Title
Typed Self-Optimization

Author
Brown, Matt

Publication Date
2013

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Typed Self-Optimization

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Matthew Scott Brown

2013

Abstract of the Thesis

Typed Self-Optimization

by

Matthew Scott Brown

Master of Science in Computer Science

University of California, Los Angeles, 2013

Professor Jens Palsberg, Chair

Researchers have studied how to type check self-applicable programs. For example, papers by Rendel, Ostermann, and Hofer, and by Jay and Palsberg have shown how to design two kinds of polymorphically typed self-interpreters. In this paper we present the first polymorphically typed self-optimizer. In contrast to a self-interpreter that often can implement each construct by itself, a self-optimizer may replace a subterm with a rather different subterm, which complicates type checking. Our language has combinators, a variant of Mitchell's subtyping, proof terms that help match types, and a novel approach to type check self-application. Via syntactic sugar, we define a surface syntax with decidable type inference. Our implementation has type checked and run our examples.

The thesis of Matthew Scott Brown is approved.

Adnan Darwiche

Todd Millstein

Jens Palsberg, Committee Chair

University of California, Los Angeles

2013

Table of Contents

List of Figures

1 Introduction

A *self-optimizer* is a program optimizer that can be applied to a representation of itself.

The problem. What is the type of a self-optimizer? The classical answer to such questions is to work with a single type for all program representations. For example, the single type could be String or it could be SyntaxTree. The single-type approach enables an optimizer to have a type such as (String $\rightarrow$ String), where the input string represents source code and where the output string represents target code. However, the single-type approach ignores that the source program type checks, and it doesn't guarantee that the output represents a typed program, or that the type of the output program is related to the type of the input program. In particular, self-application of an optimizer of type (String $\rightarrow$ String) must work with a representation of the optimizer of type String.

Our result. We present the first program optimizer that has the polymorphic type

$$\forall T.Exp[T] \rightarrow Exp[T]$$

where $Exp[T]$ is the type of a representation of a program of type T. The type enables the optimizer to be applied to a representation of itself so we use the name self-optimizer. The type says that the optimizer is *type preserving*: if the input program type checks, then the output program has the same type as the input program. Stronger type checking means better bug finding: if we check that an optimizer has type $\forall T.Exp[T] \rightarrow Exp[T]$, we will catch more bugs than if we check that an optimizer has type (String $\rightarrow$ String).

Self-interpreters. The context for our result is the recent interest in polymorphically typed self-interpreters. Jay and Palsberg [11] identified two main forms of self-interpreters that they called *self-recognizers* and *self-enactors*. If $'e$ denotes a representation of the program e, then a self-recognizer maps $'e$ to e, while a self-enactor executes $'e$ to $'v$, where v is the value of e. Self-recognizers are much studied in λ-calculus [12, 4, 16, 17, 6, 5], and self-enactors are available for Standard ML [23], Haskell [18], Scheme [3], JavaScript [8], Python [22] and Ruby [30], and have been studied for λ-calculus [16, 6, 24] and other languages [21, 14, 31]. Rendel, Ostermann, and Hofer [20] presented a self-recognizer with type

$$\forall T.Exp[T] \to T$$

and Jay and Palsberg [11] presented a self-enactor with type

$$\forall T.T \to T$$

Jay and Palsberg's result was possible because they equated $Exp[T]$ and T. We improve on the latter result and present the first self-enactor that has type

$$\forall T.Exp[T] \to Exp[T]$$

via an application of the techniques that led to our self-optimizer.

Self-optimizers. Our self-optimizer and our self-enactor are both type-preserving program transformations and they have the same polymorphic type. However, the problem to design a polymorphically typed self-optimizer is substantially harder than to design a polymorphically typed self-interpreter. To see this point, let us first examine the self-enactor of Jay and Palsberg [11]. Their self-enactor is largely meta-circular in that it implements almost every language construct via a use of itself, as in this excerpt that implements the operator K via a use of K itself:

$$B\ K\ x_2\ x_1 \longrightarrow\ \mathit{enact}\ (K\ x_2\ x_1)$$

This line of code covers the case when the self-enactor encounters a *representation* of $K\ x_2\ x_1$. In this case, the self-enactor calls itself recursively on $K\ x_2\ x_1$ and relies on the underlying execution engine to reduce $K\ x_2\ x_1$ to x_2, eventually. Jay and Palsberg give the two occurrences of K the same type, which is sufficient to make their self-enactor type check. In particular, they don't need to know details of the type derivation for $B\ K\ x_2\ x_1$.

In contrast, a self-optimizer may replace a subterm with a quite different subterm, which complicates type checking. For example, our self-optimizer replaces a representation of SK with a representation of KI, where S, K, I have their usual meanings in combinatory calculi. We can justify the optimization by noting that each of $SKxy$ and $KIxy$ reduces to y in two steps. One heavy-weight approach might be to work with a program representation that contains the entire type derivation, though we leave that for future work. Instead, we use a light-weight approach and work with a program representation that contains *no type information* at all. Our self-optimizer works in *any context* and for any type of SK, and has to discover details of type derivations at run time. In essence, the main challenge is:

> **Main challenge:** The optimizer must prove that the optimized term has all the types of the input term.

The use of *subtyping* in type derivations is a major complication.

Our approach. We use a variation of Donnelly's proof terms [7] to witness subtype relationships and to build proofs at run time. The proof terms enable us to map a typable term to an implicit representation of subtype constraints that characterize all the input term's types. Based on those constraints we build a proof that the optimized term has all the types of the input term. In our language, a proof term is much like a heap label in that both are constants that get their types from an environment. In the case of heap labels, such an environment is usually known as a store type. Our proof terms generalize Donelly's proof terms, and we use proof terms in a novel way.

We present a light-weight framework that specifies a program skeleton for how to program a type-preserving program transformation. We have used the framework to program both a polymorphically typed self-optimizer and a polymorphically typed self-enactor. Programs contain proof terms but no types or type derivations. Our language is a combinatory calculus with constructs for program representation and operator equality, along with the proof terms. Our type system uses two kinds of types, namely kind $(* \to *)$ and kind $*$, and also type equivalence and an extension of Mitchell subtyping [15]. In slogan form:

$$\begin{array}{rcl} \text{Language} & = & \text{combinators + program representation +} \\ & & \text{operator equality + proof terms} \\ \text{Types} & = & \text{two kinds + equivalence + subtyping} \\ \text{Technique} & = & \text{programming with proofs} \end{array}$$

Our main theorem is the soundness of our type system: well-typed programs cannot go wrong. We will state our lemmas and theorems in Section 7.5, and provide proofs in an appendix.

We have designed a decidable fragment of our type system, along with a type inference algorithm that we have used to type check both our self-optimizer and our self-enactor. The full type system is needed to ensure that computation preserves typing. Intuitively, we don't use all the bells and whistles of the type system to write interesting programs but we do need the entire type system to type the programs that may arise during computation.

We have implemented our language and will show results from experiments.

The rest of the paper. In Section 2 we give an overview of our framework, in

Section 3 we give an introduction to proof terms, in Section 4 we describe how to program with proof terms, in Section 5 we show our representation of programs, and in Section 6 we exhibit our self-optimizer. In Section 7 we formalize our language, type system, and type soundness theorem, in Section 8 we describe our type-inference algorithm, in Section 9 we describe our typed self-interpreter, in Section 10 we explain our experimental results, and in Section 11 we discuss related work.

Two appendices contain the proofs of our theorems and the code for three optimizations that are part of our overall self-optimizer. The code for our self-enactor is seven pages in this format and is available upon request. Sections 2-6 also serve as a gentle introduction to our language and type system.

2 Our Framework

We will now give an overview of how our framework works.

Our framework ensures that if an optimization is typable, then it is type preserving. In essence, we want to type check the implementation of an optimization $e_1 \to e_2$ and have that imply that $e_1 \to e_2$ is type preserving. Intuitively, we want the type checker to guarantee that *every* type of e_1 is also a type of e_2.

Let us begin with a non-example, namely $SK \to I$. Suppose we find that one of the possible type derivations for SK assigns $SK : T$, where T is of the form $(U \to U) \to (U \to U)$. We might also notice that we can derive $I : T$ and be tempted to think that $SK \to I$ is type preserving, though it isn't. In fact we want the type checker to reject an implementation of $SK \to I$.

Let us next look at a type-preserving optimization, namely $SK \to KI$, which we will use as a running example throughout Sections 4–6. Again, we might notice that we can derive $KI : T$ and be tempted to think that $SK \to KI$ is type preserving, which it actually is! We might also find that a second possible type derivation for SK assigns $SK : T'$ where T' is of the form $(V \to W) \to V \to V$. If V and W are distinct, then we cannot derive $I : T'$ which shows that $SK \to I$ isn't always type preserving. On the other hand, we can also derive $KI : T'$, so again we are tempted to think that $SK \to KI$ is type preserving. How can a type check of $SK \to KI$ imply that every type of SK is also a type of KI?

If we want to ensure our optimization is type preserving, we must reason about all the possible type derivations of SK. We characterize all the types of SK with a set of subtype constraints, which we represent implicitly with *proof terms*. If KI can be assigned any type that satisfies those subtype constraints, then we know that $SK \to KI$ is type preserving.

The implementation of $SK \to KI$ has the following structure:

1. **analyzeSK**: Look for an occurrence of SK in the input term and generate an implicit representation of type constraints.

2. **proveSK2KI**: Build from those constraints a proof that $\forall T, U. T \to U \to U$ is a subtype of any type of SK.

3. **constructKI**: Produce version of KI with the type $\forall T, U. T \to U \to U$.

This is the common structure for all our optimizations, though the details vary. For example, there is no construct step for $S(Ke)I \to e$, since the result is a subterm of the input term. In this case analyze produces the resulting term directly, along with the constraints.

In Section 4 we will explain how to program *proveSK2KI*, while in Section 5 we will explain how to program *analyzeSK* and *constructKI*.

3 What is a proof term?

A proof term witnesses a subtype relationship. In this section we review the literature on proof terms, and we discuss *dynamic generation of proof terms*, which may be a new idea.

Many languages have a notion of *subtyping* that is a reflexive and transitive relation $\subseteq$ on types. The idea is that if a term e has type T, and $T \subseteq U$, then e also has type U.

In some languages, subtyping can be applied implicitly via use of a subsumption rule:

$$\text{Type-Subsumption}\ \frac{\vdash e : T \qquad T \subseteq U}{\vdash e : U}$$

In other languages, such as O'Caml, F#, and the one in this paper, subtyping must be applied explicitly. We follow Donnelly [7] and use a syntax that involves a *proof term* p and a type-changing construct *coerce*:

$$\text{Type-Coerce}\ \frac{\vdash e : T \qquad \vdash p : T \dot{\leq} U}{\vdash \mathit{coerce}(e, p) : U}$$

Donnelly introduced the constraint type $T \dot{\leq} U$ to denote the type of p. If $\vdash p : T \dot{\leq} U$, then we say that p witnesses the subtype relationship $T \dot{\leq} U$. For example, Donnelly's proof term $refl$ witnesses the subtype relationship $T \subseteq T$ for any type T, and we can write $\vdash refl : T \dot{\leq} T$.

Constraint types enable us to compose proof terms in a straightforward way. For example, one of Donnelly's proof terms is $trans$ (for transitivity) that composes a proof term of type $T_1 \dot{\leq} T_2$ and a proof term of type $T_2 \dot{\leq} T_3$ into a proof term of type $T_1 \dot{\leq} T_3$. We can use $trans$ to build proof terms such as $trans(refl, refl)$, which has type $T \dot{\leq} T$. Such "programming with proofs" plays a major role in this paper, as we will explain in a later section.

Donnelly's thesis [7] introduced ten proof constants, including $refl$ and $trans$, each with a particular type rule. In Donnelly's Lemma 3.21, he proves that "Subtyping is Equality", that is, if a term p has type $T \dot{\leq} U$, then $T = U$. We have borrowed many of Donnelly's proof terms and added others of our own. Our proof terms satisfy a weaker lemma than Donnelly's Lemma 3.21, namely our Lemma 23 that says, intuitively, that if a value v has type $T \dot{\leq} U$, then $T \subseteq U$.

All Donnelly's proof terms are *static* in that his language defines ten specific ones and assigns them particular types. For our application, we also need proof terms that are *dynamic* in that we do dynamic generation of proof terms. Static proof terms come with specified types, while the types of dynamic proof terms depend on the context they are created within. For example, a key construct in this paper is a proof term called $eArrow$ which has this operational semantics:

$$eArrow\ v_1\ e \rightarrow e\ p_1\ p_2\ p_3 \quad \text{where } p_1, p_2, p_3 \text{ are fresh}$$

Here v_1 is a proof term and e is a continuation. The above rule says that when we execute $eArrow$, then we will dynamically generate three *fresh* proof terms called p_1, p_2, p_3.

The idea of dynamic generation of constants has a close analogy in dynamic generation of heap labels in imperative languages. For example, the semantics for an operator ref that generates new heap space might be of the form:

$$ref\ v_2 \rightarrow l \quad \text{where } l \text{ is fresh}$$

Here l is a fresh constant heap label, just like p_1, p_2, p_3 are fresh constant proof term in the rule for $eArrow$.

How do we type check dynamically generated proof terms? This problem is closely related to the problem to type check dynamically generated heap labels. Notice that the rules for $eArrow$ and *new* place no restrictions on the types of p_1, p_2, p_3 or l. A standard approach to solve this problem for dynamically generated heap labels is known as *store types* [1]. A store type Γ assigns a type $\Gamma(l)$ to every heap label l. We borrow this approach and use a type environment Γ to assign a type $\Gamma(p)$ to every dynamically generated proof term p. The type assigned to the heap label l will depend on the type of its initial contents v_2, and similarly the types of p_1, p_2, and p_3 will be related to the type of v_1, as defined by the type of $eArrow$.

4 Programming with Proofs

Let us consider the optimization $SK \rightarrow KI$ and show how to program $proveSK2KI$, first for a simplified notion of subtyping, and then for our full subtyping relation. We use syntactic sugar for λ-abstraction, let binding, and let rec binding.

4.1 Subtyping of Function Types

The input to $proveSK2KI$ is a set of constraints for the term SK. Those constraints are closely related to any type derivation for SK, which in the case of Subtyping of Function Types is of the form:

$$\frac{S : T_1 \rightarrow T \qquad K : T_1}{SK : T}$$

Here we assume that SK has some type T and use the Inversion Lemma for applications to conclude that there must exist a type T_1 such that $S : T_1 \rightarrow T$ and $K : T_1$. Our type system specifies types for S and K, called $Ty[S]$ and $Ty[K]$. When we match instantiations $Ty[S]$ and $Ty[T]$ with the types in the above type derivation, we have that there exist types U_1, U_2, U_3, V_1, V_2 such that these constraints are satisfied:

$$
\begin{aligned}
(U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \subseteq T_1 \to T \\
V_1 \to V_2 \to V_1 \subseteq T_1
\end{aligned}
$$

The first step of our optimization, $analyzeSK$, presents those two constraints to $proveSK2KI$ in the form of two proof terms pS and pK:

$$
\begin{aligned}
pS &: (U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \dot{\leq} T_1 \to T \\
pK &: V_1 \to V_2 \to V_1 \dot{\leq} T_1
\end{aligned}
$$

We ensure a strong connection between proof terms and subtyping: if $p : T \dot{\leq} U$ and p is a value, then $T \subseteq U$. The condition that p be a value is required to prevent nonterminating proof terms from being used to prove false statements. For these subtype derivations are based on the Sub-$\to$ rule:

$$
\text{Sub-}\to \ \frac{U_1 \subseteq T_1 \qquad T_2 \subseteq U_2}{T_1 \to T_2 \subseteq U_1 \to U_2}
$$

Let us now program $proveSK2KI$.

To construct the proof for $SK \to KI$, we first need to decompose the constraints to combine the constraints on T_1. We rely on the inversion lemma for Sub-$\to$: For all types T_1, T_2, U_1, U_2, if $T_1 \to T_2 \subseteq U_1 \to U_2$, then $U_1 \subseteq T_1$ and $T_2 \subseteq U_2$. We encode this lemma using an operator $eArrow_1$:

$$
\begin{aligned}
eArrow_1 : \forall T_1, T_2, U_1, U_2, T.& \\
&(T_1 \to T_2 \dot{\leq} U_1 \to U_2) \to \\
&((U_1 \dot{\leq} T_1) \to (T_2 \dot{\leq} U_2) \to T) \to \ T
\end{aligned}
$$

The semantics of $eArrow_1$ is:

$$eArrow_1\ p\ f \longrightarrow f\ p_1\ p_2$$

Where p_1 and p_2 are fresh proof constants. The types of the fresh proof constants are related to the type of p by the inversion lemma. In particular, if $p : T_1 \to T_2 \dot{\leq} U_1 \to U_2$, then p_1 will have type $U_1 \dot{\leq} T_1$ and p_2 will have type $T_2 \subseteq U_2$. Applying $eArrow_1$ to pS will dynamically generate new proof constants p_1 and p_2:

$$p_1 : T_1 \dot{\leq} (U_1 \to T_2 \to U_3)$$
$$p_2 : (U_1 \to T_2) \to U_1 \to U_3 \dot{\leq} T$$

Now our task becomes clearer: we need to show KI can be assigned any type $(U_1 \to T_2) \to U_1 \to U_3$. Then p_2 will prove KI can be assigned T. We cannot yet show KI has this type, so the next step is to combine pK and p_1 in order to derive constraints on U_1, T_2, and U_3. We introduce a proof constructor $trans$ to encode the transitive subtyping rule, and use it to construct a new proof p_3.

$$trans : \forall T_1, T_2, T_3.(T_1 \dot{\leq} T_2) \to (T_2 \dot{\leq} T_3) \to (T_1 \dot{\leq} T_3)$$

```
let  (p_3 : (V_1 → V_2 → V_1) ≤̇ (U_1 → T_2 → U_3)) = trans pK p_1
```

We define helper functions $eBinary_1$ and $expandK_1$. $eBinary_1$ decomposes p_3 into three components using two applications of $eArrow_1$, which $expandK_1$ combines to produce the essential constraint needed from p_3.

```
let  eBinary_1 =
  λ(p : T_1 → T_2 → T_3 ≤̇ U_1 → U_2 → U_3).
  λ(f : (U_1 ≤̇ T_1) → (U_2 ≤̇ T_2) → (T_3 ≤̇ U_3) → V).
  eArrow_1 p
```

$(\lambda(p1 : U_1 \dot{\leq} T_1).\lambda(p2 : T_2 \to U_2 \dot{\leq} U_2 \to U_3).$

$eArrow_1\ p2\ (\lambda(p3 : U_2 \dot{\leq} T_2).\lambda(p4 : T_3 \dot{\leq} U_3).\ f\ p1\ p3\ p4))$

$\texttt{let}\ \ expandK_1\ =\ \lambda(p : (T_1 \to T_2 \to T_1) \dot{\leq} (U_1 \to U_2 \to U_3))\,.$

$eBinary_1\ p\ (\lambda(p1 : U_1 \dot{\leq} T_1).\mathrm{K}\ (\lambda(p2 : T_1 \dot{\leq} U_3).\mathrm{trans}\ p1\ p2))$

$\texttt{let}\ \ (p_4 : U_1 \dot{\leq} U_3)\ =\ expandK_1\ p_3$

We can begin to gain confidence that KI is in fact type preserving for this restriction of $SK : T$. The proof term p_4 proves that U_1 must be a subtype of U_2. Therefore the type $(U_1 \to U_2) \to U_1 \to U_1$, which is known to be a type of KI, is also a subtype of $(U_1 \to U_2) \to U_1 \to U_3$. In order to construct a proof term of this fact, we need two more proof constructors:

$$refl : \forall T.T \dot{\leq} T$$

$$iArrow : \forall T_1, T_2, U_1, U_2.(U_1 \dot{\leq} T_1) \to (T_2 \dot{\leq} U_2) \to$$
$$(T_1 \to T_2 \dot{\leq} U_1 \to U_2)$$

$\texttt{let}\ \ (p_5 :\ (U_1 \to U_2) \to U_1 \to U_1 \dot{\leq} (U_1 \to U_1) \to U_1 \to U_3)\ =$

iArrow refl (iArrow refl p_6)

Now transitivity between p_5 and p_2 proves $(U_1 \to U_2) \to U_1 \to U_1 \subseteq T$. Therefore, we have proven $SK \to KI$ to be type preserving in this simplification of subtyping. The complete definition of $proveSK2KI$ is:

$\texttt{let}\ \ proveSK2KI\ =$

$\lambda(pS : (U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \dot{\leq} T_1 \to T).$

$\lambda(pK : (V_1 \to V_2 \to V_1) \dot{\leq} T_1).$

$eArrow_1\ pS$

$(\lambda(p1 : T_1 \dot{\leq} U_1 \to U_2 \to U_3).$

$\lambda(p2 : (U_1 \to U_2) \to U_1 \to U_3 \dot{\leq} T).$

$\texttt{let}\ \ (p3 : V_1 \to V_2 \to V_1 \dot{\leq} U_1 \to U_2 \to U_3) = trans\ pK\ p1\ \ \texttt{in}$

let $(p4 : U_1 \dot{\leq} U_3) = expandK_1\ p3$ in

let $(p5 : (U_1 \to U_2) \to U_1 \to U_1 \dot{\leq} (U_1 \to U_2) \to U_1 \to U_3)$

$= iArrow\ refl\ (iArrow\ refl\ p4)$ in

$trans\ p5\ p2$

4.2 Full Subtyping

Let us generalize our consideration of the derivation of $SK : T$ to include all of subtyping. We allow quantified types, distribution of quantifiers of arrows, and substitution which combines instantiation and generalization.

$$\text{Sub-Dist-}\!\to \frac{}{\forall\vec{\alpha}.T \to U \subseteq (\forall\vec{\alpha}.T) \to (\forall\vec{\alpha}.U)}$$

$$\text{Sub-Subst} \frac{}{\forall\vec{\alpha}.T \subseteq \forall\vec{\beta}.Subst[\theta]T}\ dom(\theta) = \vec{\alpha},\ \vec{\beta} \notin FV(\forall\vec{\alpha}.T)$$

In the presence of full subtyping $analyzeSK$ will produce the following subtype constraints for $T_1 \to T$ and T_1:

$$\forall\vec{\alpha}.(U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \subseteq T_1 \to T$$
$$\forall\vec{\beta}.V_1 \to V_2 \to V_1 \subseteq T_1$$

As before, our first step will be to decompose the first constraint using inversion for subtyping between quantified arrow types. The general inversion lemma states that if $\forall\vec{\alpha}.T_1 \to T_2 \subseteq \forall\vec{\gamma}.U_1 \to U_2$, there exist quantifiers $\vec{\beta}$ and a substitution θ such that $U_1 \subseteq \forall\vec{\beta}.\theta(T_1)$ and $\forall\vec{\beta}.\theta(T_2) \subseteq U_2$. This implies that any derivation of $\forall\vec{\alpha}.T_1 \to T_2 \subseteq \forall\vec{\gamma}.U_1 \to U_2$ can be normalized to the form:

$$
\begin{aligned}
& \forall \vec{\alpha}.T_1 \to T_2 & \\
&= \forall \vec{\beta}, \vec{\gamma}.\theta(T_1 \to T_2) & \text{(Sub-Subst)} \\
&\subseteq \forall \vec{\beta}.(\forall \vec{\gamma}.\theta(T_1)) \to (\forall \vec{\gamma}.\theta(T_2)) & \text{(Sub-Dist-}\to\text{)} \\
&\subseteq \forall \vec{\beta}.U_1 \to U_2 & \text{(Sub-}\to\text{)}
\end{aligned}
$$

Adding quantifiers and substitutions leads to a new challenge: how do we capture the relationship between $\vec{\alpha}$ and $\vec{\beta}$, $\vec{\gamma}$, and θ? This is complicated by the fact that sometimes one or more of $\vec{\alpha}, \vec{\beta}, \vec{\gamma}$, and θ are unknown. In particular, the inversion lemma above states there exist quantifiers $\vec{\beta}$ and a substitution θ. We need to reason about such abstract quantifiers and substitutions. In particular, we will need to use the Sub-Subst step to derive $\forall \vec{\alpha}.T \subseteq \forall \vec{\beta}, \vec{\gamma}.\theta(T)$ as long as T satisfies the side condition of Sub-Subst. Our approach is to represent quantifier sets and substitutions syntactically, using type constructors. This frees us from having to encode substitutions themselves. Instead, we note that the side condition guarantees that $\vec{\beta}$ and $\vec{\gamma}$ are introduced by θ. By α conversion, we can assume that $\vec{\beta}$ and $\vec{\gamma}$ don't occur elsewhere in the program. This allows us to perform Sub-Subst steps when the quantifiers and substitution are unknown.

