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
© Copyright by
Matthew Scott Brown
2013
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1    Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2    Our Framework</td>
<td>5</td>
</tr>
<tr>
<td>3    What is a proof term?</td>
<td>7</td>
</tr>
<tr>
<td>4    Programming with Proofs</td>
<td>9</td>
</tr>
<tr>
<td>4.1  Subtyping of Function Types</td>
<td>9</td>
</tr>
<tr>
<td>4.2  Full Subtyping</td>
<td>13</td>
</tr>
<tr>
<td>5    Program Representations</td>
<td>17</td>
</tr>
<tr>
<td>6    Our Self-optimizer</td>
<td>22</td>
</tr>
<tr>
<td>7    Our Language</td>
<td>24</td>
</tr>
<tr>
<td>7.1  Syntax</td>
<td>24</td>
</tr>
<tr>
<td>7.2  Semantics</td>
<td>25</td>
</tr>
<tr>
<td>7.3  Types</td>
<td>26</td>
</tr>
<tr>
<td>7.4  Lambda Abstraction and Let-Terms</td>
<td>34</td>
</tr>
<tr>
<td>7.5  Soundness</td>
<td>35</td>
</tr>
<tr>
<td>8    Type Inference</td>
<td>37</td>
</tr>
<tr>
<td>9    A Self-Interpreter</td>
<td>38</td>
</tr>
<tr>
<td>10   Experimental Results</td>
<td>39</td>
</tr>
<tr>
<td>11   Related Work</td>
<td>40</td>
</tr>
<tr>
<td>12   Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>A    Proofs</td>
<td>43</td>
</tr>
</tbody>
</table>
B  Optimizations ........................................... 57

C  A Self-Interpreter ................................. 66

References ............................................. 78
## List of Figures

1. Implementations of eBinary and expandK .......................... 20
2. Implementation of SK2KI ......................................... 21
3. A Self-Optimizer .................................................. 23
4. Arities of Atoms ................................................... 25
5. Operational Semantics ............................................. 26
6. Atom Types ......................................................... 33
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 → 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 → 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 → 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 \( \lambda \)-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 \( \lambda \)-calculus [16, 6, 24] and other languages [21, 14, 31]. Rendel, Ostermann, and Hofer [20] presented a self-recognizer with type

\[ \forall T. \text{Exp}[T] \to T \]

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 \( \text{Exp}[T] \) and \( T \). We improve on the latter result and present the first self-enactor that has type

\[ \forall T. \text{Exp}[T] \to \text{Exp}[T] \]

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:

\[ B \ K \ x_2 \ x_1 \rightarrow \text{enact} \ (K \ x_2 \ x_1) \]
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 $SKx$ and $KIx$ 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 \((\ast \rightarrow \ast)\) and kind \(\ast\), and also type equivalence and an extension of Mitchell subtyping [15]. In slogan form:

\[
\text{Language} = \text{combinators} + \text{program representation} + \\
\text{operator equality} + \text{proof terms} \\
\text{Types} = \text{two kinds} + \text{equivalence} + \text{subtyping} \\
\text{Technique} = \text{programming with proofs}
\]

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 $SK \rightarrow KI$ 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 $(V \rightarrow W) \rightarrow V \rightarrow V$. If $V$ and $W$ are distinct, then we cannot derive $I : T'$ which shows that $SK \rightarrow I$ isn’t always type preserving. On the other hand, we can also derive $KI : T'$, so again we are tempted to think that $SK \rightarrow KI$ is type preserving. How can a type check of $SK \rightarrow KI$ 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 $SK \rightarrow KI$ is type preserving.

The implementation of $SK \rightarrow KI$ 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 $\forall T,U. T \rightarrow U \rightarrow U$ is a subtype of any type of $SK$.

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

This is the common structure for all our optimizations, though the details vary. For example, there is no construct step for $S(Ke)I \rightarrow e$, 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:

\[
\text{Type-Subsumption} \quad \vdash e : T \quad T \subseteq U \quad \vdash e : U
\]

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*:

\[
\text{Type-Coerce} \quad \vdash e : T \quad \vdash p : T \leq U \quad \vdash \text{coerce}(e, p) : U
\]

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

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 \( T_1 \leq T_2 \) and a proof term of type \( T_2 \leq T_3 \) into a proof term of type \( T_1 \leq T_3 \). We can use *trans* to build proof terms such as *trans(refl, refl)*, which has type \( T \leq T \). 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 $T \leq U$, 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 $T \leq U$, 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 $e\text{Arrow}$ which has this operational semantics:

$$e\text{Arrow} \quad e\rightarrow e \quad p_1 \quad p_2 \quad p_3$$

where $p_1, p_2, p_3$ are fresh. Here $v_1$ is a proof term and $e$ is a continuation. The above rule says that when we execute $e\text{Arrow}$, 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:

$$ref \quad v_2 \rightarrow l$$

where $l$ 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 $e\text{Arrow}$.

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 $e\text{Arrow}$ and $new$ place no restrictions on the types

8
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 \(\Gamma\) assigns a type \(\Gamma(l)\) to every heap label \(l\). We borrow this approach and use a type environment \(\Gamma\) 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 \(\lambda\)-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:

\[
\frac{S : T_1 \rightarrow T \quad K : T_1}{SK : T}
\]

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:

\[(U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \subseteq T_1 \to T\]

\[V_1 \to V_2 \to V_1 \subseteq T_1\]

The first step of our optimization, \textit{analyzeSK}, presents those two constraints to \textit{proveSK2KI} in the form of two proof terms \(pS\) and \(pK\):

\[
pS : (U_1 \to U_2 \to U_3) \to (U_1 \to U_2) \to U_1 \to U_3 \leq T_1 \to T
\]

\[
pK : V_1 \to V_2 \to V_1 \leq T_1
\]

We ensure a strong connection between proof terms and subtyping: if \(p : T \leq U\) 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-\(\rightarrow\) rule:

\[
\text{Sub-\(\rightarrow\)}\quad \frac{U_1 \subseteq T_1 \quad T_2 \subseteq U_2}{T_1 \to T_2 \subseteq U_1 \to U_2}
\]

Let us now program \textit{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-\(\rightarrow\): 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 \(eArrow_1\):

\[
eArrow_1 : \forall T_1, T_2, U_1, U_2, T.
\]

\[
(T_1 \to T_2 \leq U_1 \to U_2) \rightarrow
((U_1 \leq T_1) \rightarrow (T_2 \leq U_2) \rightarrow T) \rightarrow T
\]
The semantics of $e\text{Arrow}_1$ is:

$$e\text{Arrow}_1 \ p \ f \rightarrow f \ p_1 \ p_2$$

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 $p : T_1 \rightarrow T_2 \leq U_1 \rightarrow U_2$, then $p_1$ will have type $U_1 \leq T_1$ and $p_2$ will have type $T_2 \subseteq U_2$. Applying $e\text{Arrow}_1$ to $pS$ will dynamically generate new proof constants $p_1$ and $p_2$:

$$p_1 : T_1 \leq (U_1 \rightarrow T_2 \rightarrow U_3)$$

$$p_2 : (U_1 \rightarrow T_2) \rightarrow U_1 \rightarrow U_3 \leq T$$

Now our task becomes clearer: we need to show $KI$ can be assigned any type $(U_1 \rightarrow T_2) \rightarrow U_1 \rightarrow U_3$. 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 $\text{trans}$ to encode the transitive subtyping rule, and use it to construct a new proof $p_3$.

$$\text{trans} : \forall T_1, T_2, T_3. (T_1 \leq T_2) \rightarrow (T_2 \leq T_3) \rightarrow (T_1 \leq T_3)$$

$$\text{let } (p_3 : (V_1 \rightarrow V_2 \rightarrow V_1) \leq (U_1 \rightarrow T_2 \rightarrow U_3)) = \text{trans} \ pK \ p_1$$

We define helper functions $e\text{Binary}_1$ and $\text{expandK}_1$. $e\text{Binary}_1$ decomposes $p_3$ into three components using two applications of $e\text{Arrow}_1$, which $\text{expandK}_1$ combines to produce the essential constraint needed from $p_3$.

$$\text{let } e\text{Binary}_1 =$$

$$\lambda (p : T_1 \rightarrow T_2 \rightarrow T_3 \leq U_1 \rightarrow U_2 \rightarrow U_3).$$

$$\lambda (f : (U_1 \leq T_1) \rightarrow (U_2 \leq T_2) \rightarrow (T_3 \leq U_3) \rightarrow V).$$

$$e\text{Arrow}_1 \ p$$
\[(\lambda(p_1 : U_1 \leq T_1).\lambda(p_2 : T_2 \rightarrow U_2 \leq U_2 \rightarrow U_3)).\]
eArrow_{1} p_2 \ (\lambda(p_3 : U_2 \leq T_2).\lambda(p_4 : T_3 \leq U_3). \ f \ p_1 \ p_3 \ p_4))

let \( \text{expand}K_1 = \lambda(p : (T_1 \rightarrow T_2 \rightarrow T_1) \leq (U_1 \rightarrow U_2 \rightarrow U_3)). \)
eBinary_{1} p \ (\lambda(p_1 : U_1 \leq T_1).K (\lambda(p_2 : T_1 \leq U_3).\text{trans} p_1 \ p_2))

let \( (p_4 : U_1 \leq U_3) = \text{expand}K_1 \ p_3 \)

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 \( (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_1 \), which is known to be a type of \( KI \), is also a subtype of \( (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_3 \). In order to construct a proof term of this fact, we need two more proof constructors:

\[ \text{refl} : \forall T.T \leq T \]
\[ iArrow : \forall T_1, T_2, U_1, U_2.(U_1 \leq T_1) \rightarrow (T_2 \leq U_2) \rightarrow (T_1 \rightarrow T_2 \leq U_1 \rightarrow U_2) \]

let \( (p_5 : (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_1 \leq (U_1 \rightarrow U_1) \rightarrow U_1 \rightarrow U_3) = \)
\[ iArrow \ \text{refl} \ (iArrow \ \text{refl} \ p_6) \]

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

let \( \text{proveSK2KI} = \)
\[ \lambda(pS : (U_1 \rightarrow U_2 \rightarrow U_3) \rightarrow (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_3 \leq T_1 \rightarrow T). \]
\[ \lambda(pK : (V_1 \rightarrow V_2 \rightarrow V_1) \leq T_1). \]
eArrow_{1} pS
\[ (\lambda(p_1 : T_1 \leq U_1 \rightarrow U_2 \rightarrow U_3). \]
\[ \lambda(p_2 : (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_3 \leq T). \]
let \( (p_3 : V_1 \rightarrow V_2 \rightarrow V_1 \leq U_1 \rightarrow U_2 \rightarrow U_3) = \text{trans} \ pK \ p_1 \)
let \((p4 : U_1 \preceq U_3) = \text{expandK}_1 p3\) in
let \((p5 : (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_1 \preceq (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_3)\) = \(iArrow \text{refl} (iArrow \text{refl} p4)\) in
\text{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.

\[
\text{Sub-Dist-} \rightarrow \forall \vec{\alpha}.T \rightarrow U \subseteq (\forall \vec{\alpha}.T) \rightarrow (\forall \vec{\alpha}.U)
\]

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

In the presence of full subtyping \(\text{analyzeSK}\) will produce the following sub-type constraints for \(T_1 \rightarrow T\) and \(T_1\):

\[
\forall \vec{\alpha}.(U_1 \rightarrow U_2 \rightarrow U_3) \rightarrow (U_1 \rightarrow U_2) \rightarrow U_1 \rightarrow U_3 \subseteq T_1 \rightarrow T
\]
\[
\forall \vec{\beta}.V_1 \rightarrow V_2 \rightarrow V_1 \subseteq T_1
\]

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 \(\forall \vec{\alpha}.T_1 \rightarrow T_2 \subseteq \forall \vec{\gamma}.U_1 \rightarrow U_2\), there exist quantifiers \(\vec{\beta}\) and a substitution \(\theta\) 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 \(\forall \vec{\alpha}.T_1 \rightarrow T_2 \subseteq \forall \vec{\gamma}.U_1 \rightarrow U_2\) can be normalized to the form:
Adding quantifiers and substitutions leads to a new challenge: how do we capture the relationship between \( \vec{\alpha} \) and \( \vec{\beta}, \vec{\gamma}, \) and \( \theta \)? This is complicated by the fact that sometimes one or more of \( \vec{\alpha}, \vec{\beta}, \vec{\gamma}, \) and \( \theta \) are unknown. In particular, the inversion lemma above states there exist quantifiers \( \vec{\beta} \) and a substitution \( \theta \). 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 \( \theta \). By \( \alpha \) 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 \( \forall \alpha.T \rightarrow T \) can be written \( \forall[\alpha](T \rightarrow T) \). When the quantifiers are unknown, as in the case of \( \vec{\gamma} \) above, we use a type variable of kind \( \ast \rightarrow \ast \) to show that some quantifier exists: \( \varphi(T \rightarrow T) \). We denote concatenation of two type constructors \( \varphi_1 \) and \( \varphi_2 \) as \( \varphi_1 \circ \varphi_2 \). We represent substitutions similarly using type variables \( \sigma \), which also have kind \( \ast \rightarrow \ast \). A type variable \( \sigma \) 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, \( (\sigma T) \rightarrow (\sigma U) \) is equivalent to \( \sigma(T \rightarrow U) \) if \( \sigma \) has sort Subst.
This equivalence forms the basis of the proof constructor \textit{factor}:

\[ \text{factor} : \forall \sigma, T_1, T_2. \sigma T_1 \rightarrow \sigma T_2 \preceq \sigma (T_1 \rightarrow T_2) \]

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 \( \rho \) to range over either sort. Congruence states that if \( T \subseteq U \), then \( \rho T \subseteq \rho U \), where \( \rho \) is any substitution or quantifier set.

\[ \text{dist} : \forall \rho, T_1, T_2. \rho(T_1 \rightarrow T_2) \preceq \rho T_1 \rightarrow \rho T_2 \]

\[ \text{congr} : \forall \rho, T_1, T_2. \left( T_1 \preceq T_2 \right) \rightarrow \left( \rho T_1 \preceq \rho T_2 \right) \]

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-\( \rightarrow \) step. It is not necessary for \( eArrow \) to introduce a proof of Sub-Dist, since it is available as an axiom via the \textit{dist} proof constructor.

