Deciding Type Equivalence in a Language with Singleton Kinds*

Christopher A. Stone
cstone+@cs.cmu.edu

Robert Harper
rwh+@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–3891

Abstract

Work on the TILT compiler for Standard ML led us to study a language with *singleton kinds*: $S(A)$ is the kind of all types provably equivalent to the type A. Singletons are interesting because they provide a very general form of definitions for type variables, allow fine-grained control of type computations, and allow many equational constraints to be expressed within the type system.

Internally, TILT represents programs using a predicative variant of Girard's F_ω enriched with singleton kinds, dependent product and function kinds (Σ and Π), and a subkinding relation. An important benefit of using a typed language as the representation of programs is that typechecking can detect many common compiler implementation errors. However, the decidability of typechecking for our particular representation is not obvious. In order to typecheck a term, we must be able to determine whether two type constructors are provably equivalent. But in the presence of singleton kinds, the equivalence of type constructors depends both on the typing context in which they are compared and on the kind at which they are compared.

In this paper we concentrate on the key issue for decidability of typechecking: determining the equivalence of well-formed type constructors. We define the $\lambda^{\Pi\Sigma S}_{\leq}$ calculus, a model of the constructors and kinds of TILT's intermediate language. Inspired by Coquand's result for type theory, we prove decidability of constructor equivalence for $\lambda^{\Pi\Sigma S}_{\leq}$ by exhibiting a novel — though slightly inefficient — type-directed comparison algorithm. The correctness of this algorithm is proved using an interesting variant of Kripke-style logical relations: unary relations are indexed by a single possible world (in our case, a typing context), but binary relations are indexed by two worlds. Using this result we can then show the correctness of a natural, practical algorithm used by the TILT compiler.

*This research was sponsored in part 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.

To appear at the 2000 Principles of Programming Languages Conference.

1 Introduction

1.1 Motivation

The TIL compiler for core Standard ML [21] was structured as a series of translations between explicitly-typed intermediate languages. Each pass of the compiler (e.g., common subexpression elimination or closure conversion) transformed the program and its type, preserving well-typedness. One advantage of this framework is that typechecking the intermediate representation can detect a wide variety of common compiler implementation errors. The typing information on terms can also be used to support type-based optimizations and efficient data representations; TIL used a type-passing interpretation of polymorphism in which types were passed and analyzed at run-time [15]. In the future, it should be possible to use such typing information for annotating binaries with a certification of safety [16, 17].

The results from TIL were very encouraging, but the compiler implementation was inefficient and could only handle complete programs written without use of modules. The Fox Project group at Carnegie Mellon therefore decided to completely re-engineer TIL to produce TILT (TIL Two), a more practical compiler which could handle separate compilation and the complete SML language.

One challenge in scaling up the compiler was properly handling the propagation of type information. For example, in the Standard ML module language we can have a structure `Set` with the signature

```
sig
   type item = int
   type set
   type setpair = set * set

   val empty     : set
   val insert    : set * item -> set
   val member    : set * item -> bool
   val union     : setpair -> set
   val intersect : setpair -> set
end
```

From this interface it is apparent that the module `Set` has three type components: the type `Set.item` known to be equal to `int`, the type `Set.set` about which nothing is known, and the type `Set.setpair` which is the type of pairs of `Set.set`'s.

There are two important points to note about this example. First, equivalences such as the one between `Set.item` and `int` are *open-scope* definitions available to "the rest of the program", which may not even be written when this module is compiled. Such definitions cannot be eliminated by a simple local substitution and forgotten. Second, in a type-passing implementation like TILT these type components really are computed and stored by the run-time code. Although it is possible get rid of type definitions in signatures by replacing all references to these components with their definitions [19] we do not wish to do so; such substitutions could substantially increase the number of type computations performed at run-time.

The choice we made for TILT was to use an typed intermediate language based on F_ω with the following kind structure (recall that kinds classify type constructors):

- A kind T classifying ordinary types such as `int` or `bool->bool`.
- Singleton kinds $S(A)$ classifying all types of kind T provably equivalent to A;
- Dependent record kinds classifying records of type constructors and dependent function kinds classifying functions mapping type constructors to type constructors[1];
- A sub-kinding relation induced by $S(A) \leq T$.

Modules are represented in this language using a phase-splitting interpretation [8, 19]. The main idea is that modules can be split into type constructor and a term, while signatures split in a parallel way into a kind and a type. Singleton kinds are used to model definitions and type sharing specifications in module signatures, dependent record kinds model the type parts of structure signatures, dependent function kinds model the type parts of functor signatures, and subkinding models (non-coercive) signature matching.

For example, the kind corresponding to the above signature is a dependent record kind saying that there are three type components: the first component *item* has kind $S(\texttt{int})$ because its definition is known; the second component *set* has kind T because its definition is not known; finally the third component *setpair* has kind $S(set \times set)$, which takes advantage of the record kind being dependent by referring to the value of a previous field.

Singletons are used to describe and control the propagation of type definitions and sharing in the compiler. The type A has kind $S(B)$ if and only if the types A and B are provably equivalent. Thus, the hypothesis that the variable α has type $S(A)$ essentially says that α is a type variable with definition A. This models open-scope definitions in the source language.

Furthermore, singletons provide "partial" definitions for variables. If α is a pair of types with kind $S(\texttt{int}) \times T$ this tells us that the first component of this pair, $\pi_1\alpha$, is `int`. However, this kind tells us nothing about the identity of the $\pi_2\alpha$. As in the above example, partial definitions allow natural modeling of definitions in a modular system, where some components of a module have known definitions and others remain abstract.

[1] A record of type constructors should not be confused with a record type, which would have kind T. Similarly, functions of type constructors are not the types of term-level functions, which would also have kind T.