When particular quantifiers $\vec{\alpha}$ are known, we use a type constructor $\forall[\vec{\alpha}]$. For example $\forall \alpha.T \to T$ can be written $\forall[\alpha](T \to T)$. When the quantifiers are unknown, as in the case of $\vec{\gamma}$ above, we use a type variable of kind $* \to *$ to show that some quantifier exists: $\varphi(T \to T)$. We denote concatentation of two type constructors φ_1 and φ_2 as $\varphi_1 \circ \varphi_2$. We represent substitutions similarly using type variables σ, which also have kind $* \to *$. A type variable σ may be instantiated to a concrete substitution $Subst[\theta]$, which performs substitution in our system. Thus, we have explicit substitutions at the type-level. Since quantifiers and substitutions are governed by different rules, we use sorts to distinguish between them. For example, $(\sigma T) \to (\sigma U)$ is equivalent to $\sigma(T \to U)$ if σ has sort Subst.

This equivalence forms the basis of the proof constructor $factor$:

$$factor : \forall \sigma, T_1, T_2.\ \sigma T_1 \to \sigma T_2 \ \dot{\leq}\ \sigma(T_1 \to T_2)$$

Some rules are valid for both quantifiers and substitutions, for example congruence and distribution over arrows. Rather than introduce separate proof constructors for distribution of quantifiers and substitutions, we use type variables ρ to range over either sort. Congruence states that if $T \subseteq U$, then $\rho T \subseteq \rho U$, where ρ is any substitution or quantifier set.

$$dist : \forall \rho, T_1, T_2.\ \rho(T_1 \to T_2)\ \dot{\leq}\ \rho T_1 \to \rho T_2$$

$$congr : \forall \rho, T_1, T_2. (T_1 \dot{\leq} T_2) \to (\rho T_1 \dot{\leq} \rho T_2)$$

Like $eArrow_1$, $eArrow$ constructs new proof constants from an existing proof term. While the input of $eArrow_1$ is a proof of subtyping between unquantified arrows, $eArrow$ accepts general proofs between quantified arrows. Therefore $eArrow$ must encode the general inversion lemma described above. $eArrow$ creates three new proof constructors, which witness the Sub-Subst step and the contravariant and covariant premises of the Sub-$\to$ step. It is not necessary for $eArrow$ to introduce a proof of Sub-Dist, since it is available as an axiom via the $dist$ proof constructor.

$$\begin{aligned}
eArrow : \forall T_1, T_2, U_1, U_2, T, \rho, \varphi_1.& \\
&(\rho(T_1 \to T_2) \dot{\leq} \varphi_1(U_1 \to T_2)) \to \\
&(\forall \varphi_3, \sigma. (\rho \hat{\leq} (\varphi_1 \circ \varphi_2 \circ \sigma)) \to \\
&\qquad (U_1 \dot{\leq} \varphi_2 \sigma T_1) \to \\
&\qquad (\varphi_2 \sigma T_2 \dot{\leq} U_2) \to \\
&\qquad T) \to T
\end{aligned}$$

The semantics of $eArrow_1$ is:

$$eArrow_1\ p\ f \longrightarrow f\ p_1\ p_2\ p_3$$

Where p_1, p_2 and p_3 are fresh proof constants. The constants p_2 and p_3 are much like the proof constants produced by $eArrow_1$. p_1 is a witness of the Sub-Subst step. If $p : \rho(T_1 \to T_2)\dot{\leq}\varphi_1(U_1 \to T_2)$, we will get the following types for the new proof constants:

$$p_1 : \rho\hat{\leq}(\varphi_1 \circ \varphi_2 \circ \sigma)$$
$$p_2 : U_1\dot{\leq}\varphi_2\sigma T_1$$
$$p_3 : \varphi_2\sigma T_2\dot{\leq}U_2$$

Proof Schemes The types φ_2 and σ will be instantiated to skolem constants, since they correspond to the existentially quantified $\vec{\gamma}$ and θ in the inversion lemma's conclusion. The proof constant p_1 is an example of a *proof scheme*, which have types of the form $\rho_1\hat{\leq}\rho_2$. A proof scheme of type $\varphi_1\hat{\leq}(\varphi_2 \circ \varphi_3 \circ \sigma)$ represents a Sub-Subst step from a type $\forall\vec{\alpha}.T$ to $\forall\vec{\beta},\vec{\gamma}.\theta(T)$ if $\varphi_1 = Forall[\vec{\alpha}]$, $\varphi_2 = Forall[\vec{\beta}]$, $\varphi_3 = Forall[\vec{\gamma}]$, and $\sigma = Subst[\theta]$. As with proof terms, we maintain a strong connection between proof schemes and subtyping: if $p : \rho_1\hat{\leq}\rho_2$, then $\rho_1 T \subseteq \rho_2 T$ for any type T.

We can apply the Sub-Subst step for any such T without violating the side conditions on the Sub-Subst step, by restricting the application of ρ_1 to types known to satisfy the side conditions. In effect, the presence of ρ_1 applied to a type T guarantees that the side condition holds. We can't always derive a type $\rho_1 T$ from a type T, but if a term exists with type $\rho_1 T$, we can derive $\rho_2 T$. As with proof terms, we define constructors to enable computing with proof schemes:

$$sTrans : \forall\varphi_1, \varphi_2, \varphi_3.(\varphi_1\hat{\leq}\varphi_2) \to (\varphi_2\hat{\leq}\varphi_3) \to (\varphi_1\hat{\leq}\varphi_3)$$
$$sCongr : \forall\varphi_1, \varphi_2, \varphi_3.(\varphi_1\hat{\leq}\varphi_2) \to (\varphi_3 \circ \varphi_1\hat{\leq}\varphi_3 \circ \varphi_2)$$

The constructor $sCongr$ encodes the Sub-Congr for proof schemes. The proof scheme constructor $sTrans$ encodes transitivity of proof schemes.

Figure 1 shows the final versions of *eBinary* and *expandK*. Note that in *eBinary*, the quantified σ in the type of f will be instantiated to $\sigma' \circ \sigma$. The definition of *expandK* is similar to $expandK_1$, except that it calls *eBinary* instead of $eBinary_1$. As before, we need only the first contravariant and the covariant proofs, so *expandK* discards the others.

Figure 2 shows the final version of $proveSK2KI$. The result of $proveSK2KI$ is again supplied via a continuation, though here the type includes substitution σ. The relationship between the types of f and p_{16} demonstrates how we can abstract away some of the irrelevant details of the type constraints.

5 Program Representations

Let us continue our study of the optimization $\mathrm{Opt} = SK \rightarrow KI$ and show how to program *analyzeSK* and *constructKI*. The challenge is to compute with program representations.

Program Representations Our typed program representations distinguish between programs and their representations, and can recover the type of a program from the type of its representation. For example, if an expression e has type T, its representation $'e$ will have type $Exp[T]$. We use the constructors Q and A to build program representations.

$$\begin{aligned} Ty[Q] &= \forall T.T \rightarrow Exp[T] \\ Ty[A] &= \forall T, U.Exp[T \rightarrow U] \rightarrow Exp[T] \rightarrow Exp[U] \end{aligned}$$

We use G to deconstruct program representations by pattern matching. When applied to an expression QO, G returns the underlying operator O. When applied to a compound $A\ e_1\ e_2$, G returns the components e_1 and e_2. The type of G

corresponds to the inversion lemma for applications: if $e_1\ e_2 : T$, there exists a type T_1 such that $e_1 : T_1 \to T$ and $e_2 : T_1$. The type T_1 is unknown, so G passes e_1 and e_2 to a continuation which can accept any T_1:

$$G\ a\ b\ (A\ c\ d) \longrightarrow b\ c\ d$$

Here G requires the type of b to be $\forall T_1.Exp[T_1 \to T] \to Exp[T_1] \to U$. The type of G is:

$$\begin{aligned}&\forall T, U.(T \to U) \to\\ &\qquad (\forall T_1.Exp[T_1 \to T] \to Exp[T_1] \to U) \to\\ &\qquad Exp[T] \to U\end{aligned}$$

When applied to a term t, $Is[O]$ will test if t is equal to O. If so, we can introduce a constraint on the type of t, namely that it is a supertype of $Ty[O]$. This is supported by the inversion lemma for operators, lemma 9. The reduction of $Is[O]$ generates a fresh proof constant as a witness for this subtype constraint, just as $eArrow$ does.

$$Is[O]\ O\ a\ b \longrightarrow\ a\ p\ O$$

If O has type T, then the proof constant p will have type $Ty[O] \dot{\leq} T$. The continuation a is also passed a copy of O at the type $Ty[O]$. The operator $IsIs$ is self-applicative, in that it can be used to recognize itself. It tests if its first argument t is any of $Is[O]$ or $IsIs$, and otherwise behaves similarly to $Is[O]$. The proof constant introduced by $IsIs$ proves that the type of t is a subtype of one of $Ty[Is[O]]$ or $Ty[IsIs]$. This is achieved by giving all such types a uniform structure. The ability of $IsIs$ to recognize itself "ties the knot", thus avoiding the potential infinite regress of operators $Is[Is[O]]$, $Is[Is[Is[O]]]$. This is a key aspect of the implementation of our typed self-interpreter described in section 9. $IsIs$ provides a copy of t at the type $Ty[t]$, which can be used to implement t metacircularly.

Analyzing SK In order to recognize that an expression $t =' (SK)$, we use G to test if t is an application of two operators $o_1 : T_1 \to T$ and $o_2 : T_1$. Then we use $Is[S]$ and $Is[K]$ to test if $o_1 = S$ and $o_2 = K$. If all these conditions are true, we will have produced the proof terms $pS : Ty[S] \dot{\leq} T_1 \to T$ and $pK : Ty[K] \dot{\leq} T_1$ needed by the $proveSK2KI$ function developed in the previous section.

We define helper functions $matchAtom$, $matchApp$, $matchS1$, and $matchK0$ that wrap $G, Is[S]$, and $Is[K]$ in order to clarify the code. Each has three arguments: false and true continuations, and an expression to match against. $matchAtom$ matches an expression against $Q\ x$ and returns x, while $matchApp$ matches against $A\ x\ y$ and returns x and y. $matchS1$ matches an expression against $A\ (Q\ S)\ x$ and returns x and the proof from $Is[S]$, and $matchK0$ matches an expression against $Q\ K$ and returns the proof from $Is[K]$. $analyzeSK$ calls $matchS1$ to get pS and x, then passes x to $matchK0$ to get pK, and returns pS and pK via the continuation $withIfSK$.

The function $constructKI$ defines an expression of $'(K\ I)$ with the type $\forall \varphi, T, U. \varphi(T \to U \to U)$ which matches the proof computed by $proveSK2KI$. The construction requires a distribution step on K to align the occurrences of φ. This is an implementation detail; we would have been justified in constructing $(K\ I)$ with the type $\forall T, U. T \to U \to U$ and introducing φ using the equivalence rule $E9$. For simplicity, our typechecker only allows rule $E9$ to be used at atoms, so so we compensate by adding the explicit $dist$ coercion.

The function $SK2KI$ assembles $analyzeSK$, $proveSK2KI$, and $constructKI$ into the complete optimization.

$$
\begin{array}{l}
\mathtt{let}\;\;(\mathit{eBinary} : \forall\varphi, T, U, V, \varphi', T', U', V', X. \\
\qquad\qquad\qquad (\varphi(T \to U \to V)\dot{\leq}\varphi'(T' \to U' \to V')) \to \\
\qquad\qquad\qquad (\forall\varphi'', \sigma.(\varphi\hat{\leq}\varphi' \circ \varphi'' \circ \sigma) \to \\
\qquad\qquad\qquad\qquad\quad (T'\dot{\leq}\varphi''\sigma T) \to (U'\dot{\leq}\varphi''\sigma U) \to (\varphi''\sigma V\dot{\leq}V') \to X) \to \\
\qquad\qquad\qquad X) = \\
\;\lambda(p_1 : \varphi(T \to U \to V)\dot{\leq}\varphi'(T' \to U' \to V')).\ \mathrm{eArrow}\ p_1 \\
\quad(\lambda(p_2 : \varphi\hat{\leq}\varphi' \circ \varphi'' \circ \sigma). \\
\quad\;\lambda(p_3 : T'\dot{\leq}\varphi''\sigma T). \\
\quad\;\lambda(p_4 : \varphi''\sigma(U \to V)\dot{\leq}U' \to V'). \\
\quad\;\mathrm{eArrow}\ p_4 \\
\qquad(\lambda(p_5 : \varphi'' \circ \sigma\hat{\leq}\varphi''' \circ \sigma').\lambda(p_6 : U'\dot{\leq}\varphi'''\sigma' U).\lambda(p_7 : \varphi'''\sigma' V\dot{\leq}V'). \\
\qquad\;\mathtt{let}\;\;(p_8 : \varphi' \circ \varphi'' \circ \sigma\hat{\leq}\varphi' \circ \varphi''' \circ \sigma') = \;p_5\;\;\mathtt{in} \\
\qquad\;\mathtt{let}\;\;(p_9 : \varphi\hat{\leq}\varphi' \circ \varphi''' \circ \sigma') = \;p_2\ p_8\;\;\mathtt{in} \\
\qquad\;\mathtt{let}\;\;(p_{10} : T'\dot{\leq}\varphi'''\sigma' T) = \mathrm{trans}\ p_3\ p_5\;\;\mathtt{in} \\
\qquad\;\lambda(f : \forall\varphi'', \sigma.(\varphi\hat{\leq}\varphi' \circ \varphi'' \circ \sigma) \to \\
\qquad\qquad\qquad (T'\dot{\leq}\varphi''\sigma T) \to (U'\dot{\leq}\varphi''\sigma U) \to (\varphi''\sigma V\dot{\leq}V') \to X). \\
\qquad\;f\ p_9\ p_{10}\ p_6\ p_7))
\end{array}
$$

$$
\begin{array}{l}
\mathtt{let}\;\;(\mathit{expandK} : \forall T, U, V.Ty[\mathrm{K}]\dot{\leq}\varphi(T \to U \to V) \to (T\dot{\leq}V)) \;= \\
\quad\lambda(pK : Ty[\mathrm{K}]\dot{\leq}\varphi(T \to U \to V)). \\
\quad\mathit{eBinary}\ pK\ (\mathrm{K}\ (\lambda(p_1 : T\dot{\leq}\varphi_2\sigma_1 X_1).\mathrm{K}\ (\lambda(p_2 : \varphi_2\sigma X_1\dot{\leq}V).\mathrm{trans}\ p_1\ p_2)))
\end{array}
$$

Figure 1: Implementations of eBinary and expandK

$$
\begin{array}{l}
\texttt{let}\ \ (analyzeSK : \forall T, U. \\
\qquad\qquad\qquad U \to (\forall T_1.(Ty[\mathrm{S}]\dot{\leq}T_1 \to T) \to (Ty[\mathrm{K}]\dot{\leq}T_1) \to U) \to \\
\qquad\qquad\qquad Exp[T] \to U)\ = \\
\ \lambda(ifNotSK : U).\lambda(ifSK : \forall T_1.(Ty[\mathrm{S}]\dot{\leq}T_1 \to T) \to (Ty[\mathrm{K}]\dot{\leq}T_1) \to U). \\
\ matchS1\ ifNotSK \\
\quad (\lambda(pS : Ty[\mathrm{S}]\dot{\leq}T_1 \to T). \\
\qquad matchK0\ ifNotSK\ (\lambda(pK : Ty[\mathrm{K}]\dot{\leq}T_1).\ ifSK\ pS\ pK))\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (proveSK2KI : \forall T_1, T, U. \\
\qquad\qquad\qquad ((Exp[\forall\varphi, U, V.\varphi(U \to V \to V)]\dot{\leq}Exp[T]) \to U) \to \\
\qquad\qquad\qquad (Ty[\mathrm{S}]\dot{\leq}T_1 \to T) \to (Ty[\mathrm{K}]\dot{\leq}T_1) \to U)\ = \\
\ \lambda(f : (Exp[\forall\varphi, U, V.\varphi(U \to V \to V)]\dot{\leq}Exp[T]) \to U). \\
\ \lambda(p_1 : Ty[\mathrm{S}]\dot{\leq}T_1 \to T).\ \lambda(p_2 : Ty[\mathrm{K}]\dot{\leq}T_1).\ \mathrm{eArrow}\ p_1 \\
\qquad (\lambda(p_3 : \varphi_1\hat{\leq}\varphi_3 \circ \sigma). \\
\qquad\ \lambda(p_4 : T_1\dot{\leq}\varphi_3\sigma(B_1 \to B \to C)). \\
\qquad\ \lambda(p_5 : \varphi_3\sigma((B_1 \to B) \to B_1 \to C)\dot{\leq}T)\,. \\
\qquad\ \texttt{let}\ \ (p_6 : \varphi_2(T \to U \to T)\dot{\leq}\varphi_3(\sigma B_1 \to \sigma B \to \sigma C))\ = \\
\qquad\quad \mathrm{trans}\ (\mathrm{trans}\ p_2\ p_4)\ (\mathrm{congr\ dist2})\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{11} : \sigma B_1\dot{\leq}\sigma C) = expandK\ p_6\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{12} : \sigma B_1 \to \sigma B_1\dot{\leq}\sigma B_1 \to \sigma C) = \mathrm{iArrow\ refl}\ p_{11}\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{13} : \sigma B_1 \to \sigma B_1\dot{\leq}\sigma(B_1 \to C)) = \mathrm{trans}\ p_{12}\ \mathrm{factor}\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{14} : \sigma(B_1 \to B) \to \sigma B_1 \to \sigma B_1\dot{\leq}\sigma(B_1 \to B) \to \sigma(B_1 \to C))\ = \\
\qquad\quad \mathrm{iArrow\ refl}\ p_{13}\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{15} : \sigma(B_1 \to B) \to \sigma B_1 \to \sigma B_1\dot{\leq}\sigma((B_1 \to B) \to B_1 \to C))\ = \\
\qquad\quad \mathrm{trans}\ p_{14}\ \mathrm{factor}\ \ \texttt{in} \\
\qquad\ \texttt{let}\ \ (p_{16} : \varphi_3(\sigma(B_1 \to B) \to \sigma B_1 \to \sigma B_1)\dot{\leq}T)\ = \\
\qquad\quad \mathrm{trans}\ (\mathrm{congr}\ p_{15})\ p_5\ \ \texttt{in} \\
\qquad\ f\ (\mathrm{iExp}\ p_{16}))\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (constructKI : Exp[\forall\varphi, T, U.\varphi(T \to U \to U)])\ = \\
\quad \mathrm{A}\ (\mathrm{Q}\ (\mathrm{coerce\ K\ dist}))\ (\mathrm{Q\ I})\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (SK2KI : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U)\ = \\
\quad \lambda(ifNoOpt : U).\lambda(ifOpt : Exp[T] \to U). \\
\quad analyzeSK\ ifNoOpt \\
\qquad (proveSK2KI(\lambda(p : Exp[\forall\varphi, U, V.\varphi(U \to V \to V)]\dot{\leq}Exp[T])\,. \\
\qquad\quad ifOpt\ (\mathrm{coerce}\ constructKI\ p)))
\end{array}
$$

Figure 2: Implementation of SK2KI

6 Our Self-optimizer

Figure 3 shows the core of our self-optimizer, consisting of 3 key functions. The complete optimizer applies optimization steps *eta*, *reduceK*, and *reduceS*, in addtion to $SK2KI$. The definitions of these optimization steps are listed in Appendix B. Each is implemented in the same style as $SK2KI$, though the specifics vary in each case. We will describe each new function in figure 3 in this section.

The core of our framework consists of several high level functions useful for implementing optimizers with type $\forall T.Exp[T] \to Exp[T]$. *composeOpt* composes two optimization steps such as $SK2KI$ into a larger step. The two arguments and result of *composeOpt* have the type of a single optimization step in the framework: $\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U$. The first argument of type U is returned if the optimization step fails to apply. The second is a continuation which accepts the optimized program as input. The third is the original program. The result of the optimization is either the first argument or the result of calling the continuation.

The main driver of our framework is *runOpt*. It builds a complete optimizer with type $\forall T.Exp[T] \to Exp[T]$ from a single optimization step, by applying it repeatedly to an input program until no further optimizations can be applied. It uses the function $traverse$ to walk over the input, applying the step at each subexpression. If an optimization is applied, the entire expression is reconstructed and a new scan begins over the result.