\[ eArrow : \forall T_1, T_2, U_1, U_2, T, \rho, \varphi_1. \]
\[ \left( \rho(T_1 \rightarrow T_2) \preceq \varphi_1(U_1 \rightarrow T_2) \right) \rightarrow \]
\[ \left( \forall \varphi_3, \sigma. \left( \rho \preceq (\varphi_1 \circ \varphi_2 \circ \sigma) \right) \rightarrow \right. \]
\[ (U_1 \preceq \varphi_2 \sigma T_1) \rightarrow \]
\[ (\varphi_2 \sigma T_2 \preceq U_2) \rightarrow \]
\[ T) \rightarrow T \]

The semantics of \( eArrow_1 \) is:

\[ eArrow_1 p f \rightarrow f p_1 p_2 p_3 \]
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 \( p : \rho(T_1 \rightarrow T_2) \leq \varphi_1(U_1 \rightarrow T_2) \), we will get the following types for the new proof constants:

\[
\begin{align*}
p_1 & : \rho \leq (\varphi_1 \circ \varphi_2 \circ \sigma) \\
p_2 & : U_1 \leq \varphi_2 \sigma T_1 \\
p_3 & : \varphi_2 \sigma T_2 \leq U_2
\end{align*}
\]

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

\[
\begin{align*}
sTrans : \forall \varphi_1, \varphi_2, \varphi_3. (\varphi_1 \leq \varphi_2) \rightarrow (\varphi_2 \leq \varphi_3) \rightarrow (\varphi_1 \leq \varphi_3) \\
sCongr : \forall \varphi_1, \varphi_2, \varphi_3. (\varphi_1 \leq \varphi_2) \rightarrow (\varphi_3 \circ \varphi_1 \leq \varphi_3 \circ \varphi_2)
\end{align*}
\]

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 $\sigma$ 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 $\sigma$. 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 $Opt = SK \rightarrow KI$ 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.

\[
Ty[Q] = \forall T. T \rightarrow Exp[T]
\]
\[
\]

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 \(e_1 : T_1 \rightarrow T\) 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 \(T_1\):

\[
G \ a \ b \ (A \ c \ d) \rightarrow b \ c \ d
\]

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

\[
\forall T,U.(T \rightarrow U) \rightarrow \\
(\forall T_1. Exp[T_1 \rightarrow T] \rightarrow Exp[T_1] \rightarrow U) \rightarrow \\
Exp[T] \rightarrow U
\]

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.

\[
Is[O] \ O \ a \ b \rightarrow a \ p \ O
\]

If \(O\) has type \(T\), then the proof constant \(p\) will have type \(Ty[O]\leq T\). 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 \( pS : Ty[S] \leq T_1 \to T \) and \( pK : Ty[K] \leq T_1 \) 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 \( \varphi \). 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 \( \varphi \) 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.
let (eBinary : \forall \phi, T, U, V, \phi', T', U', V', X.
    (\phi(T \to U \to V) \leq \phi'(T' \to U' \to V')) \to
    (\forall \phi'', \sigma.(\phi' \leq \phi' \circ \phi'' \circ \sigma) \to
    (T' \leq \phi'' \sigma T) \to (U' \leq \phi'' \sigma U) \to (\phi'' \sigma V \leq V') \to X) \to
    X =
    \lambda(p_1 : \phi(T \to U \to V) \leq \phi'(T' \to U' \to V')). \text{eArrow } p_1
    (\lambda(p_2 : \phi \leq \phi' \circ \phi'' \circ \sigma).
    \lambda(p_3 : T' \leq \phi'' \sigma T).
    \lambda(p_4 : \phi'' \sigma (U \to V) \leq U' \to V').
    \text{eArrow } p_4
    (\lambda(p_5 : \phi'' \circ \sigma \leq \phi'' \circ \sigma'). \lambda(p_6 : U' \leq \phi'' \sigma' U). \lambda(p_7 : \phi'' \sigma' V \leq V').
    \text{let } (p_8 : \phi' \circ \phi'' \circ \sigma \leq \phi' \circ \phi'' \circ \sigma') = p_5 \text{ in }
    \text{let } (p_9 : \phi \leq \phi' \circ \phi'' \circ \sigma') = p_2 p_8 \text{ in }
    \text{let } (p_{10} : T' \leq \phi'' \sigma' T) = \text{trans } p_3 p_5 \text{ in }
    \lambda(f : \forall \phi'', \sigma.(\phi \leq \phi' \circ \phi'' \circ \sigma) \to
    (T' \leq \phi'' \sigma T) \to (U' \leq \phi'' \sigma U) \to (\phi'' \sigma V \leq V') \to X).
    f p_9 p_{10} p_6 p_7))

let (expandK : \forall T, U, V.Ty[K] \leq \phi(T \to U \to V) \to (T \leq V)) =
    \lambda(pK : Ty[K] \leq \phi(T \to U \to V)).
    \text{eBinary } pK (K (\lambda(p_1 : T \leq \phi \circ X_1). K (\lambda(p_2 : \phi \circ X_1 \leq V). \text{trans } p_1 p_2)))

Figure 1: Implementations of eBinary and expandK
let (analyzeSK : ∀T,U.
               U → (∀T₁.(Ty[S] ≤ T₁ → T) → (Ty[K] ≤ T₁) → U) →
               Exp[T] → U) =
matchS1 ifNotSK
  (λ(pS : Ty[S] ≤ T₁ → T).
   matchK0 ifNotSK
     (λ(pK : Ty[K] ≤ T₁). ifSK pS pK))
in
let (proveSK2KI : ∀T₁,T,U.
            ((Exp[∀φ,U,V.φ(U → V → V)] ≤ Exp[T]) → U) →
             (Ty[S] ≤ T₁ → T) → (Ty[K] ≤ T₁) → U) =
λ(f : (Exp[∀φ,U,V.φ(U → V → V)] ≤ Exp[T]) → U).
  (λ(p₃ : φ₁ ≤ φ₃ o σ).
   λ(p₄ : T₁ ≤ φ₃(B₁ → B → C)).
   λ(p₅ : φ₂(B₁ → B) → B₁ → C ≤ T).
   let (p₆ : φ₂(T → U → T) ≤ φ₃(σB₁ → σB → σC)) =
     trans (trans p₂ p₃) (congr dist2) in
   let (p₁₁ : σB₁ ≤ σB) = expandK p₆ in
   let (p₁₂ : σB₁ → σB₁ ≤ σB → σC) = iArrow refl p₁₁ in
   let (p₁₃ : σB₁ → σB₁ ≤ σ(B₁ → C)) =
     trans p₁₂ factor in
   let (p₁₄ : σ(B₁ → B) → σB₁ → σB₁ ≤ σ(B₁ → B)) =
     iArrow refl p₁₃ in
   let (p₁₅ : σ(B₁ → B) → σB₁ → σB₁ ≤ σ((B₁ → B) → B₁ → C)) =
     trans p₁₄ factor in
   let (p₁₆ : φ₃(σB₁ → B) → σB₁ → σB₁ ≤ T) =
     trans (congr p₁₅) p₅ in
   f (iExp p₁₆))
in
let (constructKI : Exp[∀φ,T,U.φ(T → U)]) =
               A (Q (coerce K dist)) (Q I)
in
analyzeSK ifNoOpt
  (proveSK2KI(λ(p : Exp[∀φ,U,V.φ(U → V → V)] ≤ Exp[T]).
              ifOpt (coerce constructKI p)))

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 \textit{eta}, \textit{reduceK}, and \textit{reduceS}, in addition to \textit{SK2KI}. The definitions of these optimization steps are listed in Appendix B. Each is implemented in the same style as \textit{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 $\forall T.\text{Exp}[T] \rightarrow \text{Exp}[T]$. \textit{composeOpt} composes two optimization steps such as \textit{SK2KI} into a larger step. The two arguments and result of \textit{composeOpt} have the type of a single optimization step in the framework: $\forall T,U.U \rightarrow (\text{Exp}[T] \rightarrow U) \rightarrow \text{Exp}[T] \rightarrow U$. 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 \textit{runOpt}. It builds a complete optimizer with type $\forall T.\text{Exp}[T] \rightarrow \text{Exp}[T]$ from a single optimization step, by applying it repeatedly to an input program until no further optimizations can be applied. It uses the function \textit{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 \textit{SK2KI}: \textit{eta} performs $\eta$ reduction, and \textit{reduceK} and \textit{reduceS} implement the reduction rules for $K$ and $S$ respectively.
opt1 (opt2 noOpt ifOpt e) ifOpt e in

let (tryApp : U) = G (K ifNoOpt)
  (λ(e1 : Exp[T' → T]).λ(e2 : Exp[T'])).
let (tryE2 : U) = traverseF ifNoOpt
  (λ(newE2 : Exp[T']). ifOpt (A e1 newE2)) e2 in
traverseF tryE2 (λ(newE1 : Exp[T' → T]).
  traverseF (ifOpt (A newE1 e2))
  (λ(newE2 : Exp[T']). ifOpt (A newE1 newE2))
  e2) e1 e' in
f' tryApp ifOpt e' in

Exp[T] → Exp[T]) =
Exp[T] → Exp[T]) =
traverse f e (runOpt1 f) e in

// SK2KI defined in Figure 2
let (SK2KI : ∀T.U → (Exp[T] → U) → Exp[T] → U) = ... in

// eta, reduceK, and reduceS defined in Appendix B
let (eta : ∀T.U → (Exp[T] → U) → Exp[T] → U) = ... in
let (reduceK : ∀T.U → (Exp[T] → U) → Exp[T] → U) = ... in
let (reduceS : ∀T.U → (Exp[T] → U) → Exp[T] → U) = ... in

runOpt (composeOpt 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:

\[
\begin{align*}
    e & ::= e_1 e_2 | \text{atom} | p | x \\
    \text{atom} & ::= O | P | Is[O] | IsIs \\
    O & ::= S | K | I | Y | Q | A | G | \text{coerce} | e\text{Arrow} \\
    P & ::= \text{refl} | i\text{Arrow} | i\text{Exp} | e\text{Exp} \\
      & | \text{dist} | \text{distExp} | f\text{actor} | f\text{actorExp} \\
      & | \text{congr} | s\text{Congr} | \text{trans} | s\text{Trans}
\end{align*}
\]

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

\[
\begin{align*}
    \text{'O} & = Q\ O \\
    \text{'(a b)} & = A\ \text{'a}\ \text{'b}
\end{align*}
\]

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
<table>
<thead>
<tr>
<th>Arity</th>
<th>Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>refl, dist, distExp, factor, factorExp</td>
</tr>
<tr>
<td>1</td>
<td>I, Y, Q, iExp, eExp, congr, sCongr</td>
</tr>
<tr>
<td>2</td>
<td>K, A, coerce, eArrow, iArrow, trans, sTrans</td>
</tr>
<tr>
<td>3</td>
<td>S, G</td>
</tr>
</tbody>
</table>

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 constructor’s arity, and the result type of the constructor corresponding to the logical conclusion.

Definition 3. Value

\[
v ::= \text{atom } e_1 \ldots e_i, \text{ where } i < \text{arity(} \text{atom} \text{)}
\]

\[
| \text{Q } e | \text{A } e_1 e_2 \\
| \text{P } v_1 \ldots v_i, \text{ where } i = \text{arity(} P \text{)}
\]

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.
\[ S\ a\ b\ c \rightarrow\ a\ c\ (b\ c) \]
\[ K\ a\ b \rightarrow\ a \]
\[ I\ a \rightarrow\ a \]
\[ Y\ t \rightarrow\ t\ (Y\ t) \]
\[ G\ a\ b\ (Q\ c) \rightarrow\ a\ c \]
\[ G\ a\ b\ (A\ c\ d) \rightarrow\ b\ c\ d \]
\[ Is[O]\ a\ b \rightarrow\ a\ p\ O \quad p\ fresh \]
\[ Is[O]\ v\ a\ b \rightarrow\ b \quad \text{if } v \neq O \]
\[ IsIs\ v\ a\ b \rightarrow\ a\ p\ v \quad p\ fresh,\ \text{if } v \in\ \{Is[O], IsIs\} \]
\[ IsIs\ v\ a\ b \rightarrow\ b \quad \text{otherwise} \]
\[ coerce\ e\ v \rightarrow\ e \]
\[ eArrow\ v\ e \rightarrow\ e\ p_1\ p_2\ p_3 \quad \text{where } p_1, p_2, p_3\text{ are fresh} \]

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:

\[ T ::=\ \alpha | T_1 \rightarrow T_2 | Exp[T] | \varphi | \forall[^\alpha] | \sigma | Subst[\theta] | \rho \]
\[ | T_1 T_2 | T_1 \leq T_2 | T_1 \triangleq T_2 | T_1 \circ T_2 \]

We use \( \vec{\alpha} \) to denote sets of quantifiers \( T_1, T_2, \ldots, T_n \). Function values (\( atom\ e_1 \ldots e_i, \text{where } i < \text{arity}(atom) \)) are given arrow types of the form \( T_1 \rightarrow T_2 \).
An expression type $Exp[T]$ is assigned to a program representation $'e$, if the underlying program $e$ has type $T$. The proof types $T_1 \leq T_2$ and proof type schemes $T_1 \hat{\leq} T_2$ are assigned to proof terms. A proof type $T_1 \leq T_2$ 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 $T_1$ and $T_2$ 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 $\theta$, 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 $\alpha$-equivalence, and establish associativity for composition of type constructors.
**Definition 5.** *Bound and Free Type Variables*

\[
\begin{align*}
BV(\alpha) &= \emptyset \\
BV(T_1 \rightarrow T_2) &= \emptyset \\
BV(\text{Exp}[T]) &= BV(T) \\
BV(\varphi) &= \emptyset \\
BV(\forall [\vec{\alpha}]) &= \vec{\alpha} \\
BV(\sigma) &= \emptyset \\
BV(\text{Subst}[\theta]) &= \emptyset \\
BV(\rho) &= \emptyset \\
BV(T_1 T_2) &= BV(T_1) \cup BV(T_2) \\
BV(T_1 \leq T_2) &= \emptyset \\
BV(T_1 \lessdot T_2) &= \emptyset \\
BV(T_1 \circ T_2) &= BV(T_1) \cup BV(T_2)
\end{align*}
\]

\[
\begin{align*}
FV(\alpha) &= \alpha \\
FV(T_1 \rightarrow T_2) &= FV(T_1) \cup FV(T_2) \\
FV(\text{Exp}[T]) &= FV(T) \\
FV(\varphi) &= \varphi \\
FV(\forall [\vec{\alpha}]) &= \emptyset \\
FV(\sigma) &= \sigma \\
FV(\text{Subst}[\theta]) &= FV(\theta) \\
FV(\rho) &= \rho \\
FV(T_1 T_2) &= (FV(T_2) - BV(T_1)) \cup FV(T_1) \\
FV(T_1 \leq T_2) &= FV(T_1) \cup FV(T_2) \\
FV(T_1 \lessdot T_2) &= FV(T_1) \cup FV(T_2) \\
FV(T_1 \circ T_2) &= (FV(T_2) - BV(T_1)) \cup FV(T_1)
\end{align*}
\]

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

**Definition 6.**

\[
\begin{align*}
\text{Kinds} \kappa ::= & \ast \mid \ast \rightarrow \ast \\
K-\varphi & \varphi ::= \ast \rightarrow \ast \\
K-\rho & \rho ::= \ast \rightarrow \ast
\end{align*}
\]
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 $\varphi, \sigma, \rho$ 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 $\Gamma$.