Interestingly, in a language with singleton kinds we can additionally express with delimited scope (*closed-scope*) definitions. The expression

$$\mathsf{let}\,\alpha{:}T = \texttt{int}\times\texttt{int}\ \mathsf{in}\ \texttt{id}[\alpha](3,4)\mathsf{end}$$

does not typecheck when expressed as a function application

$$(\Lambda\alpha{:}T.\texttt{id}[\alpha](3,4))[\texttt{int}\times\texttt{int}];$$

the application of $\texttt{id}[\alpha]$ to a pair of integers is only well-formed if α is known to be $\texttt{int}\times\texttt{int}$, which is not apparent while checking the abstraction. We can express this information, however, by annotating the argument with a singleton kind to get the well-formed term

$$(\Lambda\alpha{:}S(\texttt{int}\times\texttt{int}).\texttt{id}[\alpha](3,4))[\texttt{int}\times\texttt{int}].$$

Now let-bindings of types can be added as a primitive notion to the calculus, and this is probably a more efficient arrangement. Nevertheless, the general ability to turn types into function arguments (particularly into new arguments of pre-existing functions) is necessary for a low-level description of type-preserving closure-conversion in the type-passing framework [14]. It also enables finer control of when type computations occur at run time, permitting optimizations such as improved common subexpression elimination of types.

Given that we wish to typecheck our intermediate representation, the question that arises is whether typechecking is decidable. This question reduces to the decidability of equivalence for well-formed type constructors. This latter question is non-trivial because the equivalence of two constructors can depend both on the singletons (definitions) in the context and — less obviously — on the kind at which the constructors are being compared. (See Section 2.2.) The common method of implementing equivalence via context-insensitive rewrite rules is inapplicable for our calculus. The goal of this paper is to show that constructor equivalence is nevertheless decidable.

1.2 Outline

In Section 2 we introduce the $\lambda_{\leq}^{\Pi\Sigma S}$ calculus (a formalization of the key features of the type constructors and kinds of the TILT intermediate representation). We explain some of the more interesting aspects of this calculus, including the dependency of equivalence on the typing context and the classifying kind. We show that singletons containing constructors of higher kinds are definable, and show that every constructor has a principal (most-specific) kind.

In Section 3 we present a sound algorithm for determining equivalence of well-formed constructors. We were inspired by Coquand's approach to $\beta\eta$-equivalence for a type theory with Π types and one universe [3]. Coquand worked with an algorithm which directly decides equivalence, rather than using a confluent and strongly-normalizing reduction relation. However, in contrast to Coquand's system we cannot compare terms by their shape alone; we must take account of both the context and the classifier. Where Coquand maintains a set of bound variables, we maintain a full typing context. Similarly, he uses shapes to guide the algorithm where we maintain a classifying kind. (For example, when he would check whether either constructor is a lambda-abstraction, we check whether the classifying kind is a function kind.) Although the natural presentation of our

algorithm defines a relation of the form $\Gamma \vdash A_1 \Leftrightarrow A_2 : K$, we cannot analyze the correctness of this algorithm directly. Asymmetries in the formulation preclude a direct proof of such simple properties as symmetry and transitivity, both of which are immediately evident in Coquand's case. Instead we analyze a related algorithm which restores symmetry by maintaining two typing context and two classifying kinds, with the form $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$.

Our main technical result is the proof in Section 4 that the algorithm of Section 3 is both complete and terminating. Our proof of completeness is inspired by Coquand's use of Kripke logical relations, but our proof differs substantially from his. Our "worlds" are full contexts rather than sets of bound variables. More importantly, we make use of a novel form of Kripke logical relations in which we employ two worlds, rather than one.

In Section 5 we use this completeness result to show the correctness of the natural algorithm. This yields the practical algorithm used in the TILT implementation.

Finally we discuss related work and conclude.

Due to space considerations, the proofs of our results have been condensed or omitted in this extended abstract. Full details can be found in the companion technical report [20].

2 The $\lambda_{\leq}^{\Pi\Sigma S}$ calculus

2.1 Overview

The $\lambda_{\leq}^{\Pi\Sigma S}$ calculus models the type constructors and kinds of the TILT intermediate language. Since the term level of this language does not enter into the model, $\lambda_{\leq}^{\Pi\Sigma S}$ can be viewed in isolation as a dependently lambda calculus. It is solely because of our particular application that we prefer to refer to the constituents as "type constructors" and "kinds" rather than "terms" and "types".

The syntax of $\lambda_{\leq}^{\Pi\Sigma S}$ is shown in Figure 1. The constants b_i of kind T represent base types such as `int`. As usual, we use the usual notation of $K_1 \times K_2$ for $\Sigma\alpha{:}K_1.K_2$ and $K_1 \rightarrow K_2$ for $\Pi\alpha{:}K_1.K_2$ when α is not free in K_2.

There is a natural notion of size for kinds where $size(T) = 1$, $size(S(A)) = 2$, and $size(\Pi\alpha{:}K.K') = size(\Sigma\alpha{:}K.K') = size(K) + size(K') + 2$. The size of a kind is preserved under substitution of terms for variables.

The declarative rules defining the kinding and equivalence system of $\lambda_{\leq}^{\Pi\Sigma S}$ are given in Appendix A. For the most part, these are the usual rules for a dependently-typed lambda calculus with $\beta\eta$ equivalence. We concentrate here on presenting the less common rules.

Since we restrict constructors within singletons to be types (constructors of kind T), we have the following well-formedness rule for singleton kinds:

$$\frac{\Gamma \vdash A : T}{\Gamma \vdash S(A)}.$$

However, Section 2.3 shows that singletons of constructors of higher kind are definable in this language.

There are two introduction rules for singletons:

$$\frac{\Gamma \vdash A : T}{\Gamma \vdash A : S(A)} \qquad \frac{\Gamma \vdash A \equiv B : T}{\Gamma \vdash A \equiv B : S(A)}$$

and a corresponding elimination rule:

$$\frac{\Gamma \vdash A : S(B)}{\Gamma \vdash A \equiv B : T}.$$

The calculus includes implicit subsumption, where the subkinding relation is generated by the rules

$$\frac{\Gamma \vdash A : T}{\Gamma \vdash S(A) \leq T} \qquad \frac{\Gamma \vdash A_1 \equiv A_2 : T}{\Gamma \vdash S(A_1) \leq S(A_2)}$$

and is lifted to subkinding at Π and Σ kinds with the usual co- and contravariance rules. Under this ordering, the singleton introduction rule above allows a constructor A of kind T to be viewed as a constructor of the subkind $S(A)$. Symmetrically, by subsumption any constructor of a singleton kind can be viewed as having the superkind T.

