4-2007

Sound and Complete Elimination of Singleton Kinds

Karl Crary
Carnegie Mellon University

Sound and Complete Elimination of Singleton Kinds

KARL CRARY
Carnegie Mellon University

Singleton kinds provide an elegant device for expressing type equality information resulting from modern module languages, but they can complicate the metatheory of languages in which they appear. I present a translation from a language with singleton kinds to one without, and prove that translation to be sound and complete. This translation is useful for type-preserving compilers generating typed target languages. The proof of soundness and completeness is done by normalizing type equivalence derivations using Stone and Harper's type equivalence decision procedure.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Languages, Theory

Additional Key Words and Phrases: Type systems, singleton kinds

1. INTRODUCTION

Type-preserving compilation, compilation using statically typed intermediate languages, offers many compelling advantages over conventional untyped compilation. A typed compiler can utilize type information to enable optimizations that would otherwise be prohibitively difficult or impossible. Internal type checking can be used to help debug a compiler by catching errors introduced into programs in optimization or transformation stages. Finally, if preserved through the compiler to its ultimate output, types can be used to certify that executables are *safe*, that is, free of certain fatal errors or malicious behavior [Morrisett et al. 1999].

For typed compilation to be practical, we require elegant yet expressive type theories for use in the compiler: expressive because they must support the full expressive power of a real source language, and elegant because they must be practical for a compiler to manipulate. One important issue arising in the design of such type theories for compiling Standard ML, Objective CAML, and similar languages is how to account for type abbreviations and sharing constraints in the module

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

language. For example, consider the following SML signature:

```
signature SIG =
  sig
    type t = int
    val x : t
    val f : t -> t
  end
```

If `S` is a structure having signature `SIG`, the type theory must ensure that `S.t` is interchangeable with `int` in any code having access to `S`.

The standard account of sharing in type theory was developed independently by Harper and Lillibridge, under the name translucent sums [Harper and Lillibridge 1994; Lillibridge 1997], and by Leroy, under the name manifest types [Leroy 1994] (and extended in Leroy [1995]). These type theories provide a facility for stating type abbreviations in signatures and (importantly) ensure the correct propagation of type information resulting from those abbreviations. (Exactly what is meant by correct propagation is discussed in Section 2.1.) Translucent sums are employed in the type-theoretic definition of Standard ML given by Harper and Stone [2000] (currently the only formal account of an entire practical programming language in type theory), and manifest types are similarly employed (somewhat less formally) by Leroy [2000] for Objective CAML.

In this paper I consider a type theory based on *singleton kinds* [Stone and Harper 2000], a variant of the translucent sum/manifest type formalism. The singleton kind calculus differs from the module-oriented presentation in that it separates the module system from the mechanisms for type abbreviations and focuses on the latter. This separation is appropriate, first, because the two issues are orthogonal (although they typically arise together in practice), but more importantly, because type abbreviations persevere even after the compiler eliminates modules [Harper et al. 1990]. Furthermore, separating modules from the issue of type propagation makes it unnecessary to compare types by name (as in most module-oriented accounts), which makes it possible to propagate more type information. (An example of this is given in Section 2.1.)

Singleton kinds provide a very elegant and uniform type-theoretic mechanism for ensuring the propagation of type information. Kinds are used in type theories containing higher-order type constructors to classify type constructors just as types classify ordinary terms. Using singleton kinds, in the above example `S.t` is given the kind $S(\texttt{int})$, the kind containing only the type `int` (and types equal to it). Propagation of type information is then obtained by augmenting the typechecker with the rule that if τ has kind $S(\tau')$, then $\tau = \tau'$.

When using singleton kinds in practice, the question arises of how singleton kinds affect typechecking, given that they provide a new (and conceivably difficult to discover) way to show types to be equal. In fact, Harper and Stone [2000] show that there exists a very simple algorithm for deciding equality of types in the presence of singleton kinds. Indeed, the algorithm is very nearly identical to the usual algorithm employed in the absence of singletons in practice (as opposed to the less-efficient algorithms often considered in theory). In this sense, singleton kinds complicate the compiler very little.

Nevertheless, there are some good reasons why one may want to compile away singleton kinds: Although the decision algorithm discussed above is simple, its proof of correctness is sophisticated and may be difficult to extend to more complicated type systems. The latter phases of a type-preserving compiler may involve some very complicated type systems indeed [Morrisett et al. 2002; Walker et al. 2000; Crary and Weirich 1999; Smith et al. 2000; Shao et al. 2002]. Extending Stone and Harper's proof to these type systems, some of which already have nontrivial decidability proofs, is a daunting prospect. Moreover, there already exist a variety of tools for manipulating low-level typed languages that, by and large, do not support singleton kinds.

In this paper, I present such a strategy for compiling away singleton kinds. To implement the source language correctly, this elimination strategy should be sound and complete relative to the singleton calculus, that is, two types should be equal in the singleton calculus if and only if they are equal after singleton elimination. This means that the elimination process does not cause any programs to cease to typecheck, nor does it allow any programs to typecheck that would not have before.[1]

The compilation process is based on the natural idea of substituting definitions for any appearances of variables having singleton kinds. However, how to do this in a sound and complete manner is not obvious because, as discussed below in Section 3.1, in the presence of internal bindings, it is difficult to determine whether or not a variable has a singleton kind. Although I show this issue can be handled elegantly, as with Stone and Harper, the correctness proof is not obvious. This proof is the central technical contribution of the paper.

The existence of a sound and complete compilation strategy does not imply that singleton kinds are useless. They provide an extremely elegant and succinct account of ML's type sharing that (with modules taken out of the picture) is essentially equivalent to the standard type-theoretic accounts employed to explain practical source languages. To exploit this result and remove singletons from consideration entirely (in the absence of some alternative) would require programmers to eliminate type abbreviations by hand, resulting in verbose, unreadable code (to no particular benefit). Moreover, singleton kinds may also be useful for some other purposes such as compression of type information, or polymorphic closure conversion [Minamide et al. 1996].

What this result does mean is using translucent sums, manifest types or singleton kinds to express sharing in the source language need not constrain the compilation strategy. One may use singleton kinds through as many compilation phases as desired, and then compile them away and proceed without them. For example, a reasonable architecture is to use singleton kinds in the compiler's front end (which performs ML-specific optimizations and transformations), but not in the back end (which may use complicated type systems for code generation and low-level transformations).

This paper is organized as follows: In Section 2, I formalize the singleton kind

[1]It may be argued that only the former property is essential to implement the source language correctly, that it is acceptable to allow more programs to typecheck provided that the post-translation type system is still sound. Nevertheless, the latter is still a desirable property, and it is obtained with no additional trouble.

$$
\begin{array}{lrcl}
\text{kinds} & K & ::= & T \mid S(c) \mid \Pi\alpha{:}K_1.K_2 \mid \Sigma\alpha{:}K_1.K_2 \\
\text{constructors} & c & ::= & \alpha \mid b \mid \lambda\alpha{:}K.c \mid c_1 c_2 \mid \langle c_1, c_2\rangle \mid \pi_1 c \mid \pi_2 c \\
\text{assignments} & \Gamma & ::= & \epsilon \mid \Gamma, \alpha{:}K
\end{array}
$$

Fig. 1. Syntax

calculus and discuss some of its subtleties that make it complicated to work with. In Section 3, I present the singleton elimination strategy and state its correctness theorem. Section 4 is dedicated to the proof of the correctness theorem, and concluding remarks appear in Section 5.

This paper assumes familiarity with type systems with higher-order type constructors and dependent types. The correctness proof draws from the work of Stone and Harper [2000] showing decidability of type equivalence in the presence of singleton kinds, but we will use their results almost entirely "off the shelf," so familiarity with their paper is not required.

2. A SINGLETON KIND CALCULUS

We begin by formalizing the singleton calculus that is the subject of this paper. The syntax of the singleton calculus is given in Figure 1. It consists of a class of type constructors (usually referred to as "constructors" for brevity) and a class of kinds, which classify constructors. The class of constructors contains variables (ranged over by α), a collection of base types (ranged over by b), and the usual introduction and elimination forms for functions and pairs over constructors. We could also add a collection of primitive type operators (such as `list` or `->`) without difficulty, but have not done so in the interest of simplicity.

The kind structure is the novelty of the singleton calculus. The base kinds include T, the kind of all types, and $S(c)$, the kind of all types definitionally equal to c. Thus, $S(c)$ represents a singleton set, up to definitional equality. The constructor c in $S(c)$ is permitted to be open, and consequently kinds may contain free constructor variables, which makes it useful to have *dependent* kinds. The kind $\Pi\alpha{:}K_1.K_2$ contains functions from K_1 to K_2, where α refers to the function's argument and may appear free in K_2. Analogously, the kind $\Sigma\alpha{:}K_1.K_2$ contains pairs of constructors from K_1 and K_2, where α refers to the left-hand member and may appear free in K_2. As usual, when α does not appear free in K_2, we write $\Pi\alpha{:}K_1.K_2$ as $K_1 \to K_2$ and $\Sigma\alpha{:}K_1.K_2$ as $K_1 \times K_2$.

In addition, the syntax provides a class of *assignments*, which assign kinds to free constructor variables, for use in the calculus's static semantics. In a practical application, the language would be extended with an additional class of terms, but for our purposes (which deal with constructor equality) we need not be concerned with terms, so they are omitted.

As usual, alpha-equivalent expressions (written $E \equiv E'$) are taken to be identical. The capture-avoiding substitution of c for α in E (where E is a kind, constructor or assignment) is written $E\{c/\alpha\}$. We also will often desire to define substitutions independent of a particular place of use, so when σ is a substitution, we denote the application of σ to the expression E by $E\{\sigma\}$. Separately defined substitutions will usually be written in the form $\{c_1/\alpha_1\}\cdots\{c_n/\alpha_n\}$, denoting a sequential

```
signature SIG2 =
  sig
    type s
    type t = int
    type u = s * t
    ... value fields ...
  end
```

```
funsig FSIG
  (S : sig
         type s
         ... value fields ...
       end) =
  sig
    type t
    type u = S.s * t
    ... value fields ...
  end
```

Fig. 2. Sample Signatures

substitution with the leftmost substitution taking place first.

As discussed in the introduction, the principal intended use of singleton kinds is in conjunction with module systems. For example, the type portion of signature `SIG2` in Figure 2 is translated to the kind:

$$\Sigma\alpha{:}T.\,\Sigma\beta{:}S(\mathtt{int}).\,S(\alpha{*}\beta)$$

Note the essential use of dependent sums in this kind. Dependent products arise from the phase splitting [Harper et al. 1990] of functors, in which the static portion of a functor (*i.e.,* its action on types) is separated from the dynamic portion. For example, after phase-splitting, the type portion of the functor signature `FSIG` in Figure 2 (given in the syntax of Standard ML of New Jersey version 110) is translated to the kind:

$$\Pi\alpha{:}T.\,(\Sigma\beta{:}T.\,S(\alpha{*}\beta))$$

2.1 Judgements

The inference rules defining the static semantics of the singleton calculus are given in Appendix A. For the reader's convenience, the rules are given in the same order and essentially the same form as in Stone and Harper [2000]. A summary of the judgements that these rules define, and their interpretations, are given in Figure 3. The context and kind equality judgements are auxiliary judgements used in theorems but not by any of the other judgements. For the most part, the static semantics consists of the usual rules for a dependently typed lambda calculus with products and sums (but lifted to the constructor level). Again, the novelty lies with the singleton kinds. Singleton kinds have two introduction rules (one for kind assignment and one for equivalence),

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash c : S(c)} \qquad \frac{\Gamma \vdash c = c' : T}{\Gamma \vdash c = c' : S(c)}$$

and one elimination rule:

$$\frac{\Gamma \vdash c : S(c')}{\Gamma \vdash c = c' : T}$$

Judgement	Interpretation
$\Gamma \vdash \mathrm{ok}$	Γ is a valid assignment
$\vdash \Gamma_1 = \Gamma_2$	Γ_1 and Γ_2 are equivalent assignments
$\Gamma \vdash K$	K is a valid kind
$\Gamma \vdash K_1 \leq K_2$	K_1 is a subkind of K_2
$\Gamma \vdash K_1 = K_2$	K_1 and K_2 are equivalent kinds
$\Gamma \vdash c : K$	c is a valid constructor with kind K
$\Gamma \vdash c_1 = c_2 : K$	c_1 and c_2 are equivalent as members of kind K

Fig. 3. Judgement Forms

These rules capture the intuition of singleton kinds: The first says that any type belongs to its own singleton kind. The second says that equivalent types are also considered equivalent as members of their singleton kind. The third says that if one type belongs to another's singleton kind, then those types are equivalent.

The complexity of the singleton calculus arises from the above rules in conjunction with the subkinding relation generated by the following two rules:

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash S(c) \leq T} \qquad \frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) \leq S(c_2)}$$

These rules are essential for singleton kinds to serve their intended purpose in a modern module system. The first allows a signature to match a supersignature obtained by removing equality specifications. For example, structures having the signature `SIG` from the introduction should also match the signature obtained by replacing the specification "`type t = int`" (which we might write `type t :` $S($`int`$)$) with simply "`type t`" (which we might write `type t :` T). The second allows a signature to match another signature obtained by replacing equality specifications with different but equivalent ones.