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 $\rho$ represents a substitution. We define a least upper bound between sorts $s_1 \sqcup s_2$, where $s_1 \sqcup s_2 = \text{Any}$ if $s_1 \neq s_2$. 

29
Definition 7.

\[(\text{sorts}) s ::= \text{Subst} \mid \text{Forall} \mid \text{Any} \]

\[
\begin{align*}
S-\varphi & \quad \varphi ::= \text{Forall} \\
S-\sigma & \quad \sigma ::= \text{Subst} \\
S-\rho & \quad \rho ::= \text{Any} \\
S-\forall & \quad \forall[\vec{\alpha}] ::= \text{Forall} \\
S-\text{Subst} & \quad \text{Subst}[\theta] ::= \text{Subst} \\
S-\text{Compose} & \quad T_1 ::= s_1 \quad T_2 ::= s_2 \\
& \quad T_1 \circ T_2 ::= (s_1 \sqcup s_2)
\end{align*}
\]

We define equivalence between types. This forms the mechanism for applying substitutions, and supports the types of several proof axioms. In particular: \textit{distExp} and \textit{factorExp} correspond with \(E_2\), and \textit{dist} for \textit{Subst} sorts and \textit{factor} correspond with \(E_3\).
**Definition 8. Type Equivalence**

(E1) \( \text{Subst}[\theta] \alpha \equiv T \) if \( \theta(\alpha) = T \)

(E2) \( \rho \text{Exp}[T] \equiv \text{Exp}[\rho T] \)

(E3) \( \sigma(T \rightarrow U) \equiv \sigma T \rightarrow \sigma U \)

(E4) \( \sigma(T_1 \leq T_2) \equiv (\sigma T_1) \leq (\sigma T_2) \)

(E5) \( \sigma(T_1 \hat{\leq} T_2) \equiv (\sigma T_1) \hat{\leq} (\sigma T_2) \)

(E6) \( \sigma \circ \forall[\vec{\alpha}] \equiv \forall[\vec{\beta}] \circ \sigma \circ \text{Subst}[\vec{\beta}/\vec{\alpha}] \)

where \( \vec{\beta} \) are fresh.

(E7) \( \forall[\vec{\alpha}] \equiv \forall[\vec{\beta}] \circ \text{Subst}[\vec{\beta}/\vec{\alpha}] \)

(E8) \( \text{Subst}[\theta](\alpha T) \equiv U(\text{Subst}[\theta]T) \)

if \( \theta(\alpha) = U \)

(E9) \( \forall[\alpha](T \leq U) \equiv T \leq \forall[\alpha]U \) if \( \alpha \notin \text{FV}(T) \)

(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) \( \text{Subst}[\theta_1 \circ \theta_2] \equiv \text{Subst}[\theta_1] \circ \text{Subst}[\theta_2] \)

Our type system extends Mitchell’s F-\( \eta \) subtyping for our type syntax.

**Definition 9. Subtyping:**

\[
\text{Sub-Refl} \quad \frac{T \subseteq T'}{T \subseteq T}
\]

\[
\text{Sub-Trans} \quad \frac{T \subseteq U \quad U \subseteq V}{T \subseteq V}
\]

\[
\text{Sub→} \quad \frac{T'' \subseteq T \quad U \subseteq U'}{T \rightarrow U \subseteq T' \rightarrow U'}
\]

\[
\text{Sub-Dist→} \quad \frac{\forall \vec{\alpha}.T \rightarrow U \subseteq (\forall \vec{\alpha}.T) \rightarrow (\forall \vec{\alpha}.U)}{}
\]

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

Sub-Exp \[
T \subseteq U \\
\text{Exp}[T] \subseteq \text{Exp}[U]
\]

Sub-Dist-Exp \[
\varphi \text{Exp}[T] \subseteq \text{Exp}[\varphi T]
\]

Sub-Dist-\preceq \[
\varphi(T \preceq U) \subseteq \varphi T \preceq \varphi U
\]

Sub-Dist-\preceq \[
\varphi(T \preceq U) \subseteq \varphi T \preceq \varphi U
\]

Sub-Proof-Inst \[
(\rho_1 \preceq \rho_2) \subseteq (\rho_1 T \preceq \rho_2 T)
\]

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-\preceq, and Sub-Dist-\preceq distributes type constructors of sort Forall into expression types, proof types, and proof type schemes, respectively.

Well-formedness of $\Gamma$ ensures that the types of all proof constants $p$ contained in $\Gamma$ 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 $I$s* and $e$Arrow.

**Definition 10.** Well-formedness of type environment $\Gamma$.

\[
\vdash \Gamma, (p : \forall \vec{\alpha}.(T \preceq U)) \quad T \subseteq U
\]

\[
\vdash \Gamma, (p : \forall \vec{\alpha}.(\sigma_1 \preceq \sigma_2)) \quad \sigma_1 T \subseteq \sigma_2 T \text{ for any } T
\]

\[
\vdash \Gamma, (x : T)
\]
Operators
\[ Ty[S] = \forall T, U, V. (T \rightarrow U \rightarrow V) \rightarrow (T \rightarrow U) \rightarrow T \rightarrow V \]
\[ Ty[K] = \forall T, U. T \rightarrow U \rightarrow T \]
\[ Ty[I] = \forall T. T \rightarrow T \]
\[ Ty[Y] = \forall T. (T \rightarrow T) \rightarrow T \]
\[ Ty[Q] = \forall T. T \rightarrow \text{Exp}[T] \]
\[ Ty[A] = \forall T, U. \text{Exp}[T \rightarrow U] \rightarrow \text{Exp}[T] \rightarrow \text{Exp}[U] \]
\[ Ty[G] = \forall T, U. (T \rightarrow U) \rightarrow (\forall V. \text{Exp}[V \rightarrow T] \rightarrow \text{Exp}[V] \rightarrow U) \rightarrow \text{Exp}[T] \rightarrow U \]
\[ Ty[\text{coerce}] = \forall T, U. T \rightarrow (T \leq U) \rightarrow U \]
\[ Ty[\text{eArrow}] = \forall \rho, T, U, T', U', V. (\rho(T \rightarrow U) \leq \rho(T' \rightarrow U')) \rightarrow \]
\[(\forall \varphi_1, \sigma_1. (\rho \leq (\varphi \circ \sigma \circ \rho)) \rightarrow (T' \leq \varphi_1 \sigma T) \rightarrow (\varphi_1 \sigma U \leq U') \rightarrow V) \rightarrow V \]
\[ Ty[\text{Is}[O]] = \forall T, U. T \rightarrow ((Ty[O] \leq T) \rightarrow Ty[O] \rightarrow U) \rightarrow U \rightarrow U \]
\[ Ty[\text{IsIs}] = \forall T, U. T \rightarrow ((\text{IsTy} \leq T) \rightarrow \text{IsTy} \rightarrow U) \rightarrow U \rightarrow U \]
where \text{IsTy} = \forall T, U, V. T \rightarrow ((V \leq T) \rightarrow V \rightarrow U) \rightarrow U \rightarrow U

Proof Axioms
\[ Ty[\text{refl}] = \forall T. (T \leq T) \]
\[ Ty[\text{dist}] = \forall \rho, T, U. (\rho(T \rightarrow U) \leq \rho T \rightarrow \rho U) \]
\[ Ty[\text{distExp}] = \forall \rho, T. (\rho \text{Exp}[T] \leq \rho \text{Exp}[\rho T]) \]
\[ Ty[\text{factor}] = \forall \sigma, T, U. (\sigma T \rightarrow \sigma U \leq \sigma (T \rightarrow U)) \]
\[ Ty[\text{factorExp}] = \forall \rho, T. (\rho \text{Exp}[\rho T] \leq \rho \text{Exp}[T]) \]

Proof Constructors
\[ Ty[\text{iArrow}] = \forall T, U, T', U'. (T' \leq T) \rightarrow (U \leq U') \rightarrow (T \rightarrow U \leq T' \rightarrow U') \]
\[ Ty[\text{iExp}] = \forall T, U. (T \leq U) \rightarrow (\text{Exp}[T] \leq \text{Exp}[U]) \]
\[ Ty[\text{eExp}] = \forall T, U. (\text{Exp}[T] \leq \text{Exp}[U]) \rightarrow (T \leq U) \]
\[ Ty[\text{congr}] = \forall \rho, T, U. (T \leq U) \rightarrow (\rho T \leq \rho U) \]
\[ Ty[\text{sCongr}] = \forall \rho, T, U. (T \leq U) \rightarrow (\rho T \leq \rho U) \]
\[ Ty[\text{trans}] = \forall T, U, V. (T \leq U) \rightarrow (U \leq V) \rightarrow (T \leq V) \]
\[ Ty[\text{sTrans}] = \forall \rho_1, \rho_2, \rho_3, \rho_4. (\rho_1 \leq \rho_2 \circ \rho_3) \rightarrow (\rho_2 \leq \rho_4) \rightarrow (\rho_1 \leq \rho_4 \circ \rho_3) \]

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 $\ast$. As mentioned previously, our syntactic kind rules for type variables allow us to check $U :: \ast$ without a kind context.

\textbf{Definition 11. Type Rules}

\begin{align*}
\text{Type-Atom} & \quad \Gamma \vdash atom : Ty[atom] \\
\text{Type-Var} & \quad x : T \in \Gamma \quad \Gamma \vdash x : T \\
\text{Type-Proof-Constant} & \quad p : T \in \Gamma \quad \Gamma \vdash p : T \\
\text{Type-App} & \quad \Gamma \vdash e_1 : T \rightarrow U \quad \Gamma \vdash e_2 : T \\ & \quad \Rightarrow \quad \Gamma \vdash e_1 e_2 : U \\
\text{Type-Subtype} & \quad \Gamma \vdash e : T \quad T \subseteq U \quad U :: \ast \\ & \quad \Rightarrow \quad \Gamma \vdash e : U
\end{align*}

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:

- $\lambda$-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 $\lambda$-abstraction can be defined by $SKI$-terms (e.g. [9]). The definition of $\lambda x.e$ is as follows.

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

**Lemma 1.** For all terms $e_1$ and $e_2$ and variable $x$ there is a reduction $(\lambda x.e_1) e_2 \rightarrow^* [u/x] e$.

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

$$
\begin{array}{c}
\Gamma, x : T \vdash e : U \\
\hline
\Gamma \vdash \lambda x.e : T \rightarrow U
\end{array}
$$

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

$$
\begin{array}{c}
\Gamma, x : T \vdash e : U \\
\hline
\Gamma \vdash \lambda x.e : \forall \vec{\alpha}.T \rightarrow U \quad \vec{\alpha} \not\in FV(\Gamma)
\end{array}
$$

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

$$
\begin{array}{c}
\Gamma \vdash e_1 : T_1 \\
\Gamma, x : T_1 \vdash e_2 : T_2 \\
\hline
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \forall \vec{\alpha}.T_2 \\
\vec{\alpha} \not\in FV(\Gamma)
\end{array}
$$

$$
\begin{array}{c}
\Gamma, x : T \vdash e_1 : T \\
\hline
\Gamma \vdash \text{let rec } x = e_1 : \forall \vec{\alpha}.T \\
\vec{\alpha} \not\in FV(\Gamma)
\end{array}
$$

**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 $\alpha \not\in FV(T)$, then $T \subseteq \forall[\alpha]U$.

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

$$
\begin{array}{c}
\Gamma \vdash e : T \\
\hline
\Gamma \vdash e : \forall[\alpha]T \quad \vec{\alpha} \cap FV(\Gamma) = \emptyset
\end{array}
$$

35
Lemma 8. Any type $T$ of the form $\sigma(T_1 \rightarrow T_2)$, where $\sigma$ is closed (closed in this context means each $\sigma$ in $\sigma$ is either $\forall[\vec{\beta}]$ or $\text{Subst}[\theta]$, i.e. not a type variable), there exists $\vec{\alpha}$, $\theta$ such that $T \equiv \forall[\vec{\alpha}].(\text{Subst}[\theta]T_1) \rightarrow (\text{Subst}[\theta]T_2)$.

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

Lemma 10. If $\Gamma \vdash e \ e_1 : T$, then there exists a type $T_1$ such that $\Gamma \vdash e : T_1 \rightarrow T$ 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: $\Gamma \vdash e : T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T$, and $\Gamma \vdash e_i : T_i$ for $i \in [1, n]$.

Lemma 12. If $\forall[\vec{\alpha}]((T \rightarrow U) \subseteq \forall[\vec{\beta}](T' \rightarrow U'))$, there exist a substitution $\theta$ and quantifiers $\vec{\gamma}$ such that: $\text{dom}(\theta) = \vec{\alpha}$, $\forall[\vec{\gamma}]\text{Subst}[\theta]T_i$ for $i \in [1, n]$, and $\forall[\vec{\beta}]\text{Subst}[\theta]T \subseteq U'$.

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

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

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

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

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

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

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

Lemma 20. If $\Gamma \vdash T :: \text{Subst}$, then $(T U \rightarrow TV) \equiv T(U \rightarrow V)$.

Lemma 21. If $\text{Exp}[A] \subseteq \text{Exp}[B]$, then $A \subseteq B$.