Constructor equivalence includes β and η rules for functions and pairs. We express the η rules as extensionality principles:

$$\frac{\begin{array}{c}\Gamma \vdash A_1 : \Pi\alpha{:}K'.K_1'' \\ \Gamma \vdash A_2 : \Pi\alpha{:}K'.K_2'' \\ \Gamma, \alpha{:}K' \vdash A_1\alpha \equiv A_2\alpha : K''\end{array}}{\Gamma \vdash A_1 \equiv A_2 : \Pi\alpha{:}K'.K''}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 : K' \\ \Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 : \{\alpha \mapsto \pi_1 A_1\}K''\end{array}}{\Gamma \vdash A_1 \equiv A_2 : \Sigma\alpha{:}K'.K''}$$

The constructor well-formedness rules may be seen as reflexive instances of equivalence rules. For example, we have the following two non-standard kinding rules corresponding to the extensionality rules:

$$\frac{\begin{array}{c}\Gamma \vdash A : \Pi\alpha{:}K'.K_1'' \\ \Gamma, \alpha{:}K' \vdash A\alpha : K''\end{array}}{\Gamma \vdash A : \Pi\alpha{:}K'.K''} \qquad \frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash \pi_1 A : K' \\ \Gamma \vdash \pi_2 A : \{\alpha \mapsto \pi_1 A\}K''\end{array}}{\Gamma \vdash A : \Sigma\alpha{:}K'.K''}$$

Similar rules have previously appeared in the literature, including the non-standard structure-typing rule of Harper, Mitchell, and Moggi [8], the "VALUE" rules of Harper and Lillibridge's translucent sums [7], the strengthening operation of Leroy's manifest type system [10], and the "self" rule of Leroy's applicative functors [11]. In the presence of singletons, these rules give constructors more precise kinds than would otherwise be possible. (See Section 2.3.)

2.2 Examples of Term Equivalence

As mentioned in the introduction, singletons in the context can act as definitions and partial definitions for variables. So the provable judgments include:

$$\begin{array}{l}\alpha : S(b_i) \vdash \alpha \equiv b_i : T \\ \alpha : S(b_i) \vdash \langle \alpha, b_i \rangle \equiv \langle b_i, \alpha \rangle : T \times T \\ \alpha : T \times S(b_i) \vdash \pi_2\alpha \equiv b_i : T \\ \alpha : \Sigma\beta{:}T.S(\beta) \vdash \pi_1\alpha \equiv \pi_2\alpha : T \\ \alpha : \Sigma\beta{:}T.S(\beta) \vdash \alpha \equiv \langle \pi_1\alpha, \pi_1\alpha \rangle : T \times T.\end{array}$$

In the last two of these equations, the assumption governing α gives a definition to $\pi_2\alpha$ (namely $\pi_1\alpha$) without specifying what the two equal components actually are. Notice that the last equation is not simply an example of surjective pairing.

Singletons behave like terminal types, so by extensionality we can prove equivalences such as:

$$\begin{array}{l}\alpha : S(b_i) \rightarrow T \vdash \alpha \equiv \lambda\beta{:}S(b_i).(\alpha b_i) : S(b_i) \rightarrow T \\ \alpha : T \rightarrow S(b_i) \vdash \alpha \equiv \lambda\beta{:}T.b_i : T \rightarrow T\end{array}$$

Contexts	$\Gamma, \Delta ::=$	$\bullet$	Empty context
		$\Gamma, \alpha{:}K$	Context extension
Kinds	$K, L ::=$	T	Kind of types
		$S(A)$	Singleton kind
		$\Pi\alpha{:}K_1.K_2$	Dependent function kind
		$\Sigma\alpha{:}K_1.K_2$	Dependent product kind
Constructors	$A, B, C ::=$	b_i	Base types
		$\alpha, \beta, \ldots$	Variables
		$\lambda\alpha{:}K.A$	Function
		AA'	Application
		$\langle A, A' \rangle$	Pair
		$\pi_i A$	Projection

Figure 1: Syntax of $\lambda^{\Pi\Sigma S}_{\leq}$

Notice that in the first of these equations, the right-hand side is not simply an η-expansion of the left-hand side.

Because of subkinding, constructors do not have unique kinds. The equivalence of two constructors depends on the kind at which they are compared; they may be equivalent at one kind but not at another. For example, one *cannot* prove

$$\vdash \lambda\alpha{:}T.\alpha \equiv \lambda\alpha{:}T.b_i : T{\to}T$$

as the identity function and constant function have distinct behaviors. However, by subsumption these two functions also have kind $S(b_i){\to}T$ and the judgment

$$\vdash \lambda\alpha{:}T.\alpha \equiv \lambda\alpha{:}T.b_i : S(b_i){\to}T$$

is provable using extensionality; the functions do yield equivalent results for every argument of kind $S(b_i)$.

The classifying kind at which constructors are compared depends on the context of their occurrence. For example, from this last equation it follows that

$$\beta : (S(b_i){\to}T){\to}T \vdash \beta(\lambda\alpha{:}T.\alpha) \equiv \beta(\lambda\alpha{:}T.b_i) : T$$

2.3 Labelled Singletons

In our calculus $S(A)$ is well-formed if and only if A is of kind T. Aspinall [1] has studied equivalence in a lambda calculus with labelled singletons of the form $S(A : K)$.[2] This represents the kind of all constructors equivalent to A at kind K. Because equivalence depends on the classifier, the label K in these labelled singletons does matter: it follows from the examples of the previous section that $S(\lambda\alpha{:}T.b_i : \Pi\alpha{:}S(b_i).T)$ and $S(\lambda\alpha{:}T.b_i : T{\to}T)$ are not equivalent; only the former classifies the identity function $\lambda\alpha{:}T.\alpha$.