Our full optimizer consists of three optimization steps other than $SK2KI$: *eta* performs η reduction, and *reduceK* and *reduceS* implement the reduction rules for K and S respectively.

$$
\begin{array}{l}
\texttt{let}\ \ (composeOpt : (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
\qquad\qquad\qquad\quad (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
\qquad\qquad\qquad\quad (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U)) \ = \\
\ \ \lambda(opt1 : \forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
\ \ \lambda(opt2 : \forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
\ \ \lambda(noOpt : U).\ \lambda(ifOpt : Exp[T] \to U).\ \lambda(e : Exp[T]). \\
\ \ opt1\ (opt2\ noOpt\ ifOpt\ e)\ ifOpt\ e\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (traverse : (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
\qquad\qquad\qquad (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U)) \ = \\
\quad \lambda(f' : \forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U)\,. \\
\quad\ \texttt{let}\ \ \texttt{rec}\ \ (traverseF : \forall T.U \to (Exp[T] \to U) \to Exp[T] \to U) \ = \\
\qquad \lambda(ifNoOpt : U).\ \lambda(ifOpt : Exp[T] \to U).\ \lambda(e' : Exp[T]). \\
\qquad\ \texttt{let}\ \ (tryApp : U) = \mathrm{G}\ (\mathrm{K}\ ifNoOpt) \\
\qquad\qquad\quad (\,\lambda(e1 : Exp[T' \to T]).\lambda(e2 : Exp[T']). \\
\qquad\qquad\qquad \texttt{let}\ \ (tryE2 : U)\ =\ traverseF\ ifNoOpt \\
\qquad\qquad\qquad\quad (\lambda(newE2 : Exp[T']).\ ifOpt\ (\mathrm{A}\ e1\ newE2))\ e2\ \ \texttt{in} \\
\qquad\qquad\qquad traverseF\ tryE2\ (\lambda(newE1 : Exp[T' \to T]). \\
\qquad\qquad\qquad\qquad traverseF\ (ifOpt\ (\mathrm{A}\ newE1\ e2)) \\
\qquad\qquad\qquad\qquad\quad (\,\lambda(newE2 : Exp[T']).\ ifOpt\ (\mathrm{A}\ newE1\ newE2)) \\
\qquad\qquad\qquad\qquad\quad e2)\ e1)\ e'\ \ \texttt{in} \\
\qquad f'\ tryApp\ ifOpt\ e'\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (runOpt : (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
\qquad\qquad\qquad Exp[T] \to Exp[T]) \ = \\
\quad\ \texttt{let}\ \ \texttt{rec}\ \ (runOpt_1 : (\forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
\qquad\qquad\qquad\qquad\qquad Exp[T] \to Exp[T]) \ = \\
\qquad \lambda(f : \forall T,U.U \to (Exp[T] \to U) \to Exp[T] \to U).\ \lambda(e : Exp[T]). \\
\qquad traverse\ f\ e\ (runOpt_1\ f)\ e\ \ \texttt{in}
\end{array}
$$

```
// SK2KI defined in Figure 2
```
$\texttt{let}\ \ (SK2KI : \forall T.U \to (Exp[T] \to U) \to Exp[T] \to U)\ =\ \ldots\ \ \texttt{in}$

```
// eta, reduceK, and reduceS defined in Appendix B
```
$\texttt{let}\ \ (eta : \forall T.U \to (Exp[T] \to U) \to Exp[T] \to U)\ =\ \ldots\ \ \texttt{in}$
$\texttt{let}\ \ (reduceK : \forall T.U \to (Exp[T] \to U) \to Exp[T] \to U)\ =\ \ldots\ \ \texttt{in}$
$\texttt{let}\ \ (reduceS : \forall T.U \to (Exp[T] \to U) \to Exp[T] \to U)\ =\ \ldots\ \ \texttt{in}$

$runOpt\ (composeOpt4\ SK2KI\ eta\ reduceK\ reduceS)$

Figure 3: A Self-Optimizer

7 Our Language

This section formalizes our term language and type system.

7.1 Syntax

Definition 1. *Our core term language is defined by the grammar:*

$$
\begin{aligned}
e &::= e_1\ e_2 \mid atom \mid p \mid x \\
atom &::= O \mid P \mid Is[O] \mid IsIs \\
O &::= S \mid K \mid I \mid Y \mid Q \mid A \mid G \mid coerce \mid eArrow \\
P &::= refl \mid iArrow \mid iExp \mid eExp \\
&\quad \mid dist \mid distExp \mid factor \mid factorExp \\
&\quad \mid congr \mid sCongr \mid trans \mid sTrans
\end{aligned}
$$

The term language is a combinatory calculus consisting of applications, atoms, proof constants, and variables. This gives the property that all programs can be uniformly represented as binary trees. We include term variables in order to model syntactic sugar for lambda abstraction, let, and let rec.

The atoms consist of the operators O, a set of proof constructors P, $Is[O]$, and $IsIs$. Each operator O has a corresponding atom $Is[O]$, which tests for equality. Similarly, $IsIs$ tests if its argument is one of $Is[O]$ or $IsIs$. The operators O include the traditional combinators S,K,I, and Y, as well as operators for computing with program representations and proof terms. As shown in definition 2, we use Q and A to construct representations, and G to deconstruct them.

Definition 2. *Quotation*

$$
\begin{aligned}
{}'O &= Q\ O \\
{}'(a\ b) &= A\ {}'a\ {}'b
\end{aligned}
$$

The atoms $Is[O]$ and $IsIs$ are similar to Church booleans, and additionally introduce proof constants p in the true case. The proof constructors P, which

Arity	Atoms
0	refl, dist, distExp, factor, factorExp
1	I, Y, Q, iExp, eExp, congr, sCongr
2	K, A, coerce, eArrow, iArrow, trans, sTrans
3	S, G

Figure 4: Arities of Atoms

are similar to those given in Donnelly's Master's thesis [7], have no semantics aside from their type. Their purpose is to construct proofs of the existence of subtype relationships in terms of known axioms, and the dynamically generated proof constants. Such a proof can be used by *coerce* to change the type of a term. Similarly, *eArrow* effectively decomposes a proof term into three component proof terms. Both *coerce* and *eArrow* validate proofs by evaluation.

7.2 Semantics

Our atoms consist of constructors and operators. A and Q construct representations of programs, while the proof constructors P construct proofs. Each atom has an arity, as defined in figure 4. Proof constructors with arity 0 are also called proof axioms. A proof constructor with arity > 0 is analogous to a deduction with a number of premises equal to contructor's arity, and the result type of the constructor corresponding to the logical conclusion.

Definition 3. *Value*

$$\begin{aligned} v ::= &\ atom\ e_1\ \ldots\ e_i,\ \text{where } i < arity(atom) \\ &\mid Q\ e\ \ \mid A\ e_1\ e_2 \\ &\mid P\ v_1\ \ldots\ v_i,\ \text{where } i = arity(P) \end{aligned}$$

A value is either a partially applied operator, or a fully applied constructor. Proof constructors are strict, so that their components are required to be fully evaluated, while the expression constructors Q and A are lazy.

$$
\begin{aligned}
S\ a\ b\ c &\longrightarrow a\ c\ (b\ c) \\
K\ a\ b &\longrightarrow a \\
I\ a &\longrightarrow a \\
Y\ t &\longrightarrow t\ (Y\ t) \\
G\ a\ b\ (Q\ c) &\longrightarrow a\ c \\
G\ a\ b\ (A\ c\ d) &\longrightarrow b\ c\ d \\
Is[O]\ O\ a\ b &\longrightarrow a\ p\ O && p \text{ fresh} \\
Is[O]\ v\ a\ b &\longrightarrow b && \text{if } v \neq O \\
IsIs\ v\ a\ b &\longrightarrow a\ p\ v && p \text{ fresh, if } v \in \{Is[O], IsIs\} \\
IsIs\ v\ a\ b &\longrightarrow b && \text{otherwise} \\
\mathit{coerce}\ e\ v &\longrightarrow e \\
\mathit{eArrow}\ v\ e &\longrightarrow e\ p_1\ p_2\ p_3 && \text{where } p_1, p_2, p_3 \text{ are fresh}
\end{aligned}
$$

Figure 5: Operational Semantics

The operational semantics is given in figure 5. S, K, I, Y are fully lazy, while G, *coerce*, $Is*$ and *eArrow* are partially strict. In particular, G is strict in its third argument, *coerce* is strict in its second argument, and *eArrow* and the $Is*$ operators are strict in their first argument. Coerce and eArrow fully evaluate their proof term argument in order to validate a corresponding subtype proposition. This prevents coercions based on nonterminating proofs terms, which could otherwise be used to prove anything.

7.3 Types

Definition 4. *Our type language is defined by the grammar:*

$$
\begin{aligned}
T ::= {} & \alpha \mid T_1 \to T_2 \mid Exp[T] \mid \varphi \mid \forall[\vec{\alpha}] \mid \sigma \mid Subst[\theta] \mid \rho \\
& \mid T_1\ T_2 \mid T_1 \dot{\leq} T_2 \mid T_1 \hat{\leq} T_2 \mid T_1 \circ T_2
\end{aligned}
$$

We use $\vec{\alpha}$ to denote sets of quantifiers $T_1, T_2, \ldots, T_n$. Function values (*atom* $e_1 \ldots e_i$, where $i < arity(atom)$) are given arrow types of the form $T_1 \to T_2$.

An expression type $Exp[T]$ is assigned to a program representation $'e$, if the underlying program e has type T. The proof types $T_1 \dot{\leq} T_2$ and proof type schemes $T_1 \hat{\leq} T_2$ are assigned to proof terms. A proof type $T_1 \dot{\leq} T_2$ proposes the existence of a subtyping relationship between T_1 and T_2. A proof type scheme $T_1 \hat{\leq} T_2$ proposes the existence of a subtype relationship between $T_1\ T$ and $T_2\ T$ for any type T. As will be shown in lemmas 22 and 23, a proof value validates the proposition corresponding to its type.

A sequence of type variables $\vec{\alpha}$ are bound by the type constructor $\forall[\vec{\alpha}]$. The type constructor $Subst[\theta]$ contains a type substitution θ, which is a partial function from type variables to types as usual. A type application $T_1 T_2$ applies a type constructors to a type. The composition of two type constructors is denoted $T_1 \circ T_2$, and is itself a type constructor.

The type constructor $Subst[\theta]$ amounts to an explicit type substitution. The usual definition of applying a type substitution to a type is now encoded using type equivalence. We define standard equivalences such as α-equivalence, and establish associativity for composition of type constructors.

Definition 5. *Bound and Free Type Variables*

$$
\begin{aligned}
BV(\alpha) &= \emptyset\\
BV(T_1 \to T_2) &= \emptyset\\
BV(Exp[T]) &= BV(T)\\
BV(\varphi) &= \emptyset\\
BV(\forall[\vec{\alpha}]) &= \vec{\alpha}\\
BV(\sigma) &= \emptyset\\
BV(Subst[\theta]) &= \emptyset\\
BV(\rho) &= \emptyset\\
BV(T_1\ T_2) &= BV(T_1) \cup BV(T_2)\\
BV(T_1 \dot{\leq} T_2) &= \emptyset\\
BV(T_1 \hat{\leq} T_2) &= \emptyset\\
BV(T_1 \circ T_2) &= BV(T_1) \cup BV(T_2)
\end{aligned}
$$

$$
\begin{aligned}
FV(\alpha) &= \alpha\\
FV(T_1 \to T_2) &= FV(T_1) \cup FV(T_2)\\
FV(Exp[T]) &= FV(T)\\
FV(\varphi) &= \varphi\\
FV(\forall[\vec{\alpha}]) &= \emptyset\\
FV(\sigma) &= \sigma\\
FV(Subst[\theta]) &= FV(\theta)\\
FV(\rho) &= \rho\\
FV(T_1 T_2) &= (FV(T_2) - BV(T_1)) \cup FV(T_1)\\
FV(T_1 \dot{\leq} T_2) &= FV(T_1) \cup FV(T_2)\\
FV(T_1 \hat{\leq} T_2) &= FV(T_1) \cup FV(T_2)\\
FV(T_1 \circ T_2) &= (FV(T_2) - BV(T_1)) \cup FV(T_1)
\end{aligned}
$$

We use $\forall \alpha_1, \ldots, \alpha_n.T$ as syntactic sugar for $\forall[\alpha_1, \ldots, \alpha_n](T)$.

Definition 6.

$$(kinds)\kappa ::= * \mid * \to *$$

$$K\text{-}\varphi\ \frac{}{\varphi :: * \to *}$$

$$K\text{-}\rho\ \frac{}{\rho :: * \to *}$$

$$\textit{K-}\sigma\ \frac{}{\sigma :: * \to *}$$

$$\textit{K-TVar}\ \frac{}{\alpha :: *}$$

$$\textit{K-}\forall\ \frac{}{\forall[\vec{\alpha}] :: * \to *}$$

$$\textit{K-Subst}\ \frac{}{\mathit{Subst}[\theta] :: * \to *}$$

$$\textit{K-Arrow}\ \frac{T_1 : * \qquad T_2 : *}{T_1 \to T_2 :: *}$$

$$\textit{K-App}\ \frac{T_1 :: * \to * \qquad T_2 :: *}{T_1\ T_2 :: *}$$

$$\textit{K-Exp}\ \frac{T :: *}{\mathit{Exp}[T] :: *}$$

$$\textit{K-Proof}\ \frac{T_1 :: * \qquad T_2 :: *}{T_1 \dot{\leq} T_2 :: *}$$

$$\textit{K-SubstProof}\ \frac{T_1 :: * \to * \qquad T_2 :: * \to *}{T_1 \hat{\leq} T_2 :: *}$$

$$\textit{K-Compose}\ \frac{T_1 :: * \to * \qquad T_2 :: * \to *}{T_1 \circ T_2 :: * \to *}$$

Each type in our system is either of kind $* \to *$, the kind of proof constructors, or the kind of types $*$. We use the convention that type variables φ, σ, ρ range over kind $* \to *$, while all others range over kind $*$. An alternative approach would be to store the kind of type variables in the context Γ.

Our definition of sorts differentiates between type constructors that represent quantifiers, those that represent substitutions, and those that represent a composite of quantifiers and substitutions. In particular, a type $\rho(T \to U)$ is equivalent to $\rho T \to \rho U$ if and only if ρ represents a substitution. We define a least upper bound between sorts $s_1 \sqcup s_2$, where $s_1 \sqcup s_2 = \mathit{Any}$ if $s_1 \neq s_2$.

Definition 7.

$$(sorts) s ::= Subst \mid Forall \mid Any$$

$$\textit{S-}\varphi \, \frac{}{\varphi ::: Forall}$$

$$\textit{S-}\sigma \, \frac{}{\sigma ::: Subst}$$

$$\textit{S-}\rho \, \frac{}{\rho ::: Any}$$

$$\textit{S-}\forall \, \frac{}{\forall[\vec{\alpha}] ::: Forall}$$

$$\textit{S-Subst} \, \frac{}{Subst[\theta] ::: Subst}$$

$$\textit{S-Compose} \, \frac{T_1 ::: s_1 \qquad T_2 ::: s_2}{T_1 \circ T_2 ::: (s_1 \sqcup s_2)}$$

We define equivalence between types. This forms the mechanism for applying substitutions, and supports the types of several proof axioms. In particular: $distExp$ and $factorExp$ correspond with $E2$, and $dist$ for Subst sorts and $factor$ correspond with $E3$.

Definition 8. *Type Equivalence*

$$
\begin{array}{lrcl}
(E1) & Subst[\theta]\alpha & \equiv & T \quad \text{if } \theta(\alpha) = T \\
(E2) & \rho Exp[T] & \equiv & Exp[\rho T] \\
(E3) & \sigma(T \to U) & \equiv & \sigma T \to \sigma U \\
(E4) & \sigma(T_1 \dot{\leq} T_2) & \equiv & (\sigma T_1)\dot{\leq}(\sigma T_2) \\
(E5) & \sigma(T_1 \hat{\leq} T_2) & \equiv & (\sigma T_1)\hat{\leq}(\sigma T_2) \\
(E6) & \sigma \circ \forall[\vec{\alpha}] & \equiv & \forall[\vec{\beta}] \circ \sigma \circ Subst[[\vec{\beta}/\vec{\alpha}]] \\
 & & & \text{where } \vec{\beta} \text{ are fresh.} \\
(E7) & \forall[\vec{\alpha}] & \equiv & \forall[\vec{\beta}] \circ Subst[[\vec{\beta}/\vec{\alpha}]] \\
(E8) & Subst[\theta](\alpha T) & \equiv & U(Subst[\theta]T) \\
 & & & \text{if } \theta(\alpha) = U \\
(E9) & \forall[\alpha](T \dot{\leq} U) & \equiv & T \dot{\leq} \forall[\alpha]U \quad \text{if } \alpha \notin FV(T) \\
(E10) & (\rho_1 \circ \rho_2)T & \equiv & \rho_1\rho_2 T \\
(E11) & \rho_1 \circ (\rho_2 \circ \rho_3) & \equiv & (\rho_1 \circ \rho_2) \circ \rho_3 \\
(E12) & \forall[\vec{\alpha}, \vec{\beta}] & \equiv & \forall[\vec{\beta}, \vec{\alpha}] \\
(E13) & \forall[\vec{\alpha}] \circ \forall[\vec{\beta}] & \equiv & \forall[\vec{\alpha}, \vec{\beta}] \\
(E14) & \forall[\emptyset]T & \equiv & T \\
(E15) & Subst[\theta_1 \circ \theta_2] & \equiv & Subst[\theta_1] \circ Subst[\theta_2]
\end{array}
$$

Our type system extends Mitchell's F-η subtyping for our type syntax.

Definition 9. *Subtyping:*

$$\textit{Sub-Refl}\,\frac{}{T \subseteq T}$$

$$\textit{Sub-Trans}\,\frac{T \subseteq U \qquad U \subseteq V}{T \subseteq V}$$

$$\textit{Sub-}\!\to\,\frac{T' \subseteq T \qquad U \subseteq U'}{T \to U \subseteq T' \to U'}$$

$$\textit{Sub-Dist-}\!\to\,\frac{}{\forall\vec{\alpha}.T \to U \subseteq (\forall\vec{\alpha}.T) \to (\forall\vec{\alpha}.U)}$$

$$\textit{Sub-Congr}\,\frac{T \subseteq U}{\rho T \subseteq \rho U}$$

$$\textit{Sub-Subst}\ \frac{}{\forall\vec{\alpha}.T \subseteq \forall\vec{\beta}.Subst[\theta]T}\ dom(\theta) = \vec{\alpha},\ \ \vec{\beta} \notin FV(\forall\vec{\alpha}.T)$$

$$\textit{Sub-Exp}\ \frac{T \subseteq U}{Exp[T] \subseteq Exp[U]}$$

$$\textit{Sub-Dist-Exp}\ \frac{}{\varphi Exp[T] \subseteq Exp[\varphi T]}$$

$$\textit{Sub-Dist-}\dot{\leq}\ \frac{}{\varphi(T\dot{\leq}U) \subseteq \varphi T\dot{\leq}\varphi U}$$

$$\textit{Sub-Dist-}\hat{\leq}\ \frac{}{\varphi(T\hat{\leq}U) \subseteq \varphi T\hat{\leq}\varphi U}$$

$$\textit{Sub-Proof-Inst}\ \frac{}{(\rho_1\hat{\leq}\rho_2) \subseteq (\rho_1 T\dot{\leq}\rho_2 T)}$$

The rules Sub-Refl, Sub-Trans, Sub-$\rightarrow$, Sub-Dist-$\rightarrow$, Sub-Congr are unchanged from Mitchell's formulation. Sub-Subst is adapted to our style of explicit type substitutions. Sub-Exp establishes congruence for expression types. Sub-Dist-Exp, Sub-Dist-$\dot{\leq}$, and Sub-Dist-$\hat{\leq}$ distributes type constructors of sort Forall into expression types, proof types, and proof type schemes, respectively.

Well-formedness of Γ ensures that the types of all proof constants p contained in Γ are valid. We rely on well-formedness in the proofs of lemmas 22 and 23, and maintain it in the cases of theorem 24 for $Is*$ and $eArrow$.

Definition 10. *Well-formedness of type environment* Γ*.*

$$\frac{\vdash \Gamma}{\vdash \Gamma, (p : \forall\vec{\alpha}.(T\dot{\leq}U))}\ T \subseteq U$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, (p : \forall\vec{\alpha}.(\sigma_1\hat{\leq}\sigma_2))}\ \sigma_1 T \subseteq \sigma_2 T\ \textit{for any}\ T$$

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

$$
\begin{aligned}
& \textbf{Operators} \\
Ty[S] =& \forall T, U, V.(T \to U \to V) \to (T \to U) \to T \to V \\
Ty[K] =& \forall T, U.T \to U \to T \\
Ty[I] =& \forall T.T \to T \\
Ty[Y] =& \forall T.(T \to T) \to T \\
Ty[Q] =& \forall T.T \to Exp[T] \\
Ty[A] =& \forall T, U.Exp[T \to U] \to Exp[T] \to Exp[U] \\
Ty[G] =& \forall T, U.(T \to U) \to (\forall V.Exp[V \to T] \to Exp[V] \to U) \to \\
& \quad Exp[T] \to U \\
Ty[coerce] =& \forall T, U.T \to (T \dot{\leq} U) \to U \\
Ty[eArrow] =& \forall \rho, \varphi, T, U, T', U', V.(\rho(T \to U) \dot{\leq} \varphi(T' \to U')) \to \\
& (\forall \varphi_1, \sigma.(\rho \hat{\leq} (\varphi \circ \varphi_1 \circ \sigma)) \to (T' \dot{\leq} \varphi_1 \sigma T) \to (\varphi_1 \sigma U \dot{\leq} U') \to V) \to \\
& V \\
Ty[Is[O]] =& \forall T, U.T \to ((Ty[O] \dot{\leq} T) \to Ty[O] \to U) \to U \to U \\
Ty[IsIs] =& \forall T, U.T \to ((IsTy \dot{\leq} T) \to IsTy \to U) \to U \to U \\
& where\ IsTy = \forall T, U, V.T \to ((V \dot{\leq} T) \to V \to U) \to U \to U \\
& \textbf{Proof Axioms} \\
Ty[refl] =& \forall T.(T \hat{\leq} T) \\
Ty[dist] =& \forall \rho, T, U.(\rho(T \to U) \dot{\leq} \rho T \to \rho U) \\
Ty[distExp] =& \forall \rho, T.(\rho Exp[T] \dot{\leq} Exp[\rho T]) \\
Ty[factor] =& \forall \sigma, T, U.(\sigma T \to \sigma U \dot{\leq} \sigma(T \to U)) \\
Ty[factorExp] =& \forall \rho, T.(Exp[\rho T] \dot{\leq} \rho Exp[T]) \\
& \textbf{Proof Constructors} \\
Ty[iArrow] =& \forall T, U, T', U'.(T' \dot{\leq} T) \to (U \dot{\leq} U') \to (T \to U \dot{\leq} T' \to U') \\
Ty[iExp] =& \forall T, U.(T \dot{\leq} U) \to (Exp[T] \dot{\leq} Exp[U]) \\
Ty[eExp] =& \forall T, U.(Exp[T] \dot{\leq} Exp[U]) \to (T \dot{\leq} U) \\
Ty[congr] =& \forall \rho, T, U.(T \dot{\leq} U) \to (\rho T \dot{\leq} \rho U) \\
Ty[sCongr] =& \forall \rho, T, U.(T \hat{\leq} U) \to (\rho T \hat{\leq} \rho U) \\
Ty[trans] =& \forall T, U, V.(T \dot{\leq} U) \to (U \dot{\leq} V) \to (T \dot{\leq} V) \\
Ty[sTrans] =& \forall \rho_1, \rho_2, \rho_3, \rho_4.(\rho_1 \hat{\leq} \rho_2 \circ \rho_3) \to (\rho_2 \hat{\leq} \rho_4) \to (\rho_1 \hat{\leq} \rho_4 \circ \rho_3)
\end{aligned}
$$

Figure 6: Atom Types

Definition 11 shows the type rules. The types of atoms $Ty[atom]$ are defined in figure 6. The rules for variables and applications are standard. The Type-Subtype rule additionally checks that the subtyping step results in a well formed type of kind $*$. As mentioned previously, our syntactic kind rules for type variables allow us to check $U :: *$ without a kind context.

Definition 11. *Type Rules*

$$\textit{Type-Atom}\ \frac{}{\Gamma \vdash atom : Ty[atom]}$$

$$\textit{Type-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\textit{Type-Proof-Constant}\ \frac{p : T \in \Gamma}{\Gamma \vdash p : T}$$

$$\textit{Type-App}\ \frac{\Gamma \vdash e_1 : T \to U \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : U}$$

$$\textit{Type-Subtype}\ \frac{\Gamma \vdash e : T \qquad T \subseteq U \qquad U :: *}{\Gamma \vdash e : U}$$

7.4 Lambda Abstraction and Let-Terms

For the purpose of practical programming, particularly of our self-optimizer and self-enactor, we use three forms of syntactic sugar:

- λ-abstraction, written $\lambda(x : T).e$
- let binding, written $let\ (x : T) = e_1\ in\ e_2$ and
- let rec binding, written $let\ rec\ (x : T) = e$

We desugar terms with such constructs before executing them. Desugaring maps closed terms to closed terms.

