Sound and Complete Elimination of Singleton Kinds

Karl Crary

Carnegie Mellon University

Abstract

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.

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 [16].

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 language. For example, in 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 [6, 13], and by Leroy, under the name manifest types [10] (and extended in Leroy [11]). 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 [9] (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 [12] for Objective CAML.

In this paper I consider a type theory based on *singleton kinds* [21], a variant of the translucent sum/manifest type formalism. The singleton kind calculus differs from the standard accounts 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 [7]. Furthermore, separating modules from the issue of type propagation makes it unnecessary to compare types by name (as in the module-based 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 [21] 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 quite complex, and may be difficult to extend to more complicated type systems. (The complexity of this proof is probably the source of the common misconception that singleton kinds make typechecking difficult.) The latter phases of a type-preserving compiler may involve some very complicated type systems indeed [15, 3, 4, 20]. 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 correctly implement the source language, 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, to substitute 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 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 [14].

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 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.

[1] It may be argued that only the former property is essential to correctly implement the source language, but the latter is nevertheless a desirable property.

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

Figure 1: Syntax

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 [21] 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 substitution with the leftmost substitution taking place first.

As discussed in the introduction, the principal intended

```
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
```

Figure 2: Sample Signatures

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 [7] 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. 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:

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

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.

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

Figure 3: Judgement Forms

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, as discussed in the introduction. 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, $\lambda\alpha{:}T.\mathtt{int}$. These functions are clearly inequivalent as members of $T \to T$; that is, the judgement $\vdash \lambda\alpha{:}T.\alpha = \lambda\alpha{:}T.\mathtt{int} : T \to T$ is not derivable. However, since $T \to T$ is a subkind of $S(\mathtt{int}) \to T$, these two functions can also be compared as members of $S(\mathtt{int}) \to T$ and in that kind they *are* equivalent. This is because the bodies α and int are compared under the assignment $\alpha{:}S(\mathtt{int})$, 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 $\beta(\lambda\alpha{:}T.\mathtt{int})$. These are well-formed under an assignment giving β the kind $(T \to T) \to T$ and also under one giving β the kind $(S(\mathtt{int}) \to T) \to T$. 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

[2] As an aside, in the module-based accounts [6, 13, 10, 11] 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 $\lambda\alpha{:}T.\mathtt{int}$ obscures the possible connection between them, which depends essentially on their actual code. (In the first-class account of Harper and Lillibridge [6, 13] 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 the module-based accounts.

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

Figure 4: Singleton Erasure

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 15 in Section 4.3.

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 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.

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.

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.

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 $\langle \texttt{int}, \texttt{bool} \rangle$.

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.

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 $\langle \pi_1\alpha, \pi_1\alpha \rangle$.

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 `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

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

known to have kind K:

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

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 `list`.

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:

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

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 internally bound variables. Nevertheless, it provides good intuition on the process of expansion over product kinds.

3.1 Internally 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 internally 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 `int` in the former constructor. What this illustrates is that when a abstraction appears in an argument position, the abstraction's domain kind can sometimes be strengthened (in this case from T to $S(\texttt{int})$). 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 $S(\texttt{int})$; in other words, it promises only to call them with `int`. The expansion process for product kinds makes this explicit. In this case, the expansion of β is $\lambda\gamma{:}(S(\texttt{int}) \to T).\,\beta(\lambda\alpha{:}S(\texttt{int}).\gamma\,\texttt{int})$. After substituting this expansion for β, each of the constructors above normalizes to $\beta(\lambda\alpha{:}S(\texttt{int}).\texttt{int})$. 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:

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

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 $\lambda\alpha{:}T.\texttt{int}$. These constructors should be considered equivalent when compared as members of kind $S(\texttt{int}) \to T$, 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 $S(\texttt{int}) \to T$, the two constructors above become $\lambda\alpha{:}S(\texttt{int}).((\lambda\alpha{:}T.\alpha)\texttt{int})$ and $\lambda\alpha{:}S(\texttt{int}).((\lambda\alpha{:}T.\texttt{int})\texttt{int})$; each of which normalize to $\lambda\alpha{:}S(\texttt{int}).\texttt{int}$. However, when considered as members of $T \to T$, the two become $\lambda\alpha{:}T.((\lambda\alpha{:}T.\alpha)\alpha)$ and $\lambda\alpha{:}T.((\lambda\alpha{:}T.\texttt{int})\alpha)$; each of which normalizes to its original form.

