ∗ → ∗. ∀B:∗. V (F (μ F) B) → V (μ F B)
decl Unfold V =
  ∀F : (∗→∗) → ∗ → ∗. ∀B : ∗. V (μ F B) → V (F (μ F) B)
```

Fig. 15. Definitions of PExp and Exp

1. $te(\mathtt{x}) = \mathtt{x}$
2. $te(\lambda\mathtt{x}{:}\mathtt{T}.\ \mathtt{e}) = \lambda\mathtt{x}.\ te(\mathtt{e})$
3. $te(\mathtt{e}_1\ \mathtt{e}_2) = te(\mathtt{e}_1)\ te(\mathtt{e}_2)$
4. $te(\Lambda\mathtt{X}{:}\mathtt{K}.\mathtt{e}) = te(\mathtt{e})$
5. $te(\mathtt{e}\ \mathtt{T}) = te(\mathtt{e})$
6. $te(\mathtt{fold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}) = te(\mathtt{e})$
7. $te(\mathtt{unfold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}) = te(\mathtt{e})$

Definition 6.2. $time^{te}_{cbv}(\mathtt{e}) = n$ if and only if $te(\mathtt{e}) \to^n \mathtt{v}$.

LEMMA 6.3. *If* $\langle\rangle \vdash \mathtt{e} : \mathtt{T}$, *then* $te(\mathtt{unquote}\ \overline{\mathtt{e}}) \Longrightarrow_s te(\mathtt{e})$.

THEOREM 6.4. mix *is Jones-optimal for* $time^{te}_{cbv}$ *and* unquote/$\overline{\cdot}$.

6.1 Implementation of mix

Our partial evaluator mix is implemented in two steps: first, we analyze the program and input for affine variable annotations, then we reduce the annotated term to specialization-safe normal form. Each of these steps is programmed by folds over the representation. To infer affine variable annotations, we do a fold that returns for each subterm 1) a tree data structure that indicates the annotation of each bound variable in the subterm, 2) how many times each free variable occurs, and 3) which free variables occur under a λ-abstraction. Items 2 and 3 are used to determine the annotation of each variable at its binding λ-abstraction. A second fold uses the annotation tree to annotate the term. A third and final fold reduces the annotated term to specialization-safe normal form. This step is based on modification of a Mogensen's partial evaluator that reduces to β-normal form [Mogensen 1995], which was also used as the basis for a partial evaluator by Carette et al. [Carette et al. 2009]. Our modifications check whether each β-redex is specialization-safe by

$$\frac{(\mathtt{x} : \mathtt{T}) \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \rhd \mathtt{x}}$$

$$\frac{\Gamma \vdash \mathtt{T1} : * \qquad \Gamma,(\mathtt{x}{:}\mathtt{T1}) \vdash \mathtt{e} : \mathtt{T2} \rhd \mathtt{q}}{\Gamma \vdash (\lambda \mathtt{x}{:}\mathtt{T1.e}) : \mathtt{T1} \to \mathtt{T2} \rhd \mathtt{abs\ T1\ T2}\ (\lambda \mathtt{x}{:}\mathtt{V\ T1.\ q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T} \rhd \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \rhd \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{T} \rhd \mathtt{app}\ \mathtt{T}_2\ \mathtt{T}\ \mathtt{q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma,(\mathtt{X}{:}\mathtt{K}) \vdash \mathtt{e} : \mathtt{T} \rhd \mathtt{q} \qquad \begin{array}{l}\mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p}\\ \mathtt{stripAll}_{\mathtt{K}} = \mathtt{s}\\ \mathtt{underAll}_{\mathtt{X,K,T}} = \mathtt{u}\end{array}}{\Gamma \vdash (\Lambda \mathtt{X}{:}\mathtt{K.e}) : (\forall \mathtt{X}{:}\mathtt{K.T}) \rhd \mathtt{tabs}\ (\forall \mathtt{X}{:}\mathtt{K.T})\ \mathtt{p\ s\ u}\ (\Lambda \mathtt{X}{:}\mathtt{K.q})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : (\forall \mathtt{X}{:}\mathtt{K.T}) \rhd \mathtt{q}\\ \Gamma \vdash \mathtt{A} : \mathtt{K}\end{array} \qquad \begin{array}{l}\mathtt{isAll}_{\mathtt{X,K,T}} = \mathtt{p}\\ \mathtt{inst}_{\mathtt{X,K,T,A}} = \mathtt{i}\end{array}}{\Gamma \vdash \mathtt{e\ A} : \mathtt{T[X{:=}A]} \rhd \mathtt{tapp}\ (\forall \mathtt{X}{:}\mathtt{K.T})\ (\mathtt{T[X{:=}A]})\ \mathtt{p\ i\ q}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{F} : (* \to *) \to * \to *\\ \qquad \Gamma \vdash \mathtt{T} : * \qquad\qquad \Gamma \vdash \mathtt{e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \rhd \mathtt{q}\end{array}}{\Gamma \vdash \mathtt{fold\ F\ T\ e} : \mu\ \mathtt{F\ T} \rhd \mathtt{fld\ F\ T\ q}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{F} : (* \to *) \to * \to *\\ \qquad \Gamma \vdash \mathtt{T} : * \qquad\qquad \Gamma \vdash \mathtt{e} : \mu\ \mathtt{F\ T} \rhd \mathtt{q}\end{array}}{\Gamma \vdash \mathtt{unfold\ F\ T\ e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \rhd \mathtt{unfld\ F\ T\ q}}$$

$$\frac{\langle\rangle \vdash \mathtt{e} : \mathtt{T} \rhd \mathtt{q}}{\begin{array}{l}\overline{\mathtt{e}} = \Lambda \mathtt{V}{:}* \to *.\\ \quad \lambda\mathtt{abs} : \mathtt{Abs\ V}.\ \lambda\mathtt{app} : \mathtt{App\ V}.\\ \quad \lambda\mathtt{tabs} : \mathtt{TAbs\ V}.\ \lambda\mathtt{tapp} : \mathtt{TApp\ V}.\\ \quad \lambda\mathtt{fld} : \mathtt{Fold\ V}.\ \lambda\mathtt{unfld} : \mathtt{Unfold\ V}.\\ \quad \mathtt{q}\end{array}}$$

Fig. 16. Tagless-final quotation and pre-quotation of $F_\omega^{\mu i}$

inspecting the affine variable annotation of the λ-abstraction and the form of the argument. If it's a value or an unlimited variable, the β-redex is specialization-safe. If it's an application or an affine variable, the β-redex is not specialization-safe and is residualized.

Figure 18 shows some highlights of our Jones-optimal partial evaluator.

At its core is the function `ssnorm`, which reduces annotated representations of terms to (unannotated) representations of their specialization-safe normal form. The function `sem` is a fold that converts an `Exp A` to a semantic object of type `Sem V A`, and `reify` extracts a representation of the evaluated term from the semantic object. The semantic object is a pair containing the specialization-safe normal form of the term and a helper function. `sem` traverses the term and creates a semantic object for each subterm. When a subterm occurs in head position of a specialization-safe redex, its helper function is used to compute the reduct.

At the bottom of the figure is `mix` itself. It first builds `e`, a representation of the program applied to the input. Then it uses `forceExp` to check that the input `x` is available, in which case `forceExp` returns `e` to be annotated and normalized. This check is an important optimization for self-application, specifically when specializing `mix` to a single input. It prevents a blow up in the output code size caused by many copies of the traversal functions from `annotate` and `sem` being residualized thoughout `f`.

```
decl Id : * → * = λA:*. A;

decl unAbs   : Abs Id   =
  ΛA:*. ΛB:*. λf : A → B. f;
decl unApp   : App Id   =
  ΛA:*. ΛB:*. λf : A → B. f;
decl unTAbs : TAbs Id =
  ΛA:*. λp:IsAll A. λs:StripAll A. λu:UnderAll A. λe:All Id Id A.
  unAll A p Id Id e;
decl unTApp : TApp Id =
  ΛA:*. ΛB:*. λp : IsAll A. λi : Inst A B. λx : A.
  i Id (sym (All Id Id A) A (unAll A p Id) Id x);
decl unFold : Fold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : F (μ F) A. fold F A x;
decl unUnfold : Unfold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : μ F A. unfold F A x;

decl unquote : ∀(A:*. Exp A → A) =
  ΛA:*. λe:Exp A. e Id unAbs unApp unTAbs unTApp unFold unUnfold;
```

Fig. 17. The $F_\omega^{\mu i}$ tagless-final HOAS self-interpreter unquote.

The type Sem of semantic objects is shown in Figure 19. It is an iso-recursive intensional type function, combining two features of $F_\omega^{\mu i}$: iso-recursive types and intensional type functions. The type Sem V A is equivalent to (SemF V) A and isomorphic (via (un)folding the recursion) to SemF (Sem V) A.

There are two variants of semantic object: neutral objects that do not form redexes (specialization-safe or otherwise) when in head-position, and non-neutral objects that do. Neutral objects contain a boolean and a representation. The representation is used to `reify` the semantic object. The boolean is `true` if that representation is of a term that type-erases to an unlimited variable. If so, then any redex in which this term is in argument position is specialization-safe.

Non-neutral objects form redexes when in head-position. Their first two components are similar to those of neutral objects. The boolean is `true` if the representation erases to a λ-abstraction. The third component is used to compute the reduct of redexes with this object in head-position. These have types of the form SemF1 (Sem V) A. SemF1 is an intensional type function that depends on the structure of A:

```
   SemF1 (Sem V) (A → B) ≡ Pair Bool (Sem V A → Sem V B)
SemF1 (Sem V) (∀X:K.T) ≡ ∀X:K. Sem V T
     SemF1 (Sem V) ( F A) ≡ Sem V (F ( F) A)
```

A non-neutral semantic object of arrow type is a λ-abstraction. For these, the third component is a pair of boolean and function from semantic objects to semantic objects. The boolean indicates whether the λ-abstraction is affine. When this semantic object is in head position of a redex, the function is applied to the semantic object of the argument to compute the semantic object of the reduct.

If a non-neutral semantic object has a quantified type, then it is a Λ-abstraction. For these, the third component is a polymorphic semantic object. If a non-neutral semantic object has a recursive type, it is for a `fold` term `fold` F A e, and the third component is the semantic object of the unfolded term e.

```
annotate : (∀V:* → *. ∀A : *. Exp A → ExpAnn Bool A)
sem      : (∀V:* → *. ∀A : *. ExpAnn Bool A → Sem V A)
reify    : (∀V:* → *. ∀A : *. Sem V A → PExp V A)
forceExp : (∀T:*. Exp T → (∀A:*. A → A))

decl ssnorm : (∀A : *. ExpAnn Bool A → Exp A) =
  ΛA:*. λe : Exp A. ΛV:* → *.
  reify V A (sem V A e);

decl mix : (∀A : *. ∀B : *. Exp (A → B) → Exp A → Exp B) =
  ΛA:*.ΛB:*. λf:Exp (A → B). λx:Exp A.
  let e : Exp B = (ΛV:* → *. app V A B (f V) (x V)) in
  ssnorm B (annotate B (forceExp A x (Exp B) e));
```

Fig. 18. Highlights of our Jones-optimal partial evaluator mix

```
decl SemF1 : (* → *) → * → * =
  λSem : * → *. λA:*.
  Typecase
    (λA1:*. λA2:*. Pair Bool (Sem A1 → Sem A2))
    Id Sem
    (λF : (* → *) → * → *. λB : *.
     Sem (F (μ F) B))
    A
;

decl SemF : (* → *) → (* → *) → * → * =
  λV : * → *. λSem : * → *. λA : *.
  Triple Bool (PExp V A) (Maybe (SemF1 Sem A))
;

decl Sem : (* → *) → * → * = λV : * → *. μ (SemF V)
```

Fig. 19. Internal types of our Jones-optimal partial evaluator.

Figure 20 shows the function semApp, the part of sem that constructs semantic objects for applications. Given semantic objects for a function f and an argument x, semApp first unpacks f and binds the names fValueOrUnlimited, rep_f, and msem to its components. fValueOrUnlimited is not used, rep_f is the representation of the specialization-safe normal form of f, and msem indicates whether f is neutral or non-neutral. If f is neutral, then the application is not a redex and is residualized. Otherwise, the application is a β-redex. To check if it's specialization-safe, we test whether f's bound variable is affine, or x is either a value or an unlimited variable. If either case is true, the β-redex is specialization-safe and we reduce. Otherwise, we residualize. Whenever we residualize, we use semNe to construct a semantic object for the (neutral) residual term. The argument false to semNe indicates that result is not a value or an unlimited variable.

6.2 Type-checking the Futamura Projections

Figure 21 shows typed versions of the Futamura projections in the concrete syntax of $F_\omega^{\mu i}$. Quotations are denoted using square brackets [-] rather than overlines, and so double-quotations

```
decl semApp : (∀V:*→*.∀B:*.∀A:*. Sem V (B → A) → Sem V B → Sem V A) =
  ΛV:* → *. ΛB:*. ΛA:*. λf:Sem V (B → A). λx:Sem V B.
  unfold (SemF V) (B → A) f
    (Sem V A)
    (λfValueOrUnlimited : Bool.
     λrep_f : PExp V (B → A)).
     λmsem : Maybe (Pair Bool (Sem V B → Sem V A)).
     msem (Sem V A)
       -- neutral
       (semNe V A false (app V B A rep_f (reify V B x)))
       -- not neutral
       (λp : Pair Bool (Sem V B → Sem V A).
        let affine : Bool = fst Bool (Sem V B → Sem V A) p in
        let sem_f : Sem V B → Sem V A = snd Bool (Sem V B → Sem V A) p in
        or affine (isValueOrUnlimited V B x)
          (Sem V A)
          -- redex is specialization-safe: reduce
          (sem_f x)
          -- redex is not safe: residualize
          (semNe V A false (app V B A rep_f (reify V B x)))))
```

Fig. 20. A key function from sem.

are denoted using nested square brackets `[[-]]` rather than stacked overlines $\overline{\overline{\cdot}}$. The first projection compiles the factorial function `fact` by specializing the self-interpreter `unquote` to it. The second projection generates a compiler by specializing `mix` to `unquote`. The third projection generates a compiler-generator by specializing `mix` to itself. The fourth projection generates the self-generating compiler-generator. It is also obtained by specializing `mix` to itself, but at different types.

This is the first time the Futamura projections have been type-checked for a partial evaluator that operates on typed representations. One consequence of this is that we can clearly see what the types of generated programs are. In particular, the type of each generated program is determined by the type of its representation.

The compiled factorial function `fact_compiled` has the expected type `Nat → Nat`. The generated compiler `compile` has the polymorphic type `∀A:*. Exp (Exp A) → Exp A`. It compiles programs by mapping double representations to representations (like the First futamura projection does). The generated compiler generator cogen has the polymorphic type

```
∀A:*. ∀B:*. Exp (Exp (A → B)) → Exp (Exp A → Exp B).
```

If we instantiate `A` to `Exp C` and `B` to `C`, we can derive the type:

```
∀C:*. Exp (Exp (Exp C → C)) → Exp (Exp (Exp C) → Exp C).
```

Then we can apply `cogen` to the (double) representation of a self-interpreter like `unquote` and generate a compiler. The self-generating compiler generator `selfgen` has the polymorphic type:

```
∀A:*. ∀B:*.
Exp (Exp (Exp (A → B) → Exp A → Exp B)) →
Exp (Exp (Exp (A → B)) → Exp (Exp A → Exp B))
```

We can apply `selfgen` to the (double) representation of `mix`, and generate `cogen` again.

```
decl fact_compiled : Exp (Nat → Nat) =
  mix (Exp (Nat → Nat)) (Nat → Nat)
    [unquote (Nat → Nat)]
    [[fact]]

decl compile : ∀A:*. Exp (Exp (Exp A) → Exp A) =
  ΛA:*.
  mix (Exp (Exp A → A)) (Exp (Exp A) → Exp A)
    [mix (Exp A) A]
    [[unquote A]]

decl cogen : (∀A:*. ∀B:*. Exp (Exp (Exp (A → B)) →
                                Exp (Exp A → Exp B))) =
  ΛA:*. ΛB:*.
  mix (Exp (Exp (A → B) → Exp A → Exp B))
      (Exp (Exp (A → B)) → Exp (Exp A → Exp B))
      [mix (Exp (A → B)) (Exp A → Exp B)]
      [[mix A B]]