One of the oldest results on computability is that λ-abstraction can be defined by SKI-terms (e.g. [9]). The definition of $\lambda x.e$ is as follows.

$$\begin{array}{rcll} \lambda x.x & = & I & \\ \lambda x.e & = & K\ e & \text{if } e \text{ avoids } x \\ \lambda x.(e_1\ e_2) & = & S(\lambda x.e_1)\ (\lambda x.e_2) & \text{otherwise .} \end{array}$$

Lemma 1. *For all terms e_1 and e_2 and variable x there is a reduction*

$$(\lambda x.e_1)\ e_2 \longrightarrow^* [u/x]e\ .$$

Lemma 2. *The following rule can be derived for abstractions*

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U}\ .$$

Corollary 3. *Mitchell's $Abs_\forall$ rule [15, p.127] can be derived:*

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : \forall\vec{\alpha}.T \to U}\ \vec{\alpha} \notin FV(\Gamma)$$

We desugar the syntax $let\ x = e_1\ in\ e_2$ to $(\lambda x.e_2)e_1$ and we de-sugar $let\ rec\ x = e$ to $Y\ (\lambda x.e)$, as usual.

Lemma 4. *The following rules can be derived for let-terms*

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \forall\vec{\alpha}.T_2}\ \vec{\alpha} \notin FV(\Gamma)$$

$$\frac{\Gamma, x : T \vdash e_1 : T}{\Gamma \vdash let\ rec\ x = e_1 : \forall\vec{\alpha}.T}\ \vec{\alpha} \notin FV(\Gamma)$$

Lemma 5. *If $\alpha \in FV(T)$ and $T \subseteq U$, then $\alpha \in FV(U)$.*

7.5 Soundness

Lemma 6. *If $T \subseteq U$ and $\alpha \notin FV(T)$, then $T \subseteq \forall[\alpha]U$.*

Lemma 7. *The following $\forall$-intro rule is admissable:*

$$\forall\text{-}intro\ \frac{\Gamma \vdash e : T}{\Gamma \vdash e : \forall[\alpha]T}\ \vec{\alpha} \cap FV(\Gamma) = \emptyset$$

Lemma 8. *Any type T of the form $\vec{\sigma}(T_1 \to T_2)$, where $\vec{\sigma}$ is closed (closed in this context means each σ in $\vec{\sigma}$ is either $\forall[\vec{\beta}]$ or Subst[θ], i.e. not a type variable), there exists $\vec{\alpha}$, θ such that $T \equiv \forall[\vec{\alpha}].(Subst[\theta]T_1) \to (Subst[\theta]T_2)$.*

Lemma 9. *If $\Gamma \vdash O : T$, then Ty[O] $\subseteq T$*

Lemma 10. *If $\Gamma \vdash e\ e_1 : T$, then there exists a type T_1 such that $\Gamma \vdash e : T_1 \to T$ and $\Gamma \vdash e_1 : T_1$.*

Lemma 11. *If $\Gamma \vdash e\ e_1 \ldots e_n : T$, then there exist types $T_1, \ldots, T_n$ such that: $\Gamma \vdash e : T_1 \to \cdots \to T_n \to T$, and $\Gamma \vdash e_i : T_i$ for $i \in [1, n]$.*

Lemma 12. *If $\forall[\vec{\alpha}](T \to U) \subseteq \forall[\vec{\beta}](T' \to U')$, there exist a substitution θ and quantifiers $\vec{\gamma}$ such that: $dom(\theta) = \vec{\alpha}$, $T' \subseteq \forall[\vec{\gamma}]Subst[\theta]T$, and $\forall[\vec{\gamma}]Subst[\theta]U \subseteq U'$.*

Lemma 13. *If $\forall[\vec{\alpha}](T_1 \to \cdots \to T_n \to T) \subseteq U_1 \to \cdots \to U_n \to U$, then there exist quantifiers $\vec{\beta}$ and subsitution θ such that dom(θ)=$\vec{\alpha}$, $U_i \subseteq \forall[\vec{\beta}]Subst[\theta]T_i$ for $i \in [1, n]$, and $\forall[\vec{\beta}]Subst[\theta]T \subseteq U$.*

Lemma 14. *If $T \subseteq U$, then for any substitution θ and quantifiers $\vec{\gamma}$, $\forall[\vec{\gamma}]Subst[\theta]T \subseteq \forall[\vec{\gamma}]Subst[\theta]U$.*

Lemma 15. *If $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq \forall[\vec{\alpha}'](\sigma_1' \hat{\leq} \sigma_2')$, there exist quantifiers $\vec{\beta}$ and a substitution θ such that $\sigma_1' = \forall[\vec{\beta}] \circ Subst[\theta] \circ \sigma_1$ and $\sigma_2' = \forall[\vec{\beta}] \circ Subst[\theta] \circ \sigma_2$.*

Lemma 16. *If $\sigma_1 T \subseteq \sigma_2 T$ for any T, and $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq \forall[\vec{\beta}](\sigma_1' \hat{\leq} \sigma_2')$, then $\sigma_1' T \subseteq \sigma_2' T$ for any T.*

Lemma 17. *If $\forall[\vec{\alpha}](T \dot{\leq} U) \subseteq \forall[\vec{\beta}](T' \dot{\leq} U')$, there exist a substitution θ and quantifiers $\vec{\gamma}$ such that: $T' = \forall[\vec{\gamma}]Subst[\theta]T$ and $U' = \forall[\vec{\gamma}]Subst[\theta]U$.*

Lemma 18. *If $T \subseteq U$ and $\forall[\vec{\alpha}](T \dot{\leq} U) \subseteq \forall[\vec{\beta}](T' \dot{\leq} U')$, then $T' \subseteq U'$.*

Lemma 19. *If $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq \forall[\vec{\beta}](T \dot{\leq} U)$, there exist a type V, a substitution θ, and quantifiers $\vec{\gamma}$ such that: $T = \forall[\vec{\gamma}]Subst[\theta]V$ and $U = \forall[\vec{\gamma}]Subst[\theta]V$.*

Lemma 20. *If $\Gamma \vdash T ::: Subst$, then $(TU \to TV) \equiv T(U \to V)$.*

Lemma 21. *If $Exp[A] \subseteq Exp[B]$, then $A \subseteq B$.*

Lemma 22. *If $\vdash \Gamma$ and $\Gamma \vdash v : (\sigma_1 \hat{\leq} \sigma_2)$ and v is a value, then for any type T, $\sigma_1 T \subseteq \sigma_2 T$.*

Lemma 23. *For all types T,U: If $\vdash \Gamma$ and $\Gamma \vdash v : T \dot{\leq} U$ and v is a value, then $T \subseteq U$.*

Theorem 24. *Preservation.*
If $\vdash \Gamma$*,* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$*, then there exists* $\Gamma' \supseteq \Gamma$ *such that* $\vdash \Gamma'$ *and* $\Gamma' \vdash e' : T$*.*

Lemma 25. *If* $\Gamma \vdash e : Exp[T]$ *and* e *is a value, then either* $e = Q\ O$ *or* $e = A\ e_1\ e_2$*.*

Theorem 26. *Progress.*
If $\Gamma \vdash e : T$*, then either* e *is a value, or there exists* e' *such that* $e \longrightarrow e'$*.*

Theorem 27. *Type Soundness.*
If $\Gamma \vdash e : T$ *and* $e \longrightarrow^* e'$*, then either* e' *is a value or there exists an* e'' *such that* $e' \longrightarrow e''$*.*

8 Type Inference

We now describe a decidable fragment of our type system. The type inference algorithm is a straightforward extension of Dan Leijen's algorithm for a variant of System F [13, Appendix B]. We have implemented the algorithm and used it to type check our self-optimizer and self-enactor. The subset is given by these restrictions of the definitions in Section 6:

- Syntax: we don't use the proof term of the form p.
- Types: we don't use type of the form $Subst[\theta]$.
- Subtyping: Like in System F, we allow only two forms of subtyping, namely substitution, $\forall\vec{\alpha}.T \subseteq \sigma T$, and Mitchell's $Abs_\forall$ rule.

The net effect is that our subset is closely related to System F plus an extra kind $(* \to *)$ of types and a nontrivial notion of type equivalence. Our algorithm extends Dan Leijen's algorithm [13, Appendix B] with a straightforward notion of type normalization that enables us to decide type equivalence. The main benefit of the type inference algorithm is that the programmer doesn't have to specify uses of substitution and $\text{Abs}_\forall$.

9 A Self-Interpreter

Our techniques for typing self-optimization can also be used to implement a typed self-interpreter, as shown in Appendix C. Self-interpretation has a different set of challenges than self-optimization. For example, the proofs needed for a self-interpreter are simpler, in that we don't need to combine proof terms introduced by multiple Is-operators. This is because identifying a redex usually requires only matching an application of a single operator to the correct number of arguments. While the implementation of G requires that we identify whether it's third argument is headed by Q or A, this doesn't introduce any new constraints on the possible types of input term. On the other hand, we must be able to match and implement the Is-operators, and some care is needed to avoid the potential for infinite regress in introducing $Is[Is[O]]$, $Is[Is[Is[O]]]$, $\ldots$ operators to do so. In this section we will describe our self-interpreter, including how we solve the problem of infinite regress.

We match redexes similarly to matching expressions for optimizations. The functions *enact*1, *enact*2, and *enact*3 match operators with arity 1,2, and 3 respectively. These in turn dispatch to *enact*$[O]$ functions, for example *enact*K and *enact*S which are also used in our optimizer. Each of the operators S,K,I, and Y is implemented directly, by replacing each redex with its reduct. The Is-operators are handled together: an Is-redex is matched by $IsIs$, and implemented metacircularly by the function *enact*Is. When an Is-redex is matched, $IsIs$ provides a proof term that reflects that the operator is an Is-operator. In particular, if the occurrence of the operator has type T, then the proof term will have type $IsTy \dot{\leq} T$. As shown in Figure 6, $IsTy$ is an abbreviation for:

$$\forall T, U, V. T \to ((V \dot{\leq} T) \to V \to U) \to U \to U$$

which generalizes the types of $Is[O]$ and $IsIs$ by quantifying V. $IsIs$ provides a copy of the matched Is-operator at the type $IsTy$, which *enactIs* uses to perform metacircular reduction step. This uniformity of the types and semantics of the Is-operators is key to "tying the knot" in our self-interpreter, avoiding the problem of infinite regress.

Our self-interpreter *enact* evaluates a representation e to the Head Normal Form $O\ e_1\ \ldots\ e_n$, where $n < arity(O)$ and e_i are representations of arbitrary expressions. The semantics of *coerce* and *eArrow* require that their proof term argument be fully evaluated to a proof value, so some extra work is needed to fully evaluate the proof term reducing a *coerce* or *eArrow* redex. A proof term in Head Normal Form is of the form $c\ e_1\ \ldots\ e_n$, where c is a proof constructor. In order to evaluate this to a proof value, we must evaluate each e_i to a proof value. This is achieved via the *enactStrict* function defined within *enact*. After fully evaluating a proof representation, it is unquoted to obtain the underlying proof. This is then used to implement *coerce* and *eArrow* metacircularly.

10 Experimental Results

We have implemented type inference, desugaring, and the semantics. The input to our tools is the Latex source that we use to display programs.

Our implementation of type inference confirms that both our self-optimizer and self-interpreter type check with the expected types. Type inference was tremendously helpful during the development.

We desugar our self-optimizer and self-enactor before execution. The optimizer in sugared form is 274 lines of code, while the desugared version consists of 7457 atoms. The enactor in sugared form is 354 lines of code, while the desugared version consists of 7692 atoms.

Our implementation of the semantics confirms that both our self-optimizer and self-enactor work correctly. We have applied them to many microbenchmarks, to themselves, and to each other.

Let us illustrate the amount of optimization that the self-optimizer can achieve. Define $e = S(S(K(SK))I)I$ and notice that e can be optimized to I.

We found that $\mathit{enact}\ \ '(\mathit{optimize}\ \ 'e)$ executes in 33.4 seconds, while

$$\begin{array}{l} \mathit{unquote}\ (\mathit{optimize}\ '\mathit{enact}) \\ \quad '((\mathit{unquote}\ (\mathit{optimize}\ '\mathit{optimize}))\ 'e) \end{array}$$

executes in 11.8 seconds. This demonstrates that our system can be used to implement optimizations that provide significant performance improvements.

11 Related Work

This paper presents the first polymorphically typed self-optimizer. Our self-optimizer builds on a wide variety of related work, particularly on self-optimization, polymorphically typed self-interpreters, subtyping, inversion, proof terms, and explicit substitutions.

Self-optimization. Our self-optimizer is inspired by Hudak and Kranz' 1984 paper [10] on a combinator-based compiler for a functional language. The first phase of Hudak and Kranz' compiler [10] generates combinator expressions and *simplifies those expressions* as they are constructed. For example, one of their simplification rules is $SK \to KI$. Our paper shows how to implement such a simplification step as a polymorphically typed self-optimizer.

Polymorphically typed self-interpreters. Pfenning and Lee wrote in their 1991 paper about metacircularity in the polymorphic lambda-calculus that "*metacircularity seems to be impossible*" [19]. Still, their paper presented worthwhile techniques. In a breakthrough paper in 2009, Rendel, Ostermann, and

Hofer [20] presented the first polymorphically self-recognizer. In 2011, Jay and Palsberg presented the first polymorphically self-enactor [11]. We have been unable to use the techniques in those papers to program a polymorphically typed self-optimizer, so our paper uses both a novel expression language and a novel type system.

Subtyping. We follow Jay and Palsberg's 2011 paper [11] and work with a combinatory calculus and a type system with subtyping, though the details are different. At the core of both paper's definitions of subtyping are ideas from Mitchell's notion of subtyping [15] (which he calls containment). Wells showed in two papers in 1995 and 1996 that Mitchell subtyping is undecidable [28] and that type inference for Mitchell's calculus is undecidable [29]. Our notion of subtyping agrees with Mitchell's notion of subtyping for function types and polymorphic types, hence it is undecidable.

Inversion. For simply typed λ-calculus, the inversion lemma for the case of function calls says that if we can derive a judgment $\Gamma \vdash e_1 e_2 : T$, then there exists a type S such that we also can derive judgments $\Gamma \vdash e_1 : S \rightarrow T$ and $\Gamma \vdash e_2 : S$. For Mitchell's notion of subtyping, we can view part of Wells' Theorem 3.2 [28] as an inversion lemma that says that if we can derive that two polymorphic function types are subtype-related, then certain items exist such that we can also derive two other subtype-relationships. Our paper uses the proof term $eArrow$ to compute the results of inversion at run time.

Proof terms. Subtyping and explicit coercions are related and both have been studied at least since the 1990s. For example, Tannen et al. [25, 26] showed in the late 1990s how to define and compute with coercions, and Palsberg et al. [27] showed how to use explicit coercions to prove strong normalization for a calculus with subtyping. We adapted our approach to coercions and proof terms from Donnelly's Master's thesis [7]. Donnelly used $T\dot{\leq}U$ to denote the type of a proof term that witnesses that T is a subtype of U. In his Lemma 3.21, he proves that "Subtyping is Equality", that is, if a term p has type $T\dot{\leq}U$, then $T = U$. We have borrowed many of Donnelly's proof terms and added others of our own. Our proof terms satisfy a weaker lemma than Donnelly's Lemma 3.21, namely our Lemma 23 that says, intuitively, that if a value v has type $T\dot{\leq}U$, then $T \subseteq U$.

Explicit substitutions. Abadi et al. defined a λ-calculus with explicit substitutions [2]. Their calculus treated substitutions as typed first-class values. In their case, a substitution replaces a program variable with a value. Our types of the form $Subst[\theta]$ are a form of explicit substitutions at the type level. In our case, a substitution has kind $* \to *$, and replaces a type variable with a type. In contrast to Abadi et al.'s paper [2], we define most of what can be done by a substitution via type equivalence.

12 Conclusion

We have demonstrated how to write a polymorphically typed self-optimizer. We wrote it in a decidable fragment of a type system with types of kinds $(* \to *)$ and $*$. Our experiments confirm our theoretical results. Our result is a step towards better bug finding for any kind of self-applicable software.

APPENDIX A

Proofs

This appendix contains the proof of each theorem, lemma, and corollary stated in section 7.

Proof of Lemma 1. The proof is by induction on the structure of the term e_1. If e_1 is x then $(\lambda x.e_1)e_2 = I\ e_2 \longrightarrow^* e_2 = [e_2/x]e_1$. If e_1 avoids x then $(\lambda x.e_1)e_2 = K\ e_1\ e_2 \longrightarrow e_1 = [e_2/x]e_1$. Otherwise, if e_1 is of the form e_3e_4 then

$$\begin{array}{rcl} (\lambda.e_1)e_2 & = & S(\lambda x.e_3)(\lambda x.e_4)e_2 \\ & \longrightarrow & (\lambda x.e_3)e_2((\lambda x.e_4)e_2) \\ & \longrightarrow^* & [e_2/x]e_3([u/x]e_4) \\ & = & [e_2/x]e_1 \end{array}$$

by two applications of induction.

Proof of Lemma 2. The proof is by induction on the structure of the type derivation for e. If the last step in the derivation is Type-Subtype, then there exists a type T_1 such that $\Gamma, x : R \vdash e : T_1$ and $T_1 \subseteq T$. By induction, we can derive $\Gamma \vdash \lambda x.e : R \rightarrow T_1$. Now Sub-$\rightarrow$ derives $R \rightarrow T_1 \subseteq R \rightarrow T$, and Type-Subtype derives $\Gamma \vdash e : R \rightarrow T$. The remaining possibilities follow the structure of e. If e is x then $R \subseteq T$. $\forall[\alpha](\alpha \rightarrow \alpha) \subseteq R \rightarrow R \subseteq R \rightarrow T$, so $\lambda x.x = I : R \rightarrow T$ as required. If x is not free in e then $\lambda x.e = K\ e$ and $\Gamma \vdash K\ e : R \rightarrow T$ as required. Otherwise, if e is an application $e_1\ e_2$ then there are types T_1 and T_2 such that $\Gamma \vdash e_1 : T_2 \rightarrow T$ and $\Gamma \vdash e_2 : T_2$. By two applications of induction, it follows that

$\Gamma \vdash \lambda x.e_1 : U \to T_2 \to T$ and $\Gamma \vdash \lambda x.e_2 : U \to T_2$ whence $\lambda x.t = S(\lambda x.e_1)(\lambda x.e_2)$ has type $U \to T$ as required.

Proof of Corollary 3. This follows from Lemmas 2 and 7.

Proof of Lemma 4. Straightforward.

Proof of Lemma 5. By straightforward induction on the derivation of $T \subseteq U$.

Proof of Lemma 6. By induction on the structure of $T \subseteq U$.

If $\alpha \notin FV(U)$, then α is a redundant quantifier, and the result holds by Sub-Subst. Therefore, assume $\alpha \in FV(U)$.

Case $T \subseteq U$ derived by Sub-$\to$. We have $T = T_1 \to T_2$, $U = U_1 \to U_2$, $U_1 \subseteq T_1$, and $T_2 \subseteq U_2$. Since $\alpha \notin FV(T_1)$, lemma 5 states $\alpha \notin FV(U_1)$. Therefore, $\alpha \in FV(U_2)$. By induction, $T_2 \subseteq \forall[\alpha]U_2$. Now α is redundant in $U_1 \to \forall[\alpha]U_2$, so Sub-Subst derives $U_1 \to \forall[\alpha]U_2 \subseteq \forall[\alpha](U_1 \to \forall[\alpha]U_2)$, and a combination of Sub-Congr, Sub-Arrow, and Sub-Subst derives $\forall[\alpha](U_1 \to \forall[\alpha]U_2) \subseteq \forall[\alpha](U_1 \to U_2)$. The result follows from Sub-Trans.

The remaining cases are straightforward.

Proof of Lemma 7. By induction on the structure of $\Gamma \vdash e : T$.

Case $\Gamma \vdash e : T$ derived by rule Type-Atom. $T = Ty[atom]$, which is closed for all atoms. Therefore α is a redundant quantifier, so Sub-Subst derives $T \subseteq \forall[\alpha]T$.

Case $\Gamma \vdash e : T$ derived by rule Type-Subtype. We have $\Gamma \vdash e : T'$ and $T' \subseteq T$. If $\alpha \in FV(T')$, then the induction hypothesis gives $\Gamma \vdash e : \forall[\alpha]T'$. Now Sub-Congr derives $\forall[\alpha]T' \subseteq \forall[\alpha]T$ as required. If $\alpha \notin FV(T')$, then $T' \subseteq \forall[\alpha]T$ by lemma 6.

Proof of Lemma 8. Straightforward.

Proof of Lemma 9. By induction on the structure of $\Gamma \vdash O : T$.

Case Type-Atom: Immediate.

Case Type-Subtype: We have $\Gamma \vdash O : U$ and $U \subseteq T$. By induction, $Ty[O] \subseteq U$, and $Ty[O] \subseteq T$ follows by Sub-Trans.

Proof of Lemma 10. By induction on the derivation of $\Gamma \vdash e\ e_1 : T$.

Case $\Gamma \vdash e\ e_1 : T$ derived by Type-App: Immediate.

Case $\Gamma \vdash e\ e_1 : T$ derived by Type-Subtype: We have $\Gamma \vdash e\ e_1 : T'$ and $T' \subseteq T$. By induction, there exists a type T_1 such that $\Gamma \vdash e\ e_1 : T_1 \to T'$ and $\Gamma \vdash e_1 : T_1$. Now Sub-$\to$ derives $T_1 \to T' \subseteq T_1 \to T$ from Sub-Refl and $T' \subseteq T$. By Type-Subtype, $\Gamma \vdash e : T_1 \to T$ as required.

Proof of Lemma 11. By induction on the number of applications n.

Case $n = 1$. Follows from lemma 10.

Case $n > 1$. By lemma 10, there exists a type T_n such that $\Gamma \vdash e\ e_1 \ldots e_{n-1} : T_n \to T$ and $\Gamma \vdash e_n : T_n$. By induction, there exist types $T_1, \ldots, T_{n-1}$ such that $\Gamma \vdash e : T_1 \to \cdots \to T_{n-1} \to T_n \to T$ and $\Gamma \vdash e_i : T_i$ for $i \in [1, n-1]$ as required.

Proof of Lemma 12. By induction on the structure of the derivation $\forall[\vec{\alpha}](T \to U) \subseteq \forall[\vec{\beta}](T' \to U')$.

Case Sub-Refl: We have $\vec{\alpha} = \vec{\beta}$, $T' = T$, and $U' = U$. Holds with $\vec{\gamma} = \emptyset$, $\theta = []$.

Case Sub-$\to$: Holds with $\vec{\gamma} = \emptyset$, $\theta = []$.

Case Sub-Dist: We have $\vec{\beta} = \emptyset$ and $\vec{\gamma} = \vec{\alpha}$. Holds with $\theta = []$.

Case Sub-Subst: We have $T' = \mathit{Subst}[\theta]T$ and $U' = \mathit{Subst}[\theta]U$. Holds with $\vec{\gamma} = \emptyset$.