The presence of subkinding makes the usual context-insensitive methods of dealing with equivalence impossible. Consider the identity function, $\lambda\alpha{:}T.\alpha$, and the constant `int` function, $\lambda\alpha{:}T.$`int`. These functions are clearly inequivalent as members of $T \to T$; that is, the judgement $\vdash \lambda\alpha{:}T.\alpha = \lambda\alpha{:}T.$`int` $: T \to T$ is not derivable. However, since $T \to T$ is a subkind of $S($`int`$) \to T$, these two functions can also be compared as members of $S($`int`$) \to T$ and in that kind they *are* equivalent. This is because the bodies α and `int` are compared under the assignment $\alpha{:}S($`int`$)$, under which α and `int` are equivalent by the singleton elimination rule. This example makes it clear that to deal with constructor equivalence in the singleton calculus, one must take into account the contexts in which the constructors appear.

The determination of equivalence is further complicated by the fact that the classifying kind may be given *implicitly*. For example, the classifying kind may be imposed by a function on its argument. Consider the constructors $\beta(\lambda\alpha{:}T.\alpha)$ and $\beta(\lambda\alpha{:}T.$`int`$)$. These are well-formed under an assignment giving β the kind $(T \to T) \to T$ and also under one giving β the kind $(S($`int`$) \to T) \to T$. However, for the same reason as above, the two constructors are equivalent under the second

assignment but not the first.[2] The classifying kind can then be made even further remote by making β a function's formal argument instead of a free variable, and so on.

2.2 A Singleton-Free System

To formalize our results, we also require a singleton-free target language into which to translate expressions from the singleton calculus. We will define the singleton-free system in terms of its differences from the singleton calculus.

We will say that a constructor c (not necessarily well-formed) syntactically belongs to the singleton-free calculus provided that c contains no singleton kinds. Note that as a consequence of containing no singleton kinds, all product and sum kinds may be written in non-dependent form. Also, all kinds in the singleton-free calculus are well-formed.

The inference rules for the singleton-free system are obtained by removing from the singleton calculus all the rules dealing with subkinding (Rules 9–13, 28 and 45) and all the rules dealing with singleton kinds (Rules 6, 15, 25, 34 and 35). Note that derivable judgements in the singleton-free system must be built using only expressions syntactically belonging to the singleton-free calculus. When a judgement is derivable in the singleton-free system, we will note this fact by marking the turnstile $\vdash_{sf}$.

3. ELIMINATION OF SINGLETON KINDS

The critical rule in the static semantics of the singleton calculus is the singleton elimination rule (Rule 34). The main aim of the singleton kind elimination process is to rewrite constructors so that any equivalences that hold for those constructors may be derived without using that rule. If this aim is achieved, any singleton kinds remaining within the constructors are not used (in any essential way) and can simply be erased, resulting in valid constructors and derivations in the singleton-free system.

This erasure process is made precise in Figure 4, which defines a mapping $(-)^\circ$ from singleton calculus kinds to singleton-free kinds that replaces all singleton kinds by T. The erasure mapping is lifted to constructors and assignments in the obvious manner. If $\Gamma \vdash c_1 = c_2 : K$ is derivable without using singleton elimination, then $\Gamma^\circ \vdash_{sf} {c_1}^\circ = {c_2}^\circ : K^\circ$ is derivable in the singleton-free system. A slightly stronger version of this fact is formalized as Lemma 25 in Section 4.4.

Thus, our goal is to rewrite constructors in such a manner that the singleton elimination rule is not necessary. As mentioned in the introduction, this rewriting

[2] As an aside, in many module-oriented accounts [Harper and Lillibridge 1994; Lillibridge 1997; Leroy 1994; 1995] it is impossible to discover that the module analogues of these types are equal because comparisons can be made only on expressions in named form. Naming the expressions $\lambda\alpha{:}T.\alpha$ and $\lambda\alpha{:}T.\texttt{int}$ obscures the possible connection between them, which depends essentially on their actual code. (In the first-class account of Harper and Lillibridge [Harper and Lillibridge 1994; Lillibridge 1997] this is essential because the equality may not hold—in addition to being impossible to discover—since a functor can inspect the store before deciding what type to return.) This is an example of when the singleton kind account can propagate more type information than those module-oriented accounts. However, it is possible to give a module-oriented account that does propagate as much type information as the singleton kind account [Dreyer et al. 2003].

$$
\begin{aligned}
T^\circ &\stackrel{\text{def}}{=} T \\
S(c)^\circ &\stackrel{\text{def}}{=} T \\
(\Pi\alpha{:}K_1.K_2)^\circ &\stackrel{\text{def}}{=} {K_1}^\circ \to {K_2}^\circ \\
(\Sigma\alpha{:}K_1.K_2)^\circ &\stackrel{\text{def}}{=} {K_1}^\circ \times {K_2}^\circ
\end{aligned}
$$

Fig. 4. Singleton Erasure

is done by substituting definitions for variables whenever singleton kinds provide such definitions. This works out quite simply in first-order cases, but higher-order cases raise some subtle issues. We will explore these issues by considering a number of examples before defining the fully general elimination process.

3.0.0.1 *Example 1.* Suppose we are working under the assignment $\alpha{:}S(\texttt{int})$, $\beta{:}S(\texttt{bool})$. Naturally, we replace all free appearances of α in the constructor in question by `int`, and replace all free appearances of β by `bool`. This is done simply by performing the substitution $\{\texttt{bool}/\beta\}\{\texttt{int}/\alpha\}$ on the constructor in question.

In this example, we refer to `int` as the *expansion* of α, and likewise `bool` is the expansion of β. In general, the elimination process will have the same gross structure as in this example. For an assignment $\Gamma = \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n$ we will define a substitution $R(\Gamma)$ of the form $\{c_n/\alpha_n\}\cdots\{c_1/\alpha_1\}$ where each c_i is the expansion of α_i.

3.0.0.2 *Example 2.* Suppose we are working under the assignment $\Gamma = \alpha{:}S(\texttt{int}), \beta{:}S(\alpha)$. In this case, analogously to the previous example, $R(\Gamma)$ is $\{\alpha/\beta\}\{\texttt{int}/\alpha\}$. Note that since this is a sequential substitution, it is equivalent to the substitution $\{\texttt{int}/\beta\}\{\texttt{int}/\alpha\}$, as one would expect.

3.0.0.3 *Example 3.* Suppose α is assigned the kind $S(\texttt{int}) \times S(\texttt{bool})$. In this case, $\pi_1\alpha$ is equal to `int` and $\pi_2\alpha$ is equal to `bool`. We can write these equalities into a constructor by substituting for α with the pair $\langle \texttt{int}, \texttt{bool}\rangle$.

3.0.0.4 *Example 4.* In the previous examples, the expansion of a variable α did not contain α, but this is not true in general. Suppose α is assigned the kind $T \times S(\texttt{int})$. In this case, $\pi_2\alpha$ is equal to `int`, but $\pi_1\alpha$ is not given a definition and should not be changed. We handle this by substituting for α with the pair $\langle \pi_1\alpha, \texttt{int}\rangle$.

As this example illustrates, a good way to understand expansions is to view them as eta-long forms[3] of constructors. This interpretation is precisely correct, provided we view the replacement of a constructor by its singleton definition as an eta-expansion. In fact, the ultimate definition of expansions will eta-expand constructors uniformly, so, for example, if α has kind $T \times T$, its expansion will be $\langle \pi_1\alpha, \pi_2\alpha\rangle$ (instead of just α). This uniformity will make the correctness proof simpler, but a practical implementation would probably optimize such cases.

[3]That is, beta-normal forms such that no eta-expansions can be performed without creating beta-redices.

3.0.0.5 *Example 5.* Suppose α is assigned the kind $\Sigma\beta{:}T.S(\beta)$. Then $\pi_2\alpha$ is known to be equal to $\pi_1\alpha$ (although its precise value is unknown). In this case the expansion of α is $\langle \pi_1\alpha, \pi_1\alpha\rangle$.

3.0.0.6 *Example 6.* Suppose α is assigned the kind $\Sigma\beta{:}S(\texttt{int}).S(\beta)$. In this case $\pi_1\alpha$ and $\pi_2\alpha$ are equal to $\texttt{int}$ and the expansion is $\langle \texttt{int}, \texttt{int}\rangle$.

Generally, if α has the kind $\Sigma\beta{:}K_1.K_2$, the expansion of α will be the pair $\langle c_1, c_2\rangle$ where c_1 is the expansion of $\pi_1\alpha$, and c_2 is the expansion of $\pi_2\alpha$ *with the additional information* that β refers to $\pi_1\alpha$ and has kind K_1. We may generalize all the examples so far with the following definition, where $R(c, K)$ is the expansion of c assuming c is known to have kind K:

$$\begin{aligned} R(c, T) &\stackrel{\text{def}}{=} c \\ R(c, S(c')) &\stackrel{\text{def}}{=} c' \\ R(c, \Sigma\alpha{:}K_1.K_2) &\stackrel{\text{def}}{=} \langle R(\pi_1 c, K_1), R(\pi_2 c, K_2\{R(\pi_1 c, K_1)/\alpha\}\rangle \end{aligned}$$

3.0.0.7 *Example 7.* Suppose α is assigned the kind $\Pi\beta{:}T.S(\texttt{list}\,\beta)$ (where $\texttt{list} : T \to T$). Then for any argument c, the application $\alpha\, c$ is equal to $\texttt{list}\, c$. Thus, the appropriate expansion of α is $\lambda\beta{:}T.\texttt{list}\,\beta$. Note that this is the eta-long form of $\texttt{list}$.

3.0.0.8 *Example 8.* Suppose α is assigned the kind $\Pi\beta{:}T.\,(T \times S(\beta))$. In this case, for any argument c, $\pi_2(\alpha\, c)$ is known to be equal to c, but no definition is given for $\pi_1(\alpha\, c)$. Thus, the expansion of α is $\lambda\beta{:}T.\langle \pi_1(\alpha\,\beta), \beta\rangle$.

These last two examples suggest the following generalization for product kinds:

$$R(c, \Pi\alpha{:}K_1.K_2) \;=\; \lambda\alpha{:}K_1.\, R(c\alpha, K_2) \qquad \text{(wrong)}$$

This is close to the right generalization, but, as we will see in the next section, it is not quite satisfactory due to the need to account for bound variables. Nevertheless, it provides good intuition on the process of expansion over product kinds.

3.1 Bound Variables

Thus far we have exclusively considered rewriting constructors to account for the kinds of their free variables. To be sure that no uses of the singleton elimination rule are necessary, we must also consider bound variables. For example, it would seem as though the constructor $\lambda\alpha{:}S(\texttt{int}).\alpha$ should be rewritten to something like $\lambda\alpha{:}S(\texttt{int}).\texttt{int}$.

A naive approach would be to traverse the constructor in question and replace every bound variable with its expansion resulting from the kind in its binding occurrence. For example, in $\lambda\alpha{:}S(\texttt{int}).\alpha$, the binding occurrence of α gives it kind $S(\texttt{int})$, so the α in the abstraction's body would be replaced by $R(\alpha, S(\texttt{int})) \equiv \texttt{int}$. However this traversal is not sufficient to account for all bound variables, nor in fact is it even necessary.

To see why a traversal is insufficient, suppose β has kind $(S(\texttt{int}) \to T) \to T$ and consider the constructors $\beta(\lambda\alpha{:}T.\alpha)$ and $\beta(\lambda\alpha{:}T.\texttt{int})$. (Recall Section 2.1.) In the former constructor, the binding occurrence of α gives it kind T, and consequently the hypothetical traversal would not replace it. However, as we saw in Section 2.1, the two constructors should be equal, and for this to happen without the singleton

elimination rule, α must be replaced by $\mathtt{int}$ in the former constructor. What this illustrates is that when an abstraction appears in an argument position, the abstraction's domain kind can sometimes be strengthened (in this case from T to $S(\mathtt{int})$). This means that the kind given in a variable's binding occurrence cannot be relied upon.

One possibility for dealing with this would be to perform a much more complicated traversal that attempts to determine the "true" kind for every bound variable. Fortunately, we may deal with this in a much simpler way by shifting the responsibility for expanding a bound variable from the abstraction where that variable is bound to all constructors that might consume that abstraction.

In the above example, β changes the effective domain of its arguments to $S(\mathtt{int})$; in other words, it promises only to call them with $\mathtt{int}$. The expansion process for product kinds makes this explicit. In this case, the expansion of β is $\lambda\gamma{:}(S(\mathtt{int}) \to T).\,\beta(\lambda\alpha{:}S(\mathtt{int}).\gamma\,\mathtt{int})$. After substituting this expansion for β, each of the constructors above normalizes to $\beta(\lambda\alpha{:}S(\mathtt{int}).\mathtt{int})$. This can again be seen as an eta-long form for β where replacement of a variable by its definition is considered an eta-expansion.