decl selfgen : (∀A:*. ∀B:*.
                  Exp (Exp (Exp (Exp (A → B) → Exp A → Exp B)) →
                       Exp (Exp (Exp (A → B)) →
                            Exp (Exp A → Exp B)))) =
  ΛA:*. ΛB:*.
  mix (Exp (Exp (Exp (A → B) → Exp A → Exp B) →
       Exp (Exp (A → B)) →
       Exp (Exp A → Exp B)))
      (Exp (Exp (Exp (A → B) → Exp A → Exp B)) →
       Exp (Exp (Exp (A → B)) → Exp (Exp A → Exp B)))
    [mix (Exp (Exp (A → B) → Exp A → Exp B))
         (Exp (Exp (A → B)) → Exp (Exp A → Exp B))]
    [[mix (Exp (A → B)) (Exp A → Exp B)]]
```

Fig. 21. The four typed Futamura projections

7 EXPERIMENTAL RESULTS

We implemented $F_{\omega}^{\mu i}$ in Haskell, including a parser, a type checker, a β-equivalence checker, and three different *time* measures. We tested that our partial evaluator type checks and the Futamura projections type check with the types given in Figure 21. We used our β-equivalence checker to verify that the Futamura projections meet their specification. In addition to formally proving that our partial evaluator is Jones-optimal for call-by-value reduction, our experiments demonstrate Jones-optimality for call-by-value, normal-order, and memoized normal-order reduction. We also demonstrate that specialization by β-normalization is Jones-optimal for normal-order reduction, but not for call-by-value or memoized normal-order.

Reduction strategies. We measure call-by-value steps to a value, and normal-order and memoized normal-order steps to a β-normal form. Normal-order reduction is related to call-by-name reduction, but evaluates to a β-normal form instead of a value. Counting call-by-name reduction steps to a value is not a useful measure, because many of our benchmark programs reduce to a value in only a few steps of call-by-name reduction, obscuring the effect of specialization on the

Interpreter	**Time**	**Slowdown**	**Overhead**
Tagless-final PHOAS	Call-by-value	10.0	900%
	Normal order	10.9	990%
	Memoized normal order	9.9	890%
Mogensen-Scott PHOAS	Call-by-value	105.3	10430%
	Normal order	136.5	13550%
	Memoized normal order	68.8	6780%
Tagless-final deBruijn	Call-by-value	24.4	2340%
	Normal order	118.3	11730%
	Memoized normal order	24.8	2380%

Table 2. Average interpretational overhead of our self-interpreters

run time. Memoized normal-order reduction is similarly related to call-by-need (lazy) reduction. It reduces a β-redex without first reducing its parameter; parameters are only reduced if and when they need to be. When a parameter is reduced to a value, the value is saved in case it is needed again later. We do not memoize the β-normal form of parameters, so as we normalize a term, the work to normalize each parameter's value could be repeated.

Definition 7.1. $time_{no}^{te}(\mathsf{e}) = n$ if $te(\mathsf{e})$ normalizes in n steps of normal-order reduction.

Definition 7.2. $time_{mno}^{te}(\mathsf{e}) = n$ if $te(\mathsf{e})$ normalizes in n steps of memoized normal-order reduction.

Self-interpreters. We experiment with three typed self-interpreters for $\mathrm{F}_{\omega}^{\mu i}$ – essentially typed versions of the untyped λ-calculus self-interpreters in Figures 4, 12, and 13. Each of these interpreters has a different amount of interpretational overhead, which also depends on which *time* function is used. We measured the mean interpretational slowdown and overhead for the cube, factorial, and Ackermann functions on a range inputs. The tagless-final PHOAS interpreter is the most efficient, with an interpretational overhead of around 900% – corresponding to a $10X$ slowdown – for each evaluation strategy. The Mogensen-Scott PHOAS interpreter is the least efficient, with an overhead of from 6780% for memoized normal order and 13550% for normal order. The tagless-final deBruijn interpreter is in between, with an overhead of from 2340% for call-by-value and 11730% for normal order.

Jones-optimality. Our partial evaluator is Jones-optimal for all 9 combinations of our three self-interpreters and three evaluation strategies. Table 3 gives a summary of our Jones optimality tests for our three $\mathrm{F}_{\omega}^{\mu i}$ self-interpreters and three evaluation strategies. Each row aggregates the result of a set of benchmarks comparing the *time* cost of each subject program with the specialization of the self-interpreter to that program. Again, we use the cube, factorial, and Ackermann functions for our subject programs, and we test with a range of inputs for each. A speedup of 1.00 indicates that the subject program runs in exactly the same amount of time as the specialization. This usually happens when the two programs are exactly the same; when specialization removes all the interpretational overhead and does nothing else, recovering the original program. A speedup greater than 1 indicates that the specialized program is faster than the original, and a speedup less than 1 indicates a slowdown. The table demonstrates that our partial evaluator is Jones-optimal for each combination: in each case, the minimum speedup is 1. It also shows that we can sometimes achieve speedups beyond removing interpretational overhead – up to approximately $1.5X$ for our benchmarks.

Interpreter	Time	Min Speedup	Mean Speedup	Max Speedup
Tagless-final PHOAS	Call-by-value	1.00	1.06	1.36
	Normal order	1.00	1.14	1.41
	Memoized normal order	1.00	1.06	1.33
Mogensen-Scott PHOAS	Call-by-value	1.00	1.14	1.50
	Normal order	1.00	1.34	1.54
	Memoized normal order	1.00	1.12	1.46
Tagless-final deBruijn	Call-by-value	1.00	1.16	1.50
	Normal order	1.00	1.35	1.54
	Memoized normal order	1.00	1.44	1.46

Table 3. Summary of Jones Optimality Tests

Interpreter	Time	Min Speedup	Mean Speedup	Max Speedup
Tagless-final PHOAS	Call-by-value	0.03	0.74	1.89
	Normal order	1.00	1.55	1.92
	Memoized normal order	0.11	0.80	1.80
Mogensen-Scott PHOAS	Call-by-value	0.03	0.74	1.89
	Normal order	1.00	1.55	1.92
	Memoized normal order	0.11	0.80	1.80
Tagless-final deBruijn	Call-by-value	0.06	0.83	1.89
	Normal order	1.00	1.53	1.92
	Memoized normal order	0.17	0.89	1.80

Table 4. Demonstration of Non-Optimality of Specialization by β-Normalization

Interpreter	Reduction	Min Speedup	Mean Speedup	Max Speedup
Tagless-final PHOAS	Call-by-value	1.27	1.29	1.29
	Normal order	1.35	1.37	1.39
	Memoized normal order	1.26	1.27	1.27
Mogensen-Scott PHOAS	Call-by-value	1.07	1.08	1.09
	Normal order	1.29	1.30	1.30
	Memoized normal order	1.10	1.12	1.12
Tagless-final deBruijn	Call-by-value	1.32	1.33	1.34
	Normal order	1.38	1.39	1.39
	Memoized normal order	1.27	1.27	1.28

Table 5. Summary of Second Futamura Projection Tests

As a point of contrast, Table 4 shows the results of the same Jones-optimality tests for a specialization by normalization partial evaluator for $F_{\omega}^{\mu i}$. The table demonstrates that specialization by β-normalization is only Jones optimal for normal order reduction, in which the value of a variable is always recomputed for each reference. Therefore, all β-reductions are specialization-safe for normal order reduction. Specialization by β-normalization can cause significant slowdowns for call-by-value or memoized normal order reduction – the $0.03X$ speedup for call-by-value corresponds to a $33X$ slowdown – the original program runs $33X$ faster than the specialized one.

Futamura Projections. The Jones optimality benchmarks in Table 3 show the speedups obtained by the first Futamura projection. We also measure the speedups produced by the second Futamura

Time	**Min Speedup**	**Mean Speedup**	**Max Speedup**
Call-by-value	1.16	1.17	1.19
Normal order	1.37	1.37	1.38
Memoized normal order	1.21	1.22	1.22

Table 6. Summary of Third Futamura Projection Benchmarks

Time	**Speedup**
Call-by-value	1.22
Normal order	1.35
Memoized normal order	1.24

Table 7. Fourth Futamura Projection Benchmarks

projection, which are listed in Table 5. For each self-interpreter and each *time* function, we use the second Futamura projection to generate a compiler, and compare its running time with that of the first Futamura projection. We compile each of the cube, factorial, and Ackermann functions.

We measure the speedup obtained by the third Futamura projection by comparing the *time* to generate compilers using `cogen` versus the second Futamura projection. For each evaluation strategy, we generate compilers using each of our self-interpreters. The results are summarized in Table 6.

We measure the speedup obtained by the fourth Futamura projection by comparing the *time* to generate a compiler generator using `selfgen` versus the third Futamura projection. The speedups are shown in Table 7.

8 RELATED WORK

The following table compares our work with five classical papers in the three main dimensions that we consider. After the table, we give a discussion of a variety of related work.

	Typed representation	Futamura projections	Jones optimality
Original mix [Jones et al. 1985]		√	
Lambda-mix [Gomard and Jones 1991]		√	
Similix [Bondorf and Danvy 1991]		√	√
Schism [Consel 1993]		√	
TDPE [Danvy 1996]		√	
This paper	√	√	√

Typed representation. We were inspired by the considerable body of work on typed self-representation and self-interpretation. The classical approach to meta-programming in a statically-typed language is to use a single *universal* type for all program representations. This incurs a tagging overhead and allows type errors in generated code. On the other hand, typed representation can eliminate tags and ensure that only well-typed programs can be represented. Further, it can provide strong correctness properties for meta-programs just by type checking them. For example, the type of our partial evaluator ensures that it always generates well-typed code of the correct type. At the core of this line of work is the challenge to define a typed self-representation. In their seminal paper, Pfenning and Lee [1991] showed how to represent System F in F_ω, and how to represent F_ω in F_ω^+, but fell short of a self-representation. Rendel et al. [2009] and Jay and Palsberg [2011] presented the first typed self-representations, for two calculi with undecidable type checking. Brown and Palsberg [2015, 2016] defined typed self-representations for two calculi with decidable type checking, the latter of which is also strongly normalizing.

Most closely related to this work is our previous work [Brown and Palsberg 2017] that defined typed self-representations and three typed self-evaluators. Each of those self-evaluators implements a particular evaluation strategy for the calculus itself. The key insight in that paper was a way to encode type equality proofs for a GADT-style self-representation. In this paper, we use the same language and some of the representation techniques first presented there, but we use a tagless-final self-representation instead of the GADT-style encoding, which is important for enabling our partial evaluator to generate the Futamura projections.

Futamura Projections. A variety of partial evaluators for untyped languages have enabled the three classical Futamura projections including the original mix [Jones et al. 1985], Lambda-mix [Gomard and Jones 1991], Schism [Consel 1993], and Similix [Bondorf and Danvy 1991]. Glück [1999, 2009] popularized the fourth Futamura projection and showed how to achieve it. Launchbury [1991] described a partial evaluator for LML, which operates on a universally typed representation and supports the first and second Futamura projections. Danvy [1996] presented type-directed partial evaluation for a simply-typed language and showed how it enables the first and second Futamura projections.

Carette et al. [2009] implemented a typed partial evaluator that operated on a typed tagless-final representation. Their object language was the simply-typed λ-calculus extended with integers, booleans, and a fixpoint operator, and their meta-languages was MetaOCaml. Their partial evaluator could not generate the Futamura projections and was not Jones-optimal.

Our partial evaluator is the first that operates on a typed representation and can be self-applied. Typed representation ensures that the partial evaluator always generates well-typed code of the correct type. This is a strong correctness properties guaranteed just by type checking.

Jones Optimality. Similix [Bondorf and Danvy 1991] achieved Jones optimality for an untyped language. Makholm [2000] discussed the challenges for Jones optimality for simply-typed languages when the partial evaluator relies on a universal type for all program representations. Later, Taha et al. [2001], and also Danvy and López [2003], achieved Jones optimality for simply-typed languages. Gluck [2002, 2008] discussed the effect of semantics-preserving program transformations ahead of partial evaluation. He showed that such transformations can enable a Jones optimal *offline* partial evaluator to achieve the effect of an *online* partial evaluator.

Some papers [Barker et al. 2007; Feigin and Mycroft 2008; Gade and Gluck 2006; Makholm 2000] work with first-order languages and define Jones optimality via the time-based comparison that we state in Definition 2.2. Other papers [Danvy and López 2003; Skalberg 1999; Taha et al. 2001] work with higher-order functional languages and define Jones Optimality by checking that the specialization of the self-interpreter to a program produces the program itself (up to α-equivalence). This rules out the possiblity that specialization could cause a slowdown, but also rules out the possibility of a speedup.

Our proof of Jones optimality is similar in spirit to previous proofs [Gade and Gluck 2006; Gomard and Jones 1991]. In particular, we make similar assumptions in choosing steps of call-by-value reduction as a measure of time – that all reductions have an equal cost, and that other factors such as program size do not affect running time. The key to our proof of Jones optimality is the specialization-safety of specialization-safe reduction. We prove this by reordering a sequence of specialization-safe steps followed by call-by-value steps to align with a sequence of only call-by-value steps. This reordering can only add steps, and the pure call-by-value sequence establishes a maximum number of steps. This technique is inspired by a proof by Alex Simpson about linear λ-calculus [Simpson 2005].

Terminating Partial Evaluation. Asai et al. [2014] proved various correctness properties of a partial evaluator, including termination. The partial evaluator is implemented in a language that is more expressive that the language being partially evaluated, so self-application is impossible. Our work differs in that we do achieve self-application while we use experiments to demonstrate termination for important cases.

Other Related Work. Bondorf and Dussart [1994], Birkedal and Welinder [1994], and Thiemann [1996] have shown how to write strikingly simple compiler generators by hand for untyped or simply-typed languages. This is a major alternative to the use of a partial evaluator to produce a compiler generator via the third Futamura projection. However, the task to write a polymorphically-typed compiler generator by hand is significantly harder. It is an open question whether one can write a polymorphically-typed compiler generator by hand that is simpler than the one we have generated automatically.

Researchers have developed techniques for proving the semantic correctness of partial evaluation, including the recent paper by Hirota and Asai [2014].

Shali and Cook [2011] described a simply-typed partial evaluator for Java, which enables the first Futamura projection. Brady and Hammond [2010] described a partial evaluator for a dependently typed language that enables the first Futamura projection. It is an open question whether those approaches can be extended to enable the other Futamura projections.

Feigin and Mycroft [2008] showed that a version of Jones optimality coincides with a known criterion for efficient virtualization. Barker, Leuschel, and Varea showed how to build a Jones optimal partial evaluator for an untyped logic programming language [Barker et al. 2007].

9 CONCLUSION

In this work we presented the first self-applicable partial evaluator that operates on typed representations, generates the Futamura projections, and is Jones optimal. Operating on typed representations avoids tagging overhead and guarantees that it produces type-correct code. We use a novel approach to a partial-evaluator that focuses on specialization-safe β-reduction that is guaranteed never to cause a slowdown at runtime. We prove that our partial evaluator is Jones optimal for one self-interpreter and a measure of running time that counts steps of call-by-value reduction. Our experiments demonstrate that it is also Jones optimal for two other self-interpreters and time measures based on normal-order and memoized normal-order reduction.

Our results open new challenges for future work, including extending specialization-safe reduction to higher-level languages with efficient base types and user-defined types, and going beyond β-reduction to include more complex specialization-safe transformations. Another direction is to explore a complex affine variable analysis, which might lead to a better partial evaluator. Our affine analysis is sufficient to achieve Jones optimality and also simple enough to enable the Futamura projections. In particular, it can be implemented without a fixpoint combinator, which helps make our partial evaluator strongly-normalizing so the Futamura projections terminate.

One limitation of our partial evaluator is that specialization doesn't always terminate. Also, that our partial evaluator is Jones-optimal does not imply that its speedups are maximal. We leave for future work the challenge to implement a typed self-applicable and Jones-optimal partial evaluator that terminates on all inputs, and the question of identifying more powerful specialization-safe reduction relations that can be used to define a typed self-applicable and Jones-optimal partial evaluator with greater speedups.