Case Sub-Congr: We have $\vec{\alpha} = \vec{\alpha}_1, \vec{\alpha}_2$, $\vec{\beta} = \vec{\alpha}_1, \vec{\beta}_2$, and $\forall[\vec{\alpha}_2](T \to U) \subseteq \forall[\vec{\beta}_2](T' \to U')$. By induction there exist quantifiers $\vec{\gamma}$ and a substitution θ' such that $dom(\theta') = \vec{\alpha}_2$ and $T' \subseteq \forall[\vec{\gamma}]Subst[\theta]T$ and $\forall[\vec{\gamma}]Subst[\theta]U \subseteq U'$. Holds $\theta = [\vec{\alpha}_1/\vec{\alpha}_1] \circ \theta'$.

Case Sub-Trans: We have $\forall[\vec{\alpha}](T \to U) \subseteq \forall[\vec{\delta}](T'' \to U'') \subseteq \forall[\vec{\beta}](T' \to U')$. By induction, there exist quantifiers $\vec{\gamma}_1, \vec{\gamma}_2$ and substitutions θ_1, θ_2 such that $dom(\theta_1) = \vec{\alpha}$, $dom(\theta_2) = \vec{\delta}$, $T'' \subseteq \forall[\vec{\gamma}_1]Subst[\theta_1]T$, $\forall[\vec{\gamma}_1]Subst[\theta_1]U \subseteq U''$, $T' \subseteq \forall[\vec{\gamma}_2]Subst[\theta_2]T''$, and $\forall[\vec{\gamma}_2]Subst[\theta_2]U'' \subseteq U'$.

Now $\forall[\vec{\gamma}_2] \circ Subst[\theta_2] \circ \forall[\vec{\gamma}_1] \circ Subst[\theta_1] \equiv \forall[\vec{\gamma}_2, \vec{\gamma}_1'] \circ Subst[\theta_2 \circ [\vec{\gamma}_1'/vec\gamma_1] \circ \theta_1]$. Holds with $\vec{\gamma} = \vec{\gamma}_2, \vec{\gamma}_1'$ and $\theta = \theta_2 \circ [\vec{\gamma}_1'/vec\gamma_1] \circ \theta_1$.

Proof of Lemma 13. By induction on n.

Case $n = 1$: Follows from lemma 12.

Case $n > 1$: By lemma 12, there exist quantifiers $\vec{\beta}_1$ and a substitution θ_1 such that $U_1 \subseteq \forall[\vec{\beta}_1]Subst[\theta_1]T_1$ and $\forall[\vec{\beta}_1]Subst[\theta_1](T_2 \to \cdots \to T_n \to T) \subseteq U_2 \to \cdots \to U_n \to U$. By induction, there exist quantifiers β_2 and a substitution θ_2 such that $U_i \subset \forall[\beta_2]Subst[\theta_2]Subst[\theta_1]T_i$ for $i \in [2, n]$, and $\forall[\vec{\beta}_2]Subst[\theta_2]T \subseteq U$. Let $\vec{\beta} = \vec{\beta}_2$, $\theta = \theta_2 \circ \theta_1$. Sub-Subst derives $\forall[\vec{\beta}_1]Subst[\theta_1]T_1 \subseteq \forall[\vec{\beta}]Subst[\theta]T_1$, and $U_1 \subseteq \forall[\vec{\beta}]Subst[\theta]T_1$ follows by Sub-Trans.

Proof of Lemma 14. Follows by two steps of Sub-Congr.

Proof of Lemma 15. By induction on the derivation $\forall[\vec{\alpha}](\sigma_1 \hat{\lesssim} \sigma_2) \subseteq \forall[\vec{\alpha}'](\sigma_1' \hat{\lesssim} \sigma_2')$.

Case Sub-Subst:

There exist quantifiers $\vec{\beta}$ and a substitution θ such that $\forall[\vec{\alpha}](\sigma_1 \hat{\lesssim} \sigma_2) \subseteq \forall[\vec{\alpha}']$ $Subst[\theta](\sigma_1 \hat{\lesssim} \sigma_2)$ Holds with $\vec{\beta} = \emptyset$.

Case Sub-Dist-$\hat{\lesssim}$:

Holds, with $\vec{\alpha} = \vec{\alpha}', \vec{\beta}$, and $\theta = []$.

Case Sub-Trans:

There exist quantifiers $\vec{\alpha}''$ and types σ_1'', σ_2'' such that $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq \forall[\vec{\alpha}''](\sigma_1'' \hat{\leq} \sigma_2'')$ and $\forall[\vec{\alpha}''](\sigma_1'' \hat{\leq} \sigma_2'') \subseteq \forall[\vec{\alpha}'](\sigma_1' \hat{\leq} \sigma_2')$. By induction, there exist quantifiers $\vec{\beta}_1, \vec{\beta}_2$ and substitutions θ_1, θ_2 such that: $\sigma_1'' = \forall[\vec{\beta}_1] \circ Subst[\theta_1] \circ \sigma_1$, $\sigma_2'' = \forall[\vec{\beta}_1] \circ Subst[\theta_1] \circ \sigma_2$, $\sigma_1' = \forall[\vec{\beta}_2] \circ Subst[\theta_2] \circ \sigma_1''$, and $\sigma_2' = \forall[\vec{\beta}_2] \circ Subst[\theta_2] \circ \sigma_2''$. Therefore, $\sigma_1' = \forall[\vec{\beta}_2] \circ Subst[\theta_2] \circ \forall[\vec{\beta}_1] \circ Subst[\theta_1] \circ \sigma_1'$ and $\sigma_2' = \forall[\vec{\beta}_2] \circ Subst[\theta_2] \circ \forall[\vec{\beta}_1] \circ Subst[\theta_1] \circ \sigma_2'$.

Let $\vec{\beta} = \vec{\beta}_2, \vec{\beta}_1'$, where $\vec{\beta}_1'$ are fresh. Let $\theta = \theta_2 \circ [\vec{\beta}_1'/\vec{\beta}] \circ \theta_1$. Now $\sigma_1' \equiv \forall[\vec{\beta}] \circ Subst[\theta] \circ \sigma_1$ and $\sigma_2' \equiv \forall[\vec{\beta}] \circ Subst[\theta] \circ \sigma_2$ as required.

Proof of Lemma 16. By lemma 15, there exist quantifiers $\vec{\gamma}$ and a substitution θ such that $\sigma_1' = \forall[\vec{\gamma}] \circ Subst[\theta] \circ \sigma_1$ and $\sigma_2' = \forall[\vec{\gamma}] \circ Subst[\theta] \circ \sigma_2$. The result follows from lemma 14.

Proof of Lemma 17. Similar to proof of lemma 15.

Proof of Lemma 18. By lemma 17, there exist quantifiers $\vec{\gamma}$ and a substitution θ such that $T' = \forall[\vec{\gamma}]Subst[\theta]T$ and $U' = \forall[\vec{\gamma}]Subst[\theta]U$. The result follows from 14.

Proof of Lemma 19. By induction on the derivation of $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq \forall[\vec{\beta}](T \dot{\leq} U)$.

Case Sub-Congr: Holds by the induction hypothesis.

Case Sub-Proof-Inst: Holds with $\vec{\gamma} = \emptyset$ and $\theta = []$.

Case Trans: We have $\forall[\vec{\alpha}](\sigma_1 \hat{\leq} \sigma_2) \subseteq A$ and $A \subseteq \forall[\vec{\beta}](T \dot{\leq} U)$.

If $A = \forall[\vec{\alpha}'](\sigma_1' \hat{\leq} \sigma_2')$, then by lemma 15, there exist quantifiers $\vec{\gamma}_1$ and a substitution θ_1 such that $\sigma_1' = \forall[\vec{\gamma}_1] \circ Subst[\theta_1] \circ \sigma_1$ and $\sigma_2' = \forall[\vec{\gamma}_1] \circ Subst[\theta_1] \circ \sigma_2$.

Now by the induction hypothesis, there exist quantifiers $\vec{\gamma}_2$, a substitution θ_2, and a type V such that $T = \forall[\vec{\gamma}_2]Subst[\theta_2]\forall[\vec{\gamma}_1]Subst[\theta_1]\sigma_1 V$ and $U = \forall[\vec{\gamma}_2]$ $Subst[\theta_2]$ $\forall[\vec{\gamma}_1]$ $Subst[\theta_1]\sigma_2 V$. Now let $\vec{\gamma} = \vec{\gamma}_2, \vec{\gamma}'_1$ and $\theta = \theta_2 \circ [\vec{\gamma}'_1/\vec{\gamma}_1] \circ \theta_1$. Now $T \equiv \forall[\vec{\gamma}]Subst[\theta]V$ and $U \equiv \forall[\vec{\gamma}]Subst[\theta]V$.

If $A = \forall[\vec{\alpha}'](T'\dot{\leq}U')$, then by the induction hypothesis, there exist quantifiers $\vec{\gamma}_1$, a substitution θ_1, and a type V such that $T' = \forall[\vec{\gamma}_1]Subst[\theta_1]\sigma_1 V$ and $U' = \forall[\vec{\gamma}_1]Subst[\theta_1]\sigma_2 V$. Now by lemma 17, there exist quantifiers $\vec{\gamma}_2$ and a substitution θ_2 such that $T = \forall[\vec{\gamma}_2]Subst[\theta_2]\forall[\vec{\gamma}_1]Subst[\theta_1]\sigma_1 V$ and $U = \forall[\vec{\gamma}_2]Subst[\theta_2]\forall[\vec{\gamma}_1]Subst[\theta_1]\sigma_2 V$. Now let $\vec{\gamma} = \vec{\gamma}_2, \vec{\gamma}'_1$ and $\theta = \theta_2 \circ [\vec{\gamma}'_1/\vec{\gamma}_1] \circ \theta_1$. Now $T \equiv \forall[\vec{\gamma}]Subst[\theta]V$ and $U \equiv \forall[\vec{\gamma}]Subst[\theta]V$.

Proof of Lemma 20. By induction on the structure of $\Gamma \vdash T ::: Subst$.

Case S-TVar: $T = \theta$. Holds by equivalence rule $(\theta U \to \theta V) \equiv \theta(U \to V)$.

Case S-Subst: $T = Subst[\theta]$. Holds by equivalence rule
$Subst[\theta](T_1 \hat{\leq} T_2) \equiv (Subst[\theta]T_1)\hat{\leq}(Subst[\theta]T_2)$.

Case S-Compose: We have $T = T_1 \circ T_2$, $\Gamma \vdash T_1 ::: s_1$, $\Gamma \vdash T_2 ::: s_2$, and $s_1 \sqcup s_2 = Subst$. Therefore, $s_1 = Subst$ and $s_2 = Subst$. Now $\theta T \to \theta U \equiv T_1 T_2 T \to T_1 T_2 U$. By induction, $T_1 T_2 T \to T_1 T_2 U \equiv T_1(T_2 T \to T_2 U) \equiv T_1 T_2 (T \to U) \equiv \theta(T \to U)$ as required.

Proof of Lemma 21. By straightforward induction.

Proof of Lemma 22. By induction on v.

Case $v = pf_l$: We have $\Gamma \vdash pf_l : \forall[\vec{\alpha}](\sigma'_1 \hat{\leq} \sigma'_2)$ and $\forall[\vec{\alpha}](\sigma'_1 \hat{\leq} \sigma'_2) \subseteq (\sigma_1 \hat{\leq} \sigma_2)$. $\vdash \Gamma$ ensures $\sigma'_1 T \subseteq \sigma'_2 T$ for any T. Result follows from lemma 16.

Case $v = refl$: By lemma 9, $\forall[A](A \hat{\leq} A) \subseteq (\sigma_1 \hat{\leq} \sigma_2)$. Sub-Refl derives $AT \subseteq AT$ for any type T, and the result follows from lemma 16.

Case Type-App: Proceed by case analysis on the structure of v.

Case $v = sCongr\ v_1$. By lemma 11, there exists a type V_1 such that $\Gamma \vdash sCongr : V_1 \to (\sigma_1 \hat{\leq} \sigma_2)$, $\Gamma \vdash v_1 : V_1$. By lemma 9, $\forall[T, U, V]((U \hat{\leq} V) \to (T \circ U \hat{\leq} T \circ V)) \subseteq V_1 \to (\sigma_1 \hat{\leq} \sigma_2)$. Therefore, there exist quantifiers $\vec{\alpha}$ and types T, U, V such that $V_1 \subseteq \forall[\vec{\alpha}](U \hat{\leq} V)$ and $\forall[\vec{\alpha}](T \circ U \hat{\leq} T \circ V) \subseteq (\sigma_1 \hat{\leq} \sigma_2)$. By induction, $UA \subseteq VA$ for any type A. By Sub-Congr, $TUA \subseteq TVA$. $\sigma_1 T \subseteq \sigma_2 T$ follows from lemma 16.

Case $v = sTrans\ v_1\ v_2$: By lemma 11, there exist types V_1, V_2 such that $\Gamma \vdash sTrans : V_1 \to V_2 \to (\sigma_1 \hat{\leq} \sigma_2)$, $\Gamma \vdash v_1 : V_1$, and $\Gamma \vdash v_2 : V_2$. By lemma 9, $\forall[\sigma_3, \sigma_4, \sigma_5, \sigma_6]((\sigma_3 \hat{\leq} \sigma_4 \circ \sigma_5) \to (\sigma_4 \hat{\leq} \sigma_6) \to (\sigma_3 \hat{\leq} \sigma_6 \circ \sigma_5)) \subseteq V_1 \to V_2 \to (\sigma_1 \hat{\leq} \sigma_2)$. Therefore, there exist quantifiers $\vec{\alpha}$ and types $\sigma_3, \sigma_4, \sigma_5, \sigma_6$ such that:

$$\begin{aligned} V_1 &\subseteq \forall[\vec{\alpha}](\sigma_3 \hat{\leq} \sigma_4 \circ \sigma_5) \\ V_2 &\subseteq \forall[\vec{\alpha}](\sigma_4 \hat{\leq} \sigma_6) \\ \forall[\vec{\alpha}](\sigma_3 \hat{\leq} \sigma_6 \circ \sigma_5) &\subseteq (\sigma_1 \hat{\leq} \sigma_2) \end{aligned}$$

By induction, $\sigma_3 T \subseteq \sigma_4 \sigma_5 T$ for all types T, and $\sigma_4 U \subseteq \sigma_6 U$ for all types U. In particular, $\sigma_4 \sigma_5 T \subseteq \sigma_6 \sigma_5 T$ for all types T. Therefore, lemma 16 gives $\sigma_1 T \subseteq \sigma_2 T$ for all types T as required.

Proof of Lemma 23. By induction on the structure of v.

Case $v = p$: We have $p : S \in \Gamma$ and $S \subseteq T \dot{\leq} U$. Now S is either of the form $\forall[\vec{\alpha}](T' \dot{\leq} U')$, or else $\forall[\vec{\alpha}]\forall[\alpha](\sigma_1 \alpha \hat{\leq} \sigma_2 \alpha)$.

In the first case, $\vdash \Gamma$ implies $T' \subseteq U'$, and the result follows from lemma 18.

In the second case, $T = (\sigma'_1 A$ and $U = \sigma'_2 A$ for some σ'_1, σ'_2, A. By lemma 22, $T \subseteq U$ as required.

Case $v = refl$: We have $\Gamma \vdash refl : S$, and $S \subseteq T \dot{\leq} U$. By lemma 9, $\forall[X].X \hat{\leq} X \subseteq S$. By lemma 19, there exist a substitution θ and quantifiers $\vec{\gamma}$ such that $T = \forall[\vec{\gamma}]Subst[\theta]X$ and $U = \forall[\vec{\gamma}]Subst[\theta]X$. Therefore, $T \subseteq U$ by Sub-Refl.

Case $v = dist$: We have $\Gamma \vdash dist : S$, and $S \subseteq T \dot{\leq} U$. By lemma 9, $\forall[\sigma, X, Y]$ $(\sigma(X \to Y) \dot{\leq} \sigma X \to \sigma Y) \subseteq S$. Sub-Dist-$\to$ derives $\sigma(X \to Y) \subseteq \sigma X \to \sigma Y$, and $T \subseteq U$ follows from lemma 18.

Case $v = distExp$:

We have $\Gamma \vdash distExp : S$, and $S \subseteq T \dot{\leq} U$. By lemma 9, $\forall[\sigma, A](\sigma Exp[A]$ $\dot{\leq}$ $Exp[\sigma A]) \subseteq (T \dot{\leq} U)$. Sub-Refl derives $\sigma Exp[A] \subseteq Exp[\sigma A]$ with $\sigma Exp[A] \equiv Exp[\sigma A]$, and $T \subseteq U$ follows from lemma 18.

Case $v = factor$:

By lemma 9, $\forall[\theta, X, Y](\theta X \to \theta Y \dot{\leq} \theta(X \to Y)) \subseteq (T \dot{\leq} U)$. Since $\Gamma \vdash \theta ::: Subst$, $\Gamma \vdash \theta X \to \theta Y \equiv \theta(X \to Y)$, so $\theta X \to \theta Y \subseteq \theta(X \to Y)$ is true by Sub-Refl. Now $T \subseteq U$ follows from lemma 18.

Case $v = factorExp$

We have $\Gamma \vdash factorExp : S$, and $S \subseteq T \dot{\leq} U$. By lemma 9, $\forall[\sigma, A](Exp[\sigma A]$ $\dot{\leq}$ $\sigma Exp[A]) \subseteq (T \dot{\leq} U)$. Sub-Refl derives $Exp[\sigma A] \subseteq \sigma Exp[A]$ with $\sigma Exp[A] \equiv Exp[\sigma A]$, and $T \subseteq U$ follows from lemma 18.

Case $v = iArrow\ v_1\ v_2\ v_3$

By lemma 11, $\Gamma \vdash iArrow : V_1 \to V_2 \to V_3 \to (T \dot{\leq} U)$, $\Gamma \vdash v_1 : V_1$, $\Gamma \vdash v_2 : V_2$, and $\Gamma \vdash v_3 : V_3$. By lemma 9, $Ty[iArrow] \subseteq V_1 \to V_2 \to V_3 \to (T \dot{\leq} U)$. Therefore, there exist types $\sigma_1, Q_2, Q_3, \theta, X, Y, X', Y'$ and quantifiers $\vec{\alpha}$ such that:

$$
\begin{aligned}
V_1 &\subseteq \forall[\vec{\alpha}](\sigma_1 \hat{\leq} Q_2 Q_3 \theta) \\
V_2 &\subseteq \forall[\vec{\alpha}](X' \dot{\leq} Q_3 \theta X) \\
V_3 &\subseteq \forall[\vec{\alpha}](Q_3 \theta Y \dot{\leq} Y') \\
\forall[\vec{\alpha}](\sigma_1 (X \to Y) \dot{\leq} Q_2 (X' \to Y')) &\subseteq (T \dot{\leq} U)
\end{aligned}
$$

By lemma 22, $\sigma_1 Z \subseteq Q_2 Q_3 \theta Z$ for all Z. In particular, $\sigma_1(X \to Y) \subseteq Q_2 Q_3 \theta (X \to Y)$. By the induction hypothesis, $X' \subseteq Q_3 \theta X$ and $Q_3 \theta Y \subseteq Y'$. Now we have:

$$
\begin{array}{ll}
\quad \sigma_1(X \to Y) & \\
\subseteq Q_2 Q_3 \theta (X \to Y) & \\
\subseteq Q_2 (Q_3 \theta X \to Q_3 \theta Y) & \text{By Sub-Congr, Sub-Dist-}\to \\
\subseteq Q_2 (X' \to Y') & \text{By Sub-Congr, Sub-}\to
\end{array}
$$

By lemma 17, there exist a substitution θ_1 and quantifiers $\vec{\beta}$ such that $T = \forall[\vec{\beta}] Subst[\theta] \sigma_1 (X \to Y)$ and $U = \forall[\vec{\beta}] Subst[\theta] Q_2 (X' \to Y')$. By lemma 14, $T \subseteq U$ as required.

Case $v = iExp\ v_1$

By lemma 11, $\Gamma \vdash iExp : V_1 \to (T \dot{\leq} U)$ and $\Gamma \vdash v_1 : V_1$. By lemma 9, $Ty[iExp] \subseteq V_1 \to (T \dot{\leq} U)$. Therefore, there exist types A, B and quantifiers $\vec{\beta}$ such that $V_1 \subseteq \forall[\vec{\beta}](A \dot{\leq} B)$ and $\forall[\vec{\beta}](Exp[A] \dot{\leq} Exp[B]) \subseteq (T \dot{\leq} U)$.

Since v_1 is a value, the induction hypothesis gives $A \subseteq B$. By Sub-Exp, $Exp[A] \subseteq Exp[B]$. $T \subseteq U$ follows from lemma 18.

Case $v = eExp\ v_1$

By lemma 11, $\Gamma \vdash eExp : V_1 \to (T \dot{\leq} U)$ and $\Gamma \vdash v_1 : V_1$. By lemma 9, $Ty[eExp] \subseteq V_1 \to (T \dot{\leq} U)$. Therefore, there exist types A, B and quantifiers $\vec{\beta}$ such that $V_1 \subseteq \forall[\vec{\beta}](Exp[A] \dot{\leq} Exp[B])$ and $\forall[\vec{\beta}](A \dot{\leq} B) \subseteq (T \dot{\leq} U)$.

Since v_1 is a value, the induction hypothesis gives $Exp[A] \subseteq Exp[B]$. By lemma 21, $A \subseteq B$. $T \subseteq U$ follows from lemma 18.

Case $v = congr\ v_1$

By lemma 11, $\Gamma \vdash congr : V_1 \to (T \dot{\leq} U)$ and $\Gamma \vdash v_1 : V_1$. By lemma 9, $Ty[congr] \subseteq V_1 \to (T \dot{\leq} U)$. Therefore, there exist types A, B, σ and quantifiers $\vec{\beta}$ such that $V_1 \subseteq \forall[\vec{\beta}](A \dot{\leq} B)$ and $\forall[\vec{\beta}](\sigma A \dot{\leq} \sigma B) \subseteq T \dot{\leq} U$.

Since v_1 is a value, the induction hypothesis gives $A \subseteq B$. By Sub-Congr, $\sigma A \subseteq \sigma B$. $T \subseteq U$ follows from lemma 18.

Case $v = trans\ v_1\ v_2$

By lemma 11, $\Gamma \vdash trans : V_1 \to V_2 \to S$, $\Gamma \vdash v_1 : V_1$, $\Gamma \vdash v_2 : V_2$, and $S \subseteq T \dot{\leq} U$. By lemma 9, $Ty[trans] \subseteq S$. Therefore, there exist quantifiers $\vec{\alpha}$ and types X, Y, Z such that $V_1 \subseteq \forall[\vec{\alpha}](X \dot{\leq} Y)$, $V_2 \subseteq \forall[\vec{\alpha}](Y \dot{\leq} Z)$, and $\forall[\vec{\alpha}](X \dot{\leq} Z) \subseteq (T \dot{\leq} U)$. Now $\Gamma \vdash v_1 : \forall[\vec{\alpha}](X \dot{\leq} Y)$ and $\Gamma \vdash v_2 : \forall[\vec{\alpha}](Y \dot{\leq} Z)$, so the induction hypothesis yields $X \subseteq Y$ and $Y \subseteq Z$. $X \subseteq Z$ follows from Sub-Trans, and $T \subseteq U$ from lemma 18.

Proof of Theorem 24. By case analysis on $e \longrightarrow e'$.