Our system does not contain such labelled singletons as a primitive notion because they are all definable; Figure 2 gives an inductive definition.

For example, if β has kind $T{\to}T$, then $S(\beta : T{\to}T)$ is defined to be $\Pi\alpha{:}T.S(\beta\alpha)$. This can be interpreted as "the kind of all functions which, when applied to an argument of kind T, yield the same answer as β would". By extensionality, this is indeed the kind of all functions equivalent to β at kind $T{\to}T$. The non-standard kinding rules mentioned in Section 2.1 are vital here, as they are necessary in order to prove that β has this kind.

The following proposition shows that the definitions of Figure 2 do have properties analogous to Aspinall's labelled singletons.

Proposition 2.1

1. *If* $\Gamma \vdash A_2 : K$ *and* $\Gamma \vdash A_1 : S(A_2 : K)$ *then* $\Gamma \vdash A_1 \equiv A_2 : K$.
2. *If* $\Gamma \vdash A_1 \equiv A_2 : K$ *then* $\Gamma \vdash A_1 \equiv A_2 : S(A_1 : K)$.
3. *If* $\Gamma \vdash A : K$ *then* $\Gamma \vdash S(A : K)$ *and* $\Gamma \vdash A : S(A : K)$.
4. *If* $\Gamma \vdash A : K$ *then* $\Gamma \vdash S(A : K) \leq K$.
5. *If* $\Gamma \vdash A_1 \equiv A_2 : K_1$ *and* $\Gamma \vdash K_1 \leq K_2$ *then* $\Gamma \vdash S(A_1 : K_1) \leq S(A_2 : K_2)$.

It is curious to note that in our system, as in Aspinall's, β-rules become *derivable* in the presence of singletons. This can be easily seen using Proposition 2.1; for example,

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma, \alpha{:}K_2 \vdash A : K}{\Gamma, \alpha{:}K_2 \vdash A : S(A : K)}}{\Gamma \vdash \lambda\alpha{:}K_2.A : \Pi\alpha{:}K_2.S(A : K)} \qquad \Gamma \vdash A_2 : K_2}{\Gamma \vdash (\lambda\alpha{:}K_2.A)A_2 : S(\{\alpha{\mapsto}A_2\}A : \{\alpha{\mapsto}A_2\}K)}}{\Gamma \vdash (\lambda\alpha{:}K_2.A)A_2 \equiv \{\alpha{\mapsto}A_2\}A : \{\alpha{\mapsto}A_2\}K}.$$

For convenience we have chosen to formulate the system with a stronger form of the β-rules (though we conjecture this does not change the system) and we do not use this admissibility result in the remainder of the paper.

2.4 Principal Kinds

Figure 3 gives an algorithm for determining the principal kind of a well-formed constructor; given a well-formed constructor A in the context Γ, then there exists K such that $\Gamma \vdash A \Uparrow K$ and k is the most-specific kind of A. Correctness is shown by the following lemma, which uses a slightly strengthened induction hypothesis.

Lemma 2.2
If $\Gamma \vdash A : L$ *then* $\Gamma \vdash A \Uparrow K$, $\Gamma \vdash A : K$, *and* $\Gamma \vdash K \leq S(A : L)$.

[2] Aspinall's notation for our $S(A : K)$ is $\{A\}_K$. Our $S(A)$ is not the same as Aspinall's unlabelled singleton $\{A\}$, but rather would correspond to $\{A\}_T$.

$$
\begin{array}{lcl}
S(A : T) & := & S(A) \\
S(A : S(A')) & := & S(A) \\
S(A : \Pi\alpha{:}K_1.K_2) & := & \Pi\alpha{:}K_1.(S(A\alpha : K_2)) \\
S(A : \Sigma\alpha{:}K_1.K_2) & := & (S(\pi_1 A : K_1)) \times (S(\pi_2 A : \{\alpha \mapsto \pi_1 A\}K_2))
\end{array}
$$

Figure 2: Encodings of Labelled Singletons

$$
\begin{array}{ll}
\Gamma \vdash b_i \Uparrow S(b_i) & \\
\Gamma \vdash \alpha \Uparrow S(\alpha : \Gamma(\alpha)) & \\
\Gamma \vdash \lambda\alpha{:}K'.A \Uparrow \Pi\alpha{:}K'.K'' & \text{where } \Gamma, \alpha : K' \vdash A \Uparrow K'' \\
\Gamma \vdash AA' \Uparrow \{\alpha \mapsto A'\}K'' & \text{where } \Gamma \vdash A \Uparrow \Pi\alpha{:}K'.K'' \\
\Gamma \vdash \langle A', A'' \rangle \Uparrow K' \times K'' & \text{where } \Gamma \vdash A' \Uparrow K' \text{ and } \Gamma \vdash A'' \Uparrow K''. \\
\Gamma \vdash \pi_1 A \Uparrow K' & \text{where } \Gamma \vdash A \Uparrow \Pi\alpha{:}K'.K'' \\
\Gamma \vdash \pi_2 A \Uparrow \{\alpha \mapsto \pi_1 A\}K' & \text{where } \Gamma \vdash A \Uparrow \Pi\alpha{:}K'.K''
\end{array}
$$

Figure 3: Algorithm for Principal Kind Synthesis

3 An Algorithm for Constructor Equivalence

Following Coquand, we present the equivalence test by defining a set of rules defining algorithmic relations, shown in Figure 4. It is clear that these rules can be translate directly into a deterministic algorithm, since for any goal there is at most one algorithmic rule which can apply. Then decidability of the algorithmic relations corresponds to termination of the algorithm.

Our algorithm is somewhat more involved than that of Coquand because of the context and kind-dependence of equivalence. We divide the algorithmic constructor equivalence rules into a kind-directed part and a structure-directed part, while Coquand needs only structural comparison. Our weak head normalization includes looking for definitions in the context. We have also extended the algorithm in the natural fashion to handle Σ kinds, pairing, and projection.