In general, the expansion that achieves this is:

$$R(c, \Pi\alpha{:}K_1.K_2) \stackrel{\text{def}}{=} \lambda\alpha{:}K_1.\,R(c\alpha, K_2)\{R(\alpha, K_1)/\alpha\}$$

Making this expansion part of the substitution for free variables accounts for all cases in which the kind of an abstraction (and therefore its domain kind) is given by some other constructor to which the abstraction is passed as an argument. The only other way a kind may be imposed on an abstraction is at the top level. Again recall Section 2.1 and consider the constructors $\lambda\alpha{:}T.\alpha$ and $\lambda\alpha{:}T.\mathtt{int}$. These constructors should be considered equivalent when compared as members of kind $S(\mathtt{int}) \to T$, but not as members of $T \to T$. Thus, the elimination process must be affected by the kinds in which a constructor is considered to lie.

This is neatly dealt with by (in addition to substituting expansions for free variables) expanding the entire constructor using the kind to which it belongs. Thus, when considered as members of $S(\mathtt{int}) \to T$, the two constructors above become $\lambda\alpha{:}S(\mathtt{int}).((\lambda\alpha{:}T.\alpha)\mathtt{int})$ and $\lambda\alpha{:}S(\mathtt{int}).((\lambda\alpha{:}T.\mathtt{int})\mathtt{int})$; each of which normalize to $\lambda\alpha{:}S(\mathtt{int}).\mathtt{int}$. However, when considered as members of $T \to T$, the two become $\lambda\alpha{:}T.((\lambda\alpha{:}T.\alpha)\alpha)$ and $\lambda\alpha{:}T.((\lambda\alpha{:}T.\mathtt{int})\alpha)$; each of which normalizes to its original form.

It is worth noting that the required top-level expansion adds very little complexity to the use of singleton elimination in practice. In this paper we have largely ignored the term-level constructs of the intermediate language in question, but, in fact, constructors lie within surrounding terms, and elimination of singleton kinds in constructors is part of an overall transformation on terms. Typically, constructors appearing within terms are simply types (the domain of a lambda, for example), and in such cases the top-level expansion has no effect at all (since $R(c, T) = c$). In other cases constructors may by considered to lie in more interesting kinds (such as with the argument to a constructor abstraction), but in all such cases the intended kind is clearly given by context and the top-level expansion is still easy to perform.

$$
\begin{aligned}
R(c, T) &\stackrel{\text{def}}{=} c \\
R(c, S(c')) &\stackrel{\text{def}}{=} c' \\
R(c, \Pi\alpha{:}K_1.K_2) &\stackrel{\text{def}}{=} \lambda\alpha{:}K_1.\, R(c\, R(\alpha, K_1), K_2\{R(\alpha, K_1)/\alpha\}) \\
&\quad \text{(where } \alpha \text{ is not free in } c \text{ or } K_1\text{)} \\
R(c, \Sigma\alpha{:}K_1.K_2) &\stackrel{\text{def}}{=} \langle R(\pi_1 c, K_1), R(\pi_2 c, K_2\{R(\pi_1 c, K_1)/\alpha\}\rangle \\
\\
R(\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n) &\stackrel{\text{def}}{=} \{R(\alpha_n, K_n)/\alpha_n\} \cdots \{R(\alpha_1, K_1)/\alpha_1\}
\end{aligned}
$$

Fig. 5. Expansions

3.2 The Elimination Process

The full definition of the expansion constructors[4] and substitutions is given in Figure 5. Using expansion, the singleton kind elimination proceeds in three steps: Given a constructor c considered to have kind K under assignment Γ, we first expand c, resulting in $R(c, K)$. Second, we substitute expansions for all free variables, resulting in $R(c, K)\{R(\Gamma)\}$. Third, we erase any remaining singleton kinds, resulting in $(R(c, K)\{R(\Gamma)\})^\circ$. This elimination process is easily seen to be terminating, since R is defined by induction over the structure of kinds.

We may state the following correctness theorem for the elimination process, which states that rewritten constructors will be equivalent if and only if the original constructors were equivalent:

THEOREM 1. *Suppose* $\Gamma \vdash c_1 : K$ *and* $\Gamma \vdash c_2 : K$. *Then* $\Gamma \vdash c_1 = c_2 : K$ *if and only if* $\Gamma^\circ \vdash_{sf} (R(c_1, K)\{R(\Gamma)\})^\circ = (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$.

The proof of the correctness theorem is the subject of the next section.

4. CORRECTNESS PROOF

The previous section's informal discussion motivates why we might expect the elimination process to be correct. Unfortunately, Theorem 1 is difficult to prove directly, because there are too many ways that a judgement might be derived, and those derivations have no particular structure in common. We may construct a meta-explanation why the proof is difficult by considering the theorem's implications. Since it is easy to determine equality of constructors in the singleton-free system, the theorem provides a simple test for equality: translate constructors into the singleton-free system and check that they are equal there. The theorem states that such a test is sound and complete. However, this also indicates that proving the theorem is at least as difficult as proving decidability of constructor equality in the full system. In essence, the proof in this paper uses the Stone-Harper equivalence algorithm (discussed below) to provide structure to equivalence derivations they do not otherwise have.

The decidability of constructor equality has already been shown by Stone and Harper [2000]. They provide an algorithm for deciding constructor equality and

[4]Expansion of constructors is shown to be well-defined by induction on the structure of the kind, ignoring the contents of singleton kinds.

prove that algorithm sound and complete using a Kripke-style logical relation. In addition to settling the decidability question, they provide a tool with which we may prove Theorem 1. One approach would be to follow Stone and Harper and prove the theorem directly using a logical relation. However, we need not go so far to take advantage of their result. In fact, we can use it almost entirely "off the shelf."

The proof works essentially by using Stone and Harper's algorithm to normalize the derivations of equality judgements. Given a derivable equality judgement, we use completeness of the algorithm to deduce the existence of a derivation in the *algorithmic system.* That derivation can have only one form, making it much easier to reason about. Thus, this proof can be regarded as "mostly syntactic"; it does rely indirectly on the construction of a model, but all the work with the model is encapsulated in Stone and Harper's completeness proof.

The only-if portion of the proof (the difficult part, as it turns out) is structured as follows:

(1) Suppose $\Gamma \vdash c_1 = c_2 : K$.

(2) Prove that constructors are equal to their expansions; that is, $\Gamma \vdash c_1 = R(c_1, K)\{R(\Gamma)\} : K$ and $\Gamma \vdash c_2 = R(c_2, K)\{R(\Gamma)\} : K$. By symmetry and transitivity it follows that the expansions are equal: $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} = R(c_2, K)\{R(\Gamma)\} : K$.

(3) By algorithmic completeness, deduce that there exists a derivation of the algorithmic judgement $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma \vdash R(c_2, K)\{R(\Gamma)\} : K$.

(4) Prove that singleton reduction (the algorithmic counterpart of the singleton elimination rule) is not used in the algorithmic derivation. This step is the heart of the proof.

(5) By algorithmic soundness, deduce that there exists a derivation of $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} = R(c_2, K)\{R(\Gamma)\} : K$ in which the singleton elimination rule (Rule 34) is not used (except within subderivations for kinding or subkinding judgements).

(6) Prove that therefore there exists a derivation of $\Gamma^\circ \vdash_{sf} (R(c_1, K)\{R(\Gamma)\})^\circ = (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$.

Once the only-if portion is proved, the converse is easily established. The converse's proof is discussed in Section 4.4.

4.1 Equality of expansions

We begin by establishing that well-formed constructors are equal to their expansions. We first state three propositions giving some properties of the inference system (these are proven in Stone and Harper [2000]), and then prove equality of expansions by a series of three lemmas.

Proposition 2 Regularity.

(1) If $\Gamma \vdash \mathcal{J}$ *then* $\Gamma \vdash$ ok.

(2) If $\Gamma \vdash c : K$ *then* $\Gamma \vdash K$ kind.

(3) If $\Gamma \vdash c_1 = c_2 : K$ *then* $\Gamma \vdash c_1 : K$ *and* $\Gamma \vdash c_2 : K$.

PROPOSITION 3.

(1) **(Weakening)** *If* $\Gamma_1, \Gamma_3 \vdash \mathcal{J}$ *and* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \text{ok}$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathcal{J}$.
(2) **(Reflexivity)** *If* $\Gamma \vdash c : K$ *then* $\Gamma \vdash c = c : K$.
(3) **(Kind reflexivity)** *If* $\Gamma \vdash K \text{ kind}$ *then* $\Gamma \vdash K = K$.
(4) **(Subkinding reflexivity)** *If* $\Gamma \vdash K_1 = K_2$ *then* $\Gamma \vdash K_1 \leq K_2$.
(5) **(Assignment reflexivity)** *If* $\Gamma \vdash \text{ok}$ *then* $\vdash \Gamma = \Gamma$.

PROPOSITION 4 SUBSTITUTION. *Suppose* $\Gamma \vdash c_1 = c_2 : K$. *Then:*

(1) *If* $\Gamma, \alpha{:}K, \Gamma' \vdash K_1 = K_2$ *then* $\Gamma, (\Gamma'\{c_1/\alpha\}) \vdash K_1\{c_1/\alpha\} = K_2\{c_2/\alpha\}$.
(2) *If* $\Gamma, \alpha{:}K, \Gamma' \vdash c_1' = c_2' : K'$ *then* $\Gamma, (\Gamma'\{c_1/\alpha\}) \vdash c_1'\{c_1/\alpha\} = c_2'\{c_2/\alpha\} : K'\{c_1/\alpha\}$.

LEMMA 5. $R(c, K)\{c'/\alpha\} \equiv R(c\{c'/\alpha\}, K\{c'/\alpha\})$

PROOF. By induction on K. □

LEMMA 6. *If* $\Gamma \vdash c : K$ *then* $\Gamma \vdash c = R(c, K) : K$.

PROOF. By induction on K.

Case 1: Suppose $K \equiv T$. Then $R(c, K) \equiv c$ and by reflexivity, $\Gamma \vdash c = c : K$.

Case 2: Suppose $K \equiv S(c')$. Then $R(c, K) \equiv c'$. By assumption, $\Gamma \vdash c : S(c')$, so by singleton elimination (Rule 34), $\Gamma \vdash c = c' : T$. Then by symmetry and Rule 35, $\Gamma \vdash c = c' : S(c')$.

Case 3: Suppose $K \equiv \Pi\alpha{:}K_1.K_2$. Choose α so that it does not appear in the domain of Γ or free in c. Then $R(c, K) \equiv \lambda\alpha{:}K_1.\, R(c\, R(\alpha, K_1), K_2\{R(\alpha, K_1)/\alpha\})$. Invoking Lemma 5, $R(c, K) \equiv \lambda\alpha{:}K_1.\, R(c\alpha, K_2)\{R(\alpha, K_1)/\alpha\}$.

By regularity and inversion, $\Gamma \vdash K_1$ kind, so by weakening, $\Gamma, \alpha{:}K_1 \vdash c : \Pi\alpha{:}K_1.K_2$. Thus $\Gamma, \alpha{:}K_1 \vdash c\alpha : K_2$. By induction, $\Gamma, \alpha{:}K_1 \vdash c\alpha = R(c\alpha, K_2) : K_2$. Also by induction, $\Gamma, \alpha{:}K_1 \vdash \alpha = R(\alpha, K_1) : K_1$. Then, by weakening and substitution, $\Gamma, \alpha{:}K_1 \vdash c\alpha = R(c\alpha, K_2)\{R(\alpha, K_1)/\alpha\} : K_2$. By product introduction (Rule 40), $\Gamma \vdash \lambda\alpha{:}K_1.c\alpha = R(c, K) : \Pi\alpha{:}K_1.K_2$.

It remains to show that $\Gamma \vdash c = \lambda\alpha{:}K_1.c\alpha : \Pi\alpha{:}K_1.K_2$. This may be shown using functionality (Rule 30) and beta reduction (Rule 29).

Case 4: Suppose $K \equiv \Sigma\alpha{:}K_1.K_2$. Choose α so that it does not appear in the domain of Γ or free in c. Then $R(c, K) \equiv \langle R(\pi_1 c, K_1), R(\pi_2 c, K_2\{R(\pi_1 c, K_1)/\alpha\}\rangle$. Note that by regularity and inversion, $\Gamma, \alpha{:}K_1 \vdash K_2$ kind.

By sum elimination (Rule 22), $\Gamma \vdash \pi_1 c : K_1$, so by induction, $\Gamma \vdash \pi_1 c = R(\pi_1 c, K_1) : K_1$. Also by sum elimination (Rule 23), $\Gamma \vdash \pi_2 c : K_2\{\pi_1 c/\alpha\}$. By reflexivity and substitution, $\Gamma \vdash K_2\{\pi_1 c/\alpha\} = K_2\{R(\pi_1 c, K_1)/\alpha\}$, and thus $\Gamma \vdash \pi_2 c : K_2\{R(\pi_1 c, K_1)/\alpha\}$. Then, by induction, $\Gamma \vdash \pi_2 c = R(\pi_2 c, K_2\{R(\pi_1 c, K_1)/\alpha\}) : K_2\{R(\pi_1 c, K_1)/\alpha\}$. By sum introduction (Rule 44) and symmetry, $\Gamma \vdash \langle \pi_1 c, \pi_2 c\rangle = R(c, K) : \Sigma\alpha{:}K_1.K_2$.