Case G f g (Q O) $\longrightarrow$ f O: By lemma 11, there exist types T_1, T_2, T_3 such that $\Gamma \vdash G : T_1 \to T_2 \to T_3 \to T$, $\Gamma \vdash f : T_1$, $\Gamma \vdash g : T_2$, and $\Gamma \vdash QO : T_3$. By lemma 9, $\forall[A, B]((A \to B) \to \forall[C](Exp[C \to A] \to Exp[C] \to B) \to Exp[A] \to B) \subseteq T_1 \to T_2 \to T_3 \to T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and types A, B such that $T_1 \subseteq \forall[\vec{\alpha}](A \to B)$, $T_2 \subseteq \forall[\vec{\alpha}]\forall[C](Exp[C \to A] \to Exp[C] \to B)$,

$T_3 \subseteq \forall[\vec{\alpha}]Exp[A]$, and $\forall[\vec{\alpha}]B \subseteq T$. By 10, there exists a type T_4 such that $\Gamma \vdash Q : T_4 \to T_3$ and $\Gamma \vdash O : T_4$. By 9, $\forall[X](X \to Exp[X]) \subseteq T_4 \to T_3$. By 12, there exist quantifiers $\vec{\beta}$ and a type X such that $T_4 \subseteq \forall[\vec{\beta}]X$ and $\forall[\vec{\beta}]Exp[X] \subseteq T_3$. By Sub-Trans, $\forall[\vec{\beta}]Exp[X] \subseteq \forall[\vec{\alpha}]Exp[A]$. By equivalence, $Exp[\forall[\vec{\beta}]X] \subseteq Exp[\forall[\vec{\alpha}]A]$. By 21, $\forall[\vec{\beta}]X \subseteq \forall[\vec{\alpha}]A$. By Type-Subtype, $\Gamma \vdash O : \forall[\vec{\alpha}]A$. By Sub-Dist-$\to$, $\forall[\vec{\alpha}](A \to B) \subseteq \forall[\vec{\alpha}]A \to \forall[\vec{\alpha}]B$, so Type-Subtype derives $\Gamma \vdash f : \forall[\vec{\alpha}]A \to \forall[\vec{\alpha}]B$. Therefore $\Gamma \vdash f\ O : \forall[\vec{\beta}]B$, and Type-Subtype derives $\Gamma \vdash f\ O : T$ as required.

Case G f g (A p q) $\longrightarrow$ g p q: By lemma 11, there exist types T_1, T_2, T_3 such that $\Gamma \vdash G : T_1 \to T_2 \to T_3 \to T$, $\Gamma \vdash f : T_1$, $\Gamma \vdash g : T_2$, and $\Gamma \vdash QO : T_3$. By lemma 9, $\forall[A, B]((A \to B) \to \forall[C](Exp[C \to A] \to Exp[C] \to B) \to Exp[A] \to B) \subseteq T_1 \to T_2 \to T_3 \to T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and types A, B such that $T_1 \subseteq \forall[\vec{\alpha}](A \to B)$, $T_2 \subseteq \forall[\vec{\alpha}]\forall[C](Exp[C \to A] \to Exp[C] \to B)$, $T_3 \subseteq \forall[\vec{\alpha}]Exp[A]$, and $\forall[\vec{\alpha}]B \subseteq T$. By lemma 11, there exist types T_4, T_5 such that $\Gamma \vdash A : T_4 \to T_5 \to T_3$, $\Gamma \vdash p : T_4$, and $\Gamma \vdash q : T_5$. By lemma 9, $\forall[X, Y](Exp[X \to Y] \to Exp[X] \to Exp[Y] \subseteq T_4 \to T_5 \to T_3$. By lemma 13, there exist quantifiers $\vec{\beta}$ and types X, Y such that $T_4 \subseteq \forall[\vec{\beta}]Exp[X \to Y]$, $T_5 \subseteq \forall[\vec{\beta}]Exp[X]$, and $\forall[\vec{\beta}]Exp[Y] \subseteq T_3$. By Sub-Trans, $\forall[\vec{\beta}]Exp[Y] \subseteq \forall[\vec{\alpha}]Exp[A]$. By equivalence, $Exp[\forall[\vec{\beta}]Y] \subseteq Exp[\forall[\vec{\alpha}]A]$. By lemma 21, $\forall[\vec{\beta}]Y \subseteq \forall[\vec{\alpha}]A$. Therefore $\forall[\vec{\beta}]Exp[X \to Y] \subseteq Exp[\forall[\vec{\beta}]X \to \forall[\vec{\alpha}]A]$, so Type-Subtype derives $\Gamma \vdash p : Exp[\forall[\vec{\beta}]X \to \forall[\vec{\alpha}]A]$. Without loss of generality, assume $\vec{\alpha} \notin FV(\forall[\vec{\beta}]X)$. Now $\forall[\vec{\alpha}]\forall[C](Exp[C \to A] \to Exp[C] \to B) \subseteq Exp[\forall[\vec{\beta}]X \to \forall[\vec{\alpha}]A] \to Exp[\forall[\vec{\beta}]X] \to \forall[\vec{\alpha}]B$. Type-Subtype derives $\Gamma \vdash g : Exp[\forall[\vec{\beta}]X \to \forall[\vec{\alpha}]A] \to Exp[\forall[\vec{\beta}]X] \to \forall[\vec{\alpha}]B$, so $\Gamma \vdash g\ p\ q : \forall[\vec{\alpha}]B$, and Type-Subtype derives $\Gamma \vdash g\ p\ q : T$ as required.

Case Is[O] O t f $\longrightarrow$ t p O: By lemma 11, there exist types T_1, T_2, T_3 such that: $\Gamma \vdash Is[O] : T_1 \to T_2 \to T_3 \to T$, $\Gamma \vdash O : T_1$, $\Gamma \vdash t : T_2$, and $\Gamma \vdash f : T_3$. By lemma 9, $Ty[Is[O]] \subseteq T_1 \to T_2 \to T_3 \to T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and a substitution θ such that: $T_1 \subseteq \forall[\vec{\alpha}]Subst[\theta]X$, $T_2 \subseteq \forall[\vec{\alpha}]Subst[\theta]((Ty[O]\dot{\leq}X) \to Ty[O] \to Y)$, and $\forall[\vec{\alpha}]Subst[\theta]Y \subseteq T$. Note that since $Ty[O]$ is closed, $Subst[\theta]((Ty[O] \dot{\leq}X) \to Ty[O] \to Y) \equiv (Ty[O] \dot{\leq} Subst[\theta]X) \to Ty[O] \to Subst[\theta]Y$. Again by lemma 9, $Ty[O] \subseteq T_1 \subseteq \forall[\vec{\alpha}]Subst[\theta]X$. Therefore, let $\Gamma' = (\Gamma, pf_l : Ty[O] \dot{\leq} \forall[\vec{\alpha}]Subst[\theta]X)$. Note that since $\vec{\alpha} \cap FV(Ty[O]) = \emptyset$, $Ty[O] \dot{\leq} \forall[\vec{\alpha}]Subst[\theta]X \equiv \forall[\vec{\alpha}](Ty[O] \dot{\leq} Subst[\theta]X)$. By two distributions of $\forall[\vec{\alpha}]$, $T_2 \subseteq (Ty[O] \dot{\leq} \forall[\vec{\alpha}]Subst[\theta]X) \to Ty[O] \to$ $(\forall[\vec{\alpha}]Subst[\theta]Y)$. Now $\Gamma' \vdash t\ pf_l\ O : \forall[\vec{\alpha}]Subst[\theta]Y$, and the result follows from $\forall[\vec{\alpha}]Subst[\theta]Y \subseteq T$.

Case eArrow p e $\longrightarrow$ e p_{l1} p_{l2} p_{l3}: By lemma 11, there exist types T_1 and T_2 such that $\Gamma \vdash eArrow : T_1 \to T_2 \to T$, $\Gamma \vdash p : T_1$, and $\Gamma \vdash e : T_2$. By lemma 9, $Ty[eArrow] \subseteq T_1 \to T_2 \to T$. By lemma 13, there exist quantifers $\vec{\alpha}, \vec{\beta}, \vec{\gamma}$ and types A, B, A', B', C such that: $T_1 \subseteq \forall[\vec{\gamma}](\forall[\vec{\alpha}](A \to B)\dot{\leq}\forall[\vec{\beta}](A' \to B'))$, $T_2 \subseteq \forall[\vec{\gamma}](\forall Q, \theta.(\forall[\vec{\alpha}]\hat{\leq}\forall[\vec{\beta}] \circ Q \circ \theta) \to (A'\dot{\leq}Q\theta A) \to (Q\theta B\dot{\leq}B') \to C)$, and $\forall[\vec{\gamma}]C \subseteq T$.

By lemma 23, $\forall[\vec{\alpha}](A \to B) \subseteq \forall[\vec{\beta}](A' \to B')$. Now by lemma 12, there exist fresh quantifiers $\vec{\delta}$ and a substitution θ_1 such that $dom(\theta_1) = \vec{\alpha}$, $A' \subseteq \forall[\vec{\delta}]Subst[\theta_1]A$ and $\forall[\vec{\delta}]Subst[\theta_1]B \subseteq B'$. Now let $\Gamma' = (\Gamma, p_{l1} : \forall[\vec{\gamma}](\forall[\vec{\alpha}]\hat{\leq}\forall[\vec{\beta}] \circ \forall[\vec{\delta}] \circ Subst[\theta_1]),\ p_{l2} : \forall[\vec{\gamma}](A'\dot{\leq}\forall[\vec{\delta}]Subst[\theta_1]A),\ p_{l3} : \forall[\vec{\gamma}](\forall[\vec{\delta}]Subst[\theta_1]B\dot{\leq}B'))$. Note that $\vdash \Gamma'$ holds.

By instantiating Q to $\forall[\vec{\delta}]$ and θ to $Subst[\theta_1]$ and distributing $\forall[\vec{\gamma}]$ three times, we can derive:

$$\begin{aligned}
&\forall[\vec{\gamma}]((\forall[\vec{\alpha}]\hat{\leq}\forall[\vec{\beta}] \circ Q \circ \theta) \rightarrow \\
&\qquad (A' \dot{\leq} Q\theta A) \rightarrow \\
&\qquad (Q\theta B \dot{\leq} B') \rightarrow \\
&\qquad C) \\
\subseteq &(\forall[\vec{\gamma}](\forall[\vec{\alpha}]\hat{\leq}\forall[\vec{\beta}] \circ \forall[\vec{\delta}] \circ Subst[\theta_1])) \rightarrow \\
&(\forall[\vec{\gamma}](A' \dot{\leq} \forall[\vec{\delta}]Subst[\theta_1]A)) \rightarrow \\
&(\forall[\vec{\gamma}](\forall[\vec{\delta}]Subst[\theta_1]B \dot{\leq} B')) \rightarrow \\
&(\forall[\vec{\gamma}]C)
\end{aligned}$$

Now since $\forall[\vec{\gamma}]C \subseteq T$, we get $\Gamma' \vdash e\ p_{l1}\ p_{l2}\ p_{l3} : T$ as required.

Case coerce e v $\longrightarrow$ e: By lemma 11, we have $\Gamma \vdash \mathit{coerce} : T_1 \rightarrow T_2 \rightarrow T$, $\Gamma \vdash e : T_1$, and $\Gamma \vdash v : T_2$. By lemma 9, $Ty[\mathit{coerce}] \subseteq T_1 \rightarrow T_2 \rightarrow T$. Therefore, there exist quantifiers $\vec{\alpha}$ and a substitution θ such that $T_1 \subseteq \forall[\vec{\alpha}]Subst[\theta]X$, $T_2 \subseteq \forall[\vec{\alpha}]Subst[\theta](X \dot{\leq} Y)$, and $\forall[\vec{\alpha}]Subst[\theta]Y \subseteq T$. Note that $\forall[\vec{\alpha}]Subst[\theta](X \dot{\leq} Y) \equiv \forall[\vec{\alpha}](Subst[\theta]X \dot{\leq} Subst[\theta]Y) \subseteq \forall[\vec{\alpha}]Subst[\theta]X \dot{\leq} \forall[\vec{\alpha}]Subst[\theta]Y$. By lemma 23, $\forall[\vec{\alpha}]Subst[\theta]X \subseteq \forall[\vec{\alpha}]Subst[\theta]Y$. Therefore, $\Gamma \vdash e : T$ as required.

Proof of Lemma 25. By contradiction. Assume $\Gamma \vdash e : Exp[T]$ and e is a value. Then either $e = O\ e_1 \ldots\ e_i$, where $i < arity(O)$, or $e = P\ v_1 \ldots\ v_i$, where $i = arity(P)$ and each v_j is a value.

In the first case, lemma 11 states there exist types $T_1, \ldots, T_i$ such that $\Gamma \vdash O : T_1 \rightarrow \cdots \rightarrow T_i \rightarrow Exp[T]$. By lemma 9, $Ty[O] \subseteq T_1 \rightarrow \cdots \rightarrow T_i \rightarrow Exp[T]$. Since $i < arity(O)$, there exist quantifies $\vec{\alpha}$ and types U_1, U_2 such that $\forall[\vec{\alpha}](U_1 \rightarrow U_2) \subseteq Exp[T]$, a contradiction.

In the second case, lemma 11 states there exist types $T_1, \ldots, T_i$ such that $\Gamma \vdash$

$P : T_1 \to \cdots \to T_i \to Exp[T]$. By lemma 9, $Ty[P] \subseteq T_1 \to \cdots \to T_i \to Exp[T]$. Since $i = arity(P)$, there exist quantifiers $\vec{\alpha}$ and types U_1, U_2 such that either $\forall[\vec{\alpha}](U_1 \dot{\leq} U_2) \subseteq Exp[T]$ or $\forall[\vec{\alpha}](U_1 \hat{\leq} U_2) \subseteq Exp[T]$. Either case is a contradiction.

Proof of Theorem 26. By induction on the structure of e.

Case $e = G\ e_1\ e_2\ e_3$. If e_3 is not a value, then by induction $e_3' \longrightarrow e_3'$ and $e \longrightarrow G\ e_1\ e_2\ e_3'$. Otherwise, by lemma 25, $e_3 = Q\ O$ or $e_3 = A\ e_4\ e_5$. In the first case, $e \longrightarrow e_1\ O$. In the second, $e \longrightarrow e_2\ e_4\ e_4$.

Case $e = O\ e_1\ \ldots\ e_i$, where $i = arity(O)$, $O \neq G$. Since O is fully applied, the appropriate reduction rule applies.

Proof of Theorem 27. Follows from theorems 24 and 26.

APPENDIX B

Optimizations

Included in this appendix are the three optimization steps which together with $SK2KI$ form our complete optimizer. We also define several helper functions to increase the readability of our optimizer. These fall into three categories. Matching functions (*matchAtom*, *matchApp*, ...) use G and the Is-operators to match particular compound expressions. For example, $matchS1$ is used to match an expression of the form $A\ (Q\ S)\ e$, that is, an application of S to a single argument.

The functions $trans2$ and $dist2$ can be understood as derived proof constructors, defined in terms of other proof constructors. For example, $trans2$ combines two $trans$ steps into a larger composite.

The function *expandI* is similar to *expandK* listed in figure 1. Given a proof term introduced by IsI, *expandI* returns the essential consequence of the proof.

The optimization steps *reduceK* and *reduceS* evaluate K and S redexes in the input term. They use the *enactK* and *enactS* functions from our self-interpreter (discussed in section 9 and listed in appendix C), without the recursive call to $enact$.

The Eta optimization step performs the equivalent of η-reduction for our combinator calculus. η-reduction is defined for the λ calculus as follows:

$$\lambda x.e\ x \longrightarrow e, \text{ if } x \notin fv(e)$$

Desugaring $\lambda x.e\ x$ yields $S\ (K\ e')\ I$, where e' is the result of desugaring e. We can see that this term is equivalent to e':

$$S\ (K\ e')\ I\ x \longrightarrow K\ e'\ x\ (I\ x) \longrightarrow e'\ (I\ x) \equiv e'\ x$$

The function *proveEta* constructs the proof for the Eta optimization step, which proves that the type of e' must be a subtype of the type of $S\ (K\ e')\ I$.

$$
\begin{aligned}
&\texttt{let}\ \ (composeOpt4 : (\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
&\qquad\qquad (\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
&\qquad\qquad (\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
&\qquad\qquad (\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) \to \\
&\qquad\qquad (\forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U))\ = \\
&\ \lambda(opt1 : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
&\ \lambda(opt2 : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
&\ \lambda(opt3 : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
&\ \lambda(opt4 : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U). \\
&\ composeOpt\ opt1\ (composeOpt\ opt2\ (composeOpt\ opt3\ opt4))\ \ \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\ \ (matchAtom : \forall T, U.U \to (T \to U) \to Exp[T] \to U)\ = \\
&\quad \lambda(ifNotAtom : U).\lambda(ifAtom : T \to U).\mathrm{G}\ ifAtom\ (\mathrm{K}\ (\mathrm{K}\ ifNotAtom))\ \ \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\ \ (matchApp : \forall T, U.U \to (\forall T_1.Exp[T_1 \to T] \to Exp[T_1] \to U) \to \\
&\qquad\qquad Exp[T] \to U)\ = \\
&\quad \lambda(ifNotApp : U).
\end{aligned}
$$

$\lambda(ifApp : \forall T_1.Exp[T_1 \to T] \to Exp[T_1] \to U).$

$\mathrm{G}\ (\mathrm{K}\ ifNotApp)\ ifApp$ in

let $(matchS1 : \forall T, U.U \to (\forall T_1.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T) \to Exp[T_1] \to U) \to$
$Exp[T] \to U) =$

$\lambda(ifNotS1 : U).\lambda(ifS1 : \forall T_1.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T) \to Exp[T_1] \to U).$

$matchApp\ ifNotS1\ (matchAtom\ (\mathrm{K}\ ifNotS1)$

$(\lambda(e_k : T_1 \to T).\mathrm{IsS}\ e_k\ (\lambda(p : Ty[\mathrm{S}] \dot{\leq} T_1 \to T).\mathrm{K}\ (ifS1\ p))(\mathrm{K}\ ifNotS1)))$ in

let $(matchK0 : \forall T, U.U \to ((Ty[\mathrm{K}] \dot{\leq} T) \to U) \to Exp[T] \to U) =$

$\lambda(ifNotK0 : U).\lambda(ifK0 : (Ty[\mathrm{K}] \dot{\leq} T) \to U).$

$matchAtom\ ifNotK0$

$(\lambda(e_i : T).\mathrm{IsK}\ e_i\ (\lambda(p : Ty[\mathrm{K}] \dot{\leq} T).\mathrm{K}\ (ifK0\ p))\ ifNotK0)$ in

let $(trans2 : \forall T, U, V, W.(T \dot{\leq} U) \to (U \dot{\leq} V) \to (V \dot{\leq} W) \to (T \dot{\leq} W)) =$

$\lambda(p_1 : T \dot{\leq} U).\lambda(p_2 : U \dot{\leq} V).\lambda(p_3 : V \dot{\leq} W).\mathrm{trans}\ p_1\ (\mathrm{trans}\ p_2\ p_3)$ in

let $(dist2 : \forall \rho, T, U, V.\rho(T \to U \to V) \dot{\leq} \rho T \to \rho U \to \rho V) =$

trans dist(iArrow refl dist) in

let $(matchApp2 : \forall T, U.U \to$
$(\forall T_1, T_2.Exp[T_1 \to T_2 \to T] \to Exp[T_1] \to Exp[T_2] \to U) \to$
$Exp[T] \to U) =$

$\lambda(ifNotApp2 : U).$

$\lambda(ifApp2 : \forall T_1, T_2.Exp[T_1 \to T_2 \to T] \to Exp[T_1] \to Exp[T_2] \to U).$

$matchApp\ ifNotApp2\ (matchApp\ (\mathrm{K}\ ifNotApp2)\ ifApp2)$ in

$$
\begin{aligned}
&\texttt{let}\ \ (matchApp3 : \forall T, U.U \to \\
&\qquad\qquad (\forall T_1, T_2, T_3.Exp[T_1 \to T_2 \to T_3 \to T] \to \\
&\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to U) \to \\
&\qquad\qquad Exp[T] \to U) \ = \\
&\quad \lambda(ifNotApp3 : U). \\
&\quad \lambda(ifApp3 : \forall T_1, T_2, T_3.Exp[T_1 \to T_2 \to T_3 \to T] \to \\
&\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to U). \\
&\quad matchApp\ ifNotApp3\ (matchApp2\ (\mathrm{K}\ ifNotApp3)\ ifApp3)\ \ \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\ \ (matchS2 : \forall T, U.U \to (\forall T_1, T_2.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T_2 \to T) \to \\
&\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to U) \to \\
&\qquad\qquad Exp[T] \to U) \ = \\
&\quad \lambda(ifNotS2 : U). \\
&\quad \lambda(ifS2 : \forall T_1, T_2.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T_2 \to T) \to Exp[T_1] \to Exp[T_2] \to U). \\
&\quad matchApp2\ ifNotS2\ (matchAtom\ (\mathrm{K}\ (\mathrm{K}\ ifNotS2)) \\
&\quad (\lambda(e_s : T_1 \to T_2 \to T). \\
&\qquad \mathrm{IsS}\ e_s\ (\lambda(p : Ty[\mathrm{S}] \dot{\leq} T_1 \to T_2 \to T).\ \mathrm{K}\ (ifS2\ p))\ (\mathrm{K}\ (\mathrm{K}\ ifNotS2))))\ \ \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\ \ (matchS3 : \forall T, U.U \to (\forall T_1, T_2, T_3.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T_2 \to T_3 \to T) \to \\
&\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to U) \to \\
&\qquad\qquad Exp[T] \to U) \ = \\
&\quad \lambda(ifNotS3 : U). \\
&\quad \lambda(ifS3 : \forall T_1, T_2, T_3.(Ty[\mathrm{S}] \dot{\leq} T_1 \to T_2 \to T_3 \to T) \to \\
&\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to U). \\
&\quad matchApp3\ ifNotS3\ (matchAtom\ (\mathrm{K}\ (\mathrm{K}\ (\mathrm{K}\ ifNotS3))) \\
&\quad (\lambda(e_s : T_1 \to T_2 \to T_3 \to T).
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad \text{IsS } e_s \\
\quad\quad\quad (\lambda(p : Ty[\mathrm{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T). \\
\quad\quad\quad\quad \mathrm{K}\ (ifS3\ p))\ (\mathrm{K}\ (\mathrm{K}\ (\mathrm{K}\ ifNotS3)))))\ \ \mathtt{in}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{let}\ \ (matchK1 : \forall T, U.U \to (\forall T_1.(Ty[\mathrm{K}]\dot{\leq}T_1 \to T) \to Exp[T_1] \to U) \to \\
\quad\quad\quad\quad\quad\quad Exp[T] \to U)\ = \\
\quad \lambda(ifNotK1 : U). \\
\quad \lambda(ifK1 : \forall T_1.(Ty[\mathrm{K}]\dot{\leq}T_1 \to T) \to Exp[T_1] \to U). \\
\quad matchApp\ ifNotK1\ (matchAtom\ (\mathrm{K}\ ifNotK1) \\
\quad (\lambda(e_k : T_1 \to T). \\
\quad\quad \text{IsK } e_k\ (\lambda(p : Ty[\mathrm{K}]\dot{\leq}T_1 \to T).\ \mathrm{K}\ (ifK1\ p))\ (\mathrm{K}\ ifNotK1)))\ \ \mathtt{in}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{let}\ \ (matchK2 : \forall T, U.U \to \\
\quad\quad\quad\quad\quad\quad (\forall T_1, T_2.(Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T) \to \\
\quad\quad\quad\quad\quad\quad\quad\quad Exp[T_1] \to Exp[T_2] \to U) \to \\
\quad\quad\quad\quad\quad\quad Exp[T] \to U)\ = \\
\quad \lambda(ifNotK2 : U). \\
\quad \lambda(ifK2 : \forall T_1, T_2.(Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T) \to Exp[T_1] \to Exp[T_2] \to U). \\
\quad matchApp2\ ifNotK2\ (matchAtom\ (\mathrm{K}\ (\mathrm{K}\ ifNotK2)) \\
\quad (\lambda(e_s : T_1 \to T_2 \to T). \\
\quad\quad \text{IsK } e_s\ (\lambda(p : Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T).\ \mathrm{K}\ (ifK2\ p))\ (\mathrm{K}\ (\mathrm{K}\ ifNotK2))))\ \ \mathtt{in}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{let}\ \ (matchI0 : \forall T, U.U \to ((Ty[\mathrm{I}]\dot{\leq}T) \to U) \to Exp[T] \to U)\ = \\
\quad \lambda(ifNotI0 : U).\lambda(ifI0 : (Ty[\mathrm{I}]\dot{\leq}T) \to U). \\
\quad matchAtom\ ifNotI0 \\
\quad (\lambda(e_i : T).
\end{array}
$$