Define an *elimination context* to be a series of applications to and projections from "$\diamond$", which we call the context's hole. If E is such a context, then $E[A]$ represents the constructor resulting by replacing the hole in E with A. If a constructor is either of the form $E[\alpha]$ or of the form $E[b_i]$ then we will call this a *path* and denote it by p.

$$E ::= \diamond \mid EA \mid \pi_1 E \mid \pi_2 E$$

The kind extraction relation $\Gamma \vdash p \uparrow K$ attempts to determine a kind for a path by taking the kind of the head variable or constant and doing appropriate substitutions and projections. A path is said to *have a definition* if its extracted kind is a singleton kind $S(B)$; in this case we say B is the definition of the path.

The extracted kind is not always the most precise kind. For example, $\alpha{:}T \vdash \alpha \uparrow T$ but the principal kind of α in this context would be $S(\alpha)$. We can show that given a well-formed path, kind extraction succeeds and returns a valid kind for this path using induction on the well-formedness proof for the path (with a strengthened induction hypothesis).

Lemma 3.1
If $\Gamma \vdash p : K$ *then* $\Gamma \vdash p \uparrow L$, $\Gamma \vdash p : L$, *and* $\Gamma \vdash S(p : L) \leq K$.

Corollary 3.2
If $\Gamma \vdash E[p] : K$ *and* $\Gamma \vdash p \uparrow S(A)$ *then* $\Gamma \vdash E[p] \equiv E[A] : K$.

The weak head reduction relation $\Gamma \vdash A \rightsquigarrow B$ contracts the head β-redex of A, if such a redex exists. Otherwise, when the head of A is a path with a definition reduction replaces the head with the definition.

Weak head normalization $\Gamma \vdash A \Downarrow B$ repeatedly applies weak head reduction to A until a weak head normal form is found. Weak head reduction and weak head normalization are deterministic, since the head β-redex is always unique if one exists, and a path can have at most one prefix with a definition.

The algorithmic constructor equivalence relation

$$\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$$

is intended to model the declarative equivalence $\Gamma_1 \vdash A_1 \equiv A_2 : K_1$, when $\vdash \Gamma_1 \equiv \Gamma_2$ and $\Gamma_1 \vdash K_1 \equiv K_2$. The method used is to use applications and projections to drive the constructors down to a base kind, at which point they can be head-normalized and compared structurally. The algorithm can terminate immediately, however, when comparing two constructors at two singleton kinds because of our assumptions that the inputs are all well-formed: $\Gamma_1 \vdash A_1 : S(B_1)$, $\Gamma_2 \vdash A_2 : S(B_2)$, $\vdash \Gamma_1 \equiv \Gamma_2$, and $\Gamma_1 \vdash S(B_1) \equiv S(B_2)$. It follows that $\Gamma_1 \vdash A_1 \equiv B_1 \equiv B_2 \equiv A_2 : T$ and so by transitivity the algorithm can immediately succeed. (This is an effect of singletons behaving as terminal types.) As an alternative for this case the algorithm could make the (redundant) check that $\Gamma_1 \vdash A_1 : T \Leftrightarrow \Gamma_2 \vdash A_2 : T$. For certain theoretical purposes this might be preferable, and the following proofs appear to go through essentially unchanged.

The algorithmic path equivalence relation

$$\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow K_2$$

will be shown to implement constructor equality for head normal paths when $\vdash \Gamma_1 \equiv \Gamma_2$. As a notational convenience, this relation explicitly includes the extracted kinds of the two paths being compared.

Finally, the algorithmic kind equivalence relation

$$\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_2 \vdash K_2$$

Kind extraction

$\Gamma \vdash b_i \uparrow T$	
$\Gamma \vdash \alpha \uparrow \Gamma(\alpha)$	
$\Gamma \vdash \pi_1 p \uparrow K_1$	if $\Gamma \vdash p \uparrow \Sigma\beta{:}K_1.K_2$
$\Gamma \vdash \pi_2 p \uparrow \{\beta \mapsto \pi_1 p\}K_2$	if $\Gamma \vdash p \uparrow \Sigma\beta{:}K_1.K_2$
$\Gamma \vdash pA \uparrow \{\beta \mapsto A\}K_2$	if $\Gamma \vdash p \uparrow \Pi\beta{:}K_1.K_2$

Weak head reduction

$\Gamma \vdash E[(\lambda\alpha{:}K.A)A'] \rightsquigarrow E[\{\alpha \mapsto A'\}A]$	
$\Gamma \vdash E[\pi_1\langle A_1, A_2\rangle] \rightsquigarrow E[A_1]$	
$\Gamma \vdash E[\pi_2\langle A_1, A_2\rangle] \rightsquigarrow E[A_2]$	
$\Gamma \vdash E[p] \rightsquigarrow E[B]$	if $\Gamma \vdash p \uparrow S(B)$

Weak head normalization

$\Gamma \vdash A \Downarrow B$	if $\Gamma \vdash A \rightsquigarrow A'$ and $\Gamma \vdash A' \Downarrow B$
$\Gamma \vdash A \Downarrow A$	otherwise

Algorithmic constructor equivalence

$\Gamma_1 \vdash A_1 : T \Leftrightarrow \Gamma_2 \vdash A_2 : T$	if $\Gamma_1 \vdash A_1 \Downarrow p_1$, $\Gamma_2 \vdash A_2 \Downarrow p_2$, $\Gamma_1 \vdash p_1 \uparrow T \leftrightarrow \Gamma_2 \vdash p_2 \uparrow T$
$\Gamma_1 \vdash A_1 : S(B_1) \Leftrightarrow \Gamma_2 \vdash A_2 : S(B_2)$	always
$\Gamma_1 \vdash A_1 : \Pi\alpha{:}K_1.L_1 \Leftrightarrow \Gamma_2 \vdash A_2 : \Pi\alpha{:}K_2.L_2$	if $\Gamma_1, \alpha{:}K_1 \vdash A_1\alpha : L_1 \Leftrightarrow \Gamma_2, \alpha{:}K_2 \vdash A_2\alpha : L_2$
$\Gamma_1 \vdash A_1 : \Sigma\alpha{:}K_1.L_1 \Leftrightarrow \Gamma_2 \vdash A_2 : \Sigma\alpha{:}K_2.L_2$	if $\Gamma_1 \vdash \pi_1 A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash \pi_1 A_2 : K_2$, and $\Gamma_1 \vdash \pi_2 A_1 : \{\alpha \mapsto \pi_1 A_1\}L_1 \Leftrightarrow \Gamma_2 \vdash \pi_2 A_2 : \{\alpha \mapsto \pi_1 A_2\}L_2$