It remains to show that $\Gamma \vdash c = \langle \pi_1 c, \pi_2 c\rangle : \Sigma\alpha{:}K_1.K_2$. This may be shown using functionality (Rule 31) and beta reduction (Rules 32 and 33).

□

LEMMA 7. *If* $\Gamma \vdash c : K$ *then* $\Gamma \vdash c = R(c, K)\{R(\Gamma)\} : K$.

PROOF. The proof is by induction on Γ' that if $\Gamma, \Gamma' \vdash c : K$ then $\Gamma, \Gamma' \vdash c = R(c, K)\{R(\Gamma')\} : K$. For empty Γ', use Lemma 6. In the inductive case, suppose $\Gamma' \equiv \alpha{:}K', \Gamma''$. Then $R(\Gamma') \equiv R(\Gamma'')\{R(\alpha, K')/\alpha\}$. By induction, $\Gamma, \alpha{:}K', \Gamma'' \vdash c = R(c, K)\{R(\Gamma'')\} : K$. Since $\Gamma, \alpha : K', \Gamma'' \vdash \alpha : K'$, by Lemma 6 it follows that $\Gamma, \alpha{:}K', \Gamma'' \vdash \alpha = R(\alpha, K') : K'$. By weakening and substitution, $\Gamma, \alpha{:}K', \Gamma'' \vdash c\{\alpha/\alpha\} = R(c, K)\{R(\Gamma'')\}\{R(\alpha, K')/\alpha\} : K$. That is, $\Gamma, \Gamma' \vdash c = R(c, K)\{R(\Gamma')\} : K$. □

COROLLARY 8. *If* $\Gamma \vdash c_1 = c_2 : K$ *then* $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} = R(c_2, K)\{R(\Gamma)\} : K$

PROOF. By regularity, Lemma 7, symmetry and transitivity. □

4.2 The Decision Algorithm

Stone and Harper's decision algorithm for constructor equivalence is given in Figure 6. This algorithm is unusual in that it is a *six-place* algorithm; it maintains two assignments and two kinds. This allows the two halves of the algorithm to operate independently, which is critical to Stone and Harper's proof and to this one. In common usage, the two assignments and the two kinds are equivalent (but often not identical). The critical singleton reduction rule appears as the ninth clause.

Originally, Stone and Harper also proved their six-place algorithm equivalent to a conventional four-place algorithm employing judgements of the form $\Gamma \vdash c_1 \Leftrightarrow c_2 : K$, which is preferable in practice. In more recent work [Stone and Harper 2004], Stone and Harper have developed a much simpler proof that applies to the four-place algorithm directly. Nevertheless, it is the six-place algorithm that is most useful to our purposes here.

The (six-place) algorithm works as follows:

(1) The algorithm is presented with a query of the form[5] $\Gamma \vdash c : K \Leftrightarrow \Gamma' \vdash c' : K'$. When $\vdash \Gamma = \Gamma'$ and $\Gamma \vdash K = K'$, this determines whether $\Gamma \vdash c = c' : K$ is derivable.

(2) The constructor equivalence rules add appropriate elimination forms (applications or projections) to the constructors being compared in order to drive them down to kind T or a singleton kind. Then those constructors are reduced to weak head normal form.

(3) Elimination contexts (E) are defined in the usual manner, as shown below. A constructor of the form $E[\alpha]$ is referred to as a *path,* and α is called the *head* of the path. We will often use the metavariable p to range over paths.

$$E ::= [\,] \mid Ec \mid \pi_1 E \mid \pi_2 E$$

A constructor is reduced to weak head normal form by alternating beta reductions and singleton reductions. Beta reduction of a constructor c is performed by placing it in the form $E[c]$ where c is a beta redex, and reducing to $E[c']$ where c' is the corresponding contractum. Repetition of this will ultimately result in a path (if the constructor is well-formed, which is assumed).

[5]It is awkward to render six-place judgements in spoken language. My preferred rendering of the algorithmic judgement is "In assignments Γ and Γ', c and c' are related at kinds K and K'."

Natural kind extraction

$\Gamma \vdash \alpha \uparrow \Gamma(\alpha)$

$\Gamma \vdash b \uparrow T$

$\Gamma \vdash \pi_1 p \uparrow K_1$ if $\Gamma \vdash p \uparrow \Sigma\alpha{:}K_1.K_2$

$\Gamma \vdash \pi_2 p \uparrow K_2\{\pi_1 p/\alpha\}$ if $\Gamma \vdash p \uparrow \Sigma\alpha{:}K_1.K_2$

$\Gamma \vdash p\,c \uparrow K_2\{c/\alpha\}$ if $\Gamma \vdash p \uparrow \Pi\alpha{:}K_1.K_2$

Weak head reduction

$\Gamma \vdash E[(\lambda\alpha{:}K.c)c'] \longrightarrow E[c\{c'/\alpha\}]$

$\Gamma \vdash E[\pi_1\langle c_1, c_2\rangle] \longrightarrow E[c_1]$

$\Gamma \vdash E[\pi_2\langle c_1, c_2\rangle] \longrightarrow E[c_2]$

$\Gamma \vdash E[p] \longrightarrow E[c]$ if $\Gamma \vdash p \uparrow S(c)$ **(singleton reduction)**

Weak head normalization

$\Gamma \vdash c \Downarrow c'$ if $\Gamma \vdash c \longrightarrow c''$ and $\Gamma \vdash c'' \Downarrow c'$

$\Gamma \vdash c \Downarrow c$ otherwise

Algorithmic constructor equivalence

$\Gamma_1 \vdash c_1 : T \Leftrightarrow \Gamma_2 \vdash c_2 : T$ if $\Gamma_1 \vdash c_1 \Downarrow p_1$ and $\Gamma_2 \vdash c_2 \Downarrow p_2$ and $\Gamma_1 \vdash p_1 \uparrow T \leftrightarrow \Gamma_2 \vdash p_2 \uparrow T$

$\Gamma_1 \vdash c_1 : S(c_1') \Leftrightarrow \Gamma_2 \vdash c_2 : S(c_2')$ if $\Gamma_1 \vdash c_1 \Downarrow p_1$ and $\Gamma_2 \vdash c_2 \Downarrow p_2$ and $\Gamma_1 \vdash p_1 \uparrow T \leftrightarrow \Gamma_2 \vdash p_2 \uparrow T$

$\Gamma_1 \vdash c_1 : \Pi\alpha{:}K_1.K_1' \Leftrightarrow \Gamma_2 \vdash c_2 : \Pi\alpha{:}K_2.K_2'$ if $\Gamma_1, \alpha{:}K_1 \vdash c_1\alpha : K_1' \Leftrightarrow \Gamma_2, \alpha{:}K_2 \vdash c_2\alpha : K_2'$

$\Gamma_1 \vdash c_1 : \Sigma\alpha{:}K_1.K_1' \Leftrightarrow \Gamma_2 \vdash c_2 : \Sigma\alpha{:}K_2.K_2'$ if $\Gamma_1 \vdash \pi_1 c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash \pi_1 c_2 : K_2$ and $\Gamma_1 \vdash \pi_2 c_1 : K_1'\{\pi_1 c_1/\alpha\} \Leftrightarrow \Gamma_2 \vdash \pi_2 c_2 : K_2'\{\pi_2 c_2/\alpha\}$

Algorithmic path equivalence

$\Gamma_1 \vdash \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \vdash \alpha \uparrow \Gamma_2(\alpha)$

$\Gamma_1 \vdash b_1 \uparrow T \leftrightarrow \Gamma_2 \vdash b_2 \uparrow T$ if $b_1 \equiv b_2$

$\Gamma_1 \vdash p_1 c_1 \uparrow K_1'\{c_1/\alpha\} \leftrightarrow \Gamma_2 \vdash p_2 c_2 \uparrow K_2'\{c_2/\alpha\}$ if $\Gamma_1 \vdash p_1 \uparrow \Pi\alpha{:}K_1.K_1' \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Pi\alpha{:}K_2.K_2'$ and $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash c_2 : K_2$

$\Gamma_1 \vdash \pi_1 p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash \pi_1 p_2 \uparrow K_2$ if $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_1.K_1' \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_2.K_2'$

$\Gamma_1 \vdash \pi_2 p_1 \uparrow K_1'\{\pi_1 p_1/\alpha\} \leftrightarrow \Gamma_2 \vdash \pi_2 p_2 \uparrow K_2'\{\pi_1 p_2/\alpha\}$ if $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_1.K_1' \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_2.K_2'$

Fig. 6. Constructor Equivalence Algorithm (Six-Place Version)

(4) Singleton reduction of a path p is performed by determining its *natural kind*, and replacing p with c whenever p's natural kind is some singleton kind $S(c)$. (Formally, the algorithm adds an elimination context, reducing $E[p]$ to $E[c]$ when p has natural kind c, but E will be empty when $E[p]$ is well-formed.)

Note that the natural kind of a path is *not* a principal kind. For example, if $\Gamma(\alpha) = T$ then the natural kind of α is T, but α has principal kind $S(\alpha)$.

(5) When no more beta or singleton reductions apply, the algorithm compares the two paths, checking that they have the same head variable and the same series of eliminations. When checking that two applications are the same, the main algorithm is reinvoked to determine whether the arguments are equal.

We may state the following correctness theorem for the algorithm:

THEOREM 9 STONE-HARPER.

(1) **(Completeness)** *If* $\Gamma \vdash c_1 = c_2 : K$ *then* $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma \vdash c_2 : K$.
(2) **(Soundness)** *Suppose* $\vdash \Gamma = \Gamma'$, $\Gamma \vdash K = K'$, $\Gamma \vdash c_1 : K$ *and* $\Gamma' \vdash c_2 : K'$. *Then if* $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma' \vdash c_2 : K'$ *then* $\Gamma \vdash c_1 = c_2 : K$.

COROLLARY 10. *If* $\Gamma \vdash c_1 = c_2 : K$ *then* $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma \vdash R(c_2, K)\{R(\Gamma)\} : K$.

There is one minor difference between this algorithm and the one presented in Stone and Harper. When checking constructor equivalence at a singleton kind, Stone and Harper's algorithm immediately succeeds, while the algorithm here behaves the same as when comparing at kind T. However, Stone and Harper's proof goes through in almost exactly the same way, with only a change to one subcase of their "Main Lemma." Their algorithm is more efficient, since it terminates early in some cases, but for our purposes we are not concerned with efficiency. The advantage of this version of the algorithm is that we may obtain the stronger version of soundness given in Theorem 12:

DEFINITION 11. *A derivation is* mostly free of singleton elimination *if every use of singleton elimination (Rule 34) in that derivation lies within a subderivation whose root is a constructor formation or subkinding judgement.*

THEOREM 12 SINGLETON-FREE SOUNDNESS. *Suppose* $\vdash \Gamma = \Gamma'$, $\Gamma \vdash K = K'$, $\Gamma \vdash c_1 : K$ *and* $\Gamma' \vdash c_2 : K'$. *Then if* $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma' \vdash c_2 : K'$ *without using singleton reduction then there exists a derivation of* $\Gamma \vdash c_1 = c_2 : K$ *that is mostly free of singleton elimination.*

PROOF. By inspection of Stone and Harper's proof. □

Theorem 12 fails with the more efficient version of the algorithm because when $\Gamma_1 \vdash c_1 : S(c_1') \Leftrightarrow \Gamma_2 \vdash c_2 : S(c_2')$, the soundness proof must use singleton elimination to show that c_1 and c_1' are equal and that c_2 and c_2' are equal, in the course of showing that c_1 and c_2 are equal.

In the next section we will show that the algorithmic derivation shown to exist by Corollary 10 is free of singleton reduction. Then Theorem 12 will permit us to conclude that the corresponding derivation in the declarative system is mostly free of singleton elimination. A derivation mostly free of singleton elimination uses singleton elimination in no significant manner; any residual uses (within constructor formation or subkinding) will be removed by singleton erasure in Section 4.4.

4.3 Absence of singleton reduction

The heart of the proof is to show that singleton reduction will not be used in a derivation of algorithmic equivalence of expanded constructors. It is here that we really show that expansion works to eliminate singleton kinds: if the algorithm is able to deduce that the two expanded terms are equal without using singleton reduction, then we have obviated the need for singleton kinds.

The proof works by defining a condition, called *protectedness,* that is satisfied by expanded constructors, that rules out any need for singleton reduction, and that is preserved by the algorithm. First we make some preliminary definitions:

Definition 13.