$\text{IsI } e_i\ (\lambda(p : Ty[\mathrm{I}]\dot{\leq}T).\ \mathrm{K}\ (ifI0\ p))\ ifNotI0)$ in

let $(analyzeEta : \forall T, U.U \rightarrow$
$(\forall T_1, T_2, T_3.Exp[T_3] \rightarrow (Ty[\mathrm{S}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T) \rightarrow$
$(Ty[\mathrm{K}]\dot{\leq}T_3 \rightarrow T_1) \rightarrow (Ty[\mathrm{I}]\dot{\leq}T_2) \rightarrow U) \rightarrow$
$Exp[T] \rightarrow U)\ =$
$\lambda(ifNotEta : U).$
$\lambda(ifEta : \forall T_1, T_2, T_3.Exp[T_3] \rightarrow (Ty[\mathrm{S}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T) \rightarrow$
$(Ty[\mathrm{K}]\dot{\leq}T_3 \rightarrow T_1) \rightarrow (Ty[\mathrm{I}]\dot{\leq}T_2) \rightarrow U).$
$matchS2\ ifNotEta$
$(\lambda(pS : Ty[\mathrm{S}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T).\ \lambda(e_1 : Exp[T_1]).\ \lambda(e_2 : Exp[T_2]).$
$matchK1\ ifNotEta$
$(\lambda(pK : Ty[\mathrm{K}]\dot{\leq}T_3 \rightarrow T_1).\ \lambda(e_3 : Exp[T_3]).$
$matchI0\ ifNotEta\ (\lambda(pI : Ty[\mathrm{I}]\dot{\leq}T_2).\ ifEta\ e_3\ pS\ pK\ pI)\ e_2)\ e_1)$ in

let $(expandI : (Ty[\mathrm{I}]\dot{\leq}\varphi(T \rightarrow U)) \rightarrow T\dot{\leq}U)\ =$
$\lambda(pIsI : Ty[\mathrm{I}]\dot{\leq}\varphi(T \rightarrow U)).$
$\text{eArrow } pIsI$
$(\lambda(p_1 : \varphi_1\hat{\leq}\varphi \circ \varphi_2 \circ \sigma_1).\ \lambda(p_2 : T\dot{\leq}\varphi_2\sigma_1 X).\ \lambda(p_3 : \varphi_2\sigma X\dot{\leq}U).$
$\text{trans } p_2\ p_3)$ in

let $(proveEta : \forall T_1, T_2, T_3.(Ty[\mathrm{S}]\dot{\leq}T_2 \rightarrow T_1 \rightarrow T) \rightarrow (Ty[\mathrm{K}]\dot{\leq}T_3 \rightarrow T_2) \rightarrow$
$(Ty[\mathrm{I}]\dot{\leq}T_1) \rightarrow T_3\dot{\leq}T)\ =$
$\lambda(pS : Ty[\mathrm{S}]\dot{\leq}T_2 \rightarrow T_1 \rightarrow T).$
$\lambda(pK : Ty[\mathrm{K}]\dot{\leq}T_3 \rightarrow T_2).$
$\lambda(pI : Ty[\mathrm{I}]\dot{\leq}T_1).$

eBinary pS

$(\lambda(p_1 : \varphi_1 \hat{\leq} \varphi_2 \circ \sigma_1).$

$\lambda(p_2 : T_2 \dot{\leq} \varphi_2\sigma_1(X_1 \to X_2 \to X_3)).$

$\lambda(p_3 : T_1 \dot{\leq} \varphi_2\sigma_1(X_1 \to X_2)).$

$\lambda(p_4 : \varphi_2\sigma_1(X_1 \to X_3) \dot{\leq} T).$

let $(p_5 : Ty[\mathrm{I}] \dot{\leq} \varphi_2(\sigma_1 X_1 \to \sigma_1 X_2)) = \text{trans } pI(\text{trans } p_3 \text{ (congrdist)})$ in

let $(p_6 : \sigma_1 X_1 \dot{\leq} \sigma_1 X_2) = expandI\ p_5$ in

eArrow pK

$(\lambda(p_7 : \varphi_3 \hat{\leq} \varphi_4 \circ \sigma_2).$

$\lambda(p_8 : T_3 \dot{\leq} \varphi_4\sigma_2 V_1).$

$\lambda(p_9 : \varphi_4\sigma_2(V_2 \to V_1) \dot{\leq} T_2).$

let $(p_{10} : \varphi_4\sigma_2(V_2 \to V_1) \dot{\leq} \varphi_2(\sigma_1 X_1 \to \sigma_1 X_2 \to \sigma_1 X_3)) =$

trans2 p_9 p_2 (congr dist2) in

eArrow p_{10}

$(\lambda(p_{11} : \varphi_4 \circ \sigma_2 \hat{\leq} \varphi_2 \circ \varphi_5 \circ \sigma_3).$

$\lambda(p_{12} : \sigma_1 X_1 \dot{\leq} \varphi_5\sigma_3 V_2).$

$\lambda(p_{13} : \varphi_5\sigma_3 V_1 \dot{\leq} \sigma_1 X_2 \to sigma_1 X_3).$

let $(p_{14} : T_3 \dot{\leq} \varphi_2\varphi_5\sigma_3 V_1) = \text{trans } p_8\ p_{11}$ in

let $(p_{15} : T_3 \dot{\leq} \varphi_2\sigma_1 X_2 \to sigma_1 X_3) = \text{trans } p_{14} \text{ (congr } p_{13})$ in

let $(p_{16} : \sigma_1 X_2 \to sigma_1 X_3 \dot{\leq} \sigma_1(X_1 \to X_3))$

$=$ trans (iArrow p_6 refl) factor in

let $(p_{17} : T_3 \dot{\leq} T) = \text{trans2 } p_{15} \text{ (congr } p_{16})\ p_4$ in

$p_{17})))$ in

let $(Eta : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) =$

$\lambda(ifNoOpt : U).\ \lambda(ifOpt : Exp[T] \to U).$

$analyzeEta\ ifNoOpt$

$(\lambda(e_3 : Exp[T_3']).$

$\lambda(pS : Ty[\mathrm{S}]\dot{\leq}T_1' \to T_2' \to T).$

$\lambda(pK : Ty[\mathrm{K}]\dot{\leq}T_3' \to T_1').$

$\lambda(pI : Ty[\mathrm{I}]\dot{\leq}T_2').$

$ifOpt$ (coerce e_3 (iExp ($proveEta\ pS\ pK\ pI$)))) in

let $(enactK : \forall T_1, T_2, T_3.(Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T_3) \to$

$Exp[T_1] \to Exp[T_2] \to Exp[T_3]) =$

$(\lambda(pIsK : Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T_3).$

$\lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).$

eBinary $pIsK$

$(\lambda(p_1 : \varphi\hat{\leq}\varphi' \circ \sigma).$

$\lambda(p_2 : T_1\dot{\leq}\varphi'\sigma X).$

$\lambda(p_3 : T_2\dot{\leq}\varphi'\sigma Z).$

$\lambda(p_4 : \varphi'\sigma X\dot{\leq}T).$

let $(p_5 : T_1\dot{\leq}T) =$ trans $p_2\ p_4$ in

let $(p_6 : Exp[T_1]\dot{\leq}Exp[T]) =$ iExp p_5

in coerce $e_1\ p_6$)) in

let $(reduceK : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U) =$

$\lambda(notK : U).\lambda(ifK : Exp[T] \to U).$

$matchK2\ notK\ (\lambda(p : Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T).\lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).$

$ifK\ (enactK\ p\ e_1\ e_2))$ in

$$
\begin{array}{l}
\texttt{let}\ \ (enactS : \forall T_1, T_2, T_3, T_4.(Ty[\mathrm{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4) \to \\
\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4])\ = \\
\ \lambda(p : Ty[\mathrm{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4). \\
\ \lambda(e_1 : Exp[T_1]),(e_2 : Exp[T_2]),(e_3 : Exp[T_3]). \\
\ \text{eTernary}\ p \\
\quad (\lambda(p_1 : \forall[X_1, X_2, X_3]\hat{\leq}(\varphi \circ \sigma)). \\
\quad\ \lambda(p_2 : T_1\dot{\leq}\varphi\sigma(X_1 \to X_2 \to X_3)). \\
\quad\ \lambda(p_3 : T_2\dot{\leq}\varphi\sigma(X_1 \to X_2)). \\
\quad\ \lambda(p_4 : T_3\dot{\leq}\varphi\sigma X_1). \\
\quad\ \lambda(p_5 : \varphi\sigma X_3\dot{\leq}T_4). \\
\quad\ \texttt{let}\ \ (p_2' : T_1\dot{\leq}\varphi\sigma X_1 \to \varphi\sigma X_2 \to \varphi\sigma X_3) = \text{trans}\ p_2\ \text{dist2}\ \ \texttt{in} \\
\quad\ \texttt{let}\ \ (p_3' : T_2\dot{\leq}\varphi\sigma X_1 \to \varphi\sigma X_2) = \text{trans}\ p_3\ \text{dist}\ \ \texttt{in} \\
\quad\ \texttt{let}\ \ (e_1' : Exp[\varphi\sigma X_1 \to \varphi\sigma X_2 \to \varphi\sigma X_3]) = \text{coerce}\ e_1\ (\text{iExp}\ p_2')\ \ \texttt{in} \\
\quad\ \texttt{let}\ \ (e_2' : Exp[\varphi\sigma X_1 \to \varphi\sigma X_2]) = \text{coerce}\ e_2\ (\text{iExp}\ p_3')\ \ \texttt{in} \\
\quad\ \texttt{let}\ \ (e_3' : Exp[\varphi\sigma X_1]) = \text{coerce}\ e_3\ (\text{iExp}\ p_4)\ \ \texttt{in} \\
\quad\ \text{coerce}\ (\text{A}\ (\text{A}\ e_1'\ e_3')\ (\text{A}\ e_2'\ e_3'))\ (\text{iExp}\ p_5))\ \ \texttt{in}
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ \ (reduceS : \forall T, U.U \to (Exp[T] \to U) \to Exp[T] \to U)\ = \\
\quad \lambda(notS : U).\lambda(ifS : Exp[T] \to U). \\
\quad matchS3\ notS\ (\lambda(p : Ty[\mathrm{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4). \\
\qquad\qquad\qquad\qquad \lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).\lambda(e_3 : Exp[T_3]). \\
\qquad\qquad\qquad\qquad ifS\ (enactS\ p\ e_1\ e_2\ e_3))\ \ \texttt{in}
\end{array}
$$

APPENDIX C

A Self-Interpreter

Our self-interpreter *enact* can be used to evaluate a representation of any expression in our language, including itself. We discuss the self-interpreter in detail in section 9.

$$
\begin{array}{l}
\texttt{let}\ \ (\mathit{unquote} : \forall X.Exp[X] \to X)\ = \\
\quad \texttt{let}\ \ \texttt{rec}\ \ (\mathit{unquote} : \forall X.Exp[X] \to X)\ = \\
\qquad \mathrm{G\ I}\ (\lambda(e_1 : Exp[T_1 \to T]).\lambda(e_2 : Exp[T_1]).\mathit{unquote}\ e_1\ (\mathit{unquote}\ e_2))\ \ \texttt{in} \\
\\
\texttt{let}\ \ \texttt{rec}\ \ (\mathit{enact} : \forall T.Exp[T] \to Exp[T])\ = \\
\\
\texttt{let}\ \ (\mathit{enactK} : \forall T_1, T_2, T_3.(Ty[\mathrm{K}] \dot{\leq} T_1 \to T_2 \to T_3) \to \\
\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3])\ = \\
\quad (\lambda(pIsK : Ty[\mathrm{K}] \dot{\leq} T_1 \to T_2 \to T_3)\,. \\
\qquad \lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]). \\
\qquad eBinary\ pIsK \\
\qquad\quad (\,\lambda(p_1 : \varphi \hat{\leq} \varphi' \circ \sigma)\,. \\
\qquad\quad\ \lambda(p_2 : T_1 \dot{\leq} \varphi' \sigma X)\,. \\
\qquad\quad\ \lambda(p_3 : T_2 \dot{\leq} \varphi' \sigma Z)\,. \\
\qquad\quad\ \lambda(p_4 : \varphi' \sigma X \dot{\leq} T)\,. \\
\qquad\quad\ \texttt{let}\ \ (p_5 : T_1 \dot{\leq} T) = \mathrm{trans}\ p_2\ p_4\ \ \texttt{in} \\
\qquad\quad\ \texttt{let}\ \ (p_6 : Exp[T_1] \dot{\leq} Exp[T]) = \mathrm{iExp}\ p_5
\end{array}
$$

$$\texttt{in}\ \ enact(\text{coerce } e_1\ p_6)))\ \ \texttt{in}$$

$$\texttt{let}\ \ (enactS : \forall T_1, T_2, T_3, T_4.(Ty[\text{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4) \to$$
$$Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4])\ =$$
$$\lambda(p : Ty[\text{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4).$$
$$\lambda(e_1 : Exp[T_1]), (e_2 : Exp[T_2]), (e_3 : Exp[T_3]).$$
$$eTernary\ p$$
$$(\lambda(p_1 : \forall[X_1, X_2, X_3]\hat{\leq}(\varphi \circ \sigma)).$$
$$\lambda(p_2 : T_1\dot{\leq}\varphi\sigma(X_1 \to X_2 \to X_3)).$$
$$\lambda(p_3 : T_2\dot{\leq}\varphi\sigma(X_1 \to X_2)).$$
$$\lambda(p_4 : T_3\dot{\leq}\varphi\sigma X_1).$$
$$\lambda(p_5 : \varphi\sigma X_3\dot{\leq}T_4).$$
$$\texttt{let}\ \ (p_2' : T_1\dot{\leq}\varphi\sigma X_1 \to \varphi\sigma X_2 \to \varphi\sigma X_3) = \text{trans } p_2 \text{ dist2}\ \ \texttt{in}$$
$$\texttt{let}\ \ (p_3' : T_2\dot{\leq}\varphi\sigma X_1 \to \varphi\sigma X_2) = \text{trans } p_3 \text{ dist}\ \ \texttt{in}$$
$$\texttt{let}\ \ (e_1' : Exp[\varphi\sigma X_1 \to \varphi\sigma X_2 \to \varphi\sigma X_3]) = \text{coerce } e_1\ (\text{iExp } p_2')\ \ \texttt{in}$$
$$\texttt{let}\ \ (e_2' : Exp[\varphi\sigma X_1 \to \varphi\sigma X_2]) = \text{coerce } e_2\ (\text{iExp } p_3')\ \ \texttt{in}$$
$$\texttt{let}\ \ (e_3' : Exp[\varphi\sigma X_1]) = \text{coerce } e_3\ (\text{iExp } p_4)\ \ \texttt{in}$$
$$enact(\text{coerce } (\text{A } (\text{A } e_1'\ e_3')\ (\text{A } e_2'\ e_3'))\ (\text{iExp } p_5)))\ \ \texttt{in}$$

$$\texttt{let}\ \ (enactStrict : \forall T.Exp[T] \to Exp[T])\ =$$
$$\texttt{let}\ \ \texttt{rec}\ \ (f : \forall T.Exp[T] \to Exp[T])\ =$$
$$\text{G Q } (\lambda(e_1 : Exp[T_1 \to T]).\lambda(e_2 : Exp[T_1]).\text{A } (f\ e_1)\ (f\ (enact\ e_2)))\ \ \texttt{in}$$

$$\texttt{let}\ \ (enactG : \forall T_1, T_2, T_3, T_4.(Ty[\text{G}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4) \to$$
$$Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4])\ =$$
$$\lambda(pIsG : Ty[\text{G}]\dot{\leq}T1 \to T2 \to T3 \to T4).$$

$$\lambda(e_1 : Exp[T1]).\lambda(e_2 : Exp[T2]).\lambda(e_3 : Exp[T3])\,.$$

$$\texttt{let}\ \ (error : Exp[T4]) \ =\ \mathrm{A}\ (\mathrm{A}\ (\mathrm{A}\ (\mathrm{Q}\ (\mathrm{coerce}\ \mathrm{G}\ pIsG))\ e_1)\ e_2)\ e_3\ \ \texttt{in}$$

$$\texttt{let}\ \ (mkFQ : \varphi(Exp[U1 \to U2] \to Exp[U1] \to Exp[U2])) \ =\ \mathrm{A}\ \ \texttt{in}$$

$$\texttt{let}\ \ (mkFA : \varphi((\forall U3.Exp[Exp[U3 \to U1] \to Exp[U3] \to U2]) \to$$

$$(\forall U3.Exp[Exp[U3 \to U1]] \to Exp[Exp[U3]] \to Exp[U2])))$$

$$=\ \lambda(x_1 : \forall U3.Exp[Exp[U3 \to U1] \to Exp[U3] \to U2])\,.$$

$$\lambda(x_2 : Exp[Exp[U3 \to U1]])\,.$$

$$\lambda(x_3 : Exp[Exp[U3]])\,.$$

$$\mathrm{A}\ (\mathrm{A}\ x_1\ x_2)\ x_3\ \ \texttt{in}$$

$$\texttt{let}\ \ (fG : \varphi((Exp[U1] \to Exp[U2]) \to$$

$$(\forall U3.Exp[Exp[U3 \to U1]] \to Exp[Exp[U3]] \to Exp[U2]) \to$$

$$Exp[Exp[U1]] \to Exp[U2]))\ \ =$$

$$\lambda(fQ : Exp[U1] \to Exp[U2])\,.$$

$$\lambda(fA : \forall U3.Exp[Exp[U3 \to U1]] \to Exp[Exp[U3]] \to Exp[U2])\,.$$

$$\mathrm{G}\ (\lambda(z : Exp[U1]).\ error)$$

$$(\lambda(e_4 : Exp[U4 \to Exp[U1]])\,.\ \ \texttt{// (Q Q) or (A (Q A) p')}$$

$$\lambda(e_5 : Exp[U4])\,.\qquad\qquad \texttt{// 'o}$$

$$\mathrm{G}\ (\lambda(e_4' : U4 \to Exp[U1])\,.\ \ \texttt{// Q}$$

$$\mathrm{IsQ}\ e_4'$$

$$(\lambda(p_7 : Ty[\mathrm{Q}] \dot{\leq} U4 \to Exp[U1])\,.$$

$$\lambda(unusedQ : Ty[\mathrm{Q}])\,.$$

$$\mathrm{eArrow}\ p_7$$

$$(\lambda(p_8 : \varphi_2 \hat{\leq} \varphi_3 \circ \sigma_1)\,.$$

$$\lambda(p_9 : U4 \dot{\leq} \varphi_3\sigma_1 U5)\,.$$

$$\lambda(p_{10} : \varphi_3\sigma_1 Exp[U5] \dot{\leq} Exp[U1])\,.$$

$$\texttt{let}\ \ (p_{11} : \varphi_3\sigma_1 U5 \dot{\leq} U1)\ \ =$$

$$
\begin{array}{l}
\qquad\qquad\qquad\quad \text{eExp (trans factorExp } p_{10}) \;\; \texttt{in} \\
\qquad\qquad\qquad \texttt{let} \;\; (p_{12} : Exp[U4] \dot{\leq} Exp[U1]) \; = \\
\qquad\qquad\qquad\quad \text{iExp (trans } p_9 \; p_{11}) \;\; \texttt{in} \\
\qquad\qquad\qquad fQ \; (\text{coerce } e_5 \; p_{12}))) \\
\qquad error) \\
// \;\; \text{Q A} \\
(\,\lambda(e_6 : Exp[U5 \rightarrow U4 \rightarrow Exp[U1]])\,. \\
\;\lambda(e_7 : Exp[U5])\,. \\
\;\text{G } (\lambda(e_6' : U5 \rightarrow U4 \rightarrow Exp[U1])\,. \\
\qquad \text{IsA } e_6' \\
\qquad\; (\,\lambda(p_{13} : Ty[\text{A}] \dot{\leq} U5 \rightarrow U4 \rightarrow Exp[U1])\,. \\
\qquad\;\; \lambda(unusedA : Ty[\text{A}])\,. \\
\qquad\;\; eBinary \; p_{13} \\
\qquad\qquad (\,\lambda(p_{14} : \varphi_4 \hat{\leq} \varphi_5 \circ \sigma_2)\,. \\
\qquad\qquad\; \lambda(p_{15} : U5 \dot{\leq} \varphi_5 \sigma_2 Exp[U6 \rightarrow U7])\,. \\
\qquad\qquad\; \lambda(p_{16} : U4 \dot{\leq} \varphi_5 \sigma_2 Exp[U6])\,. \\
\qquad\qquad\; \lambda(p_{17} : \varphi_5 \sigma_2 Exp[U7] \dot{\leq} Exp[U1])\,. \\
\qquad\qquad\; \texttt{let} \;\; (p_{18} : U5 \dot{\leq} Exp[\varphi_5 \sigma_2 U6 \rightarrow \varphi_5 \sigma_2 U7]) \; = \\
\qquad\qquad\quad \text{trans } p_{15} \text{ (trans distExp (iExp dist))} \;\; \texttt{in} \\
\qquad\qquad\; \texttt{let} \;\; (p_{19} : U4 \dot{\leq} Exp[\varphi_5 \sigma' U6]) = \text{trans } p_{16} \text{ distExp} \;\; \texttt{in} \\
\qquad\qquad\; \texttt{let} \;\; (p_{20} : \varphi_5 \sigma' U7 \dot{\leq} U1) \; = \\
\qquad\qquad\quad \text{eExp (trans factorExp } p_{17}) \;\; \texttt{in} \\
\qquad\qquad\; \texttt{let} \;\; (p_{21} : U5 \dot{\leq} Exp[\varphi_5 \sigma' U6 \rightarrow U1]) \; = \\
\qquad\qquad\quad \text{trans2 } p_{18} \text{ distExp (iExp} \\
\qquad\qquad\qquad \text{(iArrow refl (eExp (trans factorExp } p_{17})))) \;\; \texttt{in} \\
\qquad\qquad\; \texttt{let} \;\; (e_7' : Exp[Exp[\varphi_5 \sigma' U6 \rightarrow U1]]) \; =
\end{array}
$$

coerce e_7 (iExp p_{21}) in

let $(e_5' : Exp[Exp[\varphi_5 \sigma' U6]])$ =

coerce e_5 (iExp p_{19}) in

$fA\ e_7'\ e_5'$))

$error$) // IsA failed

(K (K $error$)) // e_6 = A x y

e_6)

e_4) in

$eTernary\ pIsG$

$(\lambda(p_2 : \varphi \hat{\leq} \varphi_1 \circ \sigma)$.