Algorithmic path equivalence

$\Gamma_1 \vdash b_i \uparrow T \leftrightarrow \Gamma_2 \vdash b_j \uparrow T$	if $i = j$
$\Gamma_1 \vdash \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \vdash \alpha \uparrow \Gamma_2(\alpha)$	always
$\Gamma_1 \vdash p_1 A_1 \uparrow \{\alpha \mapsto A_1\}L_1 \leftrightarrow \Gamma_2 \vdash p_2 A_2 \uparrow \{\alpha \mapsto A_2\}L_2$	if $\Gamma_1 \vdash p_1 \uparrow \Pi\alpha{:}K_1.L_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Pi\alpha{:}K_2.L_2$, and $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$.
$\Gamma_1 \vdash \pi_1 p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash \pi_1 p_2 \uparrow K_2$	if $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_1.L_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_2.L_2$.
$\Gamma_1 \vdash \pi_2 p_1 \uparrow \{\alpha \mapsto \pi_1 p_1\}L_1 \leftrightarrow \Gamma_2 \vdash \pi_2 p_2 \uparrow \{\alpha \mapsto \pi_1 p_2\}L_2$	if $\Gamma_1 \vdash p_1 \uparrow \Sigma\alpha{:}K_1.L_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow \Sigma\alpha{:}K_2.L_2$

Algorithmic kind equivalence

$\Gamma_1 \vdash T \Leftrightarrow \Gamma_2 \vdash T$	always
$\Gamma_1 \vdash S(A_1) \Leftrightarrow \Gamma_2 \vdash S(A_2)$	if $\Gamma_1 \vdash A_1 : T \Leftrightarrow \Gamma_2 \vdash A_2 : T$
$\Gamma_1 \vdash \Pi\alpha{:}K_1.L_1 \Leftrightarrow \Gamma_2 \vdash \Pi\alpha{:}K_2.L_2$	if $\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_2 \vdash K_2$ and $\Gamma_1, \alpha{:}K_1 \vdash L_1 \Leftrightarrow \Gamma_2, \alpha{:}K_2 \vdash L_2$
$\Gamma_1 \vdash \Sigma\alpha{:}K_1.L_1 \Leftrightarrow \Gamma_2 \vdash \Sigma\alpha{:}K_2.L_2$	if $\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_2 \vdash K_2$ and $\Gamma_1, \alpha{:}K_1 \vdash L_1 \Leftrightarrow \Gamma_2, \alpha{:}K_2 \vdash L_2$

Figure 4: Algorithmic Relations

determines whether two kinds are equivalent given $\vdash \Gamma_1 \equiv \Gamma_2$. This easily reduces to checking the equivalence of constructors appearing within singleton kinds.

To prove soundness of this equivalence algorithm, we first prove that weak-head normalization preserves equivalence.

Proposition 3.3
If $\Gamma \vdash E[(\lambda\alpha{:}L.A)A'] : K$ *then*
$\Gamma \vdash E[(\lambda\alpha{:}L.A)A'] \equiv E[\{\alpha \mapsto A'\}A] : K$

Proposition 3.4

1. *If* $\Gamma \vdash E[\pi_1\langle A', A''\rangle] : K$ *then* $\Gamma \vdash E[\pi_1\langle A', A''\rangle] \equiv E[A'] : K$.
2. *If* $\Gamma \vdash E[\pi_2\langle A', A''\rangle] : K$ *then* $\Gamma \vdash E[\pi_2\langle A', A''\rangle] \equiv E[A''] : K$.

Corollary 3.5
If $\Gamma \vdash A : K$ *and* $\Gamma \vdash A \Downarrow B$ *then* $\Gamma \vdash A \equiv B : K$.

Given the fact that weak-head reduction of a well-formed constructor yields a well-formed and equivalence constructor, we can then prove soundness of the algorithm by induction on derivations of the algorithmic judgments.

Theorem 3.6 (Soundness)

1. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1 \equiv K_2$, $\Gamma_1 \vdash A_1 : K_1$, $\Gamma_2 \vdash A_2 : K_2$, *and* $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$ *then* $\Gamma_1 \vdash A_1 \equiv A_2 : K_1$.
2. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash p_1 : L_1$, $\Gamma_2 \vdash p_2 : L_2$, *and* $\Gamma_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash p_2 \uparrow K_2$ *then* $\Gamma_1 \vdash K_1 \equiv K_2$ *and* $\Gamma_1 \vdash p_1 \equiv p_2 : K_1$.
3. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1$, $\Gamma_2 \vdash K_2$, *and* $\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_2 \vdash K_2$ *then* $\Gamma_1 \vdash K_1 \equiv K_2$.

A key aspect of this algorithm is that it can easily be shown to obey symmetry and transitivity properties necessary for the decidability proof. It is for this purpose that the algorithm maintains two contexts and two classifiers. (Section 5 shows that this redundancy can be eliminated in an actual implementation.)