—*Two kinds K and K' are* similar *(written $K \approx K'$) if they are the same modulo the contents of singleton kinds. That is, similarity is the least congruence such that $S(c) \approx S(c')$ for any constructors c and c'.*

—*Two assignments Γ and Γ' are* similar *(written $\Gamma \approx \Gamma'$) if they bind the same variables in the same order, and if $\Gamma(\alpha) \approx \Gamma'(\alpha)$ for all $\alpha \in \mathrm{Dom}(\Gamma)$.*

Note that a well-formed kind can be similar to an ill-formed kind, and likewise for assignments. When two kinds or two assignments are similar, they are said to have the same shape. For the proof of the absence of singleton reductions, we will be able to disregard the actual kinds and assignments being used and consider only their shapes; this will simplify the proof considerably. This works because the contents of singleton kinds are only pertinent to singleton reduction, which we are showing never takes place.

We also define *contexts* (C) as shown below. Note that contexts are defined to have exactly one hole, and note also that elimination contexts are a subclass of contexts. As we are not concerned with the contents of singleton kinds, there is no need for contexts to account for constructors appearing within the domain kind of a lambda abstraction. Instantiation of a context is defined in the usual manner; in particular, it is permissible for instantiation to capture free variables.

$$C \ ::= \ [\,] \mid \lambda\alpha{:}K.C \mid C\,c \mid c\,C \mid \langle C, c\rangle \mid \langle c, C\rangle \mid \pi_1 C \mid \pi_2 C$$

Finally, we define weak head reduction without an assignment[6] in the usual manner (that is, $E[(\lambda\alpha{:}K.c)c'] \longrightarrow E[c\{c'/\alpha\}]$ and $E[\pi_i\langle c_1, c_2\rangle] \longrightarrow E[c_i]$). Note that if $c_1 \longrightarrow c_2$ then $\Gamma \vdash c_1 \longrightarrow c_2$ (recall algorithmic weak head reduction).

We are now ready to define the protectedness property. The intuition is that a constructor is protected if every variable in that constructor appears in an elimination context (equivalently, at the head of a path) that drives it down to kind T (*i.e.,* that performs elimination operations on it resulting in a constructor of kind T). By implication, this means that no variable appears in an elimination context driving it down to a singleton kind. In other words, no path within the constructor will have a singleton natural kind and consequently singleton reduction will not take place.

In order to ensure that protectedness is preserved by the algorithm, we must strengthen the condition a bit. When the algorithm processes a constructor $\lambda\alpha{:}K_1.c$ of kind $\Pi\alpha{:}K_1.K_2$ in context Γ, the body c will be processed in the extended context $\Gamma, \alpha{:}K_1$, not simply Γ. Thus, we wish to ensure that c is protected not merely at Γ, but at $\Gamma, \alpha{:}K_1$. Thus, we define a stronger condition of *protectedness relative to a kind* that extends the context appropriately for constructors of Π kind. This stronger condition is then the primary condition used in the main lemma.

Additionally, each path used (in the definition of protectedness) to drive a variable to kind T will ultimately be processed by the algorithm, and so each constructor appearing as an argument in that path must be protected at the pertinent kind. We

[6]As opposed to the algorithm's judgement $\Gamma \vdash c_1 \longrightarrow c_2$ for weak head reduction within an assignment Γ.

refer to such a path as *appropriate,* and require, in the definition of protectedness, that the path that drives a variable to kind T must be appropriate.

DEFINITION 14. *Suppose* Γ *is an assignment and* K *is a kind. The relations* Γ-protected, K-Γ-appropriate, *and* K-Γ-protected *are the least relations such that:*

(*1*) **Protectedness**

—*A constructor* c *is* Γ*-protected if whenever* $c \equiv C[\alpha]$ *(where* $\alpha \in \mathrm{Dom}(\Gamma)$ *and* C *does not capture* α*), there exist* C' *and* E *such that* $C[\,] \equiv C'[E[\,]]$*, and* $E[\alpha]$ *is* T-Γ*-appropriate.*

(*2*) **Appropriateness**

—*A path* α *is* K-Γ*-appropriate if* $\Gamma(\alpha) \approx K$*.*

—*A path* $p\,c$ *is* K_2-Γ*-appropriate if* p *is* $(\Pi\alpha{:}K_1.K_2)$-Γ*-appropriate and* c *is* K_1-Γ*-protected.*

—*A path* $\pi_1 p$ *is* K_1-Γ*-appropriate if* p *is* $(\Sigma\alpha{:}K_1.K_2)$-Γ*-appropriate.*

—*A path* $\pi_2 p$ *is* K_2-Γ*-appropriate if* p *is* $(\Sigma\alpha{:}K_1.K_2)$-Γ*-appropriate.*

(*3*) **Protectedness relative to a kind**

—*A constructor* c *is* T-Γ*-protected if* c *is* Γ*-protected.*

—*A constructor* c *is* $S(c'')$-Γ*-protected if* c *is* Γ*-protected.*

—*A lambda abstraction* $\lambda\alpha{:}K_1'.c$ *is* $(\Pi\alpha{:}K_1.K_2)$-Γ*-protected if* c *is* K_2-$(\Gamma, \alpha{:}K_1)$*-protected.*

—*A pair* $\langle c_1, c_2\rangle$ *is* $(\Sigma\alpha{:}K_1.K_2)$-Γ*-protected if* c_1 *is* K_1-Γ*-protected and* c_2 *is* K_2-Γ*-protected.*

Note that the relations being defined appear only positively above, so Definition 14 is a valid inductive definition. Also, note that these definitions are concerned with kinds only up to similarity, and for this reason the definition can safely ignore the presence of free variables in kinds and assignments. We may immediately observe a number of easy structural facts about these definitions:

LEMMA 15.

(1) *Suppose* $\Gamma \approx \Gamma'$ *and* $K \approx K'$*, then*
   —*c is* Γ*-protected if and only if c is* Γ'*-protected,*
   —*c is* K-Γ*-protected if and only if c is* K'-Γ'*-protected, and*
   —*p is* K-Γ*-appropriate if and only if p is* K'-Γ'*-appropriate.*
(2) *If c is* Γ*-protected then* $\lambda\alpha{:}K.c$*,* $\pi_1 c$*, and* $\pi_2 c$ *are* Γ*-protected.*
(3) *If* c_1 *and* c_2 *are* Γ*-protected then* $c_1 c_2$ *and* $\langle c_1, c_2\rangle$ *are* Γ*-protected.*
(4) *If* $E[\lambda\alpha{:}K.c]$ *is* Γ*-protected then c is* Γ*-protected.*
(5) *If* $E[c_1 c_2]$ *is* Γ*-protected then* c_2 *is* Γ*-protected.*
(6) *If* $E[\langle c_1, c_2\rangle]$ *is* Γ*-protected, then* c_1 *and* c_2 *are* Γ*-protected.*
(7) *Any constructor is* ϵ*-protected.*
(8) *If c is* $(\Gamma \setminus \alpha)$*-protected and* α *is not free in c, then c is* Γ*-protected.*
(9) *If c is* Γ*-protected then c is* $(\Gamma \setminus \alpha)$*-protected.*
(10) *If c is* K-Γ*-protected then c is* Γ*-protected.*

PROOF. Parts 1–3 and 7–10 are by inspection. For part 4 observe that any path with its head in c lies entirely within c. Likewise for part 5 observe that any path with its head in c_2 lies entirely within c_2, and similarly for part 6. □

In order to show that protectedness is preserved by the algorithm, we need to show that it is preserved by weak head reduction. To show this we must first establish a substitution lemma. To do so, we will have need of the fact that any subexpression of a substitution results from one or the other participant in the substitution:

LEMMA 16. *If* $C[c] = c_1\{c_2/\alpha\}$ *and* C *does not capture* α *then either*

—*there exist contexts* C_1 *and* C_2 *such that* $c_1 \equiv C_1[\alpha]$*,* $c_2 \equiv C_2[c]$ *and* $C[\,] \equiv (C_1\{c_2/\alpha\})[C_2[\,]]$ *(that is, c results from* c_2*), or*
—*there exists a context* C_1 *and a constructor* c' *such that* $c_1 \equiv C_1[c']$*,* $c \equiv c'\{c_2/\alpha\}$*, and* $C[\,] \equiv (C_1\{c_2/\alpha\})[\,]$ *(that is, c results from some* c' *in* c_1*).*

PROOF. By induction on c_1. If C is empty then the second case is satisfied by $C_1[\,] \equiv [\,]$ and $c' \equiv c_1$. Therefore assume C is nonempty.

Case 1: Suppose $c_1 \equiv \alpha$. Then the first case is satisfied by $C_1[\,] \equiv [\,]$ and $C_2[\,] \equiv C[\,]$.

Case 2: Suppose $c_1 \equiv \beta$ where $\beta \not\equiv \alpha$. Then $C[c] \equiv \beta$, which is impossible since C is nonempty.

Case 3: Suppose $c_1 \equiv \lambda\beta{:}K.c_1'$. Then $C[\,] \equiv \lambda\beta{:}(K\{c_2/\alpha\}).(C'[\,])$. Since C does not capture α, it follows that $\beta \not\equiv \alpha$. Note that $C'[c] \equiv c_1'\{c_2/\alpha\}$. We proceed by case analysis using the induction hypothesis on $C'[c]$:

Subcase 3.1: Suppose there exist contexts C_1' and C_2 such that $c_1' \equiv C_1'[\alpha]$, $c_2 \equiv C_2[c]$ and $C'[\,] \equiv (C_1'\{c_2/\alpha\})[C_2[\,]]$. Then the first case is satisfied by $C_1[\,] \equiv \lambda\beta{:}K.(C_1'[\,])$.

Subcase 3.2: Suppose there exists a context C_1' and a constructor c' such that $c_1' \equiv C_1'[c']$, $c \equiv c'\{c_2/\alpha\}$, and $C'[\,] \equiv (C_1'\{c_2/\alpha\})[\,]$. Then the second case is satisfied by $C_1[\,] \equiv \lambda\beta{:}K.(C_1'[\,])$.

Case 4: Suppose $c_1 \equiv c_1' c_1''$. The remaining cases are similar. Then $C[\,]$ is either $(C'[\,])(c_1''\{c_2/\alpha\})$ or $(c_1'\{c_2/\alpha\})(C'[\,])$. Suppose the former; the latter is similar. Note that $C'[c] \equiv c_1'\{c_2/\alpha\}$. We proceed by case analysis using the induction hypothesis on $C'[c]$:

Subcase 4.1: Suppose there exist contexts C_1' and C_2 such that $c_1' \equiv C_1'[\alpha]$, $c_2 \equiv C_2[c]$ and $C'[\,] \equiv (C_1'\{c_2/\alpha\})[C_2[\,]]$. Then the first case is satisfied by $C_1[\,] \equiv (C_1'[\,])c_1''$.

Subcase 4.2: Suppose there exists a context C_1' and a constructor c' such that $c_1' \equiv C_1'[c']$, $c \equiv c'\{c_2/\alpha\}$, and $C'[\,] \equiv (C_1'\{c_2/\alpha\})[\,]$. Then the second case is satisfied by $C_1[\,] \equiv (C_1'[\,])c_1''$.

□

Lemma 17 Substitution.

(1) *If c_1 is Γ-protected and c_2 is Γ-protected, then $c_1\{c_2/\alpha\}$ is Γ-protected.*

(2) *If p is K-Γ-appropriate, c_2 is Γ-protected and α is not the head of p, then $p\{c_2/\alpha\}$ is K-Γ-appropriate.*

(3) *If c_1 is K-Γ-protected and c_2 is Γ-protected, then $c_1\{c_2/\alpha\}$ is K-Γ-protected.*

Proof. The proof is by induction on the derivation of the first assumption (*i.e.*, c_1 being Γ-protected, p being K-Γ-appropriate, or c_1 being K-Γ-protected, respectively.) We show part 1; the other two parts are easy using an inner induction on K.

We may assume, without loss of generality, that $\alpha \notin \mathrm{Dom}(\Gamma)$, if necessary by replacing α with a fresh variable and re-establishing protectedness of c_1 using Lemma 15 (parts 8 and 9). Suppose $C[\beta] \equiv c_1\{c_2/\alpha\}$, $\beta \in \mathrm{Dom}(\Gamma)$, and C does not capture β. By assumption, $\alpha \not\equiv \beta$, so we may alpha-vary $C[\beta]$ as necessary to ensure that C does not capture α. We proceed by case analysis using Lemma 16:

Case 1: Suppose $c_1 \equiv C_1[\alpha]$, $c_2 \equiv C_2[\beta]$ and $C[\,] \equiv (C_1\{c_2/\alpha\})[C_2[\,]]$. Since c_2 is Γ-protected, there exists C_2' and E such that $C_2[\,] \equiv C_2'[E[\,]]$ and $E[\beta]$ is T-Γ-appropriate. Then $C[\,] \equiv C'[E[\,]]$ where $C'[\,]$ is $(C_1\{c_2/\alpha\})[C_2'[\,]]$.

Case 2: Suppose $c_1 \equiv C_1[c']$, $\beta \equiv c'\{c_2/\alpha\}$, $C[\,] \equiv (C_1\{c_2/\alpha\})[\,]$. The constructor c' must be either α or β. In the former case, $c_2 \equiv \beta$, and since c_2 is Γ-protected, it follows that protection is satisfied by setting C' to C and E to empty. Therefore, assume $c' \equiv \beta$.