Lemma 22. If $\Gamma \vdash \Gamma \vdash v : (\sigma_1 \leq \sigma_2)$ 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 $\Gamma \vdash v : T \leq U$ and $v$ is a value, then $T \subseteq U$. 

36
Theorem 24. Preservation.
If $\Gamma \vdash e : T$ and $e \rightarrow e'$, then there exists $\Gamma' \supseteq \Gamma$ such that $\Gamma' \vdash e' : T$.

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

If $\Gamma \vdash e : T$, then either $e$ is a value, or there exists $e'$ such that $e \rightarrow e'$.

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

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 $\text{Subst}[\theta]$.
- Subtyping: Like in System F, we allow only two forms of subtyping, namely substitution, $\forall \vec{\alpha}.T \subseteq \sigma T$, and Mitchell’s $\text{Abs}_\forall$ rule.

The net effect is that our subset is closely related to System F plus an extra kind ($\ast \rightarrow \ast$) 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 $\text{Abs}_\forall$. 

37
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 $Is[Is[O]]$, $Is[Is[Is[O]]]$, $...$ 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 $enact1$, $enact2$, and $enact3$ match operators with arity 1, 2, and 3 respectively. These in turn dispatch to $enact[O]$ functions, for example $enactK$ and $enactS$ 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 $enactIs$. 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 $IsTy \leq T$. As shown in Figure 6, $IsTy$ is an abbreviation for:

$$\forall T,U,V.T \to ((V \leq T) \to V \to U) \to U \to U$$
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 \( n < \text{arity}(O) \) and \( e_i \) are representations of arbitrary expressions. The semantics of \textit{coerce} and \textit{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 \textit{coerce} or \textit{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 \textit{enactStrict} function defined within \textit{enact}. After fully evaluating a proof representation, it is unquoted to obtain the underlying proof. This is then used to implement \textit{coerce} and \textit{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 \( e = S(S(K(SK)))I \) and notice that \( e \) can be optimized to \( I \).

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

\[
\text{unquote}(\text{optimize}'\text{enact})'((\text{unquote}(\text{optimize}'\text{optimize})))'e)
\]

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 \( SK \rightarrowKI \). 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 $\lambda$-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 $T \leq U$ 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 $T \leq U$, 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 $T \leq U$, then $T \subseteq U$.

**Explicit substitutions.** Abadi et al. defined a $\lambda$-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 $\text{Subst}[\theta]$ are a form of explicit substitutions at the type level. In our case, a substitution has kind $\ast \rightarrow \ast$, 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 ($\ast \rightarrow \ast$) and $\ast$. 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 $(\lambda x.e_1)e_2 = I e_2 \rightarrow^* e_2 = [e_2/x]e_1$. If $e_1$ avoids $x$ then $(\lambda x.e_1)e_2 = K e_1 e_2 \rightarrow e_1 = [e_2/x]e_1$. Otherwise, if $e_1$ is of the form $e_3 e_4$ then

$$
(\lambda.e_1)e_2 = S(\lambda x.e_3)(\lambda x.e_4)e_2
$$

$$
\rightarrow (\lambda x.e_3)e_2((\lambda x.e_4)e_2)
$$

$$
\rightarrow^* [e_2/x]e_3([u/x]e_4)
$$

$$
= [e_2/x]e_1
$$

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-→ 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
Γ ⊢ λx.e₁ : U → T₂ → T and Γ ⊢ λx.e₂ : U → T₂ whence λx.t = S(λx.e₁)(λx.e₂) has type U → T as required.

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


Proof of Lemma 5. By straightforward induction on the derivation of T ⊆ U.

Proof of Lemma 6. By induction on the structure of T ⊆ U.

If α ∉ FV(U), then α is a redundant quantifier, and the result holds by Sub-Subst. Therefore, assume α ∈ FV(U).

Case T ⊆ U derived by Sub→. We have T = T₁ → T₂, U = U₁ → U₂, U₁ ⊆ T₁, and T₂ ⊆ U₂. Since α ∉ FV(T₁), lemma 5 states α ∉ FV(U₁). Therefore, α ∈ FV(U₂). By induction, T₂ ⊆ ∀[α]U₂. Now α is redundant in U₁ → ∀[α]U₂, so Sub-Subst derives U₁ → ∀[α]U₂ ⊆ ∀[α](U₁ → ∀[α]U₂), and a combination of Sub-Congr, Sub-Arrow, and Sub-Subst derives ∀[α](U₁ → ∀[α]U₂) ⊆ ∀[α](U₁ → U₂). The result follows from Sub-Trans.

The remaining cases are straightforward.

Proof of Lemma 7. By induction on the structure of Γ ⊢ e : T.

Case Γ ⊢ 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 ⊆ ∀[α]T.

Case Γ ⊢ e : T derived by rule Type-Subtype. We have Γ ⊢ e : T' and T' ⊆ T. If α ∈ FV(T'), then the induction hypothesis gives Γ ⊢ e : ∀[α]T'. Now Sub-Congr derives ∀[α]T' ⊆ ∀[α]T as required. If α ∉ FV(T'), then T' ⊆ ∀[α]T by lemma 6.

Proof of Lemma 8. Straightforward.

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 $\Gamma \vdash e_1 : T$.

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

Case $\Gamma \vdash e_1 : T$ derived by Type-Subtype: We have $\Gamma \vdash e_1 : T'$ and $T' \subseteq T$. By induction, there exists a type $T_1$ such that $\Gamma \vdash e_1 : T_1 \rightarrow T'$ and $\Gamma \vdash e_1 : T_1$. Now Sub-$\rightarrow$ derives $T_1 \rightarrow T' \subseteq T_1 \rightarrow T$ from Sub-Refl and $T' \subseteq T$. By Type-Subtype, $\Gamma \vdash e : T_1 \rightarrow T$ 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 $\Gamma \vdash e_1 \ldots e_{n-1} : T_n \rightarrow T$ and $\Gamma \vdash e_n : T_n$. By induction, there exist types $T_1, \ldots, T_{n-1}$ such that $\Gamma \vdash e : T_1 \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T$ 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 $\forall[\vec{\alpha}](T \rightarrow U) \subseteq \forall[\vec{\beta}](T' \rightarrow U')$.

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

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

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

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


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

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

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

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

Case Sub-Subst:

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

Case Sub-Dist-$\leq$:
Holds, with \( \bar{\alpha} = \bar{\alpha}', \bar{\beta}, \) and \( \theta = [\cdot] \).

Case Sub-Trans:

There exist quantifiers \( \bar{\alpha}'' \) and types \( \sigma_1'', \sigma_2'' \) such that \( \forall[\bar{\alpha}](\sigma_1 \lesssim \sigma_2) \subseteq \forall[\bar{\alpha}''] \) and \( \forall[\bar{\alpha}'](\sigma_1' \lesssim \sigma_2') \subseteq \forall[\bar{\alpha}'](\sigma_1' \lesssim \sigma_2') \). By induction, there exist quantifiers \( \bar{\beta}_1, \bar{\beta}_2 \) and substitutions \( \theta_1, \theta_2 \) such that: \( \sigma_1' = \forall[\bar{\beta}_1] \circ \text{Subst}[\theta_1] \circ \sigma_1, \sigma_2' = \forall[\bar{\beta}_1] \circ \text{Subst}[\theta_1] \circ \sigma_2, \sigma_1' = \forall[\bar{\beta}_2] \circ \text{Subst}[\theta_2] \circ \sigma_1', \) and \( \sigma_2' = \forall[\bar{\beta}_2] \circ \text{Subst}[\theta_2] \circ \sigma_2'' \). Therefore, \( \sigma_1' = \forall[\bar{\beta}_2] \circ \text{Subst}[\theta_2] \circ \forall[\bar{\beta}_1] \circ \text{Subst}[\theta_1] \circ \sigma_1 \) and \( \sigma_2' = \forall[\bar{\beta}_2] \circ \text{Subst}[\theta_2] \circ \forall[\bar{\beta}_1] \circ \text{Subst}[\theta_1] \circ \sigma_2 \).

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

Proof of Lemma 16. By lemma 15, there exist quantifiers \( \bar{\gamma} \) and a substitution \( \theta \) such that \( \sigma_1' = \forall[\bar{\gamma}] \circ \text{Subst}[\theta] \circ \sigma_1 \) and \( \sigma_2' = \forall[\bar{\gamma}] \circ \text{Subst}[\theta] \circ \sigma_2 \). 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 \( \bar{\gamma} \) and a substitution \( \theta \) such that \( T' = \forall[\bar{\gamma}] \text{Subst}[\theta]T \) and \( U' = \forall[\bar{\gamma}] \text{Subst}[\theta]U \). The result follows from 14.

Proof of Lemma 19. By induction on the derivation of \( \forall[\bar{\alpha}](\sigma_1 \lesssim \sigma_2) \subseteq \forall[\bar{\beta}](T \lesssim U) \).

Case Sub-Congr: Holds by the induction hypothesis.

Case Sub-Proof-Inst: Holds with \( \bar{\gamma} = \emptyset \) and \( \theta = [\cdot] \).

Case Trans: We have \( \forall[\bar{\alpha}](\sigma_1 \lesssim \sigma_2) \subseteq A \) and \( A \subseteq \forall[\bar{\beta}](T \lesssim U) \).

If \( A = \forall[\bar{\alpha}'](\sigma_1' \lesssim \sigma_2') \), then by lemma 15, there exist quantifiers \( \bar{\gamma}_1 \) and a substitution \( \theta_1 \) such that \( \sigma_1' = \forall[\bar{\gamma}_1] \circ \text{Subst}[\theta_1] \circ \sigma_1 \) and \( \sigma_2' = \forall[\bar{\gamma}_1] \circ \text{Subst}[\theta_1] \circ \sigma_2 \)
Now by the induction hypothesis, there exist quantifiers $\vec{\gamma}_2$, a substitution $\theta_2$, and a type $V$ such that $T = \forall[\vec{\gamma}_2]\text{Subst}[\theta_2]\forall[\vec{\gamma}_1]\text{Subst}[\theta_1]\sigma_1 V$ and $U = \forall[\vec{\gamma}_2]\text{Subst}[\theta_2] \forall[\vec{\gamma}_1] \text{Subst}[\theta_1] \sigma_2 V$. Now let $\vec{\gamma} = \vec{\gamma}_2, \vec{\gamma}_1$, and $\theta = \theta_2 \circ [\vec{\gamma}_1/\vec{\gamma}_2] \circ \theta_1$. Now $T \equiv \forall[\vec{\gamma}]\text{Subst}[\theta]V$ and $U \equiv \forall[\vec{\gamma}]\text{Subst}[\theta]V$.

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

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

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

Case S-Subst: $T = \text{Subst}[\theta]$. Holds by equivalence rule

\[
\text{Subst}[\theta](T_1 \leq T_2) \equiv (\text{Subst}[\theta]T_1) \leq (\text{Subst}[\theta]T_2).
\]

Case S-Compose: We have $T = T_1 \circ T_2$, $\Gamma \vdash T_1 :: s_1$, $\Gamma \vdash T_2 :: s_2$, and $s_1 \sqcup s_2 = \text{Subst}$. Therefore, $s_1 = \text{Subst}$ and $s_2 = \text{Subst}$. Now $\theta T \rightarrow \theta U \equiv T_1 T_2 T \rightarrow T_1 T_2 U$.

By induction, $T_1 T_2 T \rightarrow T_1 T_2 U \equiv T_1(T_2 T \rightarrow T_2 U) \equiv T_1 T_2(T \rightarrow U) \equiv \theta(T \rightarrow U)$ as required.

**Proof of Lemma 21.** By straightforward induction.

**Proof of Lemma 22.** By induction on $v$.

Case $v = pf_i$: We have $\Gamma \vdash pf_i : \forall[\vec{\alpha}](\sigma_1 \leq \sigma_2)$ and $\forall[\vec{\alpha}](\sigma'_1 \leq \sigma'_2) \subseteq (\sigma_1 \leq \sigma_2)$.

$\vdash \Gamma$ ensures $\sigma'_1 T \subseteq \sigma'_2 T$ for any $T$. Result follows from lemma 16.

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

$$V_1 \subseteq \forall [\vec{\alpha}](\sigma_3 \leq \sigma_4 \circ \sigma_5)$$

$$V_2 \subseteq \forall [\vec{\alpha}](\sigma_4 \leq \sigma_6)$$

$$\forall [\vec{\alpha}](\sigma_3 \leq \sigma_6 \circ \sigma_5) \subseteq (\sigma_1 \leq \sigma_2)$$

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.


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

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

49
In the second case, \( T = (\sigma'_1 A \text{ and } U = \sigma'_2 A \text{ for some } \sigma'_1, \sigma'_2, A). \) By lemma 22, \( T \subseteq U \) as required.

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

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

Case \( v = \text{distExp} \):

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

Case \( v = \text{factor} \):

By lemma 9, \( \forall [\theta, X, Y](\theta X \to \theta Y \leq \theta(X \to Y)) \subseteq (T \leq U). \) Since \( \Gamma \vdash \theta :::: \text{Subst}, \) \( \Gamma \vdash \theta X \to \theta Y \equiv \theta(X \to Y), \) 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 \( v = \text{factorExp} \):

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

Case \( v = \text{iArrow } v_1 \ v_2 \ v_3 \)

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

\[
\begin{align*}
&\sigma_1 (X \to Y) \\
&\subseteq Q_2 Q_3 \theta (X \to Y) \\
&\subseteq Q_2 (Q_3 \theta X \to Q_3 \theta Y) \text{ By Sub-Congr, Sub-Dist-} \\
&\subseteq Q_2 (X' \to Y') \text{ By Sub-Congr, Sub-} \\
\end{align*}
\]

By lemma 17, there exist a substitution \( \theta_1 \) and quantifiers \( \vec{\beta} \) such that \( T = \forall[\vec{\beta}] \text{Subst}[\theta] \sigma_1 (X \to Y) \) and \( U = \forall[\vec{\beta}] \text{Subst}[\theta] Q_2 (X' \to Y') \). By lemma 14, \( T \subseteq U \) as required.