Lemma 3.7 (Algorithmic PER Properties)

1. *If* $\Delta_1 \vdash A_1 : K_1 \Leftrightarrow \Delta_2 \vdash A_2 : K_2$ *then* $\Delta_2 \vdash A_2 : K_2 \Leftrightarrow \Delta_1 \vdash A_1 : K_1$.
2. *If* $\Delta_1 \vdash A_1 : K_1 \Leftrightarrow \Delta_2 \vdash A_2 : K_2$ *and* $\Delta_2 \vdash A_2 : K_2 \Leftrightarrow \Delta_3 \vdash A_3 : K_3$ *then* $\Delta_1 \vdash A_1 : K_1 \Leftrightarrow \Delta_3 \vdash A_3 : K_3$.
3. *If* $\Delta_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Delta_2 \vdash A_2 \uparrow K_2$ *then* $\Delta_2 \vdash A_2 \uparrow K_2 \leftrightarrow \Delta_1 \vdash A_1 \uparrow K_1$.
4. *If* $\Delta_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Delta_2 \vdash A_2 \uparrow K_2$ *and* $\Delta_2 \vdash A_2 \uparrow K_2 \leftrightarrow \Delta_3 \vdash A_3 \uparrow K_3$ *then* $\Delta_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Delta_3 \vdash A_3 \uparrow K_3$.
5. *If* $\Delta_1 \vdash K_1 \Leftrightarrow \Delta_2 \vdash K_2$ *then* $\Delta_2 \vdash K_2 \Leftrightarrow \Delta_1 \vdash K_1$.
6. *If* $\Delta_1 \vdash K_1 \Leftrightarrow \Delta_2 \vdash K_2$ *and* $\Delta_2 \vdash K_2 \Leftrightarrow \Delta_3 \vdash K_3$ *then* $\Delta_1 \vdash K_1 \Leftrightarrow \Delta_3 \vdash K_3$.

4 Completeness and Termination

To show the completeness and termination for the algorithm we define a collection of Kripke-style logical relations, shown in Figures 5, 6, and 7. The strategy for proving completeness of the algorithm is to define the logical relations, show that logically-related constructors are related by the algorithm, and finally show that provably-equivalent constructors are logically related. Using completeness we can then show the algorithm terminates for all well-formed inputs.

We use the notation Δ to denote a Kripke world. Worlds are restricted to contexts containing no duplicate bound variables; the partial order $\preceq$ on worlds is simply the prefix ordering.

The logical kind validity relation $(\Delta; K)$ **valid** is indexed by the world Δ and is well-defined by induction on the size of kinds. Similarly, the logical constructor validity relation $(\Delta; A; K)$ **valid** is indexed by a Δ and defined by induction on the size of K, which must itself be logically valid.

In addition to validity relations, we have logically-defined binary equivalence relations between (logically valid) kinds and constructors. The unusual part of these relations is that rather than being a binary relation indexed by a world, they are relations between two kinds or constructors which have been determined to be logically valid under *potentially different* worlds. Thus the form of the equivalence of kinds is $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ and the form of the equivalence on constructors is $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$. With this modification, the logical relations are otherwise defined in a reasonably familiar manner. At the base and singleton kinds we impose the algorithmic equivalence as the definition of the logical relation. At higher kinds we use a Kripke-style logical relations interpretation of Π and Σ.

With these definitions in hand we construct some derived relations. The relation $(\Delta_1; K_1 \leq L_1)$ **is** $(\Delta_2; K_2 \leq L_2)$ is defined to satisfy the following "subsumption-like" behavior:

$$\frac{(\Delta_1; A_1; K_1) \text{ is } (\Delta_2; A_2; K_2) \quad (\Delta_1; K_1 \leq L_1) \text{ is } (\Delta_2; K_2 \leq L_2)}{(\Delta_1; A_1; L_1) \text{ is } (\Delta_2; A_2; L_2)}$$

Finally, we have validity and equivalence relations on environments (substitutions mapping variables to constructors) which are defined by pointwise validity and pointwise equivalence.

We first give some basic properties of the logical relations.

Lemma 4.1 (Monotonicity)

1. *If* $(\Delta_1; K_1)$ **valid** *and* $\Delta_1' \succeq \Delta_1$ *then* $(\Delta_1'; K_1)$ **valid**.
2. *If* $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$, $\Delta_1' \succeq \Delta_1$, *and* $\Delta_2' \succeq \Delta_2$ *then* $(\Delta_1'; K_1)$ **is** $(\Delta_2'; K_2)$.
3. *If* $(\Delta_1; K_1 \leq L_1)$ **is** $(\Delta_2; K_2 \leq L_2)$, $\Delta_1' \succeq \Delta_1$, *and* $\Delta_2' \succeq \Delta_2$ *then* $(\Delta_1'; K_1 \leq L_1)$ **is** $(\Delta_2'; K_2 \leq L_2)$.
4. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$, $\Delta_1' \succeq \Delta_1$, *and* $\Delta_2' \succeq \Delta_2$ *then* $(\Delta_1'; A_1; K_1)$ **is** $(\Delta_2'; A_2; K_2)$.
5. *If* $(\Delta_1; A_1; K_1)$ **valid** *and* $\Delta_1' \succeq \Delta_1$ *then* $(\Delta_1'; A_1; K_1)$ **valid**.
6. *If* $(\Delta_1; \gamma_1; \Gamma_1)$ **is** $(\Delta_2; \gamma_2; \Gamma_2)$, $\Delta_1' \succeq \Delta_1$, *and* $\Delta_2' \succeq \Delta_2$ *then* $(\Delta_1'; \gamma_1; \Gamma_1)$ **is** $(\Delta_2'; \gamma_2; \Gamma_2)$

We next give a technical lemma which shows that logical equivalence of kinds is enough to get logical subkinding.

Lemma 4.2
If $(\Delta_1; L_1)$ **is** $(\Delta_2; L_2)$, $(\Delta_1; K_1)$ **is** $(\Delta_1; L_1)$, *and* $(\Delta_2; K_2)$ **is** $(\Delta_2; L_2)$ *then* $(\Delta_1; K_1 \leq L_1)$ **is** $(\Delta_2; K_2 \leq L_2)$.

An easy corollary of this lemma may be visualized as the following rule:

$$\frac{\begin{array}{ccc}(\Delta_1; A_1; K_1) & \textbf{is} & (\Delta_2; A_2; K_2) \\ (\Delta_1; K_1) & \textbf{is} & (\Delta_2; K_2) \\ \textbf{is} & & \textbf{is} \\ (\Delta_1; L_1) & \textbf{is} & (\Delta_2; L_2)\end{array}}{(\Delta_1; A_1; L_1) \quad \textbf{is} \quad (\Delta_2; A_2; L_2)}$$

The logical relations obey reflexivity, symmetry, and transitivity properties. The logical relations were carefully defined so that the following property holds:

Lemma 4.3 (Reflexivity)

1. $(\Delta; K)$ **valid** *if and only if* $(\Delta; K)$ **is** $(\Delta; K)$.
2. $(\Delta; A; K)$ **valid** *if and only if* $(\Delta; A; K)$ **is** $(\Delta; A; K)$.
3. $(\Delta; \gamma; \Gamma)$ **valid** *if and only if* $(\Delta; \gamma; \Gamma)$ **is** $(\Delta; \gamma; \Gamma)$.