Then c_1 is of the form $C_1[\beta]$ where C_1 does not capture β (since C does not). Since c_1 is Γ-protected, there must exist C_1' and E such that $C_1[\,] \equiv C_1'[E[\,]]$ and $E[\beta]$ is T-Γ-appropriate. By induction, $E[\beta]\{c_2/\alpha\}$ is T-Γ-appropriate. Then $C[\,] \equiv C'[E'[\,]]$ where $C'[\,]$ is $(C_1'\{c_2/\alpha\})[\,]$ and E' is $(E\{c_2/\alpha\})[\,]$. □

Corollary 18. *If c_1 is Γ-protected and $c_1 \longrightarrow c_2$ then c_2 is Γ-protected.*

Proof. We prove that if $E_{\mathrm{out}}[c_1]$ is Γ-protected and $c_1 \longrightarrow c_2$ then c_2 is Γ-protected. The result follows by setting $E_{\mathrm{out}} \equiv [\,]$. Let c_1 be $E[c_1']$ and c_2 be $E[c_2']$, where c_1' is a redex and c_2' is its contractum. The proof is by induction on E.

Case 1: Suppose $E \equiv [\,]$ and $c_1' \equiv (\lambda\alpha{:}K.c)c'$. By Lemma 15 (parts 4 and 5), c and c' are Γ-protected. By Lemma 17, $c\{c'/\alpha\}$ is Γ-protected.

Case 2: Suppose $E \equiv [\,]$ and $c_1' \equiv \pi_i\langle c_1, c_2\rangle$. By Lemma 15 (part 6), c_i is Γ-protected.

Case 3: Suppose $E \equiv E'c$. Then $E'[c_1'] \longrightarrow E'[c_2']$ so, by induction, $E'[c_2']$ is Γ-protected. By Lemma 15 (part 5), c is Γ-protected, so $E'[c_2']\,c$ is Γ-protected.

Case 4: Suppose $E \equiv \pi_i E'$. Then $E'[c_1'] \longrightarrow E'[c_2']$ so, by induction, $E'[c_2']$ is Γ-protected. Thus $\pi_i E'[c_2']$ is Γ-protected. □

We will also need a technical lemma regarding natural kind extraction:

LEMMA 19.

(1) *If p is K-Γ-appropriate and $\Gamma \vdash p \uparrow K'$ then $K \approx K'$.*
(2) *If $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow K_2$ then $\Gamma_1 \vdash p_1 \uparrow K_1$ and $\Gamma_2 \vdash p_2 \uparrow K_2$.*

PROOF. Part 1 is by induction on K. Part 2 is by induction on the derivation. □

We are now ready to prove the main lemma:

LEMMA 20 MAIN LEMMA.

(1) *If $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash c_2 : K_2$ is derivable, $c_1 \longrightarrow^* c_1'$, $c_2 \longrightarrow^* c_2'$, c_1' is K_1-Γ_1-protected, and c_2' is K_2-Γ_2-protected, then the derivation does not use singleton reduction.*
(2) *If $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow K_2$ is derivable, c_1 is K_1-Γ_1-appropriate, and c_2 is K_2-Γ_2-appropriate, then the derivation does not use singleton reduction.*

PROOF. By induction on the algorithmic derivation.

Case 1: Suppose the derivation's root is $\Gamma_1 \vdash c_1 : T \Leftrightarrow \Gamma_2 \vdash c_2 : T$. Then $\Gamma_1 \vdash c_1 \Downarrow p_1$, $\Gamma_2 \vdash c_2 \Downarrow p_2$, and $\Gamma_1 \vdash p_1 \uparrow T \leftrightarrow \Gamma_2 \vdash p_2 \uparrow T$. By the definitions of weak head normalization and reduction, it follows either that $c_1 \longrightarrow^* p_1$ or that $c_1 \longrightarrow^* E[p_1']$, $\Gamma_1 \vdash p_1' \uparrow S(c_1'')$, and $\Gamma_1 \vdash E[c_1''] \Downarrow p_1$. In either case c_1 beta weak head reduces to a path, so let $c_1 \longrightarrow^* p$. Since weak head reduction is deterministic and p is in (beta) weak head normal form, it follows that $c_1' \longrightarrow^* p$. By assumption c_1' is Γ_1-protected, so by Corollary 18, p is Γ_1-protected.

Suppose p singleton reduces and let p be $E[\alpha]$. Then there exist E_1 and E_2 such that $E[\,] \equiv E_1[E_2[\,]]$ and $\Gamma_1 \vdash E_2[\alpha] \uparrow S(c)$. Since p is Γ_1-protected, there also exist E_1' and E_2' such that $E[\,] \equiv E_1'[E_2'[\,]]$ and $E_2'[\alpha]$ is T-Γ_1-appropriate. One of $E_2[\alpha]$ and $E_2'[\alpha]$ must be a subpath of the other and both cases lead to a contradiction. If $E_2'[\alpha]$ is a subpath of $E_2[\alpha]$ then $\Gamma_1 \vdash E_2'[\alpha] \uparrow K$ for some K, but $K \approx T$ by Lemma 19 so it cannot be the case that $\Gamma_1 \vdash E_2[\alpha] \uparrow S(c)$. If $E_2[\alpha]$ is a subpath of $E_2'[\alpha]$ then $E_2[\alpha]$ is K-Γ_1-appropriate for some K, but $K \approx S(c)$ by Lemma 19 so it cannot be the case that $E_2'[\alpha]$ is T-Γ-appropriate.

Hence p does not singleton reduce, and consequently $c_1 \longrightarrow^* p_1$ and p_1 is Γ_1-protected. Again let p_1 be $E[\alpha]$. Since p_1 is Γ_1-protected, there exist E_1 and E_2 such that $E[\,] \equiv E_1[E_2[\,]]$ and $E_2[\alpha]$ is T-Γ_1-appropriate. Since $\Gamma_1 \vdash E_1[E_2[\alpha]] \uparrow T \leftrightarrow \Gamma_2 \vdash p_2 \uparrow T$, by Lemma 19 (part 1) $\Gamma_1 \vdash E_1[E_2[\alpha]] \uparrow T$, and therefore that $\Gamma_1 \vdash E_2[\alpha] \uparrow K$ for some K. By Lemma 19 (part 2), $K \approx T$, which means that E_1 must be empty. Therefore, p_1 is T-Γ_1-appropriate. Similarly $c_2 \longrightarrow^* p_2$ and p_2 is T-Γ_2-appropriate. The result follows by induction.

Case 2: Suppose the derivation's root is $\Gamma_1 \vdash c_1 : S(c_1') \Leftrightarrow \Gamma_2 \vdash c_2 : S(c_2')$. This case is identical to the previous case.

Case 3: Suppose the derivation's root is $\Gamma_1 \vdash c_1 : \Pi\alpha{:}K_{11}.K_{12} \Leftrightarrow \Gamma_2 \vdash c_2 : \Pi\alpha{:}K_{21}.K_{22}$. By assumption, $c_1 \longrightarrow^* c_1'$ and c_1' is of the form $\lambda\alpha{:}K_{11}'.c_1''$ where c_1''

is K_{12}-($\Gamma_1, \alpha{:}K_{11}$)-protected. Then $c_1\alpha \longrightarrow^* c_1''$. Similarly, $c_2\alpha \longrightarrow^* c_2''$ for some K_{22}-($\Gamma_2, \alpha{:}K_{21}$)-protected c_2''. The result follows by induction.

Case 4: Suppose the derivation's root is $\Gamma_1 \vdash c_1 : \Sigma\alpha{:}K_{11}.K_{12} \Leftrightarrow \Gamma_2 \vdash c_2 : \Sigma\alpha{:}K_{21}.K_{22}$. By assumption, $c_1 \longrightarrow^* c_1'$ and c_1' is of the form $\langle c_{11}, c_{12}\rangle$ where c_{11} is K_{11}-Γ_1-protected and c_{12} is K_{12}-Γ_1-protected. Then $\pi_1 c_1 \longrightarrow^* c_{11}$ and $\pi_2 c_1 \longrightarrow^* c_{12}$. Since $K_{12} \approx K_{12}\{\pi_1 c_1/\alpha\}$, it follows that c_{12} is $(K_{12}\{\pi_1 c_1/\alpha\})$-$\Gamma_1$-protected. Similarly, $\pi_1 c_2 \longrightarrow^* c_{21}$ and $\pi_2 c_2 \longrightarrow^* c_{22}$ for some K_{21}-Γ_2-protected c_{21} and some $(K_{22}\{\pi_1 c_2/\alpha\})$-$\Gamma_2$-protected c_{22}. The result follows by induction.

Case 5: Suppose the derivation's root is $\Gamma_1 \vdash \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \vdash \alpha \uparrow \Gamma_2(\alpha)$. The result follows trivially.

Case 6: Suppose the derivation's root is $\Gamma_1 \vdash b \uparrow T \leftrightarrow \Gamma_2 \vdash b \uparrow T$. The result follows trivially.

Case 7: Suppose the derivation's root is $\Gamma_1 \vdash p_1 c_1 \uparrow K_{12}\{c_1/\alpha\} \leftrightarrow \Gamma_2 \vdash p_2 c_2 \uparrow K_{22}\{c_2/\alpha\}$. Then $\Gamma_1 \vdash p_1 \uparrow \Pi\alpha{:}K_{11}.K_{12} \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Pi\alpha{:}K_{21}.K_{22}$ and $\Gamma_1 \vdash c_1 : K_{11} \Leftrightarrow \Gamma_2 \vdash c_2 : K_{21}$. Since (invoking Lemma 15 (part 1)) $p_1 c_1$ is K_{12}-Γ_1-appropriate, it follows that p_1 is $(\Pi\alpha{:}K_{11}'.K_{12})$-$\Gamma_1$-appropriate and c_1 is K_{11}'-Γ_1-protected, for some K_{11}'. However, by Lemma 19 it follows that $K_{11} \approx K_{11}'$. Thus, p_1 is $(\Pi\alpha{:}K_{11}.K_{12})$-$\Gamma_1$-appropriate and c_1 is K_{11}-Γ_1-protected. Similarly, p_2 is $(\Pi\alpha{:}K_{21}.K_{22})$-$\Gamma_2$-appropriate and c_2 is K_{21}-Γ_2-protected. The result follows by induction.

Case 8: Suppose the derivation's root is $\Gamma_1 \vdash \pi_1 p_1 \uparrow K_{11} \leftrightarrow \Gamma_2 \vdash \pi_1 p_2 \uparrow K_{21}$. Then $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_{11}.K_{12} \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_{21}.K_{22}$. Since $\pi_1 p_1$ is K_{11}-Γ_1-appropriate, it follows that p_1 is $(\Sigma\alpha{:}K_{11}.K_{12}')$-$\Gamma_1$-appropriate. However, by Lemma 19 it follows that $K_{12} \approx K_{12}'$. Thus, p_1 is $(\Sigma\alpha{:}K_{11}.K_{12})$-$\Gamma_1$-appropriate. Similarly, p_2 is $(\Sigma\alpha{:}K_{21}.K_{22})$-$\Gamma_2$-appropriate. The result follows by induction.

Case 9: Suppose the derivation's root is $\Gamma_1 \vdash \pi_2 p_1 \uparrow K_{12}\{\pi_1 p_1/\alpha\} \leftrightarrow \Gamma_2 \vdash \pi_2 p_2 \uparrow K_{22}\{\pi_1 p_2/\alpha\}$. Then $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_{11}.K_{12} \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_{21}.K_{22}$. Since (invoking Lemma 15 (part 1)) $\pi_2 p_1$ is K_{12}-Γ_1-appropriate, it follows that p_1 is $(\Sigma\alpha{:}K_{11}'.K_{12})$-$\Gamma_1$-appropriate. However, by Lemma 19 it follows that $K_{11} \approx K_{11}'$. Thus, p_1 is $(\Sigma\alpha{:}K_{11}.K_{12})$-$\Gamma_1$-appropriate. Similarly, p_2 is $(\Sigma\alpha{:}K_{21}.K_{22})$-$\Gamma_2$-appropriate. The result follows by induction. □

It remains to show that expanded constructors are protected.

Definition 21.

—*The kind T is Γ-protected.*

—*The kind $S(c)$ is Γ-protected if c is.*

—*The kinds $\Pi\alpha{:}K_1.K_2$ and $\Sigma\alpha{:}K_1.K_2$ are Γ-protected if both K_1 and K_2 are.*

Lemma 22.

(*1*) *If p is K-Γ-appropriate and K is Γ-protected then $R(p, K)$ is Γ-protected.*

(*2*) *If c and K are Γ-protected then $R(c, K)$ is K-Γ-protected.*

Proof. By induction on K.