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 $(R(c, K)\{R(\Gamma)\})^\circ$.

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 defies direct proof, because there are too

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

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

Figure 5: Expansions

many ways that a judgement might be derived, and those derivations have no particular structure in common. We may see a reason 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.

The decidability of constructor equality has recently been shown by Stone and Harper [21]. They provide an algorithm for deciding constructor equality and 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. This approach is not attractive, due to the substantial complexity of the arguments involved. However, we may still take advantage of their result.

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.

Due to space limitations, we do not present the entire proof here, and instead only present the key lemmas and definitions. The full details may be found in the companion technical report [2].

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 4.3.

We begin by stating two lemmas that establish that well-formed constructors are equal to their expansions. These are each proven by straightforward inductions. It then follows by transitivity that when constructors are equal, so are their expansions.

Lemma 2 *If $\Gamma \vdash c : K$ then $\Gamma \vdash c = R(c,K) : K$.*

Lemma 3 *If $\Gamma \vdash c : K$ then $\Gamma \vdash c = R(c,K)\{R(\Gamma)\} : K$.*

Corollary 4 *If $\Gamma \vdash c_1 = c_2 : K$ then $\Gamma \vdash R(c_1,K)\{R(\Gamma)\} = R(c_2,K)\{R(\Gamma)\} : K$*

4.1 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.[5] 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.

The algorithm works as follows:

1. The algorithm is presented with a query of the form $\Gamma \vdash c : K \Leftrightarrow \Gamma' \vdash c' : K'$. 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.

[5] Stone and Harper also prove 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.

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.

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

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).

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 evaluation 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 5 (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 6 *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 8:

Definition 7 *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 8 (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 Harper and Stone's proof.

Theorem 8 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 6 is free of singleton reduction. Then Theorem 8 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 4.3.

4.2 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 9

- *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

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 $\Gamma_1, \alpha{:}K_1 \vdash c_1\alpha : K_1' \Leftrightarrow \Gamma_2, \alpha{:}K_2 \vdash c_2\alpha : K_2'$
$\Gamma_1 \vdash c_1 : \Sigma\alpha{:}K_1.K_1' \Leftrightarrow \Gamma_2 \vdash c_2 : \Sigma\alpha{:}K_2.K_2'$	$\Gamma_1 \vdash \pi_1 c_1 : K_1 \Leftrightarrow \Gamma_2 \vdash \pi_1 c_2 : K_2$ 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 $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_1.K_1' \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_2.K_2'$

Figure 6: Constructor Equivalence Algorithm (Six-Place Version)

particular, it is permissible for instantiation to capture free variables.

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

Finally, we define weak head reduction without a context[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 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 evaluation 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 strengthen the condition so that the evaluation context that drives a variable to kind T must be *appropriate*. An evaluation context is appropriate if, for every application appearing in that context, the argument constructor is protected (and, moreover, is still protected when driven to kind T and weak head normalized).

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

Definition 10 *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 10 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 are now ready to prove the main lemma:

Lemma 11 (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, using a substitution lemma to establish that protectedness is preserved by the weak head reduction.

It remains to show that expanded constructors are protected. In the following lemma, protectedness is lifted to kinds in the obvious manner.

Lemma 12

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.*

Corollary 13 *If* $\Gamma \vdash \mathrm{ok}$ *then* $R(c,K)\{R(\Gamma)\}$ *is* K-Γ-*protected.*

Corollary 14 *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.*

4.3 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 15

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* $\Gamma \vdash K_1 \le K_2$ *then* ${K_1}^\circ \equiv {K_2}^\circ$.
4. *If* $\Gamma \vdash \mathrm{ok}$ *then* $\Gamma^\circ \vdash_{sf} \mathrm{ok}$.