Case \( v = iExp v_1 \)

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

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

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 $v = cong v_1$

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

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, $\Gamma \vdash trans : V_1 \rightarrow V_2 \rightarrow S$, $\Gamma \vdash v_1 : V_1$, $\Gamma \vdash v_2 : V_2$, and $S \subseteq T \preceq U$. By lemma 9, $Ty[trans] \subseteq S$. Therefore, there exist quantifiers $\vec{\alpha}$ and types $X, Y, Z$ such that $V_1 \subseteq \forall[\vec{\alpha}](X \preceq Y)$, $V_2 \subseteq \forall[\vec{\alpha}](Y \preceq Z)$, and $\forall[\vec{\alpha}](X \preceq Z) \subseteq (T \preceq U)$. Now $\Gamma \vdash v_1 : \forall[\vec{\alpha}](X \preceq Y)$ and $\Gamma \vdash v_2 : \forall[\vec{\alpha}](Y \preceq Z)$, 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 $e \rightarrow e'$.

Case $G f g (Q O) \rightarrow f O$: By lemma 11, there exist types $T_1, T_2, T_3$ such that $\Gamma \vdash G : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$, $\Gamma \vdash f : T_1$, $\Gamma \vdash g : T_2$, and $\Gamma \vdash QO : T_3$. By lemma 9, $\forall[A, B]((A \rightarrow B) \rightarrow \forall[C](Exp[C] \rightarrow A) \rightarrow Exp[C] \rightarrow B) \rightarrow Exp[A] \rightarrow B) \subseteq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and types $A, B$ such that $T_1 \subseteq \forall[\vec{\alpha}](A \rightarrow B)$, $T_2 \subseteq \forall[\vec{\alpha}][C](Exp[C] \rightarrow A) \rightarrow Exp[C] \rightarrow B)$,
$T_3 \subseteq \forall[\vec{\alpha}]Exp[A]$, and $\forall[\vec{\alpha}]B \subseteq T$. By 10, there exists a type $T_4$ such that $\Gamma \vdash Q : T_4 \rightarrow T_3$ and $\Gamma \vdash O : T_4$. By 9, $\forall[X](X \rightarrow Exp[X]) \subseteq T_4 \rightarrow T_3$. By 12, there exist quantifiers $\vec{\beta}$ and a type $X$ such that $T_3 \subseteq \forall[\vec{\beta}]X$ 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→, $\forall[\vec{\alpha}](A \rightarrow B) \subseteq \forall[\vec{\alpha}]A \rightarrow \forall[\vec{\alpha}]B$, so Type-Subtype derives $\Gamma \vdash f : \forall[\vec{\alpha}]A \rightarrow \forall[\vec{\alpha}]B$. Therefore $\Gamma \vdash f O : \forall[\vec{\beta}]B$, and Type-Subtype derives $\Gamma \vdash f O : T$ as required.

Case $G f g (A p q) \rightarrow g p q$: By lemma 11, there exist types $T_1, T_2, T_3$ such that $\Gamma \vdash G : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$, $\Gamma \vdash f : T_1$, $\Gamma \vdash g : T_2$, and $\Gamma \vdash QO : T_3$. By lemma 9, $\forall[A, B][(A \rightarrow B) \rightarrow \forall[C](Exp[C \rightarrow A] \rightarrow Exp[C] \rightarrow B) \rightarrow Exp[A] \rightarrow B) \subseteq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and types $A, B$ such that $T_1 \subseteq \forall[\vec{\alpha}](A \rightarrow B)$, $T_2 \subseteq \forall[\vec{\alpha}]\forall[C](Exp[C \rightarrow A] \rightarrow Exp[C] \rightarrow B)$, $T_3 \subseteq \forall[\vec{\alpha}]Exp[A]$, and $\forall[\vec{\alpha}]B \subseteq T$. By lemma 11, there exist types $T_4, T_5$ such that $\Gamma \vdash A : T_4 \rightarrow T_5 \rightarrow T_3$, $\Gamma \vdash p : T_4$, and $\Gamma \vdash q : T_5$. By lemma 9, $\forall[X, Y](Exp[X \rightarrow Y] \rightarrow Exp[X] \rightarrow Exp[Y] \subseteq T_4 \rightarrow T_5 \rightarrow T_3$. By lemma 13, there exist quantifiers $\vec{\beta}$ and types $X, Y$ such that $T_4 \subseteq \forall[\vec{\beta}]Exp[X \rightarrow Y]$, $T_5 \subseteq \forall[\vec{\beta}]Exp[X]$, 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 $\forall[\vec{\beta}]Exp[X \rightarrow Y] \subseteq Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]A]$, so Type-Subtype derives $\Gamma \vdash p : Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]A]$. Without loss of generality, assume $\vec{\alpha} \notin FV(\forall[\vec{\beta}]X)$. Now $\forall[\vec{\alpha}]\forall[C](Exp[C \rightarrow A] \rightarrow Exp[C] \rightarrow B) \subseteq Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]A] \rightarrow Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]B]$. Type-Subtype derives $\Gamma \vdash g : Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]A] \rightarrow Exp[\forall[\vec{\beta}]X \rightarrow \forall[\vec{\alpha}]B]$, 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 → t p O: By lemma 11, there exist types $T_1, T_2, T_3$ such
that: $\Gamma \vdash Is[O] : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$, $\Gamma \vdash O : T_1$, $\Gamma \vdash t : T_2$, and $\Gamma \vdash f : T_3$. By lemma 9, $Ty[Is[O]] \subseteq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T$. By lemma 13, there exist quantifiers $\vec{\alpha}$ and a substitution $\theta$ such that: $T_1 \subseteq \forall[\vec{\alpha}]\text{Subst}[\theta]X$, $T_2 \subseteq \forall[\vec{\alpha}]\text{Subst}[\theta]((Ty[O] \leq X) \rightarrow Ty[O] \rightarrow Y)$, and $\forall[\vec{\alpha}]\text{Subst}[\theta]Y \subseteq T$. Note that since $Ty[O]$ is closed, $\text{Subst}[\theta]((Ty[O] \leq X) \rightarrow Ty[O] \rightarrow Y) \equiv (Ty[O] \leq \text{Subst}[\theta]X) \rightarrow Ty[O] \rightarrow \text{Subst}[\theta]Y$. Again by lemma 9, $Ty[O] \subseteq T_1 \subseteq \forall[\vec{\alpha}]\text{Subst}[\theta]X$. Therefore, let $\Gamma' = (\Gamma, pf_1 : Ty[O] \leq \forall[\vec{\alpha}]\text{Subst}[\theta]X)$. Note that since $\vec{\alpha} \cap \text{FV}(Ty[O]) = \emptyset$, $Ty[O] \leq \forall[\vec{\alpha}]\text{Subst}[\theta]X \equiv \forall[\vec{\alpha}](Ty[O] \leq \text{Subst}[\theta]X)$. By two distributions of $\forall[\vec{\alpha}]$, $T_2 \subseteq (Ty[O] \leq \forall[\vec{\alpha}]\text{Subst}[\theta]X) \rightarrow Ty[O] \rightarrow (\forall[\vec{\alpha}]\text{Subst}[\theta]Y)$. Now $\Gamma' \vdash t pf_1 O : \forall[\vec{\alpha}]\text{Subst}[\theta]Y$, and the result follows from $\forall[\vec{\alpha}]\text{Subst}[\theta]Y \subseteq T$.

Case eArrow $p e \rightarrow e p_1 p_2 p_3$: By lemma 11, there exist types $T_1$ and $T_2$ such that $\Gamma \vdash e\text{Arrow} : T_1 \rightarrow T_2 \rightarrow T$, $\Gamma \vdash p : T_1$, and $\Gamma \vdash e : T_2$. By lemma 9, $Ty[e\text{Arrow}] \subseteq T_1 \rightarrow T_2 \rightarrow T$. By lemma 13, there exist quantifiers $\vec{\alpha}, \vec{\beta}, \vec{\gamma}$ and types $A, B, A', B', C$ such that: $T_1 \subseteq \forall[\vec{\gamma}]((\forall[\vec{\alpha}](A \rightarrow B) \leq \forall[\vec{\beta}](A' \rightarrow B'))$, $T_2 \subseteq \forall[\vec{\gamma}](\forall Q, \theta. (\forall[\vec{\alpha}] \leq \forall[\vec{\beta}] \circ Q \circ \theta) \rightarrow (A' \leq Q\theta A) \rightarrow (Q\theta B \leq B') \rightarrow C)$, and $\forall[\vec{\gamma}]C \subseteq T$.

By lemma 23, $\forall[\vec{\alpha}](A \rightarrow B) \subseteq \forall[\vec{\beta}](A' \rightarrow B')$. Now by lemma 12, there exist fresh quantifiers $\vec{\delta}$ and a substitution $\theta_1$ such that $\text{dom}(\theta_1) = \vec{\alpha}$, $A' \subseteq \forall[\vec{\delta}]\text{Subst}[\theta_1]A$ and $\forall[\vec{\delta}]\text{Subst}[\theta_1]B \subseteq B'$. Now let $\Gamma' = (\Gamma, p_{v_1} : \forall[\vec{\gamma}]((\forall[\vec{\alpha}] \leq \forall[\vec{\beta}] \circ \forall[\vec{\delta}] \circ \text{Subst}[\theta_1])$, $p_{v_2} : \forall[\vec{\gamma}](A' \leq \forall[\vec{\delta}]\text{Subst}[\theta_1]A)$, $p_{v_3} : \forall[\vec{\gamma}](\forall[\vec{\delta}]\text{Subst}[\theta_1]B \leq B'))$. Note that $\vdash \Gamma'$ holds.

By instantiating $Q$ to $\forall[\vec{\delta}]$ and $\theta$ to $\text{Subst}[\theta_1]$ and distributing $\forall[\vec{\gamma}]$ three times, we can derive:
∀[γ]((∀[α] ≤ ∀[β] ○ Q ○ θ) \rightarrow
  (A' ≤ QθA) \rightarrow
  (QθB ≤ B') \rightarrow
  C) \subseteq (∀[γ](∀[α] ≤ ∀[β] ○ Subst[θ_1])) \rightarrow
  (∀[γ](∀[α] Subst[θ_1]A)) \rightarrow
  (∀[γ][Subst[θ_1]B ≤ B']) \rightarrow
  (∀[γ]C)

Now since ∀[γ]C \subseteq T, we get Γ' \vdash e p_{11} p_{12} p_{13} : T as required.

Case coerce e v \rightarrow e: By lemma 11, we have Γ \vdash coerce : T_1 \rightarrow T_2 \rightarrow T, Γ \vdash e : T_1, and Γ \vdash v : T_2. By lemma 9, Ty[coerce] \subseteq T_1 \rightarrow T_2 \rightarrow T. Therefore, there exist quantifiers \vec{α} and a substitution θ such that T_1 \subseteq ∀[\vec{α}] Subst[θ]X, T_2 \subseteq ∀[\vec{α}] Subst[θ](X ≤ Y), and ∀[\vec{α}] Subst[θ]Y \subseteq T. Note that ∀[\vec{α}] Subst[θ](X ≤ Y) \equiv
∀[\vec{α}](Subst[θ]X ≤ Subst[θ]Y) \subseteq ∀[\vec{α}] Subst[θ]X ≤ ∀[\vec{α}] Subst[θ]Y. By lemma 23,
∀[\vec{α}] Subst[θ]X \subseteq ∀[\vec{α}] Subst[θ]Y. Therefore, Γ \vdash e : T as required.

Proof of Lemma 25. By contradiction. Assume Γ \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 Γ \vdash O : T_1 → \cdots → T_i → Exp[T]. By lemma 9, Ty[O] \subseteq T_1 → \cdots → T_i → Exp[T]. Since i < arity(O), there exist quantifiers \vec{α} and types U_1, U_2 such that ∀[\vec{α}](U_1 → U_2) \subseteq Exp[T], a contradiction.

In the second case, lemma 11 states there exist types T_1, \ldots, T_i such that Γ \vdash
$P : T_1 \rightarrow \cdots \rightarrow T_i \rightarrow Exp[T]$. By lemma 9, $Ty[P] \subseteq T_1 \rightarrow \cdots \rightarrow T_i \rightarrow Exp[T]$. Since $i = \text{arity}(P)$, there exist quantifiers $\vec{\alpha}$ and types $U_1, U_2$ such that either $\forall[\vec{\alpha}](U_1 \leq U_2) \subseteq Exp[T]$ or $\forall[\vec{\alpha}](U_1 \leq U_2) \subseteq Exp[T]$. Either case is a contradiction.


Case $e = G e_1 e_2 e_3$. If $e_3$ is not a value, then by induction $e'_3 \rightarrow e'_4$ and $e \rightarrow G e_1 e_2 e'_3$. Otherwise, by lemma 25, $e_3 = Q O$ or $e_3 = A e_4 e_5$. In the first case, $e \rightarrow e_1 O$. In the second, $e \rightarrow e_2 e_4 e_4$.