Case 1: Suppose $K \equiv T$. Part 2 is trivial. For part 1, we wish to show that p is Γ-protected. Let p be $E[\alpha]$ and suppose $p \equiv C[\beta]$. If $C \equiv E$ then the result is immediate. Otherwise C chooses β from within one of the argument positions

in the path. That is, $E[\,] \equiv E_1[(E_2[\,])(C'[\beta])]$ and $C[\,] \equiv E_1[(E_2[\alpha])(C'[\,])]$. Since p is T-Γ-appropriate, $C'[\beta]$ is K'-Γ-protected (for some K'), and thus is $C'[\beta]$ is Γ-protected. Hence there exist C'' and E' such that $C'[\,] \equiv C''[E'[\,]]$ and $E'[\beta]$ is T-Γ-appropriate. The result follows choosing $E_1[(E_2[\alpha])(C''[\,])]$ for the outer context and E' for the inner.

Case 2: Suppose $K \equiv S(c')$. Both parts are trivial, since c' is Γ-protected.

Case 3: Suppose $K \equiv \Pi\alpha{:}K_1.K_2$. Assume, without loss of generality, that $\alpha \notin \mathrm{Dom}(\Gamma)$ and α is not free in c. Then α is trivially K_1-$(\Gamma, \alpha{:}K_1)$-appropriate. Therefore, by induction, $R(\alpha, K_1)$ is $(\Gamma, \alpha{:}K_1)$-protected. By Lemma 17 (and an easy induction over K_2), it follows that $K_2\{R(\alpha, K_1)/\alpha\}$ is $(\Gamma, \alpha{:}K_1)$-protected. Using Lemma 15, $K_2\{R(\alpha, K_1)/\alpha\}$ is also Γ-protected.

(1) Since $\alpha \notin \mathrm{Dom}(\Gamma)$, α is Γ-protected. By induction, $R(\alpha, K_1)$ is K_1-Γ-protected. Thus $p\,R(\alpha, K_1)$ is K_2-Γ-appropriate. By induction, $R(p\,R(\alpha, K_1), K_2\{R(\alpha, K_1)/\alpha\})$ is Γ-protected. By Lemma 15, $R(p, K) \equiv \lambda\alpha{:}K_1.R(p\,R(\alpha, K_1), K_2\{R(\alpha, K_1)/\alpha\})$ is Γ-protected.

(2) Since α is not free in c, by Lemma 15 c is $(\Gamma, \alpha{:}K_1)$-protected. Thus $c\,R(\alpha, K_1)$ is $(\Gamma, \alpha{:}K_1)$-protected. By induction $R(c\,R(\alpha, K_1), K_2\{R(\alpha, K_1)/\alpha\})$ is K_2-$(\Gamma, \alpha{:}K_1)$-protected. Hence $R(c, K)$ is K-Γ-protected.

Case 4: Suppose $K \equiv \Sigma\alpha{:}K_1.K_2$.

(1) By definition, $\pi_1 p$ is K_1-Γ-appropriate and $\pi_2 p$ is K_2-Γ-appropriate. By induction, $R(\pi_1 p, K_1)$ is Γ-protected. By Lemma 17, $K_2\{R(\pi_1 p, K_1)/\alpha\}$ is Γ-protected, so by induction, $R(\pi_2 p, K_2\{R(\pi_1 p, K_1)/\alpha\})$ is Γ-protected. By Lemma 15, $R(p, K) \equiv \langle R(\pi_1 p, K_1), R(\pi_2 p, K_2\{R(\pi_1 p, K_1)/\alpha\})\rangle$ is Γ-protected.

(2) By Lemma 15, $\pi_1 c$ and $\pi_2 c$ are Γ-protected. By induction, $R(\pi_1 c, K_1)$ is K_1-Γ-protected. Therefore $R(\pi_1 c, K_1)$ is also Γ-protected, so by Lemma 17, $K_2\{R(\pi_1 c, K_1)/\alpha\}$ is Γ-protected. By induction $R(\pi_2 c, K_2\{R(\pi_1 c, K_1)/\alpha\})$ is K_2-Γ-protected. Hence $R(c, K)$ is K-Γ-protected.

□

LEMMA 23. *If* $\Gamma \vdash \mathrm{ok}$ *then* $R(c, K)\{R(\Gamma)\}$ *is* K-Γ-*protected.*

PROOF. Observe first that since $\Gamma \vdash \mathrm{ok}$, whenever $\Gamma \equiv \Gamma_1, \alpha{:}K', \Gamma_2$, neither α nor any variable in $\mathrm{Dom}(\Gamma_2)$ can appear free in K'. We claim that for any c', $c'\{R(\Gamma)\}$ is Γ-protected. By Lemma 5, $R(c, K)\{R(\Gamma)\} \equiv R(c\{R(\Gamma)\}, K\{R(\Gamma)\})$. It follows from the claim that $c\{R(\Gamma)\}$ and $K\{R(\Gamma)\}$ are Γ-protected, and therefore, by Lemma 22, $R(c\{R(\Gamma)\}, K\{R(\Gamma)\})$ is $(K\{R(\Gamma)\})$-Γ-protected. Then $R(c\{R(\Gamma)\}, K\{R(\Gamma)\})$ is K-Γ-protected as well, since $K \approx K\{R(\Gamma)\}$.

We prove the claim by induction on Γ. The base case is trivial. Suppose then $\Gamma \equiv \alpha{:}K', \Gamma'$. By induction, $c'\{R(\Gamma')\}$ is Γ'-protected. By the initial observation, neither α nor any variable in $\mathrm{Dom}(\Gamma')$ is free in K'. Therefore K' is Γ-protected. Also $\Gamma(\alpha) \equiv K'$ so α is K'-Γ-appropriate. By Lemma 22, $R(\alpha, K')$ is Γ-protected. We cannot immediately claim by Lemma 17 that $c'\{R(\Gamma)\}$ is Γ-protected, since $c'\{R(\Gamma')\}$ may contain free occurrences of α and thus might not be Γ-protected. However, any such occurrences are nonessential, since they will only be substituted away. We make this explicit with a change of variables. Let β be fresh. Then by

changing variables we obtain:

$$\begin{aligned} c'\{R(\Gamma)\} &\equiv c'\{R(\Gamma')\}\{R(\alpha, K')/\alpha\} \\ &\equiv c'\{R(\Gamma')\}\{\beta/\alpha\}\{R(\alpha, K')/\beta\} \end{aligned}$$

Then $c'\{R(\Gamma')\}\{\beta/\alpha\}$ is Γ-protected, since it does not contain α free. Therefore, by Lemma 17, $c'\{R(\Gamma)\}$ is Γ-protected. □

Corollary 24. *If $\Gamma \vdash c_1 = c_2 : K$ then there exists a derivation of $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} = R(c_2, K)\{R(\Gamma)\} : K$ that is mostly free of singleton elimination.*

Proof. Suppose $\Gamma \vdash c_1 = c_2 : K$. By regularity, $\Gamma \vdash \text{ok}$. By Corollary 10, $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma \vdash R(c_2, K)\{R(\Gamma)\} : K$. By Lemma 23, both $R(c_1, K)\{R(\Gamma)\}$ and $R(c_2, K)\{R(\Gamma)\}$ are K-Γ-protected, and each weak head reduces to itself, so by Lemma 20 the algorithmic derivation is free of singleton reduction. Therefore the desired derivation exists by Theorem 12. □

4.4 Wrapping up

To complete the first half of the proof, we need only the fact that singleton erasure preserves derivability of judgements with mostly singleton free derivations.

Lemma 25.

(1) *If $\Gamma \vdash c_1 = c_2 : K$ has a derivation mostly free of singleton elimination, then $\Gamma^\circ \vdash_{sf} {c_1}^\circ = {c_2}^\circ : K^\circ$.*

(2) *If $\Gamma \vdash c : K$ then $\Gamma^\circ \vdash_{sf} c^\circ : K^\circ$.*

(3) *If $\Gamma \vdash K_1 \leq K_2$ then ${K_1}^\circ \equiv {K_2}^\circ$.*

(4) *If $\Gamma \vdash \text{ok}$ then $\Gamma^\circ \vdash_{sf} \text{ok}$.*

Proof. By a straightforward induction on derivations. □

Corollary 26. *If $\Gamma \vdash c_1 = c_2 : K$ then $\Gamma^\circ \vdash_{sf} (R(c_1, K)\{R(\Gamma)\})^\circ = (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$.*

For the converse, we already have most of the facts we need at our disposal. We require two more lemmas. One states that the algorithm is symmetric and transitive. It is here that the use of a six-place algorithm is critical. For the six-place algorithm it is easy to show that symmetry and transitivity hold. For a four-place algorithm, on the other hand, it is a deep fact depending on soundness and completeness that symmetry and transitivity hold for well-formed instances, and for ill-formed instances it is not known to hold at all.

Lemma 27.

(1) *If $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash c_2 : K_2$ then $\Gamma_2 \vdash c_2 : K_2 \Leftrightarrow \Gamma_1 \vdash c_1 : K_1$.*

(2) *If $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash c_2 : K_2$ and $\Gamma_2 \vdash c_2 : K_2 \Leftrightarrow \Gamma_3 \vdash c_3 : K_3$ then $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_3 \vdash c_3 : K_3$.*

Proof. By inspection. □

The other lemma states that if singleton reduction is not employed in the algorithm, then whatever singleton kinds appear are not relevant and may be erased. Moreover, since the two halves of the algorithm operate independently (here again the six-place algorithm is critical), we may erase them from either half of the algorithm.

LEMMA 28.

(1) *If* $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash c_2 : K_2$ *without using singleton reduction, then* $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow {\Gamma_2}^\circ \vdash {c_2}^\circ : {K_2}^\circ$

(2) *If* $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow K_2$ *without using singleton reduction, then* $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow {\Gamma_2}^\circ \vdash {p_2}^\circ \uparrow {K_2}^\circ$.

PROOF. By induction on the algorithmic derivation. □

It is worth noting that the algorithmic judgement in Lemma 28 is quite peculiar, in that Γ is ordinarily not equal to Γ° and K is ordinarily not equal to K°. Although there is a valid derivation of this algorithmic judgement, the soundness theorem does not apply, so it does not correspond to any derivation in the declarative system. When we apply this lemma below we will use transitivity to bring the assignments and kinds back into agreement before invoking soundness.

LEMMA 29. *If* $\Gamma \vdash c_1 : K$, $\Gamma \vdash c_2 : K$, *and* $\Gamma^\circ \vdash_{sf} (R(c_1, K)\{R(\Gamma)\})^\circ = (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$ *then* $\Gamma \vdash c_1 = c_2 : K$.

PROOF. By Lemma 7, $\Gamma \vdash c_1 = R(c_1, K)\{R(\Gamma)\} : K$. By algorithmic completeness, $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K$. By symmetry and transitivity of the algorithm, $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K$. Then, by Lemmas 23, 20, and 28, $\Gamma \vdash R(c_1, K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma^\circ \vdash (R(c_1, K)\{R(\Gamma)\})^\circ : K^\circ$. By transitivity, $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma^\circ \vdash (R(c_1, K)\{R(\Gamma)\})^\circ : K^\circ$. Similarly, $\Gamma \vdash c_2 : K \Leftrightarrow \Gamma^\circ \vdash (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$.

Since the singleton-free system is a subsystem of the full system, we have by algorithmic completeness that $\Gamma^\circ \vdash (R(c_1, K)\{R(\Gamma)\})^\circ : K^\circ \Leftrightarrow \Gamma^\circ \vdash (R(c_2, K)\{R(\Gamma)\})^\circ : K^\circ$. Hence, by symmetry and transitivity, $\Gamma \vdash c_1 : K \Leftrightarrow \Gamma \vdash c_2 : K$. (Note that by applying transitivity, we have swept away the peculiarity noted above.) Therefore $\Gamma \vdash c_1 = c_2 : K$ by algorithmic soundness. □

This completes the proof.

5. RELATED WORK AND CONCLUSIONS

The primary purpose of this work is to allow the reification of type equality information in a type-preserving compiler for a language like Standard ML, thereby eliminating the need to complicate the metatheory of the latter phases of the compiler with singleton kinds. Within this architecture, equality (or "sharing") information would initially be expressed using singleton kinds, but at some point singleton kind elimination would be exploited to eliminate them. Thereafter, with singleton kinds no longer available, type information would be propagated by substitution, as in Harper *et al.* [1990].

Shao [1999] proposes a different approach for dealing with type equality in module languages. Shao's approach resembles the approach in this paper, in that it

substitutes definitions for variables. However, it does so less thoroughly than the approach here, since, as with most module-oriented accounts, less type information is to be propagated than in the singleton account, as mentioned in Section 2.1. In effect, Shao's substitution does not account for the issue of internal bindings discussed here in Section 3.1.

Another alternative is given in an earlier paper by Shao [1998]. In his earlier approach, equality specifications are taken as mere abbreviations and deleted from signatures. The main work arises in ensuring that the appropriate subsignature relationships hold: a signature containing a type abbreviation must be considered a subsignature of a similar one that contains that type but not the abbreviation (as required by Standard ML and the standard type-theoretic accounts). To accomplish this, when a structure matching a signature with a deleted field is used in a context where that deleted field is required, the translation coerces the structure to reinsert the deleted field. Thus, Shao's earlier approach differs from the one here in two main ways: it interprets the subsignature relation by coercion, whereas this paper's approach interprets it by inclusion; and (as with the later approach) it does not account for indirect equalities resulting from internal bindings—abbreviation occurs only where equality specifications appear syntactically.