ACKNOWLEDGMENTS

We thank Nada Amin, Jacques Carette, William Cook, Olivier Danvy, Danko Ilik, Neil Jones, Oleg Kiselyov, Philip Wadler, and the POPL reviewers for helpful discussions and suggestions. This material is based upon work supported by the National Science Foundation under Grant Number 1219240.

REFERENCES

Lars Ole Andersen. Self-applicable C program specialization. In *Proceedings of PEPM'92, Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University).

Kenichi Asai, Luminous Fennell, Peter Thiemann, and Yang Zhang. A type theoretic specification of partial evaluation. In *PPDP'14 Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, pages 57–68, 2014.

Steve Barker, Michael Leuschel, and Mauricio Varea. Efficient and flexible access control via Jones-optimal logic program specialisation. *Higher-Order and Symbolic Computation*, pages 3–35, 2007.

Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In *Programming Language Implementation and Logic Programming*, pages 198–214, 1994.

Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–10, 1994.

Edwin C. Brady and Kevin Hammond. Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation. In *Proceedings of ICFP'10, ACM SIGPLAN International Conference on Functional Programming*, 2010.

Matt Brown and Jens Palsberg. Self-Representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676988. URL http://doi.acm.org/10.1145/2676726.2676988.

Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 5–17, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837623. URL http://doi.acm.org/10.1145/2837614.2837623.

Matt Brown and Jens Palsberg. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 415–428, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009853. URL http://doi.acm.org/10.1145/3009837.3009853.

Matt Brown and Jens Palsberg. Webpage accompanying this paper. The webpage accompanying this paper is available at http://compilers.cs.ucla.edu/popl18/. The full paper with the appendix is available there, as is the source code for our implementation of System $F_{\omega}^{\mu i}$ and its self-interpreters and partial evaluator., 2018.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411226. URL http://doi.acm.org/10.1145/1411204.1411226.

Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

Olivier Danvy. Type-directed partial evaluation. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In *Proceedings of ESOP'03, European Symposium on Programming*, pages 335–347. Springer-Verlag (*LNCS*), 2003.

Boris Feigin and Alan Mycroft. Jones optimality and hardware virtualization: a report on work in progress. In *PEPM*, pages 169–175, 2008.

Yoshihiko Futamura. Evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999. First published in 1971 in System.Computers.Controls, Volume 2, Number 5, pages 45–50.

Johan Gade and Robert Gluck. On Jones-optimal specializers: A case study using unmix. In *Proceedings of APLAS'06, Asian Symposium on Programming Languages and Systems*, pages 406–422. Springer-Verlag (*LNCS* 4279), 2006.

Robert Glück. Is there a fourth Futamura projection? In *Proceedings of PEPM'99, Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 51–60, 1999.

Robert Gluck. Jones optimality, binding-time improvements, and the strength of program specializers. In *ASIA-PEPM*, pages 9–19, 2002.

Robert Gluck. An investigation of jones optimality and bti-universal specializers. *Higher-Order and Symbolic Computation*, 21(3):283–309, 2008.

Robert Glück. An experiment with the fourth Futamura projection. In *Ershov Memorial Conference*, pages 135–150, 2009.

Robert Glück. Is there a fourth futamura projection? In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 51–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-327-3. doi: 10.1145/1480945.1480954. URL http://doi.acm.org/10.1145/1480945.1480954.

Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

Noriko Hirota and Kenichi Asai. Formalizing a correctness property of a type-directed partial evaluator. In *PLPV'14 Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification*, pages 41–46, 2014.

Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of ICFP'11, ACM SIGPLAN International Conference on Functional Programming*, pages 247–258, Tokyo, September 2011.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Proceedings of Rewriting Techniques and Applications*, pages 225–282. Springer-Verlag (*LNCS* 202), 1985.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

Oleg Kiselyov. *Typed Tagless Final Interpreters*, pages 130–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32202-0. doi: 10.1007/978-3-642-32202-0_3. URL http://dx.doi.org/10.1007/978-3-642-32202-0_3.

John Launchbury. A strongly-typed self-applicable partial evaluator. In *Proceedings of FPCA'91, Sixth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 145–164, 1991.

Henning Makholm. On jones-optimal specialization for strongly typed languages. In *SAIG*, pages 129–148, 2000.

Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. See also DIKU Report D–128, Sep 2, 1994.

Torben Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 39–44, New York, NY, USA, 1995. ACM. ISBN 0-89791-720-0. doi: 10.1145/215465.215469. URL http://doi.acm.org/10.1145/215465.215469.

Tobias Nipkow. Lambda calculus. https://www21.in.tum.de/teaching/logik/SS13/lambda-en.pdf, (accessed July 7, 2017), August 2, 2012.

Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, June 2009.

Amin Shali and William Cook. Hybrid partial evaluation. In *Proceedings of OOPSLA'11, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 375–390, 2011.

Alex Simpson. Reduction in a linear lambda-calculus with applications to operational semantics. In *Proceedings of the 16th International Conference on Term Rewriting and Applications*, RTA'05, pages 219–234, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25596-6, 978-3-540-25596-3. doi: 10.1007/978-3-540-32033-3_17. URL http://dx.doi.org/10.1007/978-3-540-32033-3_17.

Sebastian Skalberg. Mechanical proof of the optimality of a partial evaluator. M.S. Thesis; DIKU, University of Copenhagen, 1999.

Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In *Proceedings of PADO'01, Programs as Data Objects, Second Symposium*, pages 257–275, 2001.

Peter Thiemann. Cogen in six lines. In *Proceedings of ICFP'96, ACM International Conference on Functional Programming*, pages 180–189, 1996.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 249–262, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944728. URL http://doi.acm.org/10.1145/944705.944728.

$$\text{NP-1}\ \frac{}{\texttt{e} > \texttt{e}}$$

$$\text{NP-2}\ \frac{\texttt{e} > \texttt{e}'}{\lambda\texttt{x}^*\texttt{.e} > \lambda\texttt{x}^*\texttt{.e}'}$$

$$\text{NP-3}\ \frac{\texttt{a} > \texttt{a}' \qquad \texttt{b} > \texttt{b}'}{\texttt{a b} > \texttt{a}'\ \texttt{b}'}$$

$$\text{NP-4}\ \frac{\texttt{e} > \texttt{e}' \qquad \texttt{v} > \texttt{v}'}{(\lambda\texttt{x.e})\texttt{v} > \texttt{e}'\texttt{[x:=v}'\texttt{]}}$$

$$\text{NP-5}\ \frac{\texttt{e} > \texttt{e}'}{(\lambda\texttt{x.e})\texttt{y} > \texttt{e}'\texttt{[x:=y]}}$$

$$\text{NP-6}\ \frac{\texttt{a} > \texttt{a}' \qquad \texttt{b} > \texttt{b}'}{(\lambda\texttt{x}^\circ\texttt{.a})\texttt{b} > \texttt{a}'\texttt{[x}^\circ\texttt{:=b}'\texttt{]}}$$

A CONFLUENCE

Following the proof technique in Nipkow's lecture notes, we define the nested parallel strong optimization relation a > b.

Lemma A.1. *If* e1 $\to_s$ e2*, then* e1 > e2

Proof. By induction on the derivation of e1 $\to_s$ e2.

Suppose e1 $\to_s$ e2 is by SSR-1. Then e1 = $(\lambda$x.e)v and e2 = e[x:=v], and e1 > e2 is derivable by NP-4.

Suppose e1 $\to_s$ e2 is by SSR-2. Then e1 = a1 b and e2 = a2 b and a1 $\to_s$ a2. By induction, a1 > a2. Therefore, e1 > e2 is derivable by NP-3.

Suppose e1 $\to_s$ e2 is by SSR-3. Then e1 = a b1 and e2 = a b2 and b1 $\to_s$ b2. By induction, b1 > b2. Therefore, e1 > e2 is derivable by NP-3.

Suppose e1 $\to_s$ e2 is by SSR-4. Then e1 = $(\lambda$x°.a)b and e2 = a[x°:=b]. Therefore, e1 > e2 is derivable by NP-6.

Suppose e1 $\to_s$ e2 is by SSR-5. Then e1 = $(\lambda$x.e)y and e2 = e[x:=y]. Therefore, e1 > e2 is derivable by NP-5.

Suppose e1 $\to_s$ e2 is by SSR-6. Then e1 = $(\lambda$x.a1) and e2 = $(\lambda$x.a2) and a1 $\to_s$ a2. By induction, a1 > a2. Therefore, e1 > e2 is derivable by NP-2.

Suppose e1 $\to_s$ e2 is by SSR-7. Then e1 = $(\lambda$x°.a1) and e2 = $(\lambda$x°.a2) and a1 $\to_s$ a2. By induction, a1 > a2. Therefore, e1 > e2 is derivable by NP-2.

□

Lemma A.2. *If* e1 > e2*, then* e1 $\to_s^*$ e2

Proof. By induction on the derivation of e1 > e2.

Suppose e1 > e2 is by NP-1. Then e1 = e2, and e1 $\to_s^0$ e2.

Suppose e1 > e2 is by NP-2. Then e1 = $(\lambda$x*.e) and e2 = $(\lambda$x*.e$'$) and e > e$'$. By induction, e $\to_s^*$ e$'$. Therefore, e1 $\to_s^*$ e2 by multiple uses of SSR-6 or SSR-7.

Suppose e1 > e2 is by NP-3. Straightforward induction.

Suppose e1 > e2 is by NP-4. Then e1 = $(\lambda$x.e)v and e2 = e$'$[x:=v$'$] and e > e$'$ and v > v$'$. By induction, e $\to_s^*$ e$'$ and v $\to_s^*$ v$'$. Therefore, e1 = $(\lambda$x.e)v $\to_s^*$ $(\lambda$x.e$'$)v$'$ $\to_s$ e$'$[x:=v$'$] = e2.

Suppose e1 > e2 is by NP-5. Then e1 = $(\lambda$x.e)y and e2 = e$'$[x:=y] and e > e$'$. By induction e $\to_s^*$ e$'$. Therefore, e1 = $(\lambda$x.e)y $\to_s^*$ $(\lambda$x.e$'$)y $\to_s$ e$'$[x:=y] = e2.

Suppose e1 > e2 is by NP-6. Then e1 = $(\lambda$x°.a)b and e2 = a$'$[x°:=b$'$] and a > a$'$ and b > b$'$. By induction, a $\to_s^*$ a$'$ and b $\to_s^*$ b$'$. Therefore, e1 = $(\lambda$x°.a)b $\to_s^*$ $(\lambda$x°.a$'$)b$'$ $\to_s$ a$'$[x°:=b$'$] = e2.

□

Lemma A.3. *If* $(\lambda x^*.a) > b$, *then there exists* a' *such that* $b = (\lambda x^*.a')$ *and* $a > a'$

Proof. Straightforward. □

Lemma A.4. *If* $e > e'$, *then* $e[x{:=}y] > e'[x{:=}y]$.

Proof. By straightforward induction on e. □

Lemma A.5. *If* $e > e'$ *and* $v > v'$, *then* $e[x{:=}v] > e'[x{:=}v']$.

Proof. By induction on e.

Suppose $e = x$. Then $e' = x$. We have that $e[x{:=}v] = v$ and $e'[x{:=}v'] = v'$. Since $v > v'$, $e[x{:=}v] > e'[x{:=}v']$.

Suppose $e = y^* \neq x$. Then $e[x{:=}v] = e$ and $e'[x{:=}v'] = e'$. Since $e > e'$, $e[x{:=}v] > e'[x{:=}v']$.

Suppose $e = (\lambda y^*.e1)$. Without loss of generality, assume $y^* \neq x$ and $y^* \notin FV(v) \cup FV(v')$. By Lemma A.3, $e' = (\lambda y^*.e1')$ and $e1 > e1'$. By induction, $e1[x{:=}v] > e1'[x{:=}v']$. Therefore, $\lambda y^*.e1[x{:=}v] > \lambda y^*.e1'[x{:=}v']$. Since $\lambda y^*.e1[x{:=}v] = (\lambda y^*.e1)[x{:=}v] = e[x{:=}v]$ and $\lambda y^*.e1'[x{:=}v'] = (\lambda y^*.e1')[x{:=}v'] = e'[x{:=}v']$, we have $e[x{:=}v] > e'[x{:=}v']$ as required.

Suppose e is an application. We proceed by inspecting the rule used to derive $e > e'$.

Suppose that $e > e'$ is by NP-3. Then $e = e1\ e2$ and $e' = e1'\ e2'$ and $e1 > e1'$ and $e2 > e2'$. By induction, $e1[x{:=}v] > e1'[x{:=}v']$ and $e2[x{:=}v] > e2'[x{:=}v']$. Therefore $e1[x{:=}v]\ e2[x{:=}v] > e1'[x{:=}v']\ e2'[x{:=}v']$. Since $e1[x{:=}v]\ e2[x{:=}v] = (e1\ e2)[x{:=}v] = e[x{:=}v]$, and $e1'[x{:=}v']\ e2'[x{:=}v'] = (e1'\ e2')[x{:=}v'] = e'[x{:=}v']$, NP-3 derives $e[x{:=}v] > e'[x{:=}v']$ as required.

Suppose that $e > e'$ is by NP-4. Then $e = (\lambda y.e1)v1$ and $e' = e1'[y{:=}v1']$ and $e1 > e1'$ and $v1 > v1'$. Without loss of generality, assume $y \neq x$ and $y \notin FV(v)$. By induction, $e1[x{:=}v] > e1'[x{:=}v']$ and $v1[x{:=}v] > v1'[x{:=}v']$. It must be the case that $v1[x{:=}v]$ is a value. Therefore, NP-4 derives $(\lambda y.e1[x{:=}v])(v1[x{:=}v]) > e1'[x{:=}v'][y{:=}v1'[x{:=}v']]$. Since $(\lambda y.e1[x{:=}v])(v1[x{:=}v]) = ((\lambda y.e1)v1)[x{:=}v] = e[x{:=}v]$ and $e1'[x{:=}v'][y{:=}v1'[x{:=}v']] = (e1'[y{:=}v1'])[x{:=}v'] = e'[x{:=}v']$, we have that $e[x{:=}v] > e'[x{:=}v']$ as required.

Suppose that $e > e'$ is by NP-5. Then $e = (\lambda y.e1)z$ and $e' = e1[y{:=}z]$ and $e1 > e1'$. Without loss of generality, assume $y \neq x$ and $y \notin FV(v)$. By induction, $e1[x{:=}v] > e1'[x{:=}v']$.

This step is key! A NP-5 step turns into a NP-4 step. If $z = x$, then $e[x{:=}v] = ((\lambda y.e1)z)[x{:=}v] = (\lambda y.e1[x{:=}v])v$ and $e'[x{:=}v'] = (e1'[y{:=}z])[x{:=}v'] = (e1'[y{:=}v'])[x{:=}v'] = (e1'[x{:=}v'])[y{:=}v']$. Since $e1[x{:=}v] > e1'[x{:=}v']$ and $v > v'$, NP-4 derives $(\lambda y.e1[x{:=}v])v > (e1'[x{:=}v'])[y{:=}v']$. Therefore, $e[x{:=}v] > e'[x{:=}v']$ as required.

If $z \neq x$, then $e[x{:=}v] = ((\lambda y.e1)z)[x{:=}v] = (\lambda y.e1[x{:=}v])z$ and $e'[x{:=}v'] = (e1'[y{:=}z])[x{:=}v'] = (e1'[x{:=}v'])[y{:=}z]$. Since $e1[x{:=}v] > e1'[x{:=}v']$, NP-5 derives $(\lambda y.e1[x{:=}v])z > (e1'[x{:=}v'])[y{:=}z]$. Therefore, $e[x{:=}v] > e'[x{:=}v']$ as required.

Suppose that $e > e'$ is by NP-6. Then $e = (\lambda y^\circ.a)b$ and $e' = a'[y^\circ{:=}v']$ and $a > a'$ and $b > b'$. Without loss of generality, assume $y^\circ \notin FV(v)$. By induction, $a[x{:=}v] > a'[x{:=}v']$ and $b[x{:=}v] > b'[x{:=}v']$. Therefore, NP-6 derives $(\lambda y^\circ.a[x{:=}v])(b[x{:=}v]) > a'[x{:=}v'][y^\circ{:=}b'[x{:=}v']]$. Since $(\lambda y^\circ.a[x{:=}v])(b[x{:=}v]) = ((\lambda y^\circ.a)b)[x{:=}v] = e[x{:=}v]$ and $a'[x{:=}v'][y^\circ{:=}b'[x{:=}v']] = (a'[y^\circ{:=}b'])[$
$= e'[x{:=}v']$, $e[x{:=}v] > e'[x{:=}v']$ as required. □

Lemma A.6. *If* $a > a'$ *and* $b > b'$, *then* $a[x^\circ{:=}b] > a[x^\circ{:=}b']$.

Note: the proof is identical to that of Lemma A.5, except that in the case that $a > a'$ is by NP-5, the argument z cannot be x°.

Proof. By induction on a.

Suppose $a = x^\circ$. Then $a' = x^\circ$, and $a[x^\circ{:=}b] = b$ and $a'[x^\circ{:=}b'] = b'$. Therefore $b > b'$ implies $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$.

Suppose $a = y^* \neq x^\circ$. $a[x^\circ{:=}b] = a$ and $a'[x^\circ{:=}b'] = a'$. Therefore $a > a'$ implies $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$.

Suppose $a = (\lambda y^*.a1)$. Without loss of generality, assume $y^* \neq x^\circ$ and $y^* \notin FV(b) \cup FV(b')$. By Lemma A.3, $a' = (\lambda y^*.a1')$ and $a1 > a1'$. By induction, $a1[x^{\circ\circ}{:=}b] > a1'[x^\circ{:=}b']$. Therefore, $\lambda y^*.a1[x^\circ{:=}b] > \lambda y^*.a1'[x^\circ{:=}b']$. Since $\lambda y^*.a1[x^\circ{:=}b] = (\lambda y^*.a1)[x^\circ{:=}b] = a[x^\circ{:=}b]$ and $\lambda y^*.a1'[x^\circ{:=}b'] = (\lambda y^*.a1')[x^\circ{:=}b'] = a'[x^\circ{:=}b']$, we have $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ as required.

Suppose a is an application. We proceed by inspecting the rule used to derive $a > a'$.

Suppose that $a > a'$ is by NP-3. Then $a = a1\ a2$ and $a' = a1'\ a2'$ and $a1 > a1'$ and $a2 > a2'$. By induction, $a1[x^\circ{:=}b] > a1'[x^\circ{:=}b']$ and $a2[x^\circ{:=}b] > a2'[x^\circ{:=}b']$. Therefore $a1[x^\circ{:=}b]\ a2[x^\circ{:=}b] > a1'[x^\circ{:=}b']\ a2'[x^\circ{:=}b']$. Since $a1[x^\circ{:=}b]\ a2[x^\circ{:=}b] = (a1\ a2)[x^\circ{:=}b] = a[x^\circ{:=}b]$, and $a1'[x^\circ{:=}b']\ a2'[x^\circ{:=}b'] = (a1'\ a2')[x^\circ{:=}b'] = a'[x^\circ{:=}b']$, NP-3 derives $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ as required.

Suppose that $a > a'$ is by NP-4. Then $a = (\lambda y.a1)v$ and $a' = a1'[y{:=}v']$ and $a1 > a1'$ and $v > v'$. Without loss of generality, assume $y \neq x^\circ$ and $y \notin FV(b)$. By induction, $a1[x^\circ{:=}b] > a1'[x^\circ{:=}b']$ and $v[x^\circ{:=}b] > v'[x^\circ{:=}b']$. It must be the case that $v[x^\circ{:=}b]$ is a value. Therefore, NP-4 derives $(\lambda y.a1[x^\circ{:=}b])(v[x^\circ{:=}b]) > a1'[x^\circ{:=}b'][y{:=}v'[x^\circ{:=}b']]$. Since $(\lambda y.a1[x^\circ{:=}b])(v[x^\circ{:=}b]) = ((\lambda y.a1)v)[x^\circ{:=}b] = a[x^\circ{:=}b]$ and $a1'[x^\circ{:=}b'][y{:=}v'[x^\circ{:=}b']] = (a1'[y{:=}v'])[x^\circ{:=}b'] = a'[x^\circ{:=}b']$, we have that $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ as required.

Suppose that $a > a'$ is by NP-5. Then $a = (\lambda y.a1)z$ and $a' = a1[y{:=}z]$ and $a1 > a1'$. Without loss of generality, assume $y \neq x^\circ$ and $y \notin FV(b)$. By induction, $a1[x^\circ{:=}b] > a1'[x^\circ{:=}b']$.

Since $z \neq x^\circ$, we have that $a[x^\circ{:=}b] = ((\lambda y.a1)z)[x^\circ{:=}b] = (\lambda y.a1[x^\circ{:=}b])z$ and $a'[x^\circ{:=}b'] = (a1'[y{:=}z])[x^\circ{:=}b'] = (a1'[x^\circ{:=}b'])[y{:=}z]$. Since $a1[x^\circ{:=}b] > a1'[x^\circ{:=}b']$, NP-5 derives $(\lambda y.a1[x^\circ{:=}b])z > (a1'[x^\circ{:=}b'])[y{:=}z]$. Therefore, $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ as required.

Suppose that $a > a'$ is by NP-6. Then $a = (\lambda y^\circ.a)b1$ and $a' = a'[y^\circ{:=}b']$ and $a > a'$ and $b1 > b1'$. Without loss of generality, assume $y^\circ \notin FV(b)$. By induction, $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ and $b1[x^\circ{:=}b] > b1'[x^\circ{:=}b']$. Therefore, NP-6 derives $(\lambda y^\circ.a[x^\circ{:=}b])(b1[x^\circ{:=}b]) > a'[x^\circ{:=}b'][y^\circ{:=}b1'[x^\circ{:=}$
Since $(\lambda y^\circ.a[x^\circ{:=}b])(b1[x^\circ{:=}b]) = ((\lambda y^\circ.a)b1)[x^\circ{:=}b] = a[x^\circ{:=}b]$ and $a'[x^\circ{:=}b'][y^\circ{:=}b1'[x^\circ{:=}b']] = (a'[y^\circ{:=}b1'])[x^\circ{:=}b'] = a'[x^\circ{:=}b']$, $a[x^\circ{:=}b] > a'[x^\circ{:=}b']$ as required. □

Lemma A.7 (Diamond property for $>$). *If* $a > b1$ *and* $a > b2$, *then there exists a term* c *such that* $b1 > c$ *and* $b2 > c$.

Proof. By induction on a.

Suppose $a = x^*$. Then $b1 = b2 = x^*$. Holds with $c = x^*$.