$\lambda(p_3 : T1 \dot{\leq} \varphi_1 \sigma (U1 \to U2))$.

$\lambda(p_4 : T2 \dot{\leq} \varphi_1 \sigma (\forall U3.Exp[U3 \to U1] \to Exp[U3] \to U2))$.

$\lambda(p_5 : T3 \dot{\leq} \varphi_1 \sigma Exp[U1])$.

$\lambda(p_6 : \varphi_1 \sigma U2 \dot{\leq} T4)$.

let $(mkFQ' : \varphi_1 \sigma Exp[U1 \to U2] \to$

$\varphi_1 \sigma (Exp[U1] \to Exp[U2])) =$

coerce $mkFQ$ (trans p_2 dist) in

let $(fQ' : \varphi_1 \sigma (Exp[U1] \to Exp[U2]))$ =

$mkFQ'$ (coerce e_1 (trans (iExp p_3) factorExp)) in

let $(mkFA' : \varphi_1 \sigma (\forall U3.Exp[Exp[U3 \to U1] \to Exp[U3] \to U2]) \to$

$\varphi_1 \sigma (\forall U3.Exp[Exp[U3 \to U1]] \to$

$Exp[Exp[U3]] \to Exp[U2]))$ =

coerce $mkFA$ (trans p_2 dist) in

let $(fA' : \varphi_1 \sigma (\forall U3.Exp[Exp[U3 \to U1]] \to$

$Exp[Exp[U3]] \to Exp[U2]))$ =

$mkFA'$ (coerce e_2 (trans (iExp p_4) factorExp)) in

$$
\begin{array}{l}
\mathtt{let}\ \ (fG' : \varphi_1\sigma(Exp[U1] \to Exp[U2]) \to \\
\qquad\qquad \varphi_1\sigma(\forall U3.Exp[Exp[U3 \to U1]] \to \\
\qquad\qquad\qquad\qquad Exp[Exp[U3]] \to Exp[U2]) \to \\
\qquad\qquad \varphi_1\sigma Exp[Exp[U1]] \to \\
\qquad\qquad \varphi_1\sigma Exp[U2]) \ = \\
\quad \text{coerce } fG \text{ (trans } p_2 \text{ (trans dist (iArrow refl dist2)))} \ \ \mathtt{in} \\
\mathtt{let}\ \ (e_3' : Exp[\varphi_1\sigma Exp[U1]]) \ = \ enact \text{ (coerce } e_3 \text{ (iExp } p_5)) \ \ \mathtt{in} \\
\mathtt{let}\ \ (e : \varphi_1\sigma Exp[U2]) = fG' \ fQ' \ fA' \text{ (coerce } e_3' \text{ factorExp)} \ \ \mathtt{in} \\
\mathtt{let}\ \ (p_6' : \varphi_1\sigma Exp[U2] \dot{\leq} Exp[T4]) = \text{trans distExp (iExp } p_6) \ \ \mathtt{in} \\
enact \text{ (coerce } e \ p_6')) \ \ \mathtt{in}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{let}\ \ (enactIs : \forall T_1, T_2, T_3, T_4.(TyIs \dot{\leq} T_1 \to T_2 \to T_3 \to T_4) \to TyIs \to \\
\qquad\qquad\qquad\qquad\qquad Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4]) \ = \\
\quad (\lambda(p_1 : TyIs \dot{\leq} X \to Y \to Z \to T4). \\
\quad \lambda(o' : TyIs). \\
\quad \mathtt{let}\ \ (mkIs : U1 \to (U2 \to U3 \to Exp[U4]) \to Exp[U4] \to Exp[U4] \to \\
\qquad\qquad Exp[U1] \to Exp[U2 \to U3 \to U4] \to Exp[U4] \to Exp[U4]) \\
\qquad = \ \lambda(isO : U1 \to (U2 \to U3 \to Exp[U4]) \to Exp[U4] \to Exp[U4]). \\
\qquad\quad \lambda(o'' : Exp[U1]). \\
\qquad\quad \lambda(eTrue : Exp[U2 \to U3 \to U4]). \\
\qquad\quad \lambda(eFalse : Exp[U4]). \\
\qquad\quad \mathrm{G}(\lambda(a : U1). \\
\qquad\qquad isO \ (unquote \ o'') \\
\qquad\qquad\quad (\lambda(p : U2).\lambda(t : U3).\mathrm{A} \ (\mathrm{A} \ eTrue \ (\mathrm{Q} \ p)) \ (\mathrm{Q} \ t)) \\
\qquad\qquad\quad eFalse) \\
\qquad\qquad (\mathrm{K} \ (\mathrm{K} \ eFalse))
\end{array}
$$

$(enact\ o'')$ in

let $(is : \varphi(Exp[U1] \to Exp[U2 \to U3 \to U4] \to Exp[U4] \to Exp[U4]))$

$=$ (coerce $mkIs$ dist) o' in

eArrow p_1

$(\lambda(p_2 : \varphi \hat{\leq} \varphi_1 \circ \sigma_1)\,.$

$\lambda(p_3 : T1 \dot{\leq} \varphi_1\sigma_1 U1)\,.$

$\lambda(p_4 : \varphi_1\sigma_1(U2 \to U3 \to U4) \to U4 \to U4 \dot{\leq} T2 \to T3 \to T4)\,.$

$\lambda(e_1 : Exp[T1])\,.$

let $(is_1 : (\varphi_1\sigma_1 Exp[U1]) \to$

$(\varphi_1\sigma_1 Exp[U2 \to U3 \to U4] \to Exp[U4] \to Exp[U4]))$ $=$

coerce is (trans p_2 dist) in

let $(is_2 : \varphi_1\sigma_1 Exp[U2 \to U3 \to U4] \to Exp[U4] \to Exp[U4])$ $=$

is_1(coerce e_1 (trans (iExp p_3) factorExp)) in

$eBinary$ p_4

$(\lambda(p_5 : \varphi_1 \circ \sigma_1 \hat{\leq} \varphi_3 \circ \sigma_3)\,.$

$\lambda(p_6 : T2 \dot{\leq} \varphi_3\sigma_3(U2 \to U3 \to U4))\,.$

$\lambda(p_7 : T3 \dot{\leq} \varphi_3\sigma_3 U4)\,.$

$\lambda(p_8 : \varphi_3\sigma_3 U4 \dot{\leq} T4)\,.$

$\lambda(e_2 : Exp[T2]), (e_3 : Exp[T3])\,.$

let $(is_3 : \varphi_3\sigma_3 Exp[U2 \to U3 \to U4] \to$

$\varphi_3\sigma_3 Exp[U4] \to \varphi_3\sigma_3 Exp[U4])$ $=$

coerce is_2 (trans p_5 dist2) in

let $(e : \varphi_3\sigma_3 Exp[U4])$ $=$

is_3 (coerce e_2 (trans (iExp p_6) factorExp))

(coerce e_3 (trans (iExp p_7) factorExp)) in

$enact$ (coerce e (trans distExp (iExp p_8)))))) in

$$
\begin{aligned}
&\texttt{let}\ \ (enactY : \forall T_1, T_2, T_3.(Ty[\mathrm{Y}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \\
&\qquad\qquad\qquad Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) \ = \\
&\lambda(p_1 : Ty[\mathrm{Y}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T_3). \\
&\lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]). \\
&eBinary\ p_1 \\
&\ \ (\lambda(p_2 : \forall[X_1, X_2]\hat{\leq}\varphi \circ \sigma). \\
&\quad \lambda(p_3 : T_1\dot{\leq}\varphi\sigma((X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_2)). \\
&\quad \lambda(p_4 : T_2\dot{\leq}\varphi\sigma X_1). \\
&\quad \lambda(p_5 : \varphi\sigma X_2\dot{\leq}T_3). \\
&\quad \texttt{let}\ \ (e_1' : Exp[\varphi\sigma(X_1 \rightarrow X_2) \rightarrow \varphi\sigma X_1 \rightarrow \varphi\sigma X_2]) \ = \\
&\qquad \text{coerce } e_1 \text{ (iExp (trans } p_3 \text{ dist2))} \ \ \texttt{in} \\
&\quad \texttt{let}\ \ (y : \varphi\sigma((X_1 \rightarrow X_2) \rightarrow (X_1 \rightarrow X_2)) \rightarrow \varphi\sigma(X_1 \rightarrow X_2)) \ = \\
&\qquad \text{coerce Ydist} \ \ \texttt{in} \\
&\quad \texttt{let}\ \ (e : Exp[\varphi\sigma X_1 \rightarrow \varphi\sigma X_2]) \ = \\
&\qquad \text{A } e_1' \text{ (A (Q } y\text{) (coerce } e_1 \text{ (iExp } p_3\text{)))} \ \ \texttt{in} \\
&\quad enact \text{ (coerce (A } e\text{(coerce } e_2 \text{ (iExp } p_4\text{))) (iExp } p_5\text{)))} \ \ \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\ \ (enactCoerce : \forall T_1, T_2, T_3.(Ty[\text{coerce}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \\
&\qquad\qquad\qquad\qquad Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) \ = \\
&\quad \lambda(pIsCoerce : Ty[\text{coerce}]\dot{\leq}T_1 \rightarrow T_2 \rightarrow T_3)\,. \\
&\qquad \texttt{let}\ \ (f : Exp[U_1] \rightarrow Exp[U_1\dot{\leq}U_2] \rightarrow Exp[U_2]) \ = \\
&\qquad\quad \lambda(a_1 : Exp[U_1])\,. \\
&\qquad\quad \lambda(a_2 : Exp[U_1\dot{\leq}U_2])\,. \\
&\qquad\quad \text{coerce } a_1 \text{ (iExp (}unquote\text{ (}enactStrict\text{ (}enact\ a_2\text{))))} \ \ \texttt{in} \\
&\qquad eBinary\ pIsCoerce
\end{aligned}
$$

$$
\begin{aligned}
&(\,\lambda(p_1 : \varphi \hat{\leq} \varphi' \circ \sigma')\,.\\
&\;\lambda(p_2 : T_1 \dot{\leq} \varphi'\sigma' V_1)\,.\\
&\;\lambda(p_3 : T_2 \dot{\leq} \varphi'\sigma'(V_1 \dot{\leq} V_2))\,.\\
&\;\lambda(p_4 : \varphi'\sigma' V_2 \dot{\leq} T_3)\,.\\
&\;\lambda(e_1 : \mathit{Exp}[T_1])\,.\\
&\;\lambda(e_2 : \mathit{Exp}[T_2])\,.\\
&\;\texttt{let}\;\; (f' : \varphi'\sigma'\mathit{Exp}[V_1] \to \varphi'\sigma'\mathit{Exp}[V_1 \dot{\leq} V_2] \to \varphi'\sigma'\mathit{Exp}[V_2]) \;=\\
&\qquad \text{coerce } f \text{ (trans } p_1 \text{ dist2)} \;\; \texttt{in}\\
&\;\texttt{let}\;\; (e_1' : \varphi'\sigma'\mathit{Exp}[V_1]) \;=\\
&\qquad \text{coerce } e_1 \text{ (trans (iExp } p_2\text{) factorExp)} \;\; \texttt{in}\\
&\;\texttt{let}\;\; (e_2' : \varphi'\sigma'\mathit{Exp}[V_1 \dot{\leq} V_2]) \;=\\
&\qquad \text{coerce } e_2 \text{ (trans (iExp } p_3\text{) factorExp)} \;\; \texttt{in}\\
&\;\texttt{let}\;\; (e : \varphi'\sigma'\mathit{Exp}[V_2]) \;=\; f' e_1' e_2' \;\; \texttt{in}\\
&\;\mathit{enact} \text{ (coerce } e \text{ (trans distExp (iExp } p_4\text{))))} \;\; \texttt{in}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let}\;\; (\mathit{enactEArrow} : \forall T_1, T_2, T_3.(\mathit{Ty}[\text{eArrow}] \dot{\leq} T_1 \to T_2 \to T_3) \to\\
&\qquad\qquad\qquad\qquad \mathit{Exp}[T_1] \to \mathit{Exp}[T_2] \to \mathit{Exp}[T_3]) \;=\\
&\;\lambda(\mathit{pIsEArrow} : \mathit{Ty}[\text{eArrow}] \dot{\leq} T_1 \to T_2 \to T_3)\,.\\
&\;\texttt{let}\;\; (f : \varphi(\mathit{Exp}[\varphi_1(U_1 \to U_2) \dot{\leq} \varphi_2(V_1 \to V_2)] \to\\
&\qquad\qquad \mathit{Exp}[\forall \varphi_3, \theta_3.(\varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3) \to\\
&\qquad\qquad\qquad (V_1 \dot{\leq} \varphi_3\theta_3 U_1) \to (\varphi_3\theta_3 U_2 \dot{\leq} V_2) \to C] \to\\
&\qquad\qquad \mathit{Exp}[C])) \;=\\
&\quad \lambda(q_1 : \mathit{Exp}[\varphi_1(U_1 \to U_2) \dot{\leq} \varphi_2(V_1 \to V_2)])\,.\\
&\quad \lambda(g : \mathit{Exp}[\forall \varphi_3, \theta_3.(\varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3) \to\\
&\qquad\qquad (V_1 \dot{\leq} \varphi_3\theta_3 U_1) \to (\varphi_3\theta_3 U_2 \dot{\leq} V_2) \to C])\,.\\
&\quad \text{eArrow } (\mathit{unquote}\ (\mathit{enactStrict}\ (\mathit{enact}\ q_1)))
\end{aligned}
$$

$(\,\lambda(q_2 : \varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3)\,.$

$\lambda(q_3 : V_1 \dot{\leq} \varphi_3 \theta_3 U_1)\,.$

$\lambda(q_4 : \varphi_3 \theta_3 U_2 \dot{\leq} V_2)\,.$

A (A (A g (Q q_2)) (Q q_3)) (Q q_4)) in

$eBinary\ pIsEArrow$

$(\,\lambda(p_1 : \varphi \hat{\leq} \varphi' \circ \theta')\,.$

$\lambda(p_2 : X \dot{\leq} \varphi'\theta'(\varphi_1(U_1 \to U_2) \dot{\leq} \varphi_2(V_1 \to V_2)))\,.$

$\lambda(p_3 : Y \dot{\leq} \varphi'\theta'(\forall \varphi_3, \theta_3.(\varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3) \to$

$(V_1 \dot{\leq} \varphi_3 \theta_3 U_1) \to (\varphi_3 \theta_3 U_2 \dot{\leq} V_2) \to C))\,.$

$\lambda(p_4 : \varphi'\theta' C \dot{\leq} T)\,.$

$\lambda(e_x : Exp[T_1]).\lambda(e_y : Exp[T_2]).$

let $(f' : \varphi'\theta' Exp[\varphi_1(U_1 \to U_2) \dot{\leq} \varphi_2(V_1 \to V_2)] \to$

$\varphi'\theta' Exp[\forall \varphi_3, \theta_3.(\varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3) \to$

$(V_1 \dot{\leq} \varphi_3 \theta_3 U_1) \to (\varphi_3 \theta_3 U_2 \dot{\leq} V_2) \to C] \to$

$\varphi'\theta' Exp[C])\ =$

coerce f (trans p_1 dist2) in

let $(e'_x : \varphi'\theta' Exp[\varphi_1(U_1 \to U_2) \dot{\leq} \varphi_2(V_1 \to V_2)])\ =$

coerce e_x (trans (iExp p_2) factorExp) in

let $(e'_y : \varphi'\theta' Exp[\forall \varphi_3, \theta_3.(\varphi_1 \hat{\leq} \varphi_2 \circ \varphi_3 \circ \theta_3) \to$

$(V_1 \dot{\leq} \varphi_3 \theta_3 U_1) \to (\varphi_3 \theta_3 U_2 \dot{\leq} V_2) \to C])\ =$

coerce e_y (trans (iExp p_3) factorExp) in

let $(e : \varphi_3 \theta_3 Exp[C])\ =\ f'\ e'_x\ e'_y$ in

enact (coerce e (trans distExp (iExp p_4)))) in

let $(enactI : \forall T_1, T_2.(Ty[\mathrm{I}] \dot{\leq} T_1 \to T_2) \to Exp[T_1] \to Exp[T_2])\ =$

$\lambda(p : Ty[\mathrm{I}] \dot{\leq} T_1 \to T_2).$

$\lambda(e_1 : Exp[T_1])$.

eArrow p

$(\lambda(p_1 : \forall[X]\hat{\leq}(\varphi \circ \sigma))$.

$\lambda(p_2 : T_1\dot{\leq}\varphi\sigma X)$.

$\lambda(p_3 : \varphi\sigma X\dot{\leq}T_2)$.

enact (coerce e_1 (iExp (trans p_2 p_3)))) in

let $(enact3 : \forall T_1, T_2, T_3, T_4.(T_1 \to T_2 \to T_3 \to T_4) \to$

$Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4]) =$

$\lambda(o : T_1 \to T_2 \to T_3 \to T_4)$.

IsS o $(\lambda(p : Ty[\mathrm{S}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4).\lambda(s : Ty[\mathrm{S}]).enactS\ p)$ (

IsG o $(\lambda(p : Ty[\mathrm{G}]\dot{\leq}T_1 \to T_2 \to T_3 \to T_4).\lambda(g : Ty[\mathrm{G}]).enactG\ p)$ (

IsIs o $enactIs$ (

$\lambda(e1 : Exp[T_1]).\lambda(e2 : Exp[T_2]).\lambda(e3 : Exp[T_3]).\mathrm{A}(\mathrm{A}(\mathrm{A}(\mathrm{Q}o)e1)e2)e3$

))) in

let $(enact2 : \forall T_1, T_2, T_3.(T_1 \to T_2 \to T_3) \to Exp[T_1] \to Exp[T_2] \to Exp[T_3]) =$

$\lambda(o : T_1 \to T_2 \to T_3)$.

IsK o $(\lambda(p : Ty[\mathrm{K}]\dot{\leq}T_1 \to T_2 \to T_3).\lambda(k : Ty[\mathrm{K}]).enactK\ p)$(

IsY o $(\lambda(p : Ty[\mathrm{Y}]\dot{\leq}T_1 \to T_2 \to T_3).\lambda(o' : Ty[\mathrm{Y}]).enactY\ p)$(

IsCoerce o $(\lambda(p : Ty[\mathrm{coerce}]\dot{\leq}T_1 \to T_2 \to T_3).\lambda(o' : Ty[\mathrm{coerce}])$.

$enactCoerce\ p)$(

IsEArrow o $(\lambda(p : Ty[\mathrm{eArrow}]\dot{\leq}T_1 \to T_2 \to T_3).\lambda(o' : Ty[\mathrm{eArrow}])$.

$enactEArrow\ p)$(

$\lambda(e1 : Exp[T_1]).\lambda(e2 : Exp[T_2]).\mathrm{A}\ (\mathrm{A}\ (\mathrm{Q}\ o)\ e1)\ e2$

)))) in

let $(enact1 : \forall T_1, T_2.(T_1 \to T_2) \to Exp[T_1] \to Exp[T_2]) =$

$\lambda(o : T_1 \to T_2).$

IsI o $(\lambda(p : Ty[\mathrm{I}] \dot{\leq} T_1 \to T_2).\lambda(o' : Ty[\mathrm{I}]).enactI\ p)($

$\lambda(e1 : Exp[T_1]).$A (Q o) $e1$

) in

G Q

$(\lambda(e1 : Exp[T_1 \to T]).$

G$(\lambda(o : T_1 \to T).enact1\ o)$

(G$(\lambda(o : T_2 \to T_1 \to T).enact2\ o)$

(G$(\lambda(o : T_3 \to T_2 \to T_1 \to T).enact3\ o)$

$(\lambda(x_1 : Exp[T_4 \to T_3 \to T_2 \to T_1 \to T]).$

$\lambda(x_2 : Exp[T_4]).\lambda(x_3 : Exp[T_3]).\lambda(x_4 : Exp[T_2]).\lambda(x_5 : Exp[T_1]).$

A (A (A (A x_1 x_2) x_3) x_4) x_5)))

$(enact\ e1))$

References

[1] Martín Abadi and Luca Cardelli. An imperative object calculus. In *Proceedings of TAPSOFT'95, Theory and Practice of Software Development*, pages 471–485. Springer-Verlag (LNCS 915), 1995.

[2] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Levy. Explicit substitutions. In *Proceedings of POPL'90, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[4] Henk Barendregt. Self-interpretations in lambda calculus. *J. Funct. Program*, 1(2):229–233, 1991.

[5] Michel Bel. A recursion theoretic self interpreter for the lambda-calculus. http://www.belxs.com/michel/#selfint.

[6] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In *CSL*, pages 85–99, 1992.

[7] Kevin Donnelly. System F with constraint types. Master's thesis, Boston University, 2008.

[8] Brendan Eich. Narcissus. `http://mxr.mozilla.org/mozilla/source/js/narcissus/jsexec.js`, 2010.

[9] R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.

[10] Paul Hudak and David A. Kranz. A combinator-based compiler for a functional language. In *Proceedings of POPL'84, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 122–132, 1984.

[11] 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.

[12] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, pages 340–353, 1936.

[13] Daan Leijen. Flexible types: Robust type inference for first-class polymorphism. In *Proceedings of POPL'09, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 66–77, 2009.

[14] Oleg Mazonka and Daniel B. Cristofani. A very short self-interpreter. http://arxiv.org/html/cs/0311032v1, November 2003.

[15] John C. Mitchell. *Lambda Calculus Models of Typed Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1984.

[16] 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.

[17] Torben Æ. Mogensen. Linear-time self-interpretation of the pure lambda calculus. *Higher-Order and Symbolic Computation*, 13(3):217–237, 2000.

[18] Matthew Naylor. Evaluating Haskell in Haskell. *The Monad.Reader*, 10:25–33, 2008.

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

[20] 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.

[21] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740. ACM Press, 1972. The paper later appeared in Higher-Order and Symbolic Computation, 11, 363–397 (1998).

[22] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA Companion*, pages 044–953, 2006.

[23] Andreas Rossberg. HaMLet. http://www.mpi-sws.org/ rossberg/hamlet, 2010.

[24] Fangmin Song, Yongsen Xu, and Yuechen Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171–181, March 2000.

[25] Val Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance and explicit coercion. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 112–129, 1989.

[26] Val Tannen, Carl A. Gunter, and Andre Scedrov. Computing with coercions. In *LFP'90, ACM Conference on Lisp and Functional Programming*, pages 44–60, 1990.

[27] Mitchell Wand, Patrick M. O'Keefe, and Jens Palsberg. Strong normalization with non-structural subtyping. *Mathematical Structures in Computer Science*, 5(3):419–430, 1995.

[28] J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95–019, Comp. Sci. Dept., Boston Univ., December 1995.

[29] J. B. Wells. Typability is undecidable for F+eta. Technical Report 96–022, Comp. Sci. Dept., Boston Univ., March 1996.

[30] Wikipedia. Rubinius. http://en.wikipedia.org/wiki/Rubinius, 2010.

[31] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of PEPM'07, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.