Corollary 16 *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 17

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* $\Gamma_1 \vdash c_1 : K_1 \Leftrightarrow \Gamma_3 \vdash c_3 : K_3$.

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 18

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* $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow {\Gamma_2}^\circ \vdash {p_2}^\circ \uparrow {K_2}^\circ$.

It is worth noting that the algorithmic judgement in Lemma 18 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 19 *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 3, $\Gamma \vdash c_1 = R(c_1,K)\{R(\Gamma)\} : K$. 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 Corollary 13 and Lemmas 11 and 18, $\Gamma \vdash R(c_1,K)\{R(\Gamma)\} : K \Leftrightarrow \Gamma^\circ \vdash (R(c_1,K)\{R(\Gamma)\})^\circ : K^\circ$. 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.* [7].

Shao [18] 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, in keeping with the module-based 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 [17]. 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 [1] 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), but various other technical differences between his system and this one make it unclear whether the same elimination process would apply to his system as well. Stone and Harper [21] compare this system to Aspinall's in greater detail.

An implementation of this paper's singleton kind elimination procedure in the context of the TILT compiler is planned, but has not yet been done. The main challenge we anticipate in this implementation, 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 could thus substantially increase the space taken up by type information. This issue could arise in two ways; first, type information could take up more space in the compiler, resulting in slower compilation, and, second, if types are constructed and passed at run time [8], inefficient type representation could result in poor performance at run time. Shao *et al.* [19] 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 [5] and doing so before performing singleton elimination; then singleton elimination will have no effect on the run-time version of type information.

References

[1] David Aspinall. Subtyping with singleton types. In *Eighth International Workshop on Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 1–15, Kazimierz, Poland, September 1994. Springer-Verlag.

[2] Karl Crary. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, Carnegie Mellon University, School of Computer Science, January 2000.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, January 1999.

[4] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[5] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.

[6] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[7] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[8] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

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

[10] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

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

[12] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 2000. To appear.

[13] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1997.

[14] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[15] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer-Verlag, March 1998. Extended version published as CMU technical report CMU-CS-98-178.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[17] Zhong Shao. Typed cross-module compilation. In *1998 ACM International Conference on Functional Programming*, pages 141–152, Baltimore, Maryland, September 1998.

[18] Zhong Shao. Transparent modules with fully syntactic signatures. In *1999 ACM International Conference on Functional Programming*, pages 220–232, Paris, September 1999.

[19] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *1998 ACM International Conference on Functional Programming*, pages 313–323, Baltimore, Maryland, September 1998.

[20] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, Berlin, Germany, March 2000.

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

A Inference Rules

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

$$\frac{}{\epsilon \vdash \mathrm{ok}} \tag{1}$$

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

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

$$\frac{}{\vdash \epsilon = \epsilon} \tag{3}$$

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

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T} \tag{5}$$

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

$$\frac{\Gamma, \alpha{:}K' \vdash K''}{\Gamma \vdash \Pi\alpha{:}K'.K''} \tag{7}$$

$$\frac{\Gamma, \alpha{:}K' \vdash K''}{\Gamma \vdash \Sigma\alpha{:}K'.K''} \tag{8}$$

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

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T \leq T} \tag{10}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' \\ \Gamma \vdash K_2' \leq K_1' \qquad \Gamma, \alpha{:}K_2' \vdash K_1'' \leq K_2''\end{array}}{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' \leq \Pi\alpha{:}K_2'.K_2''} \tag{12}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K_2'.K_2'' \\ \Gamma \vdash K_1' \leq K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' \leq K_2''\end{array}}{\Gamma \vdash \Sigma\alpha{:}K_1'.K_1'' \leq \Sigma\alpha{:}K_2'.K_2''} \tag{13}$$

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T = T} \tag{14}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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