Suppose $a = (\lambda x^*.a')$. By Lemma A.3, $b1 = (\lambda x^*.b1')$ and $b2 = (\lambda x^*.b2')$ and $a' > b1'$ and $a' > b2'$. By induction, there exists a c' such that $b1' > c'$ and $b2' > c'$. Therefore, NP-2 derives $(\lambda x^*.b1') > (\lambda x^*.c')$ and $(\lambda x^*.b2') > (\lambda x^*.c')$. Holds with $c = (\lambda x^*.c')$.

Suppose a is an application. Consider the derivation of $a > b1$ and $a > b2$.

Suppose $a > b1$ is by NP-3 and $a > b2$ is by NP-3. Then $a = d\ e$ and $b1 = d1\ e1$ and $b2 = d2\ e2$ and $d > d1$ and $d > d2$ and $e > e1$ and $e > e2$. By induction, there exist $d3$ and $e3$ such that $d1 > d3$ and $d2 > d3$ and $e1 > e3$ and $e2 > e3$. Let $c = d3\ e3$. NP-3 derives $b1 = d1\ e1 > d3\ e3 = c$ and $b2 = d2\ e2 > d3\ e3 = c$.

Suppose $a > b1$ is by NP-4 and $a > b2$ is by NP-4. Then $a = (\lambda x.e)v$ and $b1 = e1[x{:=}v1]$ and $b2 = e2[x{:=}v2]$ and $e > e1$ and $e > e2$ and $v > v1$ and $v > v2$. By induction, there exist terms $e3$ and $v3$ such that $e1 > e3$ and $e2 > e3$ and $v1 > v3$ and $v2 > v3$. Let $c = e3[x{:=}v3]$. By Lemma A.5, $b1 = e1[x{:=}v1] > e3[x{:=}v3] = c$ and $b2 = e2[x{:=}v2] > e3[x{:=}v3] = c$.

$$\text{WOPT-1}\ \frac{}{(\lambda x.e)v \to_w e[x{:=}v]}$$

$$\text{WOPT-2}\ \frac{e1 \to_w e1'}{e1\ e2 \to_w e1'\ e2}$$

$$\text{WOPT-3}\ \frac{e1 \to_w e1'}{(\lambda x.e)e1 \to_w (\lambda x.e)e1'}$$

$$\text{WOPT-4}\ \frac{}{(\lambda x^\circ.e)e' \to_w e[x^\circ{:=}e']}$$

$$\text{WOPT-5}\ \frac{e1 \to_w e1'}{(\lambda x^\circ.e)e1 \to_w (\lambda x^\circ.e)e1'}$$

Fig. 22. Weak Optimization

Suppose a > b1 is by NP-5 and a > b2 is by NP-5. Then $a = (\lambda x.e)y$ and $b1 = e1[x{:=}y]$ and $b2 = e2[x{:=}y]$ and e > e1 and e > e2. By induction, there exists a term e3 such that e1 > e3 and e2 > e3. Let $c = e3[x{:=}y]$. By Lemma A.4, $b1 = e1[x{:=}y] > e3[x{:=}y] = c$ and $b2 = e2[x{:=}y] > e3[x{:=}y] = c$.

Suppose a > b1 is by NP-6 and a > b2 is by NP-6. Then $a = (\lambda x^\circ.d)e$ and $b1 = d1[x^\circ{:=}e1]$ and $b2 = d2[x^\circ{:=}e2]$ and d > d1 and d > d2 and e > e1 and e > e2. By induction, there exist terms d3 and e3 such that d1 > d3 and d2 > d3 and e1 > e3 and e2 > e3. Let $c = d3[x^\circ{:=}e3]$. By Lemma A.5, $b1 = d1[x^\circ{:=}e1] > d3[x^\circ{:=}e3] = c$ and $b2 = d2[x^\circ{:=}e2] > d3[x^\circ{:=}e3] = c$.

Suppose a > b1 is by NP-3 and a > b2 is by NP-4. Then $a = (\lambda x.e)v$ and $b1 = (\lambda x.e1)v1$ and e > e1 and v > v1, and $b2 = e2[x{:=}v2]$ and e > e2 and v > v2. Since v is a value and v > v1, v1 must be a value. By induction, there is term v3 such that v1 > v3 and v2 > v3. Since v1 and v2 are values, v3 is a value. By induction again, there is a term e3 such that e1 > e3 and e2 > e3. Let $c = e3[x{:=}v3]$. By rule NP-4, $b1 = (\lambda x.e1)v1 > e3[x{:=}v3]$. By Lemma A.5, $b2 = e2[x{:=}v2] > e3[x{:=}v3]$.

Suppose a > b1 is by NP-3 and a > b2 is by NP-5. Then $a = (\lambda x.e)y$ and $b1 = (\lambda x.e1)y$ and $b2 = e2[x{:=}y]$ and e > e1 and e > e2. By induction, there exists an e3 such that e1 > e3 and e2 > e3. Let $c = e3[x{:=}y]$. By rule NP-5, $b1 = (\lambda x.e1)y > e3[x{:=}y] = c$. By Lemma A.4, $b2 = e2[x{:=}y] > e3[x{:=}y] = c$.

Suppose a > b1 is by NP-3 and a > b2 is by NP-6. Then $a = (\lambda x^\circ.e)e'$ and $b1 = (\lambda x^\circ.e1)e1'$ and e > e1 and e > e1′ and $b2 = e2[x^\circ{:=}e2']$ and e > e2 and e > e2′. By induction, there exist e3 and e3′ such that e1 > e3 and e2 > e3, and e1′ > e3′ and e2′ > e3′. Let $c = e3[x^\circ{:=}e3']$. By rule NP-6, $b1 = (\lambda x^\circ.e1)e1' > e3[x^\circ{:=}e3'] = c$. By Lemma A.6, $b2 = e2[x^\circ{:=}e2'] > e3[x^\circ{:=}e3'] = c$. □

Theorem 4.3. *Specialization-safe reduction is confluent.*

Proof. By Nipkow's Lemma A.2.5, and Lemmas A.1, A.2, and A.7 □

Corollary 4.4. *Specialization-safe normal forms are unique.*

B SPECIALIZATION SAFETY

Figure 7 defines call-by-value reduction on annotated terms. Figure 22 defines weak optimization. It extends call-by-value reduction with the rule WOPT-4, which reduces β-redexes without evaluating the argument when the λ-abstraction abstracts over a top-level linear variable.

The following lemma allows us to reason about multiple steps of call-by-value reduction in a big-step-style.

Lemma B.1. *For any n,* `e1 e2` $\rightarrow^{n}$ `v` *if and only if there exist integers $n1$, $n2$, and $n3$, and terms* $(\lambda \mathtt{x}^*.\mathtt{e1}')$ *and* `v2`*, such that* `e1` $\rightarrow^{n1}$ $(\lambda \mathtt{x}^*.\mathtt{e1}')$*, and* `e2` $\rightarrow^{n2}$ `v2`*, and* $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n3}$ `v`*, and* $n = n1 + n2 + n3 + 1$.

We prove each direction separately.

Proof. Suppose `e1 e2` $\rightarrow^{n}$ `v`. We want to prove there exist integers $n1$, $n2$, and $n3$, and terms $(\lambda \mathtt{x}^*.\mathtt{e1}')$ and `v2`, such that `e1` $\rightarrow^{n1}$ $(\lambda \mathtt{x}^*.\mathtt{e1}')$, and `e2` $\rightarrow^{n2}$ `v2`, and $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n3}$ `v`, and $n = n1 + n2 + n3 + 1$.

We proceed by induction on the derivation of `e1 e2` $\rightarrow^{n}$ `v`.

Suppose $n = 0$. Then `e1 e2` = `v`. This is a contradiction, since `e1 e2` is not a λ-abstraction.

Suppose $n > 0$. Then the sequence `e1 e2` $\rightarrow^{n}$ `v` contains at least one step. Consider the first step in the sequence.

If the first step is by CBV-1 or CBV-4, then we have that `e1` = $(\lambda \mathtt{x}^*.\mathtt{e1}')$ and `e2` = `v2` and $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n'}$ `v`. The result holds with $n1 = 0$ and $n2 = 0$ and $n3 = n'$.

If the first step is by CBV-2, then `e1` $\rightarrow$ $\mathtt{e1}'$ and $\mathtt{e1}'$ `e2` $\rightarrow^{n'}$ `v`. By induction, $\mathtt{e1}' \rightarrow^{n1'}$ λ`x.e` and `e2` $\rightarrow^{n2}$ `v2` and `e[x:=v2]` $\rightarrow^{n3}$ `v` and $n' = n1' + n2 + n3 + 1$. Then `e1` $\rightarrow^{n1'+1}$ λ`x.e`. The result holds with $n1 = n1' + 1$.

If the first step is by CBV-3 or CBV-5, then `e1` = $(\lambda \mathtt{x}^*.\mathtt{e1}')$ and `e2` $\rightarrow$ $\mathtt{e2}'$ and $(\lambda \mathtt{x}^*.\mathtt{e1}')$ $\mathtt{e2}'$ $\rightarrow^{n'}$ `v`. Since $(\lambda \mathtt{x}^*.\mathtt{e1}')$ is a value, it doesn't step. Therefore, the induction hypothesis states that $\mathtt{e2}' \rightarrow^{n2'}$ `v2` and $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n3}$ `v` and $n' = n2' + n3 + 1$. Therefore, `e2` $\rightarrow^{n2'+1}$ `v2`. The result holds with $n1 = 0$ and $n2 = n2' + 1$. □

Proof. Suppose `e1` $\rightarrow^{n1}$ $(\lambda \mathtt{x}^*.\mathtt{e1}')$, and `e2` $\rightarrow^{n2}$ `v2`, and $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n3}$ `v`. We want to prove `e1 e2` $\rightarrow^{n1+n2+n3+1}$ `v`.

We proceed by induction on $n1 + n2$.

Suppose $n1 + n2 = 0$. Then $n1 = 0$ and $n2 = 0$. Therefore, `e1` = $(\lambda \mathtt{x}^*.\mathtt{e1}')$ and `e2` = `v2`. We have `e1 e2` $\rightarrow$ $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}]$ and since $\mathtt{e1}'[\mathtt{x}^*{:=}\mathtt{v2}] \rightarrow^{n3}$ `v`, `e1 e2` $\rightarrow^{n3+1}$ `v`. The conclusion holds because $n3 + 1 = n1 + n2 + n3 + 1$.

Suppose $n1 + n2 > 0$. Then either $n1 > 0$, or $n1 = 0$ and $n2 > 0$.

Suppose $n1 > 0$. In particular, $n1 = n1' + 1$. Then there is a term `e` such that `e1` $\rightarrow$ `e` and `e` $\rightarrow^{n1'}$ $(\lambda \mathtt{x}^*.\mathtt{e1}')$. By induction, `e e2` $\rightarrow^{n1'+n2+n3+1}$ `v`. Therefore, `e1 e2` $\rightarrow^{1+n1'+n2+n3+1}$ `v`. The conclusion holds since $n1 = n1' + 1$.

Suppose $n1 = 0$ and $n2 > 0$. In particular, $n2 = n2' + 1$. Then `e1` = $(\lambda \mathtt{x}^*.\mathtt{e1}')$ and there exists a term `e` such that `e2` $\rightarrow$ `e` and `e` $\rightarrow^{n2'}$ `v2`. By induction, `e1 e` $\rightarrow^{n1+n2'+n3+1}$ `v`. Therefore, `e1 e2` $\rightarrow^{1+n1+n2'+n3+1}$ `v`. The conclusion holds since $n2 = n2' + 1$. □

Lemma B.2 (Weakening of unlimited context)**.** *If* $(\Gamma_1, \Gamma_2); \Sigma \vdash \mathtt{e}$*, then* $(\Gamma_1,\mathtt{x},\Gamma_2); \Sigma \vdash \mathtt{e}$.

Proof. By induction on the derivation of $(\Gamma_1, \Gamma_2); \Sigma \vdash \mathtt{e}$.

Suppose `e` is an unlimited variable $\mathtt{y} \neq \mathtt{x}$. Immediate.

Suppose `e` is an affine variable $\mathtt{y}^\circ$. Immediate.

Suppose `e` = `e1 e2`. Then $(\Gamma_1, \Gamma_2); \Sigma_1 \vdash \mathtt{e1}$ and $(\Gamma_1, \Gamma_2); \Sigma_2 \vdash \mathtt{e2}$ and $\Sigma = \Sigma_1 \uplus \Sigma_2$. By induction, $(\Gamma_1,\mathtt{x},\Gamma_2); \Sigma_1 \vdash \mathtt{e1}$, and $(\Gamma_1,\mathtt{x},\Gamma_2); \Sigma_2 \vdash \mathtt{e1}$, so $(\Gamma_1,\mathtt{x},\Gamma_2); \Sigma \vdash \mathtt{e1\ e2}$ as required.

Suppose `e` = $(\lambda \mathtt{y}.\mathtt{e}')$. Then $(\Gamma_1, \Gamma_2,\mathtt{y}); \langle\rangle \vdash \mathtt{e}'$. By induction, $(\Gamma_1,\mathtt{x}\Gamma_2,\mathtt{y}); \langle\rangle \vdash \mathtt{e}'$, so $(\Gamma_1,\mathtt{x}\Gamma_2); \Sigma \vdash (\lambda \mathtt{y}.\mathtt{e}')$ as required. □

Lemma B.3 (Weakening of affine context)**.** *If* $\Gamma; \langle\rangle \vdash \mathtt{e}$*, then* $\Gamma; \Sigma \vdash \mathtt{e}$.

Proof. By induction on the derivation of $\Gamma;\langle\rangle \vdash$ e.

Suppose e is an unlimited variable x. Immediate.

Suppose e is an affine variable x°. Contradiction.

Suppose e = e1 e2. Then $\Gamma;\langle\rangle \vdash$ e1 and $\Gamma;\langle\rangle \vdash$ e2. By induction, $\Gamma;\Sigma \vdash$ e1, so $\Gamma;\Sigma \vdash$ e1 e2 as required.

Suppose e = $(\lambda x^*.e')$. Immediate. □

Lemma B.4. *If* $(\Gamma_1,x,\Gamma_2);\Sigma \vdash$ e1 *and* $(\Gamma_1,\Gamma_2);\langle\rangle \vdash$ e2*, then* $(\Gamma_1,\Gamma_2);\Sigma \vdash$ e1[x:=e2].

Proof. By induction on the derivation of $(\Gamma_1,x,\Gamma_2);\Sigma \vdash$ e1.

Suppose e1 = x. Then e1[x:=e2] = e2, and the conclusion holds.

Suppose e1 = y $\neq$ x. Then e1[x:=e2] = y, and the conclusion holds.

Suppose e1 = y°. Since x $\neq$ y°, e1[x:=e2] = y°, and the conclusion holds.

Suppose e1 = a b. Then $(\Gamma_1,x,\Gamma_2);\Sigma_1 \vdash$ a and $(\Gamma_1,x,\Gamma_2);\Sigma_2 \vdash$ b and $\Sigma = \Sigma_1 \uplus \Sigma_2$. By induction, $(\Gamma_1,\Gamma_2);\Sigma_1 \vdash$ a[x:=e2] and $(\Gamma_1,\Gamma_2);\Sigma_2 \vdash$ b[x:=e2]. Therefore $(\Gamma_1,\Gamma_2);\Sigma_2 \vdash$ (a[x:=e2]) (b[x:=e2]). The conclusion holds since e1[x:=e2] = (a b)[x:=e2] = (a[x:=e2]) (b[x:=e2]).

Suppose e1 = $(\lambda y.a)$. Then $(\Gamma_1,x,\Gamma_2,y);\langle\rangle \vdash$ a. Without loss of generality, assume x $\neq$ y and y does not occur free in e2. By induction, $(\Gamma_1,\Gamma_2,y);\langle\rangle \vdash$ a[x:=e2]. Therefore, $(\Gamma_1,\Gamma_2);\langle\rangle \vdash$ $(\lambda y.a[x:=e2])$. The conclusion holds since e1[x:=e2] = $(\lambda y.a)$[x:=e2] = $(\lambda y.a[x:=e2])$.

Suppose e1 = $(\lambda y^\circ.a)$. This case is similar to the previous case. □

Lemma B.5. *If* $\Gamma;x^\circ \vdash$ e1 *and* $\Gamma;\Sigma \vdash$ e2*, then* $\Gamma;\Sigma \vdash$ e1[x°:=e2].

Proof. By induction on the derivation of $\Gamma;x^\circ \vdash$ e1.

Suppose e1 = y $\in \Gamma$. Then e1[x°:=e2] = y, and the conclusion holds.

Suppose e1 = x°. Then e1[x°:=e2] = e2, and the conclusion holds.

Suppose e1 = a b. Then $\Gamma;\Sigma_1 \vdash$ a and $\Gamma;\Sigma_1 \vdash$ b and $x^\circ = \Sigma_1 \uplus \Sigma_2$. Therefore, either $\Sigma_1 = x^\circ$ and $\Sigma_2 = \langle\rangle$, or else $\Sigma_1 = \langle\rangle$ and $\Sigma_2 = x^\circ$. The two cases are similar, so we will only consider the first. Suppose $\Sigma_1 = x^\circ$ and $\Sigma_2 = \langle\rangle$. Then by induction, we have $\Gamma;\Sigma \vdash$ a[x°:=e2]. Since $\Gamma;\langle\rangle \vdash$ b, x° does not occur free in b, so b[x°:=e2] = b. Therefore, e1[x°:=e2] = (a b)[x°:=e2] = (a[x°:=e2]) b. Since $\Gamma;\Sigma \vdash$ a[x°:=e2] and $\Gamma;\langle\rangle \vdash$ b, we have $\Gamma;\Sigma \vdash$ (a[x°:=e2]) b as required.

Suppose e1 = $(\lambda y.a)$. Then $(\Gamma,y);\langle\rangle \vdash$ a, so x° does not occur free in a. Therefore, x° does not occur free in e1, so e1[x°]:=e2 = e1. Finally, since $(\Gamma,y);\langle\rangle \vdash$ a, $\Gamma;\Sigma \vdash$ $(\lambda y.a)$ as required.

Suppose e = $(\lambda y^\circ.a)$. This case is similar to the previous case.

□

Lemma B.6. *If* $\Gamma;\Sigma \vdash$ v*, then* $\Gamma;\langle\rangle \vdash$ v.

Proof. We have either v = $(\lambda x.e)$ or v = $(\lambda x^\circ.e)$. If v = $(\lambda x.e)$, then $(\Gamma,x);\langle\rangle \vdash$ e, so $\Gamma;\langle\rangle \vdash$ $(\lambda x.e)$ as required. If v = $(\lambda x^\circ.e)$, then $\Gamma;x^\circ \vdash$ e, so $\Gamma;\langle\rangle \vdash$ $(\lambda x^\circ.e)$ as required. □

Theorem 4.2. *If* $\Gamma;\Sigma \vdash$ e *and* e $\rightarrow_s$ e'*, then* $\Gamma;\Sigma \vdash$ e'.

Proof. By induction on the derivation of e $\rightarrow_s$ e'.

Suppose e $\rightarrow_s$ e' is by SSR-1. Then e = $(\lambda x.a)$v and e' = a[x:=v]. We have that $(\Gamma,x);\langle\rangle \vdash$ a, and $\Gamma;\Sigma \vdash$ v. By Lemma B.6, $\Gamma;\Sigma \vdash$ v. By Lemma B.4, $\Gamma;\langle\rangle \vdash$ a[x:=v]. By Lemma B.3, $\Gamma;\Sigma \vdash$ a[x:=v] as required.

Suppose e $\rightarrow_s$ e' is by SSR-2. Then e = e1 e2 and e' = e1' e2 and e1 $\rightarrow_s$ e1'. We have $\Gamma,\Sigma_1 \vdash$ e1 and $\Gamma,\Sigma_2 \vdash$ e2 and $\Sigma = \Sigma_1 \uplus \Sigma_2$. By induction, $\Gamma,\Sigma_1 \vdash$ e1', so $\Gamma,\Sigma \vdash$ e1' e2 as required.

Suppose e $\rightarrow_s$ e' is by SSR-3. This case is simliar to the previous case.

Suppose e $\rightarrow_s$ e' is by SSR-4. Then e = $(\lambda x^\circ.a)$b and e' = a[x°:=b]. We have $\Gamma;x^\circ \vdash$ a and $\Gamma;\Sigma \vdash$ b. By Lemma B.5, $\Gamma;\Sigma \vdash$ a[x°:=b] as required.

Suppose $\mathtt{e} \rightarrow_s \mathtt{e}'$ is by SSR-5. Then $\mathtt{e} = (\lambda\mathtt{x}.\mathtt{a})\mathtt{y}$ and $\mathtt{e}' = \mathtt{e}[\mathtt{x}:=\mathtt{y}]$. We have $(\Gamma,\mathtt{x});\langle\rangle \vdash \mathtt{a}$ and $\Gamma;\Sigma \vdash \mathtt{y}$. Since y is unlimited, $\mathtt{y} \notin \Sigma$, so $\Gamma;\langle\rangle \vdash \mathtt{y}$. Therefore, by Lemma B.4, $\Gamma;\langle\rangle \vdash \mathtt{a}[\mathtt{x}:=\mathtt{y}]$ as required.

Suppose $\mathtt{e} \rightarrow_s \mathtt{e}'$ is by SSR-6. Then $\mathtt{e} = (\lambda\mathtt{x}.\mathtt{a})$ and $\mathtt{e}' = (\lambda\mathtt{x}.\mathtt{a}')$ and $\mathtt{a} \rightarrow_s \mathtt{a}'$. We have that $(\Gamma,\mathtt{x});\langle\rangle \vdash \mathtt{a}$, so by induction $(\Gamma,\mathtt{x});\langle\rangle \vdash \mathtt{a}'$. Therefore, $\Gamma;\Sigma \vdash (\lambda\mathtt{x}.\mathtt{a}')$ as required.

Suppose $\mathtt{e} \rightarrow_s \mathtt{e}'$ is by SSR-7. Then $\mathtt{e} = (\lambda\mathtt{x}^\circ.\mathtt{a})$ and $\mathtt{e}' = (\lambda\mathtt{x}^\circ.\mathtt{a}')$ and $\mathtt{a} \rightarrow_s \mathtt{a}'$. We have that $\Gamma;\mathtt{x}^\circ \vdash \mathtt{a}$, so by induction $\Gamma;\mathtt{x}^\circ \vdash \mathtt{a}'$. Therefore, $\Gamma;\Sigma \vdash (\lambda\mathtt{x}^\circ.\mathtt{a}')$ as required. □

Lemma B.7. *Suppose* $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{e1}$ *and* $\mathtt{e2} \rightarrow^j \mathtt{v2}$*.Then, if* $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^k \mathtt{v}$ *then* $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k'} \mathtt{v}$ *for some* $k' \leq j + k$*.*

Proof. By induction on the derivation of $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{e1}$.

Suppose $\mathtt{e1} = \mathtt{x}^\circ$. Then $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e2}$ and $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] = \mathtt{v2}$. Since $\mathtt{e2} \rightarrow^j \mathtt{v2}$, we have that $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^j \mathtt{v2}$. Since v2 is a value, $\mathtt{v2} \rightarrow^0 \mathtt{v2}$ so $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^0 \mathtt{v2}$. The conclusion holds with $\mathtt{v} = \mathtt{v2}$, $i = j$, $k = 0$. The inequality $i \leq j + k$ simplifies to $j \leq j$.

Suppose $\mathtt{e1} = \mathtt{y}^*$ where $\mathtt{y}^* \neq \mathtt{x}^\circ$. Then we have $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{y}^*$, a contradiction.

Suppose $\mathtt{e1} = (\lambda\mathtt{y}.\mathtt{e1}')$. Then we have $\langle\rangle;\mathtt{x}^\circ \vdash (\lambda\mathtt{y}.\mathtt{e1}')$, which implies $(\langle\rangle,\mathtt{y});\langle\rangle \vdash \mathtt{e1}'$. Therefore, $\mathtt{x}^\circ$ does occur free in e1.Therefore, $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] = (\lambda\mathtt{y}.\mathtt{e1}')$. Let $\mathtt{v} = (\lambda\mathtt{y}.\mathtt{e1}')$. Then $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^0 \mathtt{v}$ and $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^0 \mathtt{v}$, so $i = k = 0$ and $i \leq j + k$.

Suppose $\mathtt{e1} = (\lambda\mathtt{y}^\circ.\mathtt{e1}')$. Then we have $\langle\rangle;\mathtt{x}^\circ \vdash (\lambda\mathtt{y}^\circ.\mathtt{e1}')$, which implies $\langle\rangle;\mathtt{y}^\circ \vdash \mathtt{e1}'$. Therefore, $\mathtt{x}^\circ$ does occur free in e1.Therefore, $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] = (\lambda\mathtt{y}^\circ.\mathtt{e1}')$. Let $\mathtt{v} = (\lambda\mathtt{y}^\circ.\mathtt{e1}')$. Then $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^0 \mathtt{v}$ and $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^0 \mathtt{v}$, so $i = k = 0$ and $i \leq j + k$.

Suppose $\mathtt{e1} = \mathtt{e3}\ \mathtt{e4}$. Then either $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{e3}$ and $\langle\rangle;\langle\rangle \vdash \mathtt{e4}$, or $\langle\rangle;\langle\rangle \vdash \mathtt{e3}$ and $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{e4}$.