Case $e = O e_1 \ldots e_i$, where $i = \text{arity}(O)$, $O \neq G$. 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 $\eta$-reduction for our combinator calculus. $\eta$-reduction is defined for the $\lambda$ calculus as follows:
\( \lambda x.e \ x \rightarrow e, \text{ if } x \not\in \text{fv}(e) \)

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 \( e' \):

\[ S \ (K \ e') \ I \ x \rightarrow K \ e' \ x \ (I \ x) \rightarrow e' \ (I \ x) \equiv e' \ x \]

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

\[
\begin{align*}
\text{let } &\ (\text{composeOpt}4 : (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U)) \rightarrow \\
&\ (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) \rightarrow \\
&\ (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) \rightarrow \\
&\ (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) \rightarrow \\
&\ (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) \rightarrow \\
&\ (\forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U)) = \\
&\lambda(\text{opt1 : } \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U). \lambda(\text{opt2 : } \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U). \lambda(\text{opt3 : } \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U). \lambda(\text{opt4 : } \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U). \\
&\text{composeOpt opt1 (composeOpt opt2 (composeOpt opt3 opt4)) in} \\
&\text{let } (\text{matchAtom : } \forall T, U. U \rightarrow (T \rightarrow U) \rightarrow Exp[T] \rightarrow U) = \\
&\lambda(\text{if NotAtom : } U).\lambda(\text{if Atom : } T \rightarrow U).G \text{ if Atom } (K \ (K \text{ if NotAtom}))) \text{ in} \\
&\text{let } (\text{matchApp : } \forall T, U. U \rightarrow (\forall T_1. Exp[T_1] \rightarrow T) \rightarrow Exp[T_1] \rightarrow U) \rightarrow \\
&\quad Exp[T] \rightarrow U) = \\
&\lambda(\text{if NotApp : } U). \end{align*}
\]
\[
\lambda (\text{ifApp} : \forall T_1. \text{Exp}[T_1 \to T] \to \text{Exp}[T_1] \to U).
\]

\[G (K \text{ifNotApp}) \text{ifApp in}\]

let \((\text{matchS1} : \forall T,U.U \to (\forall T_1.(Ty[S] \preceq T_1 \to T) \to \text{Exp}[T_1] \to U) \to \text{Exp}[T] \to U) =\]

\[\lambda (\text{ifNotS1} : U).\lambda (\text{iS1} : (Ty[S] \preceq T_1 \to T).K (\text{iS1} p))(K \text{iNotS1})) \text{ in}\]

let \((\text{matchK0} : \forall T,U.U \to ((Ty[K] \preceq T) \to U) \to \text{Exp}[T] \to U) =\]

\[\lambda (\text{iK0} : (Ty[K] \preceq T)).\lambda (\text{K} \text{K0} p) \text{ifNotK0}) \text{ in}\]

let \((\text{trans2} : \forall T,U,V,W.(T \preceq U) \to (U \preceq V) \to (V \preceq W) \to (T \preceq W)) =\]

\[\lambda (\text{p1} : T \preceq U).\lambda (\text{p2} : U \preceq V).\lambda (\text{p3} : V \preceq W).\text{trans p1 (trans p2 p3) in}\]

let \((\text{dist2} : \forall \rho,T,U,V,\rho(T \to U \to V) \preceq \rho T \to \rho U \to \rho V) =\]

\[\text{trans dist(iArrow refl dist) in}\]

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

\[\lambda (\text{ifNotApp2} : U).\lambda (\text{App2} : \forall T_1,T_2.\text{Exp}[T_1 \to T_2 \to T] \to \text{Exp}[T_1] \to \text{Exp}[T_2] \to U).\text{matchApp ifNotApp2 (matchApp (K ifNotApp2) ifApp2) in}\]}
let (matchApp3 : \forall T, U. U \rightarrow
\forall T_1, T_2, T_3. \text{Exp}[T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T] \rightarrow
\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow U) \rightarrow
\text{Exp}[T] \rightarrow U) =
\lambda (\text{ifNotApp3} : U).
\lambda (\text{ifApp3} : \forall T_1, T_2, T_3. \text{Exp}[T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T] \rightarrow
\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow U).
matchApp3 \text{ ifNotApp3} (\text{matchApp2} (\text{K } i f N o t A p p 3 ) \text{ i fApp3}) \text{ in }

let (matchS2 : \forall T, U. U \rightarrow \forall T_1, T_2. (\text{Ty}[S] \preceq T_1 \rightarrow T_2 \rightarrow T) \rightarrow
\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow U) \rightarrow
\text{Exp}[T] \rightarrow U) =
\lambda (\text{ifNotS2} : U).
\lambda (\text{ifS2} : \forall T_1, T_2. (\text{Ty}[S] \preceq T_1 \rightarrow T_2 \rightarrow T) \rightarrow \text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow U).
matchApp2 \text{ ifNotS2} (\text{matchAtom} (\text{K } (\text{K } \text{ifNotS2})))
(\lambda (e_s : T_1 \rightarrow T_2 \rightarrow T).
\text{IsS } e_s (\lambda (p : \text{Ty}[S] \preceq T_1 \rightarrow T_2 \rightarrow T). \text{K} (\text{ifS2 } p)) (\text{K} (\text{K } \text{ifNotS2}))) \text{ in }

let (matchS3 : \forall T, U. U \rightarrow \forall T_1, T_2, T_3. (\text{Ty}[S] \preceq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T) \rightarrow
\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow U) \rightarrow
\text{Exp}[T] \rightarrow U) =
\lambda (\text{ifNotS3} : U).
\lambda (\text{ifS3} : \forall T_1, T_2, T_3. (\text{Ty}[S] \preceq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T) \rightarrow
\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow U).
matchApp3 \text{ ifNotS3} (\text{matchAtom} (\text{K} (\text{K} (\text{K } \text{ifNotS3})))
(\lambda (e_s : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T).
IsS e_s
(\lambda(p : T y[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T).
K \ (i f \ S \ 3 \ p)) \ (K \ (K \ i f \ N o t \ S \ 3)))) \ in

let \ (matchK1 : \forall T, U. U \rightarrow (\forall T_1 . (T y[K] \leq T_1 \rightarrow T) \rightarrow E x p[T_1] \rightarrow U) \rightarrow
E x p[T] \rightarrow U) \ =
\lambda(i f \ N o t K1 : U).
\lambda(i f K1 : \forall T_1 . (T y[K] \leq T_1 \rightarrow T) \rightarrow E x p[T_1] \rightarrow U).
matchApp \ i f \ N o t K1 \ (matchAtom \ (K \ i f \ N o t K1))
(\lambda(e_k : T_1 \rightarrow T).
IsK e_k \ (\lambda(p : T y[K] \leq T_1 \rightarrow T). \ K \ (i f K1 p)) \ (K \ (K \ i f \ N o t K1)))) \ in

let \ (matchK2 : \forall T, U. U \rightarrow
(\forall T_1, T_2 . (T y[K] \leq T_1 \rightarrow T_2 \rightarrow T) \rightarrow
E x p[T_1] \rightarrow E x p[T_2] \rightarrow U) \rightarrow
E x p[T] \rightarrow U) \ =
\lambda(i f \ N o t K2 : U).
\lambda(i f K2 : \forall T_1, T_2 . (T y[K] \leq T_1 \rightarrow T_2 \rightarrow T) \rightarrow E x p[T_1] \rightarrow E x p[T_2] \rightarrow U).
matchApp2 \ i f \ N o t K2 \ (matchAtom \ (K \ (K \ i f \ N o t K2))
(\lambda(e_s : T_1 \rightarrow T_2 \rightarrow T).
IsK e_s \ (\lambda(p : T y[K] \leq T_1 \rightarrow T_2 \rightarrow T). \ K \ (i f K2 p)) \ (K \ (K \ i f \ N o t K2)))) \ in

let \ (matchI0 : \forall T, U. U \rightarrow ((T y[I] \leq T) \rightarrow U) \rightarrow E x p[T] \rightarrow U) \ =
\lambda(i f \ N o t I0 : U). \lambda(i f I0 : (T y[I] \leq T) \rightarrow U).
matchAtom \ i f \ N o t I0
(\lambda(e_i : T).
\[ \text{IsI } e_i (\lambda(p : Ty[I] \leq T). \ K (ifI0 \ p)) \ \text{ifNotI0} \ \text{in} \]

let (analyzeEta : \forall T, U. U \rightarrow
\begin{align*}
(\forall T_1, T_2, T_3. \ Exp[T_3] \rightarrow (Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T) \rightarrow \\
(Ty[K] \leq T_3 \rightarrow T_1) \rightarrow (Ty[I] \leq T_2) \rightarrow U) \rightarrow \\
Exp[T] \rightarrow U) = \\
\lambda(\text{ifNotEta} : U).
\end{align*}

\[ \lambda(\text{ifEta} : \forall T_1, T_2, T_3. \ Exp[T_3] \rightarrow (Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T) \rightarrow \\
(Ty[K] \leq T_3 \rightarrow T_1) \rightarrow (Ty[I] \leq T_2) \rightarrow U). \]

matchS2 \text{ifNotEta}
\begin{align*}
(\lambda(pS : Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T). \ \lambda(e_1 : Exp[T_1]). \ \lambda(e_2 : Exp[T_2]). \\
matchK1 \text{ifNotEta}
\end{align*}
\begin{align*}
(\lambda(pK : Ty[K] \leq T_3 \rightarrow T_1). \ \lambda(e_3 : Exp[T_3]). \\
matchI0 \text{ifNotEta}
\end{align*}
\begin{align*}
(\lambda(pI : Ty[I] \leq T_2). \ \text{ifEta} \ e_3 \ pS \ pK \ pI \ e_2) \ e_1) \ \text{in} \\
\text{let (expandI} : (Ty[I] \leq \varphi(T \rightarrow U)) \rightarrow T \leq U) = \\
\lambda(pIsI : Ty[I] \leq \varphi(T \rightarrow U)).
\end{align*}
\[ \text{eArrow pIsI}
\begin{align*}
(\lambda(p_1 : \varphi_1 \leq \varphi \circ \varphi_2 \circ \sigma_1). \ \lambda(p_2 : T \leq \varphi_2 \sigma_1 X). \ \lambda(p_3 : \varphi_2 \sigma X \leq U). \\
\text{trans } p_2 \ p_3) \ \text{in} \\
\text{let (proveEta} : \forall T_1, T_2, T_3. (Ty[S] \leq T_2 \rightarrow T_1 \rightarrow T) \rightarrow (Ty[K] \leq T_3 \rightarrow T_2) \rightarrow \\
(Ty[I] \leq T_1) \rightarrow T_3 \leq T) = \\
\lambda(pS : Ty[S] \leq T_2 \rightarrow T_1 \rightarrow T). \\
\lambda(pK : Ty[K] \leq T_3 \rightarrow T_2). \\
\lambda(pI : Ty[I] \leq T_1). \]

62
eBinary $pS$

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

$\lambda(p_2 : T_2 \leq \varphi_2 \sigma_1(X_1 \rightarrow X_2 \rightarrow X_3))$.

$\lambda(p_3 : T_1 \leq \varphi_2 \sigma_1(X_1 \rightarrow X_2))$.

$\lambda(p_4 : \varphi_2 \sigma_1(X_1 \rightarrow X_3) \leq T)$.

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

let $(p_6 : \sigma_1 X_1 \leq \sigma_1 X_2) = \text{expand}I \ p_5$ in

eArrow $pK$

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

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

$\lambda(p_9 : \varphi_4 \sigma_2(V_2 \rightarrow V_1) \leq T_2)$.

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

trans2 $p_9$ $p_2$ (congr dist2) in

eArrow $p_{10}$

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

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

$\lambda(p_{13} : \varphi_5 \sigma_3 V_1 \leq \sigma_1 X_2 \rightarrow \text{sigma}_1 X_3)$.

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

let $(p_{15} : T_3 \leq \varphi_2 \sigma_1 X_2 \rightarrow \text{sigma}_1 X_3) = \text{trans} \ p_{14} \ (\text{congr} \ p_{13})$ in

let $(p_{16} : \sigma_1 X_2 \rightarrow \text{sigma}_1 X_3 \leq \sigma_1(X_1 \rightarrow X_3))$

$= \text{trans} \ (\text{iArrow} \ p_6 \ \text{refl}) \ \text{factor} \ \text{in}$

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

$(p_{17}))$ in

let $(\text{Eta} : \forall T,U.U \rightarrow (\text{Exp}[T] \rightarrow U) \rightarrow \text{Exp}[T] \rightarrow U) =$

$\lambda(\text{ifNoOpt} : U). \ \lambda(\text{ifOpt} : \text{Exp}[T] \rightarrow U)$. 

63
analyzeEta ifNoOpt

\( (\lambda e_3 : Exp[T_3']) \).
\( \lambda (pS : Ty[S] \leq T'_1 \rightarrow T'_2 \rightarrow T) \).
\( \lambda (pK : Ty[K] \leq T'_3 \rightarrow T'_1) \).
\( \lambda (pI : Ty[I] \leq T'_2) \).

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

\[
\begin{align*}
\text{let } (\text{enactK} : \forall T_1, T_2, T_3. (Ty[K] \leq T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \) \\
Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) = \\
(\lambda (pIsK : Ty[K] \leq T_1 \rightarrow T_2 \rightarrow T_3). \\
\lambda (e_1 : Exp[T_1]). \lambda (e_2 : Exp[T_2])) .
\end{align*}
\]

eBinary pIsK

\( (\lambda (p_1 : \varphi \leq \varphi' \circ \sigma)) . \)
\( \lambda (p_2 : T_1 \leq \varphi' \sigma X) . \)
\( \lambda (p_3 : T_2 \leq \varphi' \sigma Z) . \)
\( \lambda (p_4 : \varphi' \sigma X \leq T) . \)

\[
\begin{align*}
\text{let } (p_5 : T_1 \leq T) = \text{trans } p_2 p_4 \text{ in } \\
\text{let } (p_6 : Exp[T_1] \leq Exp[T]) = \text{iExp } p_5 \\
in \text{coerce } e_1 p_6)) \text{ in }
\end{align*}
\]

\[
\begin{align*}
\text{let } (\text{reduceK} : \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) = \\
\lambda (\text{notK} : U). \lambda (\text{ifK} : Exp[T] \rightarrow U).
\end{align*}
\]

matchK2 notK \( (\lambda (p : Ty[K] \leq T_1 \rightarrow T_2 \rightarrow T). \lambda (e_1 : Exp[T_1]). \lambda (e_2 : Exp[T_2])) . \)

\[
\begin{align*}
\text{ifK } (\text{enactK } p \ e_1 \ e_2)) \text{ in }
\end{align*}
\]
\begin{verbatim}
let (enactS : \forall T_1, T_2, T_3, T_4. (Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4) \rightarrow
    Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3] \rightarrow Exp[T_4]) =
\lambda(p : Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4).
\lambda(e_1 : Exp[T_1]), (e_2 : Exp[T_2]), (e_3 : Exp[T_3]).

let (enactS p)
(\lambda(p_1 : \forall [X_1, X_2, X_3] \leq (\varphi \circ \sigma)).
\lambda(p_2 : T_1 \leq \varphi \sigma (X_1 \rightarrow X_2 \rightarrow X_3)).
\lambda(p_3 : T_2 \leq \varphi \sigma (X_1 \rightarrow X_2)).
\lambda(p_4 : T_3 \leq \varphi \sigma X_1).
\lambda(p_5 : \varphi \sigma X_3 \leq T_4).

let (p'_2 : T_1 \leq \varphi \sigma X_1 \rightarrow \varphi \sigma X_2 \rightarrow \varphi \sigma X_3) = trans p_2 dist2 in
let (p'_3 : T_2 \leq \varphi \sigma X_1 \rightarrow \varphi \sigma X_2) = trans p_3 dist in
let (e'_1 : Exp[\varphi \sigma X_1 \rightarrow \varphi \sigma X_2 \rightarrow \varphi \sigma X_3]) = coerce e_1 (iExp p'_2) in
let (e'_2 : Exp[\varphi \sigma X_1 \rightarrow \varphi \sigma X_2]) = coerce e_2 (iExp p'_3) in
let (e'_3 : Exp[\varphi \sigma X_1]) = coerce e_3 (iExp p_4) in
coerce (A (A e'_1 e'_3) (A e'_2 e'_3)) (iExp p_5)) in

let (reduceS : \forall T, U. U \rightarrow (Exp[T] \rightarrow U) \rightarrow Exp[T] \rightarrow U) =
\lambda(notS : U). \lambda(ifS : Exp[T] \rightarrow U).