Aspinall [1994] studies in detail a related type system with singleton types. The difference between singleton kinds and his singleton types is entirely cosmetic—this work could just as easily be presented as singleton type elimination by taking everything down one level. However, there exist important technical difference between this system and Aspinall's that prevent our result from being applied directly to Aspinall's system. Most importantly, our system relies crucially on eta-conversion, since the elimination process places constructors into eta-long form, but Aspinall's system does not support eta-conversion. Stone and Harper [2000] compare this system to Aspinall's in greater detail.

Some other systems also include types that have a bearing on equality; for example, Martin-Löf type theories [Martin-Löf 1975] contain types representing the proposition that two objects are equal. The result discussed here does not immediately apply for such systems, as our proof relies on an algorithm to structure equivalence derivations. For Martin-Löf type theories that do enjoy decidable equivalence, it may be possible to develop an appropriate notion of equality-type elimination. However, even in such cases, one would not necessarily expect that the appropriate notion of reduction would closely resemble the one here, unless the equivalence algorithm closely resembled that of Stone and Harper.

Singleton kind elimination has been implemented in the TILT compiler for Standard ML. The main issue that arises when putting this work into practice is that singleton kinds, in addition to expressing type equality information from the module language, are also very useful for expressing type information compactly. The elimination of singleton kinds can thus substantially increase the space taken up by type information. (In the limit, a particularly naive implementation could result in exponential blowup of type information by breaking DAGs into trees.) This issue arises in two ways; first, type information can take up more space in the compiler, resulting in slower compilation, and, second, if types are constructed and passed at run time [Harper and Morrisett 1995] as they are in TILT, inefficient type rep-

resentation can result in poor performance at run time. Shao *et al.* [1998] discuss a number of ways to deal with the former issue, such as hash-consing and using explicit substitutions. The latter issue can be addressed by making the construction and passing of type information explicit [Crary et al. 2002] and doing so before performing singleton elimination; then singleton elimination will have no effect on the run-time version of type information.

A. INFERENCE RULES

Well-Formed Context $\boxed{\Gamma \vdash \mathrm{ok}}$

$$\frac{}{\epsilon \vdash \mathrm{ok}} \quad (1)$$

$$\frac{\Gamma \vdash K \qquad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma, \alpha{:}K \vdash \mathrm{ok}} \quad (2)$$

Context Equivalence $\boxed{\vdash \Gamma_1 = \Gamma_2}$

$$\frac{}{\vdash \epsilon = \epsilon} \quad (3)$$

$$\frac{\vdash \Gamma_1 = \Gamma_2 \qquad \Gamma_1 \vdash K_1 = K_2 \qquad \alpha \notin \mathrm{Dom}(\Gamma_1)}{\vdash \Gamma_1, \alpha{:}K_1 = \Gamma_2, \alpha{:}K_2} \quad (4)$$

Well-Formed Kind $\boxed{\Gamma \vdash K}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T} \quad (5)$$

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash S(c)} \quad (6)$$

$$\frac{\Gamma, \alpha{:}K' \vdash K''}{\Gamma \vdash \Pi\alpha{:}K'.K''} \quad (7)$$

$$\frac{\Gamma, \alpha{:}K' \vdash K''}{\Gamma \vdash \Sigma\alpha{:}K'.K''} \quad (8)$$

Subkinding $\boxed{\Gamma \vdash K \leq K'}$

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash S(c) \leq T} \quad (9)$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T \leq T} \quad (10)$$

$$\frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) \leq S(c_2)} \quad (11)$$

$$\frac{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' \qquad \Gamma \vdash K_2' \leq K_1' \qquad \Gamma, \alpha{:}K_2' \vdash K_1'' \leq K_2''}{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' \leq \Pi\alpha{:}K_2'.K_2''} \quad (12)$$

$$\frac{\Gamma \vdash \Sigma\alpha{:}K_2'.K_2'' \qquad \Gamma \vdash K_1' \leq K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' \leq K_2''}{\Gamma \vdash \Sigma\alpha{:}K_1'.K_1'' \leq \Sigma\alpha{:}K_2'.K_2''} \quad (13)$$

Kind Equivalence $\boxed{\Gamma \vdash K_1 = K_2}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T = T} \quad (14)$$

$$\frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) = S(c_2)} \quad (15)$$

$$\frac{\Gamma \vdash K_2' = K_1' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' = \Pi\alpha{:}K_2'.K_2''} \quad (16)$$

$$\frac{\Gamma \vdash K_1' = K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Sigma\alpha{:}K_1'.K_1'' = \Sigma\alpha{:}K_2'.K_2''} \quad (17)$$

Well-Formed Constructor $\boxed{\Gamma \vdash c : K}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash b : T} \quad (18)$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \alpha : \Gamma(\alpha)} \quad (19)$$

$$\frac{\Gamma, \alpha{:}K' \vdash c : K''}{\Gamma \vdash \lambda\alpha{:}K'.c : \Pi\alpha{:}K'.K''} \quad (20)$$

$$\frac{\Gamma \vdash c : \Pi\alpha{:}K'.K'' \qquad \Gamma \vdash c' : K'}{\Gamma \vdash cc' : K''\{c'/\alpha\}} \quad (21)$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_1 c : K'} \quad (22)$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_2 c : K''\{\pi_1 c/\alpha\}} \quad (23)$$

$$\frac{\Gamma \vdash \Sigma\alpha{:}K'.K'' \qquad \Gamma \vdash c_1 : K' \qquad \Gamma \vdash c_2 : K''\{c_1/\alpha\}}{\Gamma \vdash \langle c_1, c_2\rangle : \Sigma\alpha{:}K'.K''} \quad (24)$$

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash c : S(c)} \quad (25)$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash \pi_1 c : K' \\ \Gamma \vdash \pi_2 c : K''\{\pi_1 c/\alpha\}\end{array}}{\Gamma \vdash c : \Sigma\alpha{:}K'.K''} \quad (26)$$

$$\frac{\begin{array}{c}\Gamma \vdash c : \Pi\alpha{:}K'.K_1'' \\ \Gamma, \alpha{:}K' \vdash c\alpha : K''\end{array}}{\Gamma \vdash c : \Pi\alpha{:}K'.K''} \quad (27)$$

$$\frac{\Gamma \vdash c : K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash c : K_2} \quad (28)$$

Constructor Equivalence

$\boxed{\Gamma \vdash c = c' : K}$

$$\frac{\Gamma, \alpha{:}K' \vdash c_1 = c_2 : K'' \quad \Gamma \vdash c_1' = c_2' : K'}{\Gamma \vdash (\lambda\alpha{:}K'.c_1)c_1' = c_2\{c_2'/\alpha\} : K''\{c_1'/\alpha\}} \quad (29)$$

$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \Pi\alpha{:}K'.K_1'' \\ \Gamma \vdash c_2 : \Pi\alpha{:}K'.K_2'' \\ \Gamma, \alpha{:}K' \vdash c_1\alpha = c_2\alpha : K''\end{array}}{\Gamma \vdash c_1 = c_2 : \Pi\alpha{:}K'.K''} \quad (30)$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash \pi_1 c_1 = \pi_1 c_2 : K' \\ \Gamma \vdash \pi_2 c_1 = \pi_2 c_2 : K''\{\pi_1 c_1/\alpha\}\end{array}}{\Gamma \vdash c_1 = c_2 : \Sigma\alpha{:}K'.K''} \quad (31)$$

$$\frac{\Gamma \vdash c_1 = c_1' : K_1 \qquad \Gamma \vdash c_2 : K_2}{\Gamma \vdash \pi_1\langle c_1, c_2\rangle = c_1' : K_1} \quad (32)$$

$$\frac{\Gamma \vdash c_1 : K_1 \qquad \Gamma \vdash c_2 = c_2' : K_2}{\Gamma \vdash \pi_2\langle c_1, c_2\rangle = c_2' : K_2} \quad (33)$$

$$\frac{\Gamma \vdash c : S(c')}{\Gamma \vdash c = c' : T} \quad (34)$$

$$\frac{\Gamma \vdash c = c' : T}{\Gamma \vdash c = c' : S(c)} \quad (35)$$

$$\frac{\Gamma \vdash c' = c : K}{\Gamma \vdash c = c' : K} \quad (36)$$

$$\frac{\Gamma \vdash c = c' : K \qquad \Gamma \vdash c' = c'' : K}{\Gamma \vdash c = c'' : K} \quad (37)$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash b = b : T} \quad (38)$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \alpha = \alpha : \Gamma(\alpha)} \quad (39)$$

$$\frac{\Gamma \vdash K_1' = K_2' \qquad \Gamma, \alpha{:}K_1' \vdash c_1 = c_2 : K''}{\Gamma \vdash \lambda\alpha{:}K_1'.c_1 = \lambda\alpha{:}K_2'.c_2 : \Pi\alpha{:}K'.K''} \quad (40)$$

$$\frac{\Gamma \vdash c = c' : \Pi\alpha{:}K_1.K_2 \quad \Gamma \vdash c_1 = c_1' : K_1}{\Gamma \vdash cc_1 = c'c_1' : K_2\{c_1/\alpha\}} \quad (41)$$

$$\frac{\Gamma \vdash c_1 = c_2 : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_1 c_1 = \pi_1 c_2 : K'} \quad (42)$$

$$\frac{\Gamma \vdash c_1 = c_2 : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_2 c_1 = \pi_2 c_2 : K''\{\pi_1 c_1/\alpha\}} \quad (43)$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash c_1' = c_2' : K' \\ \Gamma \vdash c_1'' = c_2'' : K''\{c_1'/\alpha\}\end{array}}{\Gamma \vdash \langle c_1', c_1''\rangle = \langle c_2', c_2''\rangle : \Sigma\alpha{:}K'.K''} \quad (44)$$

$$\frac{\Gamma \vdash c_1 = c_2 : K \qquad \Gamma \vdash K \leq K'}{\Gamma \vdash c_1 = c_2 : K'} \quad (45)$$

REFERENCES

Aspinall, D. 1994. Subtyping with singleton types. In *Eighth International Workshop on Computer Science Logic*. Lecture Notes in Computer Science, vol. 933. Springer-Verlag, Kazimierz, Poland, 1–15.

Crary, K. and Weirich, S. 1999. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*. Paris, 233–248.

Crary, K., Weirich, S., and Morrisett, G. 2002. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming 12,* 6 (Nov.), 567–600.

Dreyer, D., Crary, K., and Harper, R. 2003. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages*. New Orleans, Louisiana, 236–249.

Harper, R. and Lillibridge, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*. Portland, Oregon, 123–137.

Harper, R., Mitchell, J. C., and Moggi, E. 1990. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*. San Francisco, 341–354.

Harper, R. and Morrisett, G. 1995. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*. San Francisco, 130–141.

Harper, R. and Stone, C. 2000. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press. Extended version published as CMU technical report CMU-CS-97-147.

Leroy, X. 1994. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*. Portland, Oregon, 109–122.

Leroy, X. 1995. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM Symposium on Principles of Programming Languages*. San Francisco.

Leroy, X. 2000. A modular module system. *Journal of Functional Programming 10,* 3.

Lillibridge, M. 1997. Translucent sums: A foundation for higher-order module systems. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania.

Martin-Löf, P. 1975. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*. Studies in Logic and the Foundations of Mathematics, vol. 80. North-Holland, 73–118.

Minamide, Y., Morrisett, G., and Harper, R. 1996. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*. St. Petersburg, Florida, 271–283.

Morrisett, G., Crary, K., Glew, N., and Walker, D. 2002. Stack-based typed assembly language. *Journal of Functional Programming 12,* 1 (Jan.), 43–88.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21,* 3 (May), 527–568. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

Shao, Z. 1998. Typed cross-module compilation. In *1998 ACM International Conference on Functional Programming*. Baltimore, Maryland, 141–152.

Shao, Z. 1999. Transparent modules with fully syntactic signatures. In *1999 ACM International Conference on Functional Programming*. Paris, 220–232.

Shao, Z., League, C., and Monnier, S. 1998. Implementing typed intermediate languages. In *1998 ACM International Conference on Functional Programming*. Baltimore, Maryland, 313–323.

Shao, Z., Saha, B., Trifonov, V., and Papaspyrou, N. 2002. A type system for certified binaries. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*. Portland, Oregon, 217–232.

Smith, F., Walker, D., and Morrisett, G. 2000. Alias types. In *European Symposium on Programming*. Berlin, Germany.

Stone, C. and Harper, R. 2004. Personal Communciation.

Stone, C. A. and Harper, R. 2000. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*. Boston. Extended version published as CMU technical report CMU-CS-99-155.

Walker, D., Crary, K., and Morrisett, G. 2000. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems 22,* 4 (July). An earlier version appeared in the 1999 Symposium on Principles of Programming Languages.

Received August 2002; revised January 2004 and May 2005; accepted May 2005