Suppose $\langle\rangle;\mathtt{x}^\circ \vdash \mathtt{e3}$ and $\langle\rangle;\langle\rangle \vdash \mathtt{e4}$. Then $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e3}[\mathtt{x}^\circ:=\mathtt{e2}]\ \mathtt{e4}$, and $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{v2}] = \mathtt{e3}[\mathtt{x}^\circ:=\mathtt{v2}]\ \mathtt{e4}$. Since $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^k \mathtt{v}$ Lemma B.1 states that $\mathtt{e3}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^{k1} (\lambda\mathtt{y}^*.\mathtt{e3}')$, and $\mathtt{e4} \rightarrow^{k2} \mathtt{v4}$, and $\mathtt{e3}'[\mathtt{y}^*:=\mathtt{v4}] \rightarrow^{k3} \mathtt{v}$, and $(\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^{k1+k2+k3+1} \mathtt{v}$, and $k = k1+k2+k3+1$. By induction, $\mathtt{e3}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k1'} (\lambda\mathtt{y}^*.\mathtt{e3}')$ for some $k1' \leq j+k1$. We have that $\mathtt{e3}[\mathtt{x}:=\mathtt{e2}] \rightarrow^{k1'} (\lambda\mathtt{y}^*.\mathtt{e3}')$, and $\mathtt{e4} \rightarrow^{k2} \mathtt{v4}$, and $\mathtt{e3}'[\mathtt{y}^*:=\mathtt{v4}] \rightarrow^{k3} \mathtt{v}$. Therefore, Lemma B.1 states that $\mathtt{e3}[\mathtt{x}^\circ:=\mathtt{e2}]\ \mathtt{e4} \rightarrow^{k1'+k2+k3+1} \mathtt{v}$. Since $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e3}[\mathtt{x}^\circ:=\mathtt{e2}]\ \mathtt{e4}$, we also have that $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k1'+k2+k3+1} \mathtt{v}$. Let $k' = k1' + k2 + k3 + 1 \leq j + k1 + k2 + k3 + 1 = j + k$. The conclusion holds since $k' \leq j + k$.

Suppose $\langle\rangle;\langle\rangle \vdash \mathtt{e3}$ and $\langle\rangle;\mathtt{x} \vdash \mathtt{e4}$. Then $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e3}\ (\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{e2}])$, and $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{v2}] = \mathtt{e3}\ (\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{v2}])$. Since $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^k \mathtt{v}$, Lemma B.1 states that $\mathtt{e3} \rightarrow^{k1} (\lambda\mathtt{y}^*.\mathtt{e3}')$, and $\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{v2}] \rightarrow^{k2} \mathtt{v4}$, and $\mathtt{e3}'[\mathtt{y}^*:=\mathtt{v4}] \rightarrow^{k3} \mathtt{v}$, and $k = k1 + k2 + k3 + 1$. By induction, $\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k2'} \mathtt{v4}$ for some $k2' \leq j + k2$. Therefore, we have $\mathtt{e3} \rightarrow^{k1} (\lambda\mathtt{y}^*.\mathtt{e3}')$, and $\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k2'} \mathtt{v4}$, and $\mathtt{e3}'[\mathtt{y}^*:=\mathtt{v4}] \rightarrow^{k3} \mathtt{v}$, so Lemma B.1 states that $\mathtt{e3}\ (\mathtt{e4}[^\circ:=\mathtt{e2}]) \rightarrow^{k1+k2'+k3+1} \mathtt{v}$. Since $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] = (\mathtt{e3}\ \mathtt{e4})[\mathtt{x}^\circ:=\mathtt{e2}] = \mathtt{e3}\ (\mathtt{e4}[\mathtt{x}^\circ:=\mathtt{e2}])$, we also have that $\mathtt{e1}[\mathtt{x}^\circ:=\mathtt{e2}] \rightarrow^{k1+k2'+k3+1} \mathtt{v}$. Let $k' = k1 + k2' + k3 + 1 \leq j + k1 + k2 + k3 + 1 = j + k$. The conclusion holds since $k' \leq j + k$. □

Lemma B.8. *If* $\Gamma;\mathtt{x}^\circ \vdash \mathtt{e}$ *and* $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{v1}] \rightarrow^* \mathtt{v}$ *and* $\mathtt{e1} \rightarrow^* \mathtt{v1}$*, then* $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{e1}] \rightarrow^* \mathtt{v}$*.*

Proof. By induction on e.

Suppose $\mathtt{e} = \mathtt{x}^\circ$. Then $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{v1}] = \mathtt{v1}$ and $\mathtt{v} = \mathtt{v1}$. But then $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{e1}] \rightarrow^* \mathtt{v1} = \mathtt{v}$ as required.

Suppose $\mathtt{e} = \mathtt{y}^\circ \neq \mathtt{x}^\circ$. Contradiction: $\Gamma;\mathtt{x}^\circ \vdash \mathtt{e}$ cannot be derived.

Suppose $\mathtt{e} = \mathtt{y}$. Then $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{v1}] = \mathtt{e}[\mathtt{x}^\circ:=\mathtt{e1}] = \mathtt{e}$. Trivial.

Suppose $\mathtt{e} = \lambda\mathtt{y}^*.\mathtt{e}'$. Since $\Gamma;\mathtt{x}^\circ \vdash \mathtt{e}$, $\mathtt{x}^\circ \notin \mathtt{FV}(\mathtt{e}')$, so $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{v1}] = \mathtt{e}[\mathtt{x}^\circ:=\mathtt{e1}] = \mathtt{e}$. Trivial.

Suppose $\mathtt{e} = \mathtt{a}\ \mathtt{b}$. Then either $\Gamma;\mathtt{x}^\circ \vdash \mathtt{a}$ and $\Gamma;\langle\rangle \vdash \mathtt{b}$, or else $\Gamma;\langle\rangle \vdash \mathtt{a}$ and $\Gamma;\mathtt{x}^\circ \vdash \mathtt{b}$. The two cases are similar, so we consider on the first. Suppose $\Gamma;\mathtt{x}^\circ \vdash \mathtt{a}$ and $\Gamma;\langle\rangle \vdash \mathtt{b}$. Then $\mathtt{x}^\circ \notin \mathtt{FV}(\mathtt{b})$, so $\mathtt{e}[\mathtt{x}^\circ:=\mathtt{v1}]$

= $\mathtt{a[x^{\circ}:=v1]\ b}$. Now consider $\mathtt{e[x^{\circ}:=v1]} \to^{*} \mathtt{v}$. By Lemma B.1, $\mathtt{a[x^{\circ}:=v1]} \to^{*} \lambda\mathtt{y^{*}.a'}$ and $\mathtt{b} \to^{*} \mathtt{v2}$ and $\mathtt{a'[y^{*}:=v2]} \to^{*} \mathtt{v}$. By induction, $\mathtt{a[x^{\circ}:=e1]} \to^{*} \lambda\mathtt{y^{*}.a'}$, so the conclusion holds. □

Lemma B.9. *If* $\mathtt{e1} \to_{s} \mathtt{e2}$, *then* $\mathtt{e1[x^{*}:=v]} \to_{s} \mathtt{e2[x^{*}:=v]}$.

Proof. By induction on the derivation of $\mathtt{e1} \to_{s} \mathtt{e2}$.

If $\mathtt{e1} \to_{s} \mathtt{e2}$ is by SSR-1, then $\mathtt{e1} = (\lambda\mathtt{y.e)v1}$ and $\mathtt{e2} = \mathtt{e[y:=v1]}$. Without loss of generality, $\mathtt{y} \neq \mathtt{x^{*}}$ and $\mathtt{y} \notin \mathtt{FV(v)}$, so $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{y.e[x^{*}:=v])(v1[x^{*}:=v])}$, and $\mathtt{e2[x^{*}:=v]} = \mathtt{e[y:=v1][x^{*}:=v]}$ = $\mathtt{e[x^{*}:=v][y:=v1[x^{*}:=v]]}$. Since $\mathtt{v1}$ is a value, $\mathtt{v1[x:=v]}$ is a value. Therefore, $\mathtt{e1[x^{*}:=v]}$ = $(\lambda\mathtt{y.e[x^{*}:=v])(v1[x^{*}:=v])} \to_{s} \mathtt{e[x^{*}:=v][y:=v1[x^{*}:=v]]} = \mathtt{e2[x^{*}:=v]}$ by SSR-1.

If $\mathtt{e1} \to \mathtt{e2}$ is by SSR-2 or SSR-3, the result follows by straightforward induction.

If $\mathtt{e1} \to_{s} \mathtt{e2}$ is by SSR-4, then $\mathtt{e1} = (\lambda\mathtt{y^{\circ}.a)b}$. Without loss of generality, $\mathtt{x^{*}} \neq \mathtt{y^{\circ}}$ and $\mathtt{y^{\circ}} \notin \mathtt{FV(a)}$, so $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{y^{\circ}.a[x^{*}:=v])(b[x^{*}:=v])}$. Therefore, $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{y^{\circ}.a[x^{*}:=v])(b[x^{*}:=v])}$ $\to_{s} \mathtt{a[x:=v][y:=b[x:=v]]} = \mathtt{a[y:=b][x:=v]} = \mathtt{e2[x:=v]}$ is derivable by SSR-4..

If $\mathtt{e1} \to_{s} \mathtt{e2}$ is by SSR-5, then $\mathtt{e1} = (\lambda\mathtt{z.e)y}$ and $\mathtt{e2} = \mathtt{e[z:=y]}$. Without loss of generality, assume $\mathtt{z} \notin \mathtt{FV(v)}$. If $\mathtt{x^{*}} = \mathtt{y}$, then $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{z.e[x^{*}:=v])v}$ and $\mathtt{e2[x^{*}:=v]} = \mathtt{e[z:=x][x^{*}:=v]}$ = $\mathtt{e[x^{*}:=v][z:=v]}$. Therefore, $\mathtt{e1[x^{*}:=v]} \to_{s} \mathtt{e2[x^{*}:=v]}$ is derivable using SSR-1. If $\mathtt{x} \neq \mathtt{y}$, then $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{z.e[x^{*}:=v])y}$ and $\mathtt{e2[x^{*}:=v]} = \mathtt{e[z:=y][x^{*}:=v]} = \mathtt{e[x^{*}:=v][z:=y]}$. Therefore, $\mathtt{e1[x^{*}:=v]} \to_{s} \mathtt{e2[x^{*}:=v]}$ is derivable using SSR-5.

If $\mathtt{e1} \to_{s} \mathtt{e2}$ is by SSR-6, then $\mathtt{e1} = (\lambda\mathtt{y.e})$ and $\mathtt{e2} = (\lambda\mathtt{y.e'})$ and $\mathtt{e} \to_{s} \mathtt{e'}$. By induction, $\mathtt{e[x^{*}:=v]}$ $\to_{s} \mathtt{e'[x^{*}:=v]}$. Therefore, $\mathtt{e1[x^{*}:=v]} = (\lambda\mathtt{y.e[x^{*}:=v]}) \to_{s} (\lambda\mathtt{y.e'[x^{*}:=v]}) = \mathtt{e2[x^{*}:=v]}$.

The case for when $\mathtt{e1} \to_{s} \mathtt{e2}$ is by SSR-7 is similar to that for SSR-6. □

Lemma B.10. *If* $\mathtt{e1} \to_{s} \mathtt{e2}$, *then* $\mathtt{e[x^{*}:=e1]} \to_{s}^{*} \mathtt{e[x^{*}:=e2]}$.

Proof. By induction on $\mathtt{e}$.

If $\mathtt{e} = \mathtt{x^{*}}$, then $\mathtt{e[x^{*}:=e1]} = \mathtt{e1}$ and $\mathtt{e[x^{*}:=e2]} = \mathtt{e2}$, and $\mathtt{e1} \to_{s} \mathtt{e2}$ implies $\mathtt{e[x^{*}:=e1]} \to_{s}^{*}$ $\mathtt{e[x^{*}:=e2]}$.

If $\mathtt{e} = \mathtt{y^{*}} \neq \mathtt{x^{*}}$, then $\mathtt{e[x^{*}:=e1]} = \mathtt{y^{*}} = \mathtt{e[x^{*}:=e2]}$, and $\mathtt{e[x^{*}:=e1]} \to_{s}^{*} \mathtt{e[x^{*}:=e2]}$ is derivable by reflexivity.

If $\mathtt{e} = (\lambda\mathtt{y.e'})$, then $\mathtt{e[x^{*}:=e1]} = (\lambda\mathtt{y.e'[x^{*}:=e1]})$ and $\mathtt{e[x^{*}:=e2]} = (\lambda\mathtt{y.e'[x^{*}:=e2]})$. By induction, $\mathtt{e'[x^{*}:=e1]} \to_{s}^{*} \mathtt{e'[x^{*}:=e2]}$. Therefore, $(\lambda\mathtt{y.e'[x^{*}:=e1]}) \to_{s}^{*} (\lambda\mathtt{y.e'[x^{*}:=e2]})$ can be derived by repeated uses of SSR-6.

If $\mathtt{e} = (\lambda\mathtt{y^{\circ}.e'})$, then $\mathtt{e[x^{*}:=e1]} = (\lambda\mathtt{y^{\circ}.e'[x^{*}:=e1]})$ and $\mathtt{e[x^{*}:=e2]} = (\lambda\mathtt{y^{\circ}.e'[x^{*}:=e2]})$. By induction, $\mathtt{e'[x^{*}:=e1]} \to_{s}^{*} \mathtt{e'[x^{*}:=e2]}$. Therefore, $(\lambda\mathtt{y^{\circ}.e'[x^{*}:=e1]}) \to_{s}^{*} (\lambda\mathtt{y^{\circ}.e'[x^{*}:=e2]})$ can be derived by repeated uses of SSR-7.

If $\mathtt{e} = \mathtt{a1\ a2}$, then $\mathtt{e[x^{*}:=e1]} = \mathtt{(a1[x^{*}:=e1])\ (a2[x^{*}:=e1])}$ and $\mathtt{e[x^{*}:=e2]} = \mathtt{(a1[x^{*}:=e2])}$ $\mathtt{(a2[x^{*}:=e2])}$. By induction, $\mathtt{a1[x^{*}:=e1]} \to_{s}^{*} \mathtt{a1[x^{*}:=e2]}$ and $\mathtt{a2[x^{*}:=e1]} \to_{s}^{*} \mathtt{a2[x^{*}:=e2]}$. Then $\mathtt{e[x^{*}:=e1]} \to_{s}^{*} \mathtt{e[x^{*}:=e2]}$ can be derived by repeated uses of SSR-2 and SSR-3. □

Definition B.1. *A step* $\mathtt{e1} \to_{s} \mathtt{e2}$ *is called a* **weak step** *if* $\mathtt{e1} \to_{w} \mathtt{e2}$ *is derivable. Otherwise, it is called a* **strong step**.

Lemma B.11. *If* $\mathtt{e} \to_{s} \mathtt{v}$ *and* $\mathtt{e}$ *is closed, then either* $\mathtt{e} \to_{s} \mathtt{v}$ *is a weak step, or* $\mathtt{e}$ *is a value.*

Proof. By induction on the derivation of $\mathtt{e} \to_{s} \mathtt{v}$.

If $\mathtt{e} \to_{s} \mathtt{v}$ is by SSR-1, then $\mathtt{e} \to_{w} \mathtt{v}$ is derivable by WSSR-1. If $\mathtt{e} \to_{s} \mathtt{v}$ is by SSR-2 or SSR-3, then $\mathtt{v}$ is an application. Contradiction. If $\mathtt{e} \to_{s} \mathtt{v}$ is by SSR-5, then $\mathtt{e}$ is not closed. Contradiction. If $\mathtt{e}$ $\to_{s} \mathtt{v}$ is by SSR-6, then $\mathtt{e} = \lambda\mathtt{x.e'}$, so $\mathtt{e}$ is a value. If $\mathtt{e} \to_{s} \mathtt{v}$ is by SSR-7, then $\mathtt{e} = \lambda\mathtt{x^{\circ}.e'}$, so $\mathtt{e}$ is a value. □

Lemma B.12. *If* $\texttt{e} \to_s^* \texttt{v}$ *and* e *is closed, then either* e *is a value or* $\texttt{e} \to_s^* \texttt{v}$ *contains at least one weak step.*

Proof. By induction on the number n of steps in $\texttt{e} \to_s^* \texttt{v}$.

Suppose $n = 0$. Then e = v.

Suppose $n = n' + 1$. Then $\texttt{e} \to_s^{n'} \texttt{e}' \to_s \texttt{v}$. Since e is closed, so is e′. By Lemma B.11 on $\texttt{e}' \to_s \texttt{v}$, either $\texttt{e}' \to_s \texttt{v}$ is a weak step, or else e′ is a value. If $\texttt{e}' \to_s \texttt{v}$ is a weak step, then $\texttt{e} \to_s^* \texttt{v}$ contains at least one weak step. If e′ is a value, then by induction on $\texttt{e} \to_s^{n-1} \texttt{e}'$, either e is a value or e $\to_s^{n-1}$ e′ contains at least one weak step. If $\texttt{e} \to_s^{n-1} \texttt{e}'$ contains at least one weak step, then so does $\texttt{e} \to_s^n \texttt{v}$. □

Lemma B.13. *Let* e, e1, *and* e′ *be closed terms. If* $\texttt{e} \to_s \texttt{e1} \to_w \texttt{e}'$, *then there exist closed terms* e2 *and* e3 *such that* $\texttt{e} \to_w \texttt{e2} \to_w^* \texttt{e3} \to_s^* \texttt{e}'$.

Proof. **Suppose** $\texttt{e1} \to_w \texttt{e}'$ **is by WSSR-1.**

Then e1 = $(\lambda$x.a)v and e′ = a[x:=v]. Now, consider the step $\texttt{e} \to_s \texttt{e1}$.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-1, then $\texttt{e} \to_w \texttt{e1}$ and we have $\texttt{e} \to_w \texttt{e1} \to_w \texttt{e}'$ as required.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-2, then e = e1′ v and e1′ $\to_s$ $(\lambda$x.a). By Lemma B.11, then either 1) e1′ $\to_w$ $(\lambda$x.a), or 2) e1′ is a value. If 1), we have e1′ v $\to_w$ $(\lambda$x.a) v $\to_w$ a[x:=v] as required. If 2), then e1′ = $(\lambda$x.a′) and it must be that that a′ $\to_s$ a. By Lemma B.9, a′[x:=v] $\to_s$ a[x:=v]. Therefore, e = $(\lambda$x.a′)v $\to_w$ a′[x:=v] $\to_s$ a[x:=v] = e′.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-3, then e = $(\lambda$x.a)b and b $\to_s$ v. By Lemma B.11, b $\to_w$ v or else b is a value v′. In the first case, we have $(\lambda$x.a)b $\to_w$ $(\lambda$x.a)v $\to_w$ a[x:=v] as required. In the second case, we have v′ $\to_s$ v, so by Lemma B.10, a[x:=v′] $\to_s^*$ a[x:=v]. Therefore, e = $(\lambda$x.a)v′ $\to_w$ a[x:=v′] $\to_s^*$ a[x:=v] = e′.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-4, then $\texttt{e} \to_w \texttt{e1}$ by WSSR-4, so $\texttt{e} \to_w \texttt{e1} \to_w \texttt{e}'$ and the conclusion holds.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-5, then e = $(\lambda$x.a)y, which is not closed. Contradiction.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-6, then e1 = $(\lambda$x.e1′), which contradicts that e1 = $(\lambda$x.a)v.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-7, then e1 = $(\lambda$x°.e1′), which contradicts that e1 = $(\lambda$x.a)v.

Suppose $\texttt{e1} \to_w \texttt{e}'$ **is by WSSR-4.** This case is similar to the case for WR-1.

Suppose $\texttt{e1} \to_w \texttt{e}'$ **is by WSSR-2.** Then e1 = a1 b and e′ = a2 b and a1 $\to_w$ a2. Now consider $\texttt{e} \to_s \texttt{e1}$.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-1, then $\texttt{e} \to_w \texttt{e1} \to_w^* \texttt{e}$.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-2, then e = a0 b and a0 $\to_s$ a1. By induction, there exist terms a3,a4 such that a0 $\to_w$ a3 $\to_w^*$ a4 $\to_s^*$ a2. Therefore, e = a0 b $\to_w$ a3 b $\to_w^*$ a4 b $\to_s^*$ a2 b = e′.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-3, then e = a1 b0 and b0 $\to_s$ b. Then e = a1 b0 $\to_w$ a2 b0 $\to_s$ a2 b = e′.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-4, then $\texttt{e} \to_w \texttt{e1}$ by rule WSSR-4, so $\texttt{e} \to_w \texttt{e1} \to_w \texttt{e}'$ and the conclusion holds.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-5, then e is not closed. Contradiction.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-6 or SSR-7, then e1 is a λ-abstraction. Contradicts e1 = a1 b.

Suppose $\texttt{e1} \to_w \texttt{e}'$ **is by WSSR-3.** Then e1 = $(\lambda$x.a)b1 and e′ = $(\lambda$x.a)b2 and b1 $\to_w$ b2. Now consider $\texttt{e} \to_s \texttt{e1}$.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-1, then $\texttt{e} \to_w \texttt{e1} \to_w^* \texttt{e}$.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-2, then e = a0 b1 and a0 $\to_s$ $(\lambda$x.a). By Lemma B.11, either 1) a0 $\to_w$ $(\lambda$x.a), or 2) a0 = $(\lambda$x.a′). If 1), we have a0 b1 $\to_w$ $(\lambda$x.a) b1 $\to_w$ $(\lambda$x.a) b2 as required. If 2), we have a0 = $(\lambda$x.a′), so a′ $\to_s$ a. Therefore, e = $(\lambda$x.a′) b1 $\to_w$ $(\lambda$x.a′) b2 $\to_s$ $(\lambda$x.a) b2 = e′.

If $\texttt{e} \to_s \texttt{e1}$ is by SSR-3, then e = $(\lambda$x.a) b0 and b0 $\to_s$ b1. By induction, there exist closed terms b3,b4 such that b0 $\to_w$ b3 $\to_w^*$ b4 $\to_s^*$ b2. Therefore, e = $(\lambda$x.a) b0 $\to_w$ $(\lambda$x.a) b3 $\to_w^*$ $(\lambda$x.a) b4 $\to_s^*$ $(\lambda$x.a) b2 = e′.

If $\texttt{e} \rightarrow_s \texttt{e1}$ is by SSR-4, then $\texttt{e} \rightarrow_w \texttt{e1}$ by rule WSSR-4, so $\texttt{e} \rightarrow_w \texttt{e1} \rightarrow_w \texttt{e}'$ and the conclusion holds.

If $\texttt{e} \rightarrow_s \texttt{e1}$ is by SSR-5, then $\texttt{e}$ is not closed. Contradiction.

If $\texttt{e} \rightarrow_s \texttt{e1}$ is by SSR-6 or SSR-7, then $\texttt{e1}$ is a λ-abstraction. Contradicts $\texttt{e1} = (\lambda\texttt{x}.\texttt{a})\texttt{b2}$.

Suppose $\texttt{e1} \rightarrow_w \texttt{e}'$ is by WSSR-5. This case is similar to the case for WR-3. □

Lemma B.14. *Suppose $\texttt{e1} \rightarrow_w \texttt{e2}$ is derived without using WSSR-4. Then $\texttt{e1} \rightarrow \texttt{e2}$.*

Proof. By induction on the derivation of $\texttt{e1} \rightarrow_w \texttt{e2}$.

If $\texttt{e1} \rightarrow_w \texttt{e2}$ is by WSSR-1, then $\texttt{e1} \rightarrow \texttt{e2}$ is derivable by CBV-1.

If $\texttt{e1} \rightarrow_w \texttt{e2}$ is by WSSR-2, then $\texttt{e1} = \texttt{a1 b}$ and $\texttt{e2} = \texttt{a2 b}$ and $\texttt{a1} \rightarrow_w \texttt{a2}$ is derived without using WSSR-4. By induction, $\texttt{a1} \rightarrow \texttt{a2}$, so $\texttt{e1} = \texttt{a1 b} \rightarrow \texttt{a2 b} = \texttt{e2}$ is derivable by CBV-2.

If $\texttt{e1} \rightarrow_w \texttt{e2}$ is by WSSR-3, then $\texttt{e1} = (\lambda\texttt{x}.\texttt{e})\ \texttt{a1}$ and $\texttt{e2} = (\lambda\texttt{x}.\texttt{e})\ \texttt{a2}$ and $\texttt{a1} \rightarrow_w \texttt{a2}$ is derived without using WSSR-4. By induction, $\texttt{a1} \rightarrow \texttt{a2}$, so $\texttt{e1} = (\lambda\texttt{x}.\texttt{e})\ \texttt{a1} \rightarrow (\lambda\texttt{x}.\texttt{e})\ \texttt{a2} = \texttt{e2}$ is derivable by CBV-3.