matchS3 notS (\lambda(p : Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4).
\lambda(e_1 : Exp[T_1]), \lambda(e_2 : Exp[T_2]), \lambda(e_3 : Exp[T_3]).
\lambda(ifS (enactS p e_1 e_2 e_3)) in
\end{verbatim}
APPENDIX C

A Self-Interpreter

Our self-interpreter \textit{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{verbatim}
let (unquote : \forall X. Exp[X] \rightarrow X) =
    let rec (unquote : \forall X. Exp[X] \rightarrow X) =
        GI (\lambda(e_1 : Exp[T_1 \rightarrow T_1]).\lambda(e_2 : Exp[T_1]).unquote e_1 (unquote e_2)) in

let rec (enact : \forall T. Exp[T] \rightarrow Exp[T]) =

let (enactK : \forall T_1, T_2, T_3.(Ty[K] \leq T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow
    Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) =
    (\lambda(pIsK : Ty[K] \leq T_1 \rightarrow T_2 \rightarrow T_3).\lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).
eBinary pIsK
    (\lambda(p_1 : \varphi \leq \varphi' \circ \sigma)).
    \lambda(p_2 : T_1 \leq \varphi' \sigma X).
    \lambda(p_3 : T_2 \leq \varphi' \sigma Z).
    \lambda(p_4 : \varphi' \sigma X \leq T).
    let (p_5 : T_1 \leq T) = trans p_2 p_4 in
    let (p_6 : Exp[T_1] \leq Exp[T]) = iExp p_5
\end{verbatim}
\[
\text{in } enact(\text{coerce } e_1 \ p_6)) \text{ in }
\]

\[
\text{let } (enactS : \forall T_1, T_2, T_3, T_4. (Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4) \rightarrow \\
\quad \text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow \text{Exp}[T_4]) = \\
\quad \lambda (p : Ty[S] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4).
\]

\[
\lambda (e_1 : \text{Exp}[T_1]), (e_2 : \text{Exp}[T_2]), (e_3 : \text{Exp}[T_3]).
\]

\[
eTernary \ p
\]

\[
(\lambda (p_1 : \forall [X_1, X_2, X_3] \leq (\varphi \circ \sigma)).
\lambda (p_2 : T_1 \leq \varphi \sigma (X_1 \rightarrow X_2 \rightarrow X_3)).
\lambda (p_3 : T_2 \leq \varphi \sigma (X_1 \rightarrow X_2)).
\lambda (p_4 : T_3 \leq \varphi \sigma X_1).
\lambda (p_5 : \varphi \sigma X_3 \leq T_4).
\]

\[
\text{let } (p'_2 : T_1 \leq \varphi \sigma X_1 \rightarrow \varphi \sigma X_2 \rightarrow \varphi \sigma X_3) = \text{trans } p_2 \ \text{dist2} \text{ in }
\]

\[
\text{let } (p'_3 : T_2 \leq \varphi \sigma X_1 \rightarrow \varphi \sigma X_2) = \text{trans } p_3 \ \text{dist} \text{ in }
\]

\[
\text{let } (e'_1 : \text{Exp}[\varphi \sigma X_1 \rightarrow \varphi \sigma X_2 \rightarrow \varphi \sigma X_3]) = \text{coerce } e_1 \ (\text{iExp } p'_2) \text{ in }
\]

\[
\text{let } (e'_2 : \text{Exp}[\varphi \sigma X_1 \rightarrow \varphi \sigma X_2]) = \text{coerce } e_2 \ (\text{iExp } p'_3) \text{ in }
\]

\[
\text{let } (e'_3 : \text{Exp}[\varphi \sigma X_1]) = \text{coerce } e_3 \ (\text{iExp } p_4) \text{ in }
\]

\[
enact(\text{coerce } (A (A e'_1 e'_3) (A e'_2 e'_3)) (\text{iExp } p_5))) \text{ in }
\]

\[
\text{let } (enactStrict : \forall T. \text{Exp}[T] \rightarrow \text{Exp}[T]) = 
\]

\[
\text{let rec } (f : \forall T. \text{Exp}[T] \rightarrow \text{Exp}[T]) = 
\]

\[
G \ Q (\lambda (e_1 : \text{Exp}[T_1] \rightarrow T_1).\lambda (e_2 : \text{Exp}[T_1]).A (f e_1) (f (enact e_2))) \text{ in }
\]

\[
\text{let } (enactG : \forall T_1, T_2, T_3, T_4. (Ty[G] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4) \rightarrow \\
\quad \text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow \text{Exp}[T_4]) = \\
\quad \lambda (pIsG : Ty[G] \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4).
\]

67
\(\lambda(e_1 : \text{Exp}[T1]).\lambda(e_2 : \text{Exp}[T2]).\lambda(e_3 : \text{Exp}[T3]).\)

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

\(\text{let } (mkFQ : \varphi(\text{Exp}[U1 \rightarrow U2] \rightarrow \text{Exp}[U1] \rightarrow \text{Exp}[U2])) = A \text{ in}
\)

\(\text{let } (mkFA : \varphi((\forall U3.\text{Exp}[\text{Exp}[U3 \rightarrow U1] \rightarrow \text{Exp}[U3] \rightarrow U2]) \rightarrow
\)

\((\forall U3.\text{Exp}[\text{Exp}[U3 \rightarrow U1] \rightarrow \text{Exp}[\text{Exp}[U3] \rightarrow U2]) \rightarrow
\
\text{Exp}[\text{Exp}[U1] \rightarrow \text{Exp}[U2]]) =
\)

\(\lambda(fQ : \text{Exp}[U1] \rightarrow \text{Exp}[U2]).
\)

\(\lambda(fA : \forall U3.\text{Exp}[\text{Exp}[U3 \rightarrow U1] \rightarrow \text{Exp}[\text{Exp}[U3] \rightarrow U2]).
\)

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

\(\lambda(e_4 : \text{Exp}[U4 \rightarrow \text{Exp}[U1]]). \text{ // (Q Q) or (A (Q A) p’ )}
\)

\(\lambda(e_5 : \text{Exp}[U4]). \text{ // ’ o}
\)

\(\text{G } (\lambda(e_4' : U4 \rightarrow \text{Exp}[U1]). \text{ // Q }
\)

\(\text{IsQ } e_4'
\)

\(\lambda(p_7 : Ty[Q] \leq U4 \rightarrow \text{Exp}[U1]).
\)

\(\lambda(unusedQ : Ty[Q]).
\)

\(\text{eArrow } p_7
\)

\(\lambda(p_8 : \varphi_2 \leq \varphi_3 \circ \sigma_1).
\)

\(\lambda(p_9 : U4 \leq \varphi_3 \sigma_1 U5).
\)

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

\(\text{let } (p_{11} : \varphi_3 \sigma_1 U5 \leq U1) =
\)

68
eExp (trans factorExp p10) in
let (p12 : Exp[U4] ≤ Exp[U1]) =
iExp (trans p9 p11) in
fQ (coerce e5 p12))

error)

// Q A
(λ(e6 : Exp[U5 → U4 → Exp[U1]]) .
λ(e7 : Exp[U5]) .
G (λ(e'_6 : U5 → U4 → Exp[U1]) .
IsA e'_6
λ(unusedA : Ty[A]) .
eBinary p13
(λ(p14 : ϕ_4 ≤ ϕ_5 ∘ σ_2) .
λ(p15 : U5 ≤ ϕ_5 σ_2 Exp[U6 → U7]) .
λ(p16 : U4 ≤ ϕ_5 σ_2 Exp[U6]) .
λ(p17 : ϕ_5 σ_2 Exp[U7] ≤ Exp[U1]) .
let (p18 : U5 ≤ Exp[ϕ_5 σ_2 U6 → ϕ_5 σ_2 U7]) =
trans p15 (trans distExp (iExp dist)) in
let (p19 : U4 ≤ Exp[ϕ_5 σ' U6]) = trans p16 distExp in
let (p20 : ϕ_5 σ' U7 ≤ U1) =
eExp (trans factorExp p17) in
let (p21 : U5 ≤ Exp[ϕ_5 σ' U6 → U1]) =
trans2 p18 distExp (iExp
(iArrow refl (eExp (trans factorExp p17)))) in
let (e'_7 : Exp[Exp[ϕ_5 σ' U6 → U1]] =
coerce e₇ (iExp p₂₁) in
let (e'_₅ : Exp[Exp[ϕ₅σ[U₆]]]) =
coerce e₅ (iExp p₁₉) in
fA e'_₇ e'_₅)

error) // IsA failed
(K (K error)) // e₆ = A x y

e₄) in
eTernary plIsG

(λ(p₂ : ϕ₁≤φ₁ o σ).
λ(p₃ : T₁≤φ₁σ(U₁ → U₂)).
λ(p₄ : T₂≤φ₁σ(∀U₃.Exp[U₃ → U₁] → Exp[U₃] → U₂)).
λ(p₅ : T₃≤φ₁σExp[U₁]).
λ(p₆ : φ₁σU₂≤T₄).
let (mkFQ' : φ₁σExp[U₁ → U₂] →
ϕ₁σ(Exp[U₁] → Exp[U₂])) =
coerce mkFQ (trans p₂ dist) in
let (fQ' : φ₁σ(Exp[U₁] → Exp[U₂])) =
mkFQ' (coerce e₁ (trans (iExp p₃) factorExp)) in
let (mkFA' : φ₁σ(∀U₃.Exp[Exp[U₃ → U₁] → Exp[U₃] → U₂]) →
φ₁σ(∀U₃.Exp[Exp[U₃ → U₁] →
Exp[Exp[U₃]] → Exp[U₂])) =
coerce mkFA (trans p₂ dist) in
let (fA' : φ₁σ(∀U₃.Exp[Exp[U₃ → U₁]]) →
Exp[Exp[U₃]] → Exp[U₂]) =
mkFA' (coerce e₂ (trans (iExp p₄) factorExp)) in
let \( fG' : \varphi_1 \sigma(\text{Exp}[U_1] \rightarrow \text{Exp}[U_2]) \rightarrow \varphi_1 \sigma(\forall U_3.\text{Exp}[\text{Exp}[U_3] \rightarrow U_1] \rightarrow \text{Exp}[\text{Exp}[U_3]] \rightarrow \text{Exp}[U_2]) \rightarrow \varphi_1 \sigma\text{Exp}[\text{Exp}[U_1]] \rightarrow \varphi_1 \sigma\text{Exp}[\text{Exp}[U_2]] = \) coerce \( fG \) (trans \( p_2 \) (trans dist (iArrow refl dist2))) in

let \( e'_3 : \text{Exp}[\varphi_1 \sigma\text{Exp}[U_1]] = \text{enact} \ (\text{coerce} \ e_3 \ (\text{iExp} p_5)) \) in

let \( e : \varphi_1 \sigma\text{Exp}[U_2] = fG' \ fQ' \ fA' \) (coerce \( e'_3 \) factorExp) in

let \( p'_6 : \varphi_1 \sigma\text{Exp}[U_2] \leq \text{Exp}[T_4] = \text{trans distExp} \ (\text{iExp} p_6) \) in \( \text{enact} \ (\text{coerce} \ e p'_6) \) in

let \( \text{enactIs} : \forall T_1, T_2, T_3, T_4. (TyIs \leq T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4) \rightarrow TyIs \rightarrow \text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3] \rightarrow \text{Exp}[T_4] = \) (\( \lambda(p_1 : TyIs \leq X \rightarrow Y \rightarrow Z \rightarrow T_4). \lambda(o' : TyIs). \lambda(mkIs : U_1 \rightarrow (U_2 \rightarrow U_3 \rightarrow \text{Exp}[U_4]) \rightarrow \text{Exp}[U_4] \rightarrow \text{Exp}[U_4] \rightarrow \text{Exp}[U_1] \rightarrow \text{Exp}[U_2] \rightarrow U_3 \rightarrow U_4 \rightarrow \text{Exp}[U_4] \rightarrow \text{Exp}[U_4]) \) = \( \lambda(isO : U_1 \rightarrow (U_2 \rightarrow U_3 \rightarrow \text{Exp}[U_4]) \rightarrow \text{Exp}[U_4] \rightarrow \text{Exp}[U_4]). \lambda(o'' : \text{Exp}[U_1]). \lambda(eTrue : \text{Exp}[U_2 \rightarrow U_3 \rightarrow U_4]). \lambda(eFalse : \text{Exp}[U_4]). \lambda(a : U_1). \)

isO (unquote o'')

\( (\lambda(p : U_2).\lambda(t : U_3).\lambda(A eTrue (Q p)) (Q t)). eFalse \)

(K (K eFalse))
(enact o'') in
let (is : \(\varphi(\text{Exp}[U1] \to \text{Exp}[U2 \to U3 \to U4] \to \text{Exp}[U4] \to \text{Exp}[U4])\))
  = (coerce mkIs dist) o' in
eArrow p_{1}
  (\lambda(p_{2} : \varphi \leq \varphi_{1} \circ \sigma_{1}).
   \lambda(p_{3} : T_{1} \leq \varphi_{1} \sigma_{1} U_{1}).
   \lambda(p_{4} : \varphi_{1} \sigma_{1}(U_{2} \to U_{3} \to U_{4}) \to U_{4} \leq U_{4} \to T_{2} \to T_{3} \to T_{4}).
   \lambda(e_{1} : \text{Exp}[T_{1}]).
   let (is_{1} : (\varphi_{1} \sigma_{1} \text{Exp}[U1]) \to
   \quad (\varphi_{1} \sigma_{1} \text{Exp}[U2 \to U3 \to U4] \to \text{Exp}[U4] \to \text{Exp}[U4])) =
   coerce is (trans p_{2} dist) in
   let (is_{2} : \varphi_{1} \sigma_{1} \text{Exp}[U2 \to U3 \to U4] \to \text{Exp}[U4] \to \text{Exp}[U4]) =
   is_{1}(coerce e_{1} (trans (iExp p_{3}) factorExp)) in
eBinary p_{4}
  (\lambda(p_{5} : \varphi_{1} \circ \sigma_{1} \leq \varphi_{3} \circ \sigma_{3}).
   \lambda(p_{6} : T_{2} \leq \varphi_{3} \sigma_{3}(U_{2} \to U_{3} \to U_{4})).
   \lambda(p_{7} : T_{3} \leq \varphi_{3} \sigma_{3} U_{4}).
   \lambda(p_{8} : \varphi_{3} \sigma_{3} U_{4} \leq T_{4}).
   \lambda(e_{2} : \text{Exp}[T_{2}]), (e_{3} : \text{Exp}[T_{3}]).
   let (is_{3} : \varphi_{3} \sigma_{3} \text{Exp}[U2 \to U3 \to U4] \to
   \quad \varphi_{3} \sigma_{3} \text{Exp}[U4] \to \varphi_{3} \sigma_{3} \text{Exp}[U4]) =
   coerce is_{2} (trans p_{5} dist2) in
   let (e : \varphi_{3} \sigma_{3} \text{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

72
\[\text{let } (\text{enact}\ Y : \forall T_1, T_2, T_3. (Ty[Y] \leq T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \]
\[Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) =
\]
\[\lambda(p_1 : Ty[Y] \leq T_1 \rightarrow T_2 \rightarrow T_3).
\]
\[\lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).
\]
\[e\text{Binary } p_1
\]
\[(\lambda(p_2 : \forall X_1, X_2 \hat{\leq} \varphi \circ \sigma).
\]
\[\lambda(p_3 : T_1 \hat{\leq} \varphi \sigma((X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_2)).
\]
\[\lambda(p_4 : T_2 \hat{\leq} \varphi \sigma X_1).
\]
\[\lambda(p_5 : \varphi \sigma X_2 \hat{\leq} T_3).
\]
\[\text{let } (e'_1 : Exp[\varphi \sigma(X_1 \rightarrow X_2) \rightarrow \varphi \sigma X_1 \rightarrow \varphi \sigma X_2]) =
\]
\[\text{coerce } e_1 \text{ (iExp (trans } p_3 \text{ dist2)) in}
\]
\[\text{let } (y : \varphi \sigma((X_1 \rightarrow X_2) \rightarrow (X_1 \rightarrow X_2)) \rightarrow \varphi \sigma(X_1 \rightarrow X_2)) =
\]
\[\text{coerce } Y\text{dist in}
\]
\[\text{let } (e : Exp[\varphi \sigma X_1 \rightarrow \varphi \sigma X_2]) =
\]
\[A \ e'_1 \ (A \ (Q \ y) \ (\text{coerce } e_1 \ (\text{iExp } p_3))) \text{ in}
\]
\[\text{enact } (\text{coerce } (A \ e(\text{coerce } e_2 \ (\text{iExp } p_4))) \ (\text{iExp } p_5))) \text{ in}
\]
\[\text{let } (\text{enactCoerce} : \forall T_1, T_2, T_3. (Ty[coerce] \hat{\leq} T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \\
Exp[T_1] \rightarrow Exp[T_2] \rightarrow Exp[T_3]) =
\]
\[\lambda(p\text{IsCoerce} : Ty[coerce] \hat{\leq} T_1 \rightarrow T_2 \rightarrow T_3).
\]
\[\text{let } (f : Exp[U_1] \rightarrow Exp[U_1 \hat{\leq} U_2] \rightarrow Exp[U_2]) =
\]
\[\lambda(a_1 : Exp[U_1]) .
\]
\[\lambda(a_2 : Exp[U_1 \hat{\leq} U_2]) .
\]
\[\text{coerce } a_1 \ (\text{iExp (unquote (enactStrict (enact } a_2))) \text{ in}
\]
\[\text{eBinary } p\text{IsCoerce}
\]
\[(\lambda(p_1 : \varphi \trianglelefteq \varphi' \circ \sigma').\]
\[\lambda(p_2 : T_1 \trianglelefteq \varphi' \circ \sigma' V_1).\]
\[\lambda(p_3 : T_2 \trianglelefteq \varphi' \circ \sigma' (V_1 \trianglelefteq V_2)).\]
\[\lambda(p_4 : \varphi' \circ \sigma' V_2 \trianglelefteq T_3).\]
\[\lambda(e_1 : \text{Exp}[T_1]).\]
\[\lambda(e_2 : \text{Exp}[T_2]).\]
\[\text{let } (f' : \varphi' \circ \text{Exp}[V_1] \rightarrow \varphi' \circ \text{Exp}[V_1 \trianglelefteq V_2] \rightarrow \varphi' \circ \text{Exp}[V_2]) =\]
\[\text{coerce } f \text{ (trans } p_1 \text{ dist2) in}\]
\[\text{let } (e'_1 : \varphi' \circ \text{Exp}[V_1]) =\]
\[\text{coerce } e_1 \text{ (trans } e_{\text{Exp}} p_2 \text{ factorExp) in}\]
\[\text{let } (e'_2 : \varphi' \circ \text{Exp}[V_1 \trianglelefteq V_2]) =\]
\[\text{coerce } e_2 \text{ (trans } e_{\text{Exp}} p_3 \text{ factorExp) in}\]
\[\text{let } (e : \varphi' \circ \text{Exp}[V_2]) = f' e'_1 e'_2 \text{ in}\]
\[\text{enact (coerce } e \text{ (trans distExp } e_{\text{Exp}} p_4)))\text{ in}\]

\[\text{let } (\text{enactEArrow} : \forall T_1, T_2, T_3. (\text{Ty}[\text{eArrow}] \trianglelefteq T_1 \rightarrow T_2 \rightarrow T_3) \rightarrow \]
\[\text{Exp}[T_1] \rightarrow \text{Exp}[T_2] \rightarrow \text{Exp}[T_3]) =\]
\[\lambda(p\text{IsEArrow} : \text{Ty}[\text{eArrow}] \trianglelefteq T_1 \rightarrow T_2 \rightarrow T_3).\]
\[\text{let } (f : \varphi(\text{Exp}[\varphi_{1}(U_1 \rightarrow U_2) \trianglelefteq \varphi_{2}(V_1 \rightarrow V_2)]) \rightarrow \]
\[\text{Exp}[\forall \varphi_3, \theta_3. (\varphi_1 \trianglelefteq \varphi_2 \circ \varphi_3 \circ \theta_3) \rightarrow \]
\[\quad (V_1 \trianglelefteq \varphi_3 \theta_3 U_1) \rightarrow (\varphi_3 \theta_3 U_2 \trianglelefteq V_2) \rightarrow C) \rightarrow \]
\[\text{Exp}[C]) =\]
\[\lambda(q_1 : \text{Exp}[\varphi_{1}(U_1 \rightarrow U_2) \trianglelefteq \varphi_{2}(V_1 \rightarrow V_2))].\]
\[\lambda(g : \text{Exp}[\forall \varphi_3, \theta_3. (\varphi_1 \trianglelefteq \varphi_2 \circ \varphi_3 \circ \theta_3) \rightarrow \]
\[\quad (V_1 \trianglelefteq \varphi_3 \theta_3 U_1) \rightarrow (\varphi_3 \theta_3 U_2 \trianglelefteq V_2) \rightarrow C)].\]
\[\text{eArrow } (\text{unquote } (\text{enactStrict } (\text{enact } q_1)))\]
(\lambda(q_2 : \varphi_1 \leq \varphi_2 \circ \varphi_3 \circ \theta_3).
\lambda(q_3 : V_1 \leq \varphi_3 \theta_3 U_1).
\lambda(q_4 : \varphi_3 \theta_3 U_2 \leq V_2).
A (A (A (g \ (Q \ q_2)) \ (Q \ q_3)) \ (Q \ q_4)) \text{ in } eBinary pIsEArrow
(\lambda(p_1 : \varphi \leq \varphi' \circ \theta').
\lambda(p_2 : X \leq \varphi' \theta'((\varphi_1(U_1 \rightarrow U_2) \leq \varphi_2(V_1 \rightarrow V_2))).
\lambda(p_3 : Y \leq \varphi' \theta'(\forall \varphi_3, \theta_3. (\varphi_1 \leq \varphi_2 \circ \varphi_3 \circ \theta_3) \rightarrow
(V_1 \leq \varphi_3 \theta_3 U_1) \rightarrow (\varphi_3 \theta_3 U_2 \leq V_2) \rightarrow C)).
\lambda(p_4 : \varphi' \theta' C \leq T).
\lambda(e_x : \text{Exp}[T_1]), \lambda(e_y : \text{Exp}[T_2]).
\text{let } (f' : \varphi' \theta' \text{Exp}[(\varphi_1(U_1 \rightarrow U_2) \leq \varphi_2(V_1 \rightarrow V_2))] \rightarrow
\varphi' \theta' \text{Exp}[(\forall \varphi_3, \theta_3. (\varphi_1 \leq \varphi_2 \circ \varphi_3 \circ \theta_3) \rightarrow
(V_1 \leq \varphi_3 \theta_3 U_1) \rightarrow (\varphi_3 \theta_3 U_2 \leq V_2) \rightarrow C)] \rightarrow
\varphi' \theta' \text{Exp}[C]) =
\text{coerce } f \ (\text{trans } p_1 \ \text{dist2}) \ \text{in}
\text{let } (e'_x : \varphi' \theta' \text{Exp}[(\varphi_1(U_1 \rightarrow U_2) \leq \varphi_2(V_1 \rightarrow V_2))] =
\text{coerce } e_x \ (\text{trans } (\text{iExp } p_2) \ \text{factorExp}) \ \text{in}
\text{let } (e'_y : \varphi' \theta' \text{Exp}[(\forall \varphi_3, \theta_3. (\varphi_1 \leq \varphi_2 \circ \varphi_3 \circ \theta_3) \rightarrow
(V_1 \leq \varphi_3 \theta_3 U_1) \rightarrow (\varphi_3 \theta_3 U_2 \leq V_2) \rightarrow C)] =
\text{coerce } e_y \ (\text{trans } (\text{iExp } p_3) \ \text{factorExp}) \ \text{in}
\text{let } (e : \varphi_3 \theta_3 \text{Exp}[C]) = f' e'_x e'_y \ \text{in}
\text{enact } (\text{coerce } e \ (\text{trans distExp } (\text{iExp } p_4)))) \ \text{in}
\text{let } (\text{enactI} : \forall T_1, T_2. (T y[l] \leq T_1 \rightarrow T_2) \rightarrow \text{Exp}[T_1] \rightarrow \text{Exp}[T_2]) =
\lambda(p : T y[l] \leq T_1 \rightarrow T_2).
\[ \lambda(e_1 : Exp[T_1]). \]

\[ \text{eArrow } p \]
\[ (\lambda(p_1 : \forall[X] \leq (\varphi \circ \sigma)). \]
\[ \lambda(p_2 : T_1 \leq \varphi \sigma X). \]
\[ \lambda(p_3 : \varphi \sigma X \leq T_2). \]
\[ \text{enact (coerce } e_1 \text{ (iExp (trans } p_2 p_3))) \] in

### let (enact3 : \forall T_1, T_2, T_3, T_4.(T_1 \to T_2 \to T_3 \to T_4) \to 
\[ Exp[T_1] \to Exp[T_2] \to Exp[T_3] \to Exp[T_4] = \]
\[ \lambda(o : T_1 \to T_2 \to T_3 \to T_4). \]
\[ \text{IsS o } (\lambda(p : Ty[S] \leq T_1 \to T_2 \to T_3 \to T_4).\lambda(s : Ty[S]).\text{enactS p}) (\]
\[ \text{IsG o } (\lambda(p : Ty[G] \leq T_1 \to T_2 \to T_3 \to T_4).\lambda(g : Ty[G]).\text{enactG p}) (\]
\[ \text{IsIs o enactIs} (\]
\[ \lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).\lambda(e_3 : Exp[T_3]).\lambda(A(A(Q_o)e_1)e_2)e_3 \]
\[ ))) \] 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]) = 
\[ \lambda(o : T_1 \to T_2 \to T_3). \]
\[ \text{IsK o } (\lambda(p : Ty[K] \leq T_1 \to T_2 \to T_3).\lambda(k : Ty[K]).\text{enactK p})(\]
\[ \text{IsY o } (\lambda(p : Ty[Y] \leq T_1 \to T_2 \to T_3).\lambda(o' : Ty[Y]).\text{enactY p})(\]
\[ \text{IsCoerce o } (\lambda(p : Ty[coerce] \leq T_1 \to T_2 \to T_3).\lambda(o' : Ty[coerce]). 
\[ \text{enactCoerce p})(\]
\[ \text{IsEArrow o } (\lambda(p : Ty[eArrow] \leq T_1 \to T_2 \to T_3).\lambda(o' : Ty[eArrow]). 
\[ \text{enactEArrow p})(\]
\[ \lambda(e_1 : Exp[T_1]).\lambda(e_2 : Exp[T_2]).\lambda(A(Q_o)e_1)e_2 \]
\[ ))) \] in

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

76
\[ \lambda(o : T_1 \rightarrow T_2). \]
\[ \text{IsI } o \ (\lambda(p : T y[I] \leq T_1 \rightarrow T_2). \lambda(o' : T y[I]). \text{enactI } p)( \]
\[ \lambda(e_1 : Exp[T_1]). \text{A } (Q \ o) \ e_1 \]
\) in \[ G \ Q \]
\[ (\lambda(e_1 : Exp[T_1 \rightarrow T]). \]
\[ G(\lambda(o : T_1 \rightarrow T). \text{enact1 } o) \]
\[ (G(\lambda(o : T_2 \rightarrow T_1 \rightarrow T). \text{enact2 } o) \]
\[ (G(\lambda(o : T_3 \rightarrow T_2 \rightarrow T_1 \rightarrow T). \text{enact3 } o) \]
\[ (\lambda(x_1 : Exp[T_4 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1 \rightarrow T]). \]
\[ \lambda(x_2 : Exp[T_4]). \lambda(x_3 : Exp[T_3]). \lambda(x_4 : Exp[T_2]). \lambda(x_5 : Exp[T_1]). \]
\[ \text{A } (A (A (A (x_1 \ x_2) \ x_3) \ x_4) \ x_5)) \]
\[ (\text{enact } e_1)) \]
References