If $\texttt{e1} \rightarrow_w \texttt{e2}$ is by WSSR-5, then $\texttt{e1} = (\lambda\texttt{x}^\circ.\texttt{e})\ \texttt{a1}$ and $\texttt{e2} = (\lambda\texttt{x}^\circ.\texttt{e})\ \texttt{a2}$ and $\texttt{a1} \rightarrow_w \texttt{a2}$ is derived without using WSSR-4. By induction, $\texttt{a1} \rightarrow \texttt{a2}$, so $\texttt{e1} = (\lambda\texttt{x}^\circ.\texttt{e})\ \texttt{a1} \rightarrow (\lambda\texttt{x}^\circ.\texttt{e})\ \texttt{a2} = \texttt{e2}$ is derivable by CBV-5. □

Lemma B.15. *Suppose $\texttt{e1} \rightarrow_w^i \texttt{e2}$ contains no uses of WSSR-4. Then $\texttt{e1} \rightarrow^i \texttt{e2}$.*

Proof. By induction on the length of $\texttt{e1} \rightarrow_w^i \texttt{e2}$.

Suppose $i = 0$. Trivial.

Suppose $i = i' + 1$. Then we have $\texttt{e1} \rightarrow_w \texttt{e1}' \rightarrow_w^{i'} \texttt{e2}$. By induction $\texttt{e1}' \rightarrow^{i'} \texttt{e2}$. Since $\texttt{e1} \rightarrow_w \texttt{e1}'$ is derived without using WSSR-4, Lemma B.14 states $\texttt{e1} \rightarrow \texttt{e1}'$. Therefore, $\texttt{e1} \rightarrow^{i'+1} \texttt{e2}$. □

Lemma B.16. *If $\Gamma;\Sigma \vdash \texttt{e}$ and $\texttt{e} \rightarrow_w^i \texttt{e}'$ and $\texttt{e} \rightarrow^j \texttt{v}$, then there exists an i' such that $\texttt{e}' \rightarrow^{i'} \texttt{v}$ and $i + i' \leq j$.*

Proof. By induction on the number of WSSR-4 reductions in $\texttt{e} \rightarrow_w^i \texttt{e}'$.

If $\texttt{e} \rightarrow_w^i \texttt{e}'$ contains no WSSR-4 reductions, then by Lemma B.15, $\texttt{e} \rightarrow^i \texttt{e}'$. Since call-by-value reduction is deterministic, $\texttt{e} \rightarrow^i \texttt{e}'$ must be a prefix of $\texttt{e} \rightarrow^j \texttt{v}$.

If $\texttt{e} \rightarrow_w^i \texttt{e}'$ contains $n > 0$ WSSR-4 reductions, then there exist $\texttt{e1}$, $\texttt{e2}$, $i1$, and $i2$ such that $\texttt{e} \rightarrow^{i1} \texttt{e1} \rightarrow_w \texttt{e2} \rightarrow_w^{i2} \texttt{e}'$, where $i = i1 + i2 + 1$ and $\texttt{e1} \rightarrow_w \texttt{e2}$ is the first WSSR-4 reduction in the sequence. So $\texttt{e1} = (\lambda\texttt{x}^\circ.\texttt{a})\texttt{b}$, and $\texttt{e2} = \texttt{a}[\texttt{x}^\circ\texttt{:=b}]$.

Since $\texttt{e} \rightarrow^{i1} (\lambda\texttt{x}^\circ.\texttt{a})\texttt{b}$, the derivation of $\texttt{e} \rightarrow^j \texttt{v}$ is of the form $\texttt{e} \rightarrow^{i1} (\lambda\texttt{x}^\circ.\texttt{a})\texttt{b} \rightarrow^{j-i1} \texttt{v}$. By Lemma B.1, there exists a $\texttt{v1}$ such that $\texttt{b} \rightarrow^{j1} \texttt{v1}$ and $\texttt{a}[\texttt{x}^\circ\texttt{:=v1}] \rightarrow^{j-i1-j1-1} \texttt{v}$. By Lemma B.8, $\texttt{a}[\texttt{x}^\circ\texttt{:=b}] \rightarrow^{j2} \texttt{v}$ for some $j2$.

We have that $\texttt{a}[\texttt{x}^\circ\texttt{:=b}] \rightarrow_w^{i2} \texttt{e}'$ contains $n-1$ WSSR-4 reductions, and $\texttt{a}[\texttt{x}^\circ\texttt{:=b}] \rightarrow^{j2} \texttt{v}$. Therefore, it follows by induction that $\texttt{e}' \rightarrow^{i'} \texttt{v}$ and $i2 + i' \leq j2$.

Since $\texttt{b} \rightarrow^{j1} \texttt{v1}$, and $\texttt{a}[\texttt{x}^\circ\texttt{:=v1}] \rightarrow^{j-i1-j1-1} \texttt{v}$, and $\texttt{a}[\texttt{x}^\circ\texttt{:=b}] \rightarrow^{j2} \texttt{v}$, if follows by Lemma B.7 that $j2 \leq j1 + j - i1 - j1 - 1 = j - i1 - 1$. Therefore, $i2 + i' \leq j - i1 - 1$. Therefore, $i + i' = i1 + i2 + 1 + i' \leq i1 + 1 + (j - i1 - 1) = j$. □

Corollary B.1. *If $\texttt{e} \rightarrow_w^i \texttt{v1}$ and $\texttt{e} \rightarrow^j \texttt{v2}$, then $i \leq j$ and $\texttt{v1} = \texttt{v2}$.*

Proof. Suppose $\texttt{e} \rightarrow_w^i \texttt{v1}$ and $\texttt{e} \rightarrow^j \texttt{v2}$. By Lemma B.16, there exists an i' such that $\texttt{v1} \rightarrow^{i'} \texttt{v2}$ and $i + i' \leq j$. But i' must equal 0, so $i \leq j$ and $\texttt{v1} = \texttt{v2}$. □

Lemma B.17. *Suppose* $\langle\rangle;\langle\rangle \vdash \mathsf{a}$, *and* $\mathsf{a} \to_s^* \mathsf{b}$ *is contains* i *weak steps, and* $\mathsf{a} \to^n \mathsf{v}$. *Then there exists a reduction sequence* $\mathsf{a} \to_w^j \mathsf{a}' \to_s^* \mathsf{b}$ *such that* $i \leq j$ *and* $\mathsf{a}' \to_s^* \mathsf{b}$ *contains no weak steps.*

Proof. For a given sequence of steps $\mathsf{e1} \to_s^* \mathsf{e2}$, let p be the number of weak steps before the first strong step; and let q be the length of the first subsequence of strong steps. By Lemma B.16, $p \leq n$, so $n - p \geq 0$. We proceed by induction on the lexicographic order of $(n-p, q)$.

Suppose $n - p = 0$, so $p = n$. Then there is an a' such that $\mathsf{a} \to_w^n \mathsf{a}' \to_s^* \mathsf{b}$. Since $\mathsf{a} \to_w^n \mathsf{a}'$ and $\mathsf{a} \to^n \mathsf{v}$, Lemma B.16 states that there is an n' such that $\mathsf{a}' \to^{n'} \mathsf{v}$ and $n + n' \leq n$. Then $n' = 0$, so $\mathsf{a}' = \mathsf{v}$. We have $\mathsf{v} \to_s^* \mathsf{b}$, which contains no weak steps because v is a value. Therefore $i = n$. Let $j = n$, so $j \leq i$. Since $\mathsf{a} \to_w^j \mathsf{v} \to_s^* \mathsf{b}$ and $j \leq i$ and $\mathsf{v} \to_s^* \mathsf{b}$ contains no weak steps, the conclusion holds.

Suppose $n - p > 0$, so $p < n$. If $q = 0$, then we have $\mathsf{a} \to_w^p \mathsf{b}$, and the conclusion holds. If $q \geq 1$, we have $\mathsf{a} \to_w^p \mathsf{a1} \to_s^q \mathsf{b}$. If $\mathsf{a1} \to_s^q \mathsf{b}$ contains only strong steps, then the conclusion holds with $\mathsf{a}' = \mathsf{a1}$. Otherwise, we have $\mathsf{a} \to_w^p \mathsf{a1} \to_s^{q-1} \mathsf{a2} \to_s \mathsf{a3} \to_w \mathsf{a4} \to_s^* \mathsf{b}$, where $\mathsf{a3} \to_w \mathsf{a4}$ is the first weak reduction step that follows the first sequence of strong reduction steps.

Since $\mathsf{a2} \to_s \mathsf{a3} \to_w \mathsf{a4}$, Lemma B.13 states that there exist terms $\mathsf{c1}$ and $\mathsf{c2}$ such that $\mathsf{a2} \to_w \mathsf{c1} \to_w^* \mathsf{c2} \to_s^* \mathsf{a4}$.

Replace $\mathsf{a2} \to_s \mathsf{a3} \to_w \mathsf{a4}$ with $\mathsf{a2} \to_w \mathsf{c1} \to_w^* \mathsf{c2} \to_s^* \mathsf{a4}$ in the full sequence. If the original sequence had i weak steps, the new sequence has $i' \geq i$ weak steps.

Suppose now that $q = 1$. Then we added $k \geq 1$ reduction steps to the initial sequence of weak reduction steps, so the new sequence will have measure $(n - (p + k), l)$ for some $l \geq 0$. Since $n - (p + k) < n - p$, we have that $(n - (p + k), l) < (n, q)$ Therefore, by induction, there exists a weak reduction sequence $\mathsf{a} \to_w^j \mathsf{a}' \to_s^* \mathsf{b}$ such that $j \geq i' \geq i$ and $\mathsf{a}' \to_s^* \mathsf{b}$ contains no weak steps.

Suppose instead that $q > 1$. Then the new sequence will have measure $(n - p, q - 1)$. Since $q - 1 < q$, $(n - p, q - 1) < (n - p, q)$. Therefore, by induction, there exists a weak reduction sequence $\mathsf{a} \to_w^j \mathsf{a}' \to_s^* \mathsf{b}$ such that $j \geq i' \geq i$ and $\mathsf{a}' \to_s^* \mathsf{b}$ contains no weak steps. □

Lemma B.18 (Preservation of termination behavior). *Suppose* $\langle\rangle;\langle\rangle \vdash \mathsf{a}$, *and Suppose* $\mathsf{a} \to^* \mathsf{v}$ *and* $\mathsf{a} \to_s^* \mathsf{a1}$. *Then there exists a value* $\mathsf{v1}$ *such that* $\mathsf{a1} \to^* \mathsf{v1}$.

Proof. By contradiction.

Suppose there is no such value $\mathsf{v1}$. Then for any n, there exists a term $\mathsf{a2}$ such that $\mathsf{a1} \to^n \mathsf{a2}$. Now suppose $\mathsf{a} \to^* \mathsf{v}$ contains i steps. Then $\mathsf{a} \to_s^* \mathsf{a1} \to_s^{i+1} \mathsf{a2}$ contains at least $i + 1$ steps of weak reduction. By Lemma B.17, there exists a reduction sequence $\mathsf{a} \to_w^j \mathsf{a3} \to_s^* \mathsf{a2}$ such that $i + 1 \leq j$. But by Lemma B.16, $\mathsf{a} \to_w^j \mathsf{a3}$ and $\mathsf{a} \to^i \mathsf{v}$ implies $\mathsf{a3} \to^k \mathsf{v}$ for some k such that $j + k \leq i$. Therefore, we have $j \leq j + k \leq i < i + 1 \leq j$. Contradiction.

□

Theorem 4.5. *If* $\langle\rangle;\langle\rangle \vdash \mathsf{e}$ *and* $\mathsf{e} \to_s^* \mathsf{e}'$ *and* $\mathsf{e} \to^i \mathsf{v}$, *then there exists an* $i' \leq i$ *and a value* v' *such that* $\mathsf{e}' \to^{i'} \mathsf{v}'$, *and* $\mathsf{v} \to_s^* \mathsf{v}'$.

Proof. By Lemma B.18, there exists an i' and a value v' such that $\mathsf{e}' \to^{i'} \mathsf{v}'$.

Therefore, we have that $\mathsf{e} \to_s^* \mathsf{e}' \to^{i'} \mathsf{v}'$, and since each $\to$ step is also a $\to_w$ step, we have $\mathsf{e} \to_s^* \mathsf{e}' \to_w^{i'} \mathsf{v}'$. Suppose $\mathsf{e} \to_s^* \mathsf{e}' \to_w^{i'} \mathsf{v}'$ contains $i1$ steps of weak reduction. It is clear that $i' \leq i1$. By Lemma B.17, there exists an a' such that $\mathsf{e} \to_w^j \mathsf{a}' \to_s^* \mathsf{v}'$ for some j, and $i1 \leq j$, and $\mathsf{a}' \to_s^* \mathsf{v}'$ contains no weak steps. Since e is closed, a' is closed, so by Lemma B.12 on $\mathsf{a}' \to_s^* \mathsf{v}'$, a' is a value v''. By Corollary B.1, $j \leq i$ and $\mathsf{v}'' = \mathsf{v}$. Therefore, $i' \leq i1 \leq j \leq i$, and $\mathsf{v} \to_s^* \mathsf{v}'$. □

Lemma 5.2. *For any self-interpreter* u *for a representation function* $\widehat{\cdot}$, *and for any term* p, *if* $\mathsf{u}\ \widehat{\mathsf{p}} \Rightarrow_s \mathsf{p}$, *then* `mix` *is Jones-optimal for* u *and* $time_{cbv}$.

$$\begin{tikzcd}
& \mathsf{e}' \arrow[rr, dashed, "\text{Lemma B.18}", "i'"'] && \mathsf{v}' \\
\mathsf{e} \arrow[ru, "*_s"] \arrow[rr, "i"] \arrow[rr, dashed, bend right, "j", "\text{Lemma B.17, Corollary B.1}"'] && \mathsf{v} \arrow[ru, dashed, "*_s"'] &
\end{tikzcd}$$

$$i' \leq j \leq i$$

Fig. 23. Illustration of Theorem 4.5

$$\frac{}{\mathsf{x}^* \rhd \mathsf{x}^*}$$

$$\frac{\mathsf{e} \rhd \mathsf{q}}{\lambda\mathsf{x}^*.\mathsf{e} \rhd \mathsf{abs}\ (\lambda\mathsf{x}^*.\mathsf{q})}$$

$$\frac{\mathsf{e1} \rhd \mathsf{q1} \qquad \mathsf{e2} \rhd \mathsf{q2}}{\mathsf{e1}\ \mathsf{e2} \rhd \mathsf{app}\ \mathsf{q1}\ \mathsf{q2}}$$

$$\frac{\mathsf{e} \rhd \mathsf{q}}{\overline{\mathsf{e}} = \lambda\mathsf{abs}.\ \lambda\mathsf{app}.\ \mathsf{q}}$$

$$\mathsf{u} = \lambda\mathsf{e}^\circ.\ \mathsf{e}^\circ\ (\lambda\mathsf{x}^\circ.\mathsf{x}^\circ)\ (\lambda\mathsf{x}^\circ.\mathsf{x}^\circ)$$

Fig. 24. Mogensen's Church-encoded represenation self-interpreter with affine variable annotations.

Proof. Let u be an unannotated self-interpreter, let p be an unannotated term, and suppose $\mathsf{u}\ \overline{\mathsf{p}} \Rightarrow_s \mathsf{p}$. We want to show that for any unannotated term d, $time_{cbv}(\mathsf{p}'\ \mathsf{d}) \leq time_{cbv}(\mathsf{p}\ \mathsf{d})$, where $\mathsf{mix}\ \overline{\mathsf{u}}\ \overline{\overline{\mathsf{p}}} \equiv_\beta \overline{\mathsf{p}'}$. By the specification of mix, we have that $\overline{\mathsf{p}'} \equiv_\beta \overline{erase(\mathsf{NF}_s(annotate(\mathsf{u}\ \overline{\mathsf{p}})))}$. By the definition of $\overline{\cdot}$, $\mathsf{p}' \equiv_\alpha erase(\mathsf{NF}_s(annotate(\mathsf{u}\ \overline{\mathsf{p}})))$. Let p'' be the annotated term $\mathsf{NF}_s(annotate(\mathsf{u}\ \overline{\mathsf{p}}))$. Then $erase(\mathsf{p}'') = \mathsf{p}'$. Since $\mathsf{u}\ \overline{\mathsf{p}} \Rightarrow_s \mathsf{p}$, we have that $annotate(\mathsf{u}\ \overline{\mathsf{p}}) \to_s^* \mathsf{p}'''$ for some annotated term p''' such that $erase(\mathsf{p}''') = \mathsf{p}$. Since $\to_s$ is confluent, $\mathsf{p}'' = \mathsf{NF}_s(annotate(\mathsf{u}\ \overline{\mathsf{p}})) = \mathsf{NF}_s(\mathsf{p}''')$. Therefore, $\mathsf{p}''' \to_s^* \mathsf{p}''$. By Corollary 4.6, $time_{cbv}(\mathsf{p}''\ annotate(\mathsf{d})) \leq time_{cbv}(\mathsf{p}'''\ annotate(\mathsf{d}))$. By Lemma 5.1, $time_{cbv}(erase(\mathsf{p}''\ annotate(\mathsf{d}))) \leq time_{cbv}(\mathsf{p}'''\ annotate(\mathsf{d}))$. Therefore, $time_{cbv}(\mathsf{p}'\ \mathsf{d}) \leq time_{cbv}(\mathsf{p}\ \mathsf{d})$ as required. □

C JONES OPTIMALITY FOR ULC

C.1 Church encoding

Figure 24 shows Mogensen's Church encoding and self-interpreter u with affine variable annotations. Our theorems do not depend on how the term is annotated before quotation – it can be unannotated or maximally annotated, and the theorems hold.

Lemma C.1. *If* $\langle\rangle; \langle\rangle \vdash \mathsf{e}$ *and* $\mathsf{e} \rhd \mathsf{q}$, *then* $(\langle\rangle,\mathsf{abs},\mathsf{app}); \langle\rangle \vdash \mathsf{q}$

Note: abs and app may be affine or not, depending on how many abstractions or applications are in e. Opimality does not depend on whether abs and app are affine or universal. We assume they are universal.

Proof. Straightforward induction. □

Lemma C.2. *If* $\langle\rangle;\langle\rangle \vdash$ e, $\langle\rangle;\langle\rangle \vdash \overline{\text{e}}$

Proof. Straightforward by Lemma C.1. □

Lemma C.3. $\langle\rangle;\langle\rangle \vdash$ `unquote`

Proof. Trivial. □

Lemma C.4. *If* e q, *then* q[abs := (λx°.x°), app := (λx°.x°)] $\to_s^*$ *e.*

Proof. By induction on e.

Suppose e = x*. Then q = x^*, so q[abs := (λx°.x°), app := (λx°.x°)] = e.

Suppose e = (λy*.e′). Then q = abs (λy*.q′) and e′ q′. By induction, q′[abs := (λx°.x°), app := (λx°.x°)] $\to_s^*$ e′. Therefore, we can derive:

q[abs := (λx°.x°), app := (λx°.x°)]
= (abs (λy*.q′))[abs := (λx°.x°), app := (λx°.x°)]
= (λx°.x°) (λy*.q′[abs := (λx°.x°), app := (λx°.x°)])
$\to_s^*$ (λx°.x°) (λy*.e′)
$\to_s$ (λy*.e′)
= e

Suppose e = e1 e2. Then q = app q1 q2 and e1 q1 and e2 q2. By induction, q1[abs := (λx°.x°), app := (λx°.x°)] $\to_s^*$ e1 and q2[abs := (λx°.x°), app := (λx°.x°)] $\to_s^*$ e2. Therefore, we can derive:

q[abs := (λx°.x°), app := (λx°.x°)]
= (app q1 q2)[abs := (λx°.x°), app := (λx°.x°)]
= (λx°.x°) (q1[abs := (λx°.x°), app := (λx°.x°)]) (q2[abs := (λx°.x°), app := (λx°.x°)])
$\to_s^*$ (λx°.x°) e1 (q2[abs := (λx°.x°), app := (λx°.x°)])
$\to_s^*$ (λx°.x°) e1 e2
$\to_s$ e1 e2
= e

□

Lemma 5.3. *For any closed term* e, u $\overline{\text{e}} \Longrightarrow_s$ e.

Proof. By definition, $\overline{\text{e}}$ = λabs.λapp.q, where e q. By Lemma C.4, q[abs := (λx°.x°), app := (λx°.x°)] $\to_s^*$ e. Therefore, we can derive:

unquote $\overline{\text{e}}$
= (λe°. e° (λx°.x°) (λx°.x°)) $\overline{\text{e}}$
$\to_s$ $\overline{\text{e}}$ (λx°.x°) (λx°.x°)
= (λabs.λapp.q) (λx°.x°) (λx°.x°)
$\to_s^2$ q[abs := (λx°.x°), app := (λx°.x°)]
$\to_s^*$ e

□

C.2 Mogensen-Scott encoding

In this section we prove Lemma 5.5, which states that for any term e, `msint` $\widehat{\text{e}} \Longrightarrow_s$ e. We proceed by defining annotated versions of `msint` and then proving for the annotated version that `msint` $\widehat{\text{e}}$ $\to_s^*$ e.

Define:

$$\begin{aligned}
\texttt{fix}_{\texttt{f}} &= (\lambda\texttt{x}.\ \texttt{f}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\ (\lambda\texttt{x}.\ \texttt{f}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\\
\texttt{fix} &= \lambda\texttt{f}.\ \texttt{fix}_{\texttt{f}}\\
\\
\texttt{u} &= \lambda\texttt{u}.\ \lambda\texttt{e}^\circ.\ \texttt{e}^\circ\ (\lambda\texttt{x}^\circ.\texttt{x}^\circ)\ (\lambda\texttt{f}.\lambda\texttt{x}.\texttt{u}\ (\texttt{f}\ ((\lambda\texttt{y}^\circ.\texttt{x})\texttt{x})))\ (\lambda\texttt{f}.\lambda\texttt{x}^\circ.\texttt{u}\ \texttt{f}\ (\texttt{u}\ \texttt{x}^\circ))\\
\texttt{msint} &= \texttt{fix}\ \texttt{u}
\end{aligned}$$

Lemma C.5. $\langle\rangle;\langle\rangle \vdash \texttt{fix}$

PROOF. Straightforward. □

Lemma C.6. $\texttt{msint} \rightarrow_s \texttt{fix}_{\texttt{u}}$

PROOF.

$$\begin{aligned}
&\texttt{msint}\\
=\ &\texttt{fix}\ \texttt{u}\\
\rightarrow_s\ &(\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\ (\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\\
=\ &\texttt{fix}_{\texttt{u}}
\end{aligned}$$

□

Lemma C.7. $\texttt{fix}_{\texttt{u}} \rightarrow_s \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)$

PROOF.

$$\begin{aligned}
&\texttt{fix}_{\texttt{u}}\\
=\ &(\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\ (\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\\
\rightarrow_s\ &\texttt{u}\ (\lambda\texttt{y}^\circ.\ (\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\ (\lambda\texttt{x}.\ \texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{x}\ \texttt{x}\ \texttt{y}^\circ))\ \texttt{y}^\circ)\\
=\ &\texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)
\end{aligned}$$

□

Lemma C.8. *For any* e, $\texttt{fix}_{\texttt{u}}\ \widehat{\texttt{e}} \rightarrow_s^* \texttt{e}$.

PROOF. By induction on e.

The annotations on $\widehat{\texttt{e}}$ don't affect this proof, so we will leave all variables its variables unannotated.

Suppose e = x. Then $\widehat{\texttt{e}} = \lambda\texttt{var}.\lambda\texttt{abs}.\lambda\texttt{app}.\texttt{var}\ \texttt{x}$. Therefore, we can derive:

$$\begin{aligned}
&\texttt{fix}_{\texttt{u}}\ \widehat{\texttt{e}}\\
\rightarrow_s\ &\texttt{u}\ (\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)\ \widehat{\texttt{e}}\\
\rightarrow_s^2\ &\widehat{\texttt{e}}\ (\lambda\texttt{x}^\circ.\texttt{x}^\circ)\\
&\quad(\lambda\texttt{f}.\lambda\texttt{x}.(\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)\ (\texttt{f}\ ((\lambda\texttt{y}^\circ.\texttt{x})\texttt{x})))\\
&\quad(\lambda\texttt{f}.\lambda\texttt{x}^\circ.(\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)\ \texttt{f}\ ((\lambda\texttt{y}^\circ.\ \texttt{fix}_{\texttt{u}}\ \texttt{y}^\circ)\ \texttt{x}^\circ))\\
\rightarrow_s^3\ &(\lambda\texttt{x}^\circ.\texttt{x}^\circ)\ \texttt{x}\\
\rightarrow_s\ &\texttt{x}\\
=\ &\texttt{e}
\end{aligned}$$

Suppose $\mathtt{e} = \lambda\mathtt{x}.\mathtt{e}'$. Then $\widehat{\mathtt{e}} = \lambda\mathtt{var}.\lambda\mathtt{abs}.\lambda\mathtt{app}.\mathtt{abs}\ (\lambda\mathtt{x}.\widehat{\mathtt{e}'})$. By induction, $\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}'} \to_s^* \mathtt{e}'$. Therefore, can derive:

$$\begin{aligned}
&\quad \mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}}\\
&\to_s \mathtt{u}\ (\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \widehat{\mathtt{e}}\\
&\to_s^2 \widehat{\mathtt{e}}\ (\lambda\mathtt{x}^\circ.\mathtt{x}^\circ)\\
&\qquad (\lambda\mathtt{f}.\lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ (\mathtt{f}\ ((\lambda\mathtt{y}^\circ.\mathtt{x})\mathtt{x})))\\
&\qquad (\lambda\mathtt{f}.\lambda\mathtt{x}^\circ.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{f}\ ((\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{x}^\circ))\\
&\to_s^3 (\lambda\mathtt{f}.\lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ (\mathtt{f}\ ((\lambda\mathtt{y}^\circ.\mathtt{x})\mathtt{x})))\ (\lambda\mathtt{x}.\widehat{\mathtt{e}'})\\
&\to_s \lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ ((\lambda\mathtt{x}.\widehat{\mathtt{e}'})\ ((\lambda\mathtt{y}^\circ.\mathtt{x})\mathtt{x}))\\
&\to_s \lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ ((\lambda\mathtt{x}.\widehat{\mathtt{e}'})\ \mathtt{x})\\
&\to_s \lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \widehat{\mathtt{e}'}\\
&\to_s \lambda\mathtt{x}.\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}'}\\
&\to_s^* \lambda\mathtt{x}.\mathtt{e}'\\
&= \mathtt{e}
\end{aligned}$$

Suppose $\mathtt{e} = \mathtt{e1}\ \mathtt{e2}$. Then $\widehat{\mathtt{e}} = \lambda\mathtt{var}.\lambda\mathtt{abs}.\lambda\mathtt{app}.\mathtt{app}\ \widehat{\mathtt{e1}}\ \widehat{\mathtt{e2}}$. By induction, $\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e1}} \to_s^* \mathtt{e1}$ and $\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e2}} \to_s^* \mathtt{e2}$. Therefore, we can derive:

$$\begin{aligned}
&\quad \mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}}\\
&\to_s \mathtt{u}\ (\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \widehat{\mathtt{e}}\\
&\to_s^2 \widehat{\mathtt{e}}\ (\lambda\mathtt{x}^\circ.\mathtt{x}^\circ)\\
&\qquad (\lambda\mathtt{f}.\lambda\mathtt{x}.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ (\mathtt{f}\ ((\lambda\mathtt{y}^\circ.\mathtt{x})\mathtt{x})))\\
&\qquad (\lambda\mathtt{f}.\lambda\mathtt{x}^\circ.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{f}\ ((\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{x}^\circ))\\
&\to_s^3 (\lambda\mathtt{f}.\lambda\mathtt{x}^\circ.(\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{f}\ ((\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \mathtt{x}^\circ))\ \widehat{\mathtt{e1}}\ \widehat{\mathtt{e2}}\\
&\to_s^2 (\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \widehat{\mathtt{e1}}\ ((\lambda\mathtt{y}^\circ.\ \mathtt{fix}_\mathtt{u}\ \mathtt{y}^\circ)\ \widehat{\mathtt{e2}})\\
&\to_s^2 \mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e1}}\ (\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e2}})\\
&\to_s^* \mathtt{e1}\ (\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e2}})\\
&\to_s^* \mathtt{e1}\ \mathtt{e2}\\
&= \mathtt{e}
\end{aligned}$$

□

Lemma 5.5. *For any term* $\mathtt{e}$, $\mathtt{msint}\ \widehat{\mathtt{e}} \Longrightarrow_s \mathtt{e}$.

Proof. By Lemma C.6, $\mathtt{msint}\ \widehat{\mathtt{e}} \to_s \mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}}$, and by Lemma C.8, $\mathtt{fix}_\mathtt{u}\ \widehat{\mathtt{e}} \to_s^* \mathtt{e}$. □

C.3 deBruijn Indices

Figure 13 defines our tagless-final representation using deBruijn indices to represent variables.

Here are annotated versions of fst and snd:

$\mathtt{fst} = (\lambda\mathtt{x}^\circ.\ \mathtt{x}^\circ\ (\lambda\mathtt{y}.\lambda\mathtt{z}^\circ.\mathtt{y}))$

$\mathtt{snd} = (\lambda\mathtt{x}^\circ.\ \mathtt{x}^\circ\ (\lambda\mathtt{y}^\circ.\lambda\mathtt{z}^\circ.\mathtt{z}^\circ))$

Define the mapping $\overline{\cdot}$ from contexts of unannotated variables to annotated terms as follows:

$\overline{\langle\rangle} = (\lambda\mathtt{y}^\circ.\mathtt{y}^\circ)$

$\overline{\Gamma,\mathtt{x}} = \lambda\mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \overline{\Gamma}$

Note that every unannotated variable x becomes unlimited in the annotated term.

Lemma C.1. *If* $\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{q}$, *then* $\mathtt{q}\ \overline{\Gamma} \to_s^* \mathtt{x}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{q}$.

Suppose $\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{q}$ is by the first rule. Then $\Gamma = \Gamma', \mathtt{x}$ and $\mathtt{q} = \mathtt{fst}$. We have:

$$\begin{array}{ll}
& \mathtt{q}\ \overline{\Gamma} \\
= & \mathtt{fst}\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'}) \\
= & (\lambda \mathtt{x}^\circ.\ \mathtt{x}^\circ\ (\lambda \mathtt{y}.\lambda \mathtt{z}^\circ.\mathtt{y})) \\
& \quad (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'}) \\
\to_s & (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'})\ (\lambda \mathtt{y}.\lambda \mathtt{z}^\circ.\mathtt{y}) \\
\to_s & (\lambda \mathtt{y}.\lambda \mathtt{z}^\circ.\mathtt{y})\ \mathtt{x}_i\ \overline{\Gamma'} \\
\to_s^2 & \mathtt{x}_i
\end{array}$$

where the last step relies on that $\mathtt{x}_i$ is unlimited.

Suppose $\Gamma \vdash \mathtt{x} \blacktriangleright \mathtt{q}$ is by the second rule. Then $\Gamma = \Gamma', \mathtt{y}$, and Then $\Gamma' \vdash \mathtt{x} \blacktriangleright \mathtt{q}'$, and and $\mathtt{q} = (\lambda \mathtt{p}^\circ.\ \mathtt{q}'\ (\mathtt{snd}\ \mathtt{p}^\circ))$. By induction, $\mathtt{q}'\ \overline{\Gamma'} \to_s^* \mathtt{x}$. Therefore, we have:

$$\begin{array}{ll}
& \mathtt{q}\ \overline{\Gamma} \\
= & (\lambda \mathtt{p}^\circ.\ \mathtt{q}'\ (\mathtt{snd}\ \mathtt{p}^\circ)) \\
& \quad (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'}) \\
\to_s & \mathtt{q}'\ (\mathtt{snd}\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'})) \\
\to_s & \mathtt{q}'\ ((\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}_i\ \overline{\Gamma'})\ (\lambda \mathtt{y}^\circ.\lambda \mathtt{z}^\circ.\mathtt{z}^\circ)) \\
\to_s & \mathtt{q}'\ ((\lambda \mathtt{y}^\circ.\lambda \mathtt{z}^\circ.\mathtt{z}^\circ)\ \mathtt{x}_i\ \overline{\Gamma'}) \\
\to_s^2 & \mathtt{q}'\ \overline{\Gamma'} \\
\to_s^* & \mathtt{x}
\end{array}$$

□

Lemma C.2. *If* $\Gamma \vdash \mathtt{e} \rhd \mathtt{q}$, *then there exists a value* $\mathtt{v}$ *such that* $\mathtt{q[var:=dbVar,abs:=dbAbs,app:=dbApp]} \to_s^* \mathtt{v}$ *and* $\mathtt{v}\ \overline{\Gamma} \to_s^* \mathtt{e}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{e} \rhd \mathtt{q}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $\Gamma \vdash \mathtt{e} \blacktriangleright \mathtt{q}'$, and $\mathtt{q} = \mathtt{var}\ \mathtt{q}'$. Let $\mathtt{v} = \mathtt{q}'$. We have that:

$$\begin{array}{ll}
& \mathtt{q[var:=dbVar,abs:=dbAbs,app:=dbApp]} \\
= & \mathtt{dbVar}\ \mathtt{q}' \\
= & (\lambda \mathtt{f}.\lambda \mathtt{e}^\circ.\mathtt{f}\ \mathtt{e})\ \mathtt{q}' \\
\to_s & \mathtt{q}'
\end{array}$$

where the last step is because $\mathtt{q}'$ is a value. The conclusion follows from Lemma C.1, which says $\mathtt{q}'\ \overline{\Gamma} \to_s^* \mathtt{x}$.

Suppose $\mathtt{e} = (\lambda \mathtt{x}.\mathtt{e}')$. Then $\Gamma, \mathtt{x} \vdash \mathtt{e}' \rhd \mathtt{q}'$, and $\mathtt{q} = \mathtt{abs}\ \mathtt{q}'$. By induction, $\mathtt{q'[var:=dbVar,abs:=dbAbs,app:=dbApp]} \to_s^* \mathtt{v}'$, and $\mathtt{v}'\ \overline{\Gamma, \mathtt{x}} \to_s^* \mathtt{e}'$. Therefore, we have:

$$\begin{array}{ll}
& \mathtt{q[var:=dbVar,abs:=dbAbs,app:=dbApp]} \\
= & \mathtt{dbAbs}\ (\mathtt{q'[var:=dbVar,abs:=dbAbs,app:=dbApp]}) \\
\to_s^* & \mathtt{dbAbs}\ \mathtt{v}' \\
= & (\lambda \mathtt{b}.\lambda \mathtt{e}.\lambda \mathtt{x}.\mathtt{b}\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \mathtt{e}))\ \mathtt{v}' \\
\to_s & (\lambda \mathtt{e}.\lambda \mathtt{x}.\mathtt{v}'\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \mathtt{e}))
\end{array}$$

Let $\mathtt{v} = (\lambda \mathtt{e}.\lambda \mathtt{x}.\mathtt{v}'\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \mathtt{e}))$. Then, we have:

$$\begin{array}{ll}
& \mathtt{v}\ \overline{\Gamma} \\
= & (\lambda \mathtt{e}.\lambda \mathtt{x}.\mathtt{v}'\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \mathtt{e}))\ \overline{\Gamma} \\
\to_s & (\lambda \mathtt{x}.\mathtt{v}'\ (\lambda \mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{x}\ \overline{\Gamma}))\quad (\overline{\Gamma} \text{ is a value}) \\
= & (\lambda \mathtt{x}.\mathtt{v}'\ \overline{\Gamma'})\quad (\mathtt{x} \text{ is unlimited}) \\
\to_s^* & (\lambda \mathtt{x}.\mathtt{e}') \\
= & \mathtt{e}
\end{array}$$

Suppose $\mathtt{e} = \mathtt{e1}\ \mathtt{e2}$. Then $\Gamma \vdash \mathtt{e1} \rhd \mathtt{q1}$, and $\Gamma \vdash \mathtt{e2} \rhd \mathtt{q2}$, and $\mathtt{q} = \mathtt{app}\ \mathtt{q1}\ \mathtt{q2}$. By induction, $\mathtt{q1[var:=dbVar,abs:=dbAbs,app:=dbApp]} \to_s^* \mathtt{v1}$ and $\mathtt{v1}\ \overline{\Gamma} \to_s^* \mathtt{e1}$, and q2[var:=dbVar,abs:=dbAbs,app:=dbApp] $\to_s^*$ v2 and v2 $\overline{\Gamma}$ $\to_s^*$ e2. Therefore, we have:

q[var:=dbVar,abs:=dbAbs,app:=dbApp]
= dbApp (q1[var:=dbVar,abs:=dbAbs,app:=dbApp])
(q2[var:=dbVar,abs:=dbAbs,app:=dbApp])
$\to_s^*$ dbApp v1 v2
= (λa.λb.λe.a e (b e)) v1 v2
$\to_s^2$ (λe.v1 e (v2 e))
Let v = (λe.v1 e (v2 e)). Then we have that:
v $\overline{\Gamma}$
= (λe.v1 e (v2 e)) $\overline{\Gamma}$
$\to_s^2$ v1 $\overline{\Gamma}$ (v2 $\overline{\Gamma}$)
$\to_s^*$ e1 e2 □

Lemma 5.7. *For any closed term* e*,* dbint $\widehat{\texttt{e}}$ $\Longrightarrow_s$ e.

Proof. We have that $\widehat{\texttt{e}}$ = (λvar.λabs.λapp.q) and ⟨⟩ ⊢ e ▷ q. By Lemma C.2, q[var:=dbVar,abs:=dbAbs,app:=dbApp] $\overline{⟨⟩}$ $\to_s^*$ e. Therefore, we have:

dbint $\widehat{\texttt{e}}$
= (λq°. q dbVar dbAbs dbApp (λx°.x°)) $\widehat{\texttt{e}}$
$\to_s$ ($\widehat{\texttt{e}}$ dbVar dbAbs dbApp (λx°.x°))
$\to_s^3$ q[var:=dbVar,abs:=dbAbs,app:=dbApp] (λx°.x°)
= q[var:=dbVar,abs:=dbAbs,app:=dbApp] $\overline{⟨⟩}$
$\to_s^*$ e □

D REPRESENTATION THEOREMS FOR $F_\omega^{\mu i}$

In this section, we prove that every closed and well-typed $F_\omega^{\mu i}$ term has a representation in $F_\omega^{\mu i}$, and that all representations produced by our quoter are β-normal forms.

The following definition relates the environment used to typecheck a term with the environment used to typecheck its pre-representation.

Definition D.1 (Environment mapping for pre-representations).

$$\begin{aligned} \overline{⟨⟩} &= ⟨⟩ \\ \overline{\Gamma,X{:}K} &= \overline{\Gamma},X{:}K \\ \overline{\Gamma,x^*{:}T} &= \overline{\Gamma},x^*{:}V\ T \end{aligned}$$

Lemma D.1. *If* Γ ⊢ T : K*, then* $\overline{\Gamma}$ ⊢ T : K

Proof. Straightforward, since $\bar{\cdot}$ does not affect the presence, order, or kinds of type variables in the environment. □

Lemma D.2. *If* Γ ⊢ e : T *and* e *contains no free term variables, then* $\overline{\Gamma}$ ⊢ e : T

Proof. Straightforward, using Lemma D.1. □

Lemma D.3. *If* Γ ⊢ (∀X:K.T) : ∗*, then*

(1) Γ ⊢ $\texttt{tcAll}_{X,K,T}$: TcAll (∀X:K.T)
(2) Γ ⊢ $\texttt{unAll}_{X,K,T}$: UnAll (∀X:K.T)
(3) Γ ⊢ $\texttt{isAll}_{X,K,T}$: IsAll (∀X:K.T)

Proof. Suppose $\Gamma \vdash$ (∀X:K.T) : ∗. Then Γ,X:K $\vdash$ T : ∗.

1) $\texttt{tcAll}_{X,K,T}$ = ΛArr:∗ → ∗ → ∗. ΛOut:∗ → ∗. ΛIn:∗ → ∗. ΛMu:((∗ → ∗) → ∗ → ∗) → ∗ → ∗.refl (Out(∀X:K.In T)). It is easily checked that $\texttt{tcAll}_{X,K,T}$ has the type Eq (Out(∀X:K.In T)) (Out(∀X:K.In T)). But Typecase Arr Out In Mu (∀X:K.T) ≡ Out (∀X:K.In T), and also All Out In (∀X:K.T) ≡ Out (∀X:K.In T), so we have that $\texttt{tcAll}_{X,K,T}$ has the type TcAll (∀X:K.T) as required.

2) Follows from reasoning similar to 1.

3) Follows from 1 and 2. □

Lemma D.4. *If* $\Gamma \vdash$ (∀X:K.T) : ∗ *and* $\Gamma \vdash$ S : K, *then*

(1) $\overline{\Gamma} \vdash \texttt{underAll}_{X,K,T}$: UnderAll (∀X:K.T)
(2) $\overline{\Gamma} \vdash \texttt{stripAll}_{K}$: StripAll (∀X:K.T)
(3) $\overline{\Gamma} \vdash \texttt{inst}_{X,K,T,S}$: Inst (∀X:K.T) (T[X:=S])

Proof. First, note that $\texttt{underAll}_{X,K,T}$, $\texttt{stripAll}_{K}$, and $\texttt{inst}_{X,K,T,S}$ contain no free term variables. Therefore, by Lemma D.2 it is sufficient to show

(1) $\Gamma \vdash \texttt{underAll}_{X,K,T}$: UnderAll (∀X:K.T)
(2) $\Gamma \vdash \texttt{stripAll}_{K}$: StripAll (∀X:K.T)
(3) $\Gamma \vdash \texttt{inst}_{X,K,T,S}$: Inst (∀X:K.T) (T[X:=S])

For 1, it is easily checked that $\Gamma \vdash \texttt{underAll}_{X,K,T}$: (∀F1:∗ → ∗. ∀F2:∗ → ∗. (∀A:∗. F1 A → F2 A) → (∀X:K. F1 T) → (∀X:K. F2 T). The result follows from the type equivalences All Id F1 (∀X:K.T) ≡ (∀X:K. F1 T) and All Id F2 (∀X:K.T) ≡ (∀X:K. F2 T).

The cases for 2 and 3 are similar. □

Lemma D.5. *If* $\Gamma \vdash$ e : T, *then there exists a unique* q *such that* $\Gamma \vdash$ e : T q *and* $\overline{\Gamma} \vdash$ q : V T.

Proof. By straightforward induction on the derivation of $\Gamma \vdash$ e : T, using the types of the constructors, Lemma D.3, and Theorem D.4.

Suppose $\Gamma \vdash$ e : T is by the rule for variables. Then e = x* and (x*:T) ∈ Γ and $\Gamma \vdash$ x* : T x*. By the definition of $\overline{\cdot}$, (x* : V T) ∈ $\overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash$ x* : V T as required.

Suppose $\Gamma \vdash$ e : T is by the rule for λ-abstractions. Then e = (λ^*x:T1. e1) and Γ,x*:T1 $\vdash$ e1 : T2 and T = T1 → T2. Also, Γ,x*:T1 $\vdash$ e1 : T2 q1 and $\Gamma \vdash$ (λx*:T1.e1) abs V T1 T2 (λx*:V T1. q1) By induction, q1 is unique and $\overline{\Gamma\texttt{,x}^*\texttt{:T1}} \vdash$ q1 : V T2. But $\overline{\Gamma\texttt{,x:T1}}$ = $\overline{\Gamma}$,x:V T1, so $\overline{\Gamma} \vdash$ (λx:V T1. q1) : V T1 → V T2. Therefore, q = abs V T1 T2 (λx:PExp V T1. q1) is unique and by the type of abs, it is easily checked that $\overline{\Gamma} \vdash$ abs V T1 T2 (λx:V T1. q1) : V (T1 → T2).

The cases for applications, and fold and unfold expressions are similar.

Suppose $\Gamma \vdash$ e : T is by the rule for type abstractions. Then e = (ΛX:K.e1) and Γ,X:K $\vdash$ e1 : T1 and T = (∀X:K.T1). Also, Γ,X:K $\vdash$ e1 : T1 q1 and $\Gamma \vdash$ (ΛX:K.e1) tabs V (∀X:K.T1) p s u (ΛX:K.q1), where p = $\texttt{isAll}_{X,K,T}$, s = $\texttt{stripAll}_{K}$ = s, and u = $\texttt{underAll}_{X,K,T}$. By induction, q1 is unique and $\overline{\Gamma\texttt{,X:K}} \vdash$ q1 : V T1. Since $\overline{\Gamma\texttt{,X:K}}$ = $\overline{\Gamma}$,X:K, we have that $\overline{\Gamma} \vdash$ (ΛX:K.q1) : (∀X:K. V T1). It follows from (All Id V (∀X:K.T1)) ≡ (∀X:K. V T1) that $\overline{\Gamma} \vdash$ (ΛX:K.q1) : (All Id V (∀X:K.T1)). By Theorem D.3, $\Gamma \vdash$ p : IsAll (∀X:K.T). But p does not contain any free term variables, so by Lemma D.2 we have that $\overline{\Gamma} \vdash$ p : IsAll (∀X:K.T). By Theorem D.4, we have that $\overline{\Gamma} \vdash$ s : StripAll (∀X:K.T) and $\overline{\Gamma} \vdash$ u : UnderAll (∀X:K.T). Therefore, q = tabs V (∀X:K.T1) p s u (ΛX:K.q1) is unique and by the type of tabs, we have that $\overline{\Gamma} \vdash$ tabs V (∀X:K.T1) p s u (ΛX:K.q1) : V (∀X:K.T1) as required.

The case for type applications is similar.

Suppose $\Gamma \vdash \texttt{e} : \texttt{T}$ is by the rule for type conversion. The $\Gamma \vdash \texttt{e} : \texttt{S}$ and $\texttt{S} \equiv \texttt{T}$. By the induction hypothesis, there exists a unique $\texttt{q}$ such that $\Gamma \vdash \texttt{e} : \texttt{S} \quad \texttt{q}$ and $\overline{\Gamma} \vdash \texttt{q} : \texttt{PExp V S}$. Since $\texttt{V S} \equiv \texttt{V T}$, we have that $\overline{\Gamma} \vdash \texttt{q} : \texttt{V T}$ as required.

□

Theorem 6.1. *If* $\langle\rangle \vdash \texttt{e} : \texttt{T}$, *then* $\langle\rangle \vdash \overline{\texttt{e}} : \texttt{Exp T}$.

Proof. Follows straightforwardly from Theorem D.5. □

Lemma D.6. *For any type* $\texttt{T}$, $\texttt{refl T}$ *is* β*-normal.*

Proof. Straightforward. □

Lemma D.7. *If* $\Gamma \vdash (\forall \texttt{X:K.T}) : *$, *then* $\texttt{isAll}_{\texttt{X,K,T}}$ *is* β*-normal.*

Proof. Straightforward using Lemma D.6, which states that $\texttt{tcAll}_{\texttt{X,K,T}}$ and $\texttt{unAll}_{\texttt{X,K,T}}$ are β-normal. □

Lemma D.8. *If* $\Gamma \vdash (\forall \texttt{X:K.T}) : *$ *and* $\Gamma \vdash \texttt{S} : \texttt{K}$, *then* $\texttt{inst}_{\texttt{X,K,T,S}}$ *is* β*-normal.*

Proof. Straightforward. □

Lemma D.9. *For any kind* $\texttt{K}$, $\texttt{stripAll}_{\texttt{K}}$ *is* β*-normal.*

Proof. Straightforward. □

Lemma D.10. *If* $\Gamma \vdash (\forall \texttt{X:K.T}) : *$, *then* $\texttt{underAll}_{\texttt{X,K,T}}$ *is* β*-normal.*

Proof. Straightforward. □

Lemma D.11. *If* $\Gamma \vdash \texttt{e} : \texttt{T} \quad \texttt{q}$, *then* $\texttt{q}$ *is* β*-normal.*

Proof. By straightforward induction on the derivation of $\Gamma \vdash \texttt{e} : \texttt{T} \quad \texttt{q}$, using Lemmas D.7, D.8, D.9, and D.10. □

Theorem 6.2. *If* $\langle\rangle \vdash \texttt{e} : \texttt{T}$, *then* $\overline{\texttt{e}}$ *is* β*-normal.*

Proof. We have that $\overline{\texttt{e}} = \Lambda \texttt{V}:* \rightarrow *.\ \texttt{q}$ and $\langle\rangle \vdash \texttt{e} : \texttt{T} \quad \texttt{q}$. By Lemma D.11, $\texttt{q}$ is β-normal. Therefore $\overline{\texttt{e}}$ is also. □

E JONES OPTIMALITY FOR $F_\omega^{\mu i}$

In this section, we prove that a strong optimization under type-erasure can reduce all interpretation work for $F_\omega^{\mu i}$. We define the original and type-erased versions of the helper functions used by the quoter, quotation itself, and the original and type-erased versions of the self-interpreter `unquote`. The representation and interpreter are tagless-final-style – essentially a typed version of the Church encoding for ULC from Section C.1.

Define:

```
θu(e) = e[abs:=unAbs,app:=unApp,
         tabs:=unTAbs,tapp:=unTApp,
         fld:=unUnfold,unfld:=unUnfold]
```

Lemma E.1. *If* $\Gamma \vdash \texttt{e} : \texttt{T} \quad \texttt{q}$, *then* $te(\theta_u(\texttt{q})) \rightarrow_s^* te(\texttt{e})$.

```
decl Eq : * → * =
 λA:*. λB:*. ∀F:* → *. F A → F B;

decl refl : (∀A:*. Eq A A) =
 ΛA:*. ΛF:* → *. λx° : F A. x°;

decl sym : (∀A:*. ∀B:*. Eq A B → Eq B A) =
 ΛA:*. ΛB:*. λeq° : Eq A B.
 eq° (ΛT:*. Eq T A) (refl A);

decl trans : (∀A:*. ∀B:*. ∀C:*.
              Eq A B → Eq B C → Eq A C) =
 ΛA:*. ΛB:*. ΛC:*. λeqAB:Eq A B. λeqBC:Eq B C.
 ΛF:* → *. λx°:F A. eqBC F (eqAB F x°);

decl eqApp : (∀A:*. ∀B:*. ∀F:* → *.
             Eq A B → Eq (F A) (F B)) =
  ΛA:*. ΛB:*. ΛF:* → *. λeq° : Eq A B.
  eq° (λT:*. Eq (F A) (F T)) (refl (F A);

decl arrL : (∀A1:*. ∀A2:*. ∀B1:*. ∀B2:*.
             Eq (A1 → A2) (B1 → B2) →
             Eq A1 B1) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrL;

decl arrR : (∀A1:*. ∀A2:*. ∀B1:*. ∀B2:*.
             Eq (A1 → A2) (B1 → B2) →
             Eq A2 B2) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrR;

decl coerce : (∀A:*. ∀B:*. Eq A B → A → B) =
 ΛA:*. ΛB:*. λeq°:Eq A B. eq° Id;
```

Fig. 25. Implementation of type equality proofs in $F_{\omega}^{\mu i}$.

```
decl refl^te = λx°. x°;
decl sym^te = λeq°. eq° refl^te;
decl trans^te = λeqAB. λeqBC. λx°. eqBC (eqAB x°);
decl eqApp^te = λeq°. eq° refl^te;
decl arrL^te = eqApp^te;
decl arrR^te = eqApp^te;
decl coerce^te = λeq°. eq°;
```

Fig. 26. The type-erasure of type equality proofs.

Proof. By induction on the derivation of $\Gamma \vdash$ e : T q.
Suppose e = x*. Then te(e) = x*, and q = te(q) = x*, and $\theta_u(te(\texttt{q}))$ = x* = te(e).

```
decl TcAll : * → * =
 λT.∀Arr.∀Out.∀In.∀Mu.
 Eq (Typecase Arr Out In Mu T) (All Out In T);
decl UnAll : * → * =
 λT.∀Out. Eq (All Out (λA:*.A) T) (Out T);
decl IsAll : * → * = λT. ∀A. TcAll T → UnAll T → A;

decl tcAll : IsAll T → TcAll T =
  λp° : IsAll T. p° (TcAll T) (λx:TcAll T. λy°:UnAll T. x);
decl unAll : IsAll T → UnAll T =
  λp° : IsAll T. p° (UnAll T) (λx°:TcAll T. λy°:UnAll T. y°);
```

$$
\begin{aligned}
&\texttt{tcAll}_{X,K,T} = \Lambda\texttt{Arr}.\Lambda\texttt{Out}.\Lambda\texttt{In}.\Lambda\texttt{Mu}.\texttt{refl (Out(}\forall\texttt{X:K.In T))}\\
&\texttt{unAll}_{X,K,T} = \Lambda\texttt{Out.refl (Out (}\forall\texttt{X:K.T))}\\
&\texttt{isAll}_{X,K,T} =\\
&\quad \Lambda\texttt{A}.\ \lambda\texttt{f}^\circ\texttt{:TcAll (}\forall\texttt{X:K.T)} \to \texttt{UnAll (}\forall\texttt{X:K.T)} \to \texttt{A}.\\
&\quad \texttt{f}^\circ\ \texttt{tcAll}_{X,K,T}\ \texttt{unAll}_{X,K,T}
\end{aligned}
$$

Fig. 27. IsAll proofs.

$$
\begin{aligned}
&\texttt{decl tcAll}^{te} = \lambda\texttt{p}^\circ.\ \texttt{p}^\circ\ (\lambda\texttt{x}.\ \lambda\texttt{y}^\circ.\ \texttt{x});\\
&\texttt{decl unAll}^{te} = \lambda\texttt{p}^\circ.\ \texttt{p}^\circ\ (\lambda\texttt{x}^\circ.\ \lambda\texttt{y}^\circ.\ \texttt{y}^\circ);
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{tcAll}^{te}_{X,K,T} = \texttt{refl}^{te}\\
&\texttt{unAll}^{te}_{X,K,T} = \texttt{refl}^{te}\\
&\texttt{isAll}^{te}_{X,K,T} = \lambda\texttt{f}^\circ.\ \texttt{f}^\circ\ \texttt{tcAll}^{te}_{X,K,T}\ \texttt{unAll}^{te}_{X,K,T}
\end{aligned}
$$

Fig. 28. Type-erasure of IsAll proofs.

```
decl Id : * → * = λA:*. A;

decl UnderAll : * → * =
 λT:*. ∀F1:* → *. ∀F2:* → *.
 (∀A:*. F1 A → F2 A) →
 All Id F1 A → All Id F2 A;
decl StripAll : * → * =
  λT:*. ∀A:*. All Id (λB:*. A) T → A;
decl Inst : * → * → * =
  λA:*. λB:*. ∀F:*→*. All Id F A → F B;
```

$$
\begin{aligned}
&\texttt{underAll}_{X,K,T} =\\
&\ \Lambda\texttt{F1}.\ \Lambda\texttt{F2}.\ \lambda\texttt{f : (}\forall\texttt{A:*. F1 A} \to \texttt{F2 A)}.\\
&\ \lambda\texttt{e}^\circ\ \texttt{: (}\forall\texttt{X:K. F1 T)}.\ \Lambda\texttt{X:K. f T (e}^\circ\ \texttt{X)}\\
&\texttt{stripAll}_{K} = \Lambda\texttt{A}.\ \lambda\texttt{e}^\circ\texttt{:(}\forall\texttt{X:K.A)}.\ \texttt{e}^\circ\ \texttt{T}_{K}\\
&\texttt{inst}_{X,K,T,S} = \Lambda\texttt{F}.\lambda\texttt{f}^\circ\texttt{:(}\forall\texttt{X:K.F T).f}^\circ\ \texttt{S}
\end{aligned}
$$

Fig. 29. `underAll`, `stripAll`, and `inst` functions

```
underAll^te_X,K,T = λf.λe°. f e°
stripAll^te_K = λe°. e°
inst^te_X,K,T,S = λf°.f°
```

Fig. 30. Type-erasure of underAll, stripAll, and inst functions

```
decl unAbs^te = λf°. f°;
decl unApp^te = λf°. f°;
decl unTAbs^te = λp. λs. λu. λe°. unAll^te p e°;
decl unTApp^te = λp. λi. λx°. i (sym^te (unAll^te p) x°);
decl unFold^te = λx°. x°;
decl unUnfold^te = λx°. x°;

decl unquote^te =
  λe°. e° unAbs^te unApp^te unTAbs^te unTApp^te unFold^te unUnfold^te
```

Fig. 31. The type-erasure of unquote.

Suppose e = $(\lambda x^*:T1.e1)$. Then q = abs T1 T2 $(\lambda x^*:$ V T1. q1), and $\Gamma, (x^*:T1) \vdash$ e1 : T2 q1. By induction, $te(\theta_u(q1)) \to_s^* te(e1)$. Therefore, we can derive:

$te(\theta_u(q))$
$= te($unAbs T1 T2 $(\lambda x^*:$ V T1. $\theta_u(q1)))$
$=$ unAbs^{te} $(\lambda x^*.\ te(\theta_u(q1)))$
$= (\lambda f°.f°)\ (\lambda x^*.\ te(\theta_u(q1)))$
$\to_s (\lambda x^*.\ te(\theta_u(q1)))$
$\to_s^* (\lambda x^*.\ te(e1))$
$= te(e)$

Suppose e = e1 e2. Then q = app T2 T q1 q2 and $\Gamma \vdash$ e1 : T2 $\to$ T q1 and $\Gamma \vdash$ e2 : T2 q2. By induction, $te(\theta_u(q1)) \to_s^* te(e1)$ and $te(\theta_u(q2)) \to_s^* te(e2)$. Therefore, we can derive:

$te(\theta_u(q))$
$= te($unApp T2 T $\theta_u(q1)\ \theta_u(q2))$
$=$ unApp^{te} $te(\theta_u(q1))\ te(\theta_u(q2))$
$= (\lambda f°.\ f°)\ te(\theta_u(q1))\ te(\theta_u(q2))$
$\to_s te(\theta_u(q1))\ te(\theta_u(q2))$
$\to_s^* te(e1)\ te(e2)$
$= te(e1\ e2)$
$= te(e)$

Suppose $\mathtt{e} = \Lambda\mathtt{X{:}K.e1}$. Then $\mathtt{q} = \mathtt{tabs}\ (\forall\mathtt{X{:}K.T})\ \mathtt{isAll}_{\mathtt{X,K,T}}\ \mathtt{stripAll}_{\mathtt{K}}\ \mathtt{underAll}_{\mathtt{X,K,T}}\ (\Lambda\mathtt{X{:}K.q1})$ and $\Gamma, (\mathtt{X{:}K}) \vdash \mathtt{e1}\ :\ \mathtt{T}\quad \mathtt{q1}$. By induction, $te(\theta_u(\mathtt{q1})) \to_s^* te(\mathtt{e1})$. Therefore, we can derive:

$te(\theta_u(\mathtt{q}))$

$= te(\mathtt{unTAbs}\ (\forall\mathtt{X{:}K.T})\ \mathtt{isAll}_{\mathtt{X,K,T}}\ \mathtt{stripAll}_{\mathtt{K}}\ \mathtt{underAll}_{\mathtt{X,K,T}}\ (\Lambda\mathtt{X{:}K.}\theta_u(\mathtt{q1})))$

$= \mathtt{unTAbs}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{stripAll}^{te}_{\mathtt{K}}\ \mathtt{underAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{p}.\ \lambda\mathtt{s}.\ \lambda\mathtt{u}.\ \lambda\mathtt{e}^\circ.\ \mathtt{unAll}^{te}\ \mathtt{p}\ \mathtt{e}^\circ)$
$\quad \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{stripAll}^{te}_{\mathtt{K}}\ \mathtt{underAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$\to_s^4 \mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{p}^\circ.\ \mathtt{p}^\circ\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ))\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$\to_s \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{tcAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{unAll}^{te}_{\mathtt{X,K,T}})\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ te(\theta_u(\mathtt{q1}))$

$\to_s (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ \mathtt{tcAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{unAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$\to_s^2 \mathtt{unAll}^{te}_{\mathtt{X,K,T}}\ te(\theta_u(\mathtt{q1}))$

$= \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{x}^\circ.\ \mathtt{x}^\circ)\ te(\theta_u(\mathtt{q1}))$

$\to_s te(\theta_u(\mathtt{q1}))$

$\to_s^* te(\mathtt{e1})$

$= te(\Lambda\mathtt{X{:}K.e1})$

$= te(\mathtt{e})$

Suppose $\mathtt{e} = \mathtt{e1}\ \mathtt{A}$. Then $\mathtt{q} = \mathtt{tapp}\ (\forall\mathtt{X{:}K.T})\ (\mathtt{T[X{:=}A]})\ \mathtt{isAll}_{\mathtt{X,K,T}}\ \mathtt{inst}_{\mathtt{X,K,T,A}}\ \mathtt{q1}$ and $\Gamma \vdash \mathtt{e1}\ :\ (\forall\mathtt{X{:}K.T})\quad \mathtt{q1}$. By induction, $te(\theta_u(\mathtt{q1})) \to_s^* \mathtt{e1}$. Therefore, we can derive:

$te(\theta_u(\mathtt{q}))$

$= te(\mathtt{unTApp}\ (\forall\mathtt{X{:}K.T})\ (\mathtt{T[X{:=}A]})\ \mathtt{isAll}_{\mathtt{X,K,T}}\ \mathtt{inst}_{\mathtt{X,K,T,A}}\ \theta_u(\mathtt{q1}))$

$= \mathtt{unTApp}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{inst}^{te}_{\mathtt{X,K,T,A}}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{p}.\ \lambda\mathtt{i}.\ \lambda\mathtt{x}^\circ.\ \mathtt{i}\ (\mathtt{sym}^{te}\ (\mathtt{unAll}^{te}\ \mathtt{p})\ \mathtt{x}^\circ))$
$\quad \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{inst}^{te}_{\mathtt{X,K,T,A}}\ te(\theta_u(\mathtt{q1}))$

$\to_s^3 \mathtt{inst}^{te}_{\mathtt{X,K,T,A}}\ (\mathtt{sym}^{te}\ (\mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}})\ te(\theta_u(\mathtt{q1})))$

$= (\lambda\mathtt{f}^\circ.\mathtt{f}^\circ)\ (\mathtt{sym}^{te}\ (\mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}})\ te(\theta_u(\mathtt{q1})))$

$\to_s \mathtt{sym}^{te}\ (\mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}})\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{eq}^\circ.\ \mathtt{eq}^\circ\ \mathtt{refl}^{te})\ (\mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}})\ te(\theta_u(\mathtt{q1}))$

$\to_s \mathtt{unAll}^{te}\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{p}^\circ.\ \mathtt{p}^\circ\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ))\ \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$\to_s \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= \mathtt{isAll}^{te}_{\mathtt{X,K,T}}\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{f}^\circ.\ \mathtt{f}^\circ\ \mathtt{tcAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{unAll}^{te}_{\mathtt{X,K,T}})\ (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$\to_s (\lambda\mathtt{x}^\circ.\ \lambda\mathtt{y}^\circ.\ \mathtt{y}^\circ)\ \mathtt{tcAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{unAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$\to_s^2 \mathtt{unAll}^{te}_{\mathtt{X,K,T}}\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= \mathtt{refl}^{te}\ \mathtt{refl}^{te}\ te(\theta_u(\mathtt{q1}))$

$= (\lambda\mathtt{x}^\circ.\ \mathtt{x}^\circ)\ (\lambda\mathtt{x}^\circ.\ \mathtt{x}^\circ)\ te(\theta_u(\mathtt{q1}))$

$\to_s^2 te(\theta_u(\mathtt{q1}))$

$\to_s^* te(\mathtt{e1})$

$= te(\mathtt{e1\ A})$

$= te(\mathtt{e})$

Suppose $\texttt{e} = \texttt{fold F T e1}$. Then $\texttt{q} = \texttt{fld F T q1}$ and $\Gamma \vdash \texttt{e1 : F (F) T q1}$. By induction $te(\theta_u(\texttt{q1}))$ $\rightarrow_s^* te(\texttt{e1})$. Therefore, we can derive:

$te(\theta_u(\texttt{q}))$
$= te(\theta_u(\texttt{fld F T q1}))$
$= te(\texttt{unFold F T } \theta_u(\texttt{q1}))$
$= \texttt{unFold}^{te}\ te(\theta_u(\texttt{q1}))$
$= (\lambda \texttt{x}^\circ.\ \texttt{x}^\circ)\ te(\theta_u(\texttt{q1}))$
$\rightarrow_s te(\theta_u(\texttt{q1}))$
$\rightarrow_s^* te(\texttt{e1})$
$= te(\texttt{fold F T e1})$
$= te(\texttt{e})$

Suppose $\texttt{e} = \texttt{unfold F T e1}$. Then $\texttt{q} = \texttt{unfld F T q1}$ and $\Gamma \vdash \texttt{e1 : F T q1}$. By induction $te(\theta_u(\texttt{q1}))$ $\rightarrow_s^* te(\texttt{e1})$. Therefore, we can derive:

$te(\theta_u(\texttt{q}))$
$= te(\theta_u(\texttt{unfld F T q1}))$
$= te(\texttt{unUnfold F T } \theta_u(\texttt{q1}))$
$= \texttt{unUnfold}^{te}\ te(\theta_u(\texttt{q1}))$
$= (\lambda \texttt{x}^\circ.\ \texttt{x}^\circ)\ te(\theta_u(\texttt{q1}))$ □
$\rightarrow_s te(\theta_u(\texttt{q1}))$
$\rightarrow_s^* te(\texttt{e1})$
$= te(\texttt{unfold F T e1})$
$= te(\texttt{e})$

Lemma 6.3. *If* $\langle\rangle \vdash \texttt{e : T}$, *then* $te(\texttt{unquote } \overline{\texttt{e}}) \Longrightarrow_s te(\texttt{e})$.

Proof. Suppose $\langle\rangle \vdash \texttt{e : T}$. Then $\langle\rangle \vdash \texttt{e : T q}$ and
$\overline{\texttt{e}} = \Lambda \texttt{V}{:}* \rightarrow *.$
$\lambda\texttt{abs : Abs V.}\ \lambda\texttt{app : App V.}$
$\lambda\texttt{tabs : TAbs V.}\ \lambda\texttt{tapp : TApp V.}$
$\lambda\texttt{fld : Fold V.}\ \lambda\texttt{unfld : Unfold V.}$
$\texttt{q}$

Therefore, $te(\overline{\texttt{e}}) = \lambda\texttt{abs.}\lambda\texttt{app.}\lambda\texttt{tabs.}\lambda\texttt{tapp.}\lambda\texttt{fld.}\lambda\texttt{unfld.}te(\texttt{q})$.
By induction, $te(\theta_u(\texttt{q})) \rightarrow_s^* te(\texttt{e})$.
Therefore, we can derive:

$te(\texttt{unquote } \overline{\texttt{e}})$
$= \texttt{unquote}^{te}\ te(\overline{\texttt{e}})$
$= \texttt{unquote}^{te}\ (\lambda\texttt{abs.}\lambda\texttt{app.}\lambda\texttt{tabs.}\lambda\texttt{tapp.}\lambda\texttt{fld.}\lambda\texttt{unfld.}te(\texttt{q}))$
$= (\lambda \texttt{e}^\circ.\ \texttt{e}^\circ\ \texttt{unAbs}^{te}\ \texttt{unApp}^{te}\ \texttt{unTAbs}^{te}\ \texttt{unTApp}^{te}\ \texttt{unFold}^{te}\ \texttt{unUnfold}^{te})$
$(\lambda\texttt{abs.}\lambda\texttt{app.}\lambda\texttt{tabs.}\lambda\texttt{tapp.}\lambda\texttt{fld.}\lambda\texttt{unfld.}te(\texttt{q}))$
$\rightarrow_s^7 te(\texttt{q})[\texttt{abs:=unAbs}^{te}\texttt{,app:=unApp}^{te}\texttt{,tabs:=unTAbs}^{te}\texttt{,tapp:=unTApp}^{te}\texttt{,fld:=unFold}^{te}\texttt{,unfld:=unUnfol}$
$= te(\theta_u(\texttt{q}))$
$\rightarrow_s^* te(\texttt{e})$

□