Symmetry is straightforward and exactly analogous to the symmetry properties of the algorithmic relations.

Lemma 4.4 (Symmetry)

1. *If* $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ *then* $(\Delta_2; K_2)$ **is** $(\Delta_1; K_1)$
2. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$ *then* $(\Delta_2; A_2; K_2)$ **is** $(\Delta_1; A_1; K_1)$.
3. *If* $(\Delta_1; \gamma_1; \Gamma_1)$ **is** $(\Delta_2; \gamma_2; \Gamma_2)$ *then* $(\Delta_2; \gamma_2; \Gamma_2)$ **is** $(\Delta_1; \gamma_1; \Gamma_1)$.

In contrast, the logical relation cannot be easily shown to obey the same transitivity property as the algorithm; it does hold at the base kind but does not lift to function kinds. We therefore prove a slightly weaker property, which is nevertheless what we need for the remainder of the proof. The key difference is that the transitivity property for the algorithm involves three contexts/worlds whereas the following lemma only involves two.

- $(\Delta_1; K_1)$ **valid** iff
  1. – $K_1 = T$
     – Or, $K_1 = S(A_1)$ and $(\Delta_1; A_1; T)$ **valid**
     – Or, $K_1 = \Pi\alpha{:}K_1'.K_1''$ and $(\Delta_1; K_1')$ **valid** and $\forall \Delta_1' \succeq \Delta_1, \Delta_1'' \succeq \Delta_1$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_1''; A_2; K_1')$ then $(\Delta_1'; \{\alpha \mapsto A_1\}K_1'')$ **is** $(\Delta_1''; \{\alpha \mapsto A_2\}K_1'')$
     – Or, $K_1 = \Sigma\alpha{:}K_1'.K_1''$ and $(\Delta_1; K_1')$ **valid** and $\forall \Delta_1' \succeq \Delta_1, \Delta_1'' \succeq \Delta_1$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_1''; A_2; K_1')$ then $(\Delta_1'; \{\alpha \mapsto A_1\}K_1'')$ **is** $(\Delta_1''; \{\alpha \mapsto A_2\}K_1'')$
- $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ iff
  1. $(\Delta_1; K_1)$ **valid** and $(\Delta_2; K_2)$ **valid**.
  2. And,
     – $K_1 = T$ and $K_2 = T$
     – Or, $K_1 = S(A_1)$ and $K_2 = S(A_2)$ and $(\Delta_1; A_1; T)$ **is** $(\Delta_2; A_2; T)$
     – Or, $K_1 = \Pi\alpha{:}K_1'.K_1''$ and $K_2 = \Pi\alpha{:}K_2'.K_2''$ and $(\Delta_1; K_1')$ **is** $(\Delta_2; K_2')$ and $\forall \Delta_1' \succeq \Delta_1, \Delta_2' \succeq \Delta_2$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_2'; A_2; K_2')$ then $(\Delta_1'; \{\alpha \mapsto A_1\}K_1'')$ **is** $(\Delta_2'; \{\alpha \mapsto A_2\}K_2'')$
     – Or, $K_1 = \Sigma\alpha{:}K_1'.K_1''$ and $K_2 = \Sigma\alpha{:}K_2'.K_2''$ and $(\Delta_1; K_1')$ **is** $(\Delta_2; K_2')$ and $\forall \Delta_1' \succeq \Delta_1, \Delta_2' \succeq \Delta_2$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_2'; A_2; K_2')$ then $(\Delta_1'; \{\alpha \mapsto A_1\}K_1'')$ **is** $(\Delta_2'; \{\alpha \mapsto A_2\}K_2'')$
- $(\Delta_1; K_1 \leq L_1)$ **is** $(\Delta_2; K_2 \leq L_2)$ iff
  1. $\forall \Delta_1' \succeq \Delta_1, \Delta_2' \succeq \Delta_2$ if $(\Delta_1'; A_1; K_1)$ **is** $(\Delta_2'; A_2; K_2)$ then $(\Delta_1'; A_1; L_1)$ **is** $(\Delta_2'; A_2; L_2)$.

Figure 5: Logical Relations on Kinds

- $(\Delta; A; K_1)$ **valid** iff
  1. $(\Delta; K_1)$ **valid**
  2. And,
     – $K_1 = T$ and $\Delta \vdash A : T \Leftrightarrow \Delta \vdash A : T$.
     – Or, $K_1 = S(B)$ and $(\Delta; A; T)$ **is** $(\Delta; B; T)$.
     – Or, $K_1 = \Pi\alpha{:}K.L$, and $\forall \Delta' \succeq \Delta, \Delta'' \succeq \Delta$ if $(\Delta'; B'; K)$ **is** $(\Delta''; B''; K)$ then $(\Delta'; AB'; \{\alpha \mapsto B'\}L)$ **is** $(\Delta''; AB''; \{\alpha \mapsto B''\}L)$.
     – Or, $K_1 = \Sigma\alpha{:}K.L$, $(\Delta; \pi_1 A; K)$ **valid** and $(\Delta; \pi_2 A; \{\alpha \mapsto \pi_1 A\}L)$ **valid**
- $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$ iff
  1. $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$
  2. And, $(\Delta_1; A_1; K_1)$ **valid** and $(\Delta_2; A_2; K_2)$ **valid**
  3. And,
     – $K_1 = K_2 = T$ and $\Delta_1 \vdash A_1 : T \Leftrightarrow \Delta_2 \vdash A_2 : T$.
     – Or, $K_1 = S(B_1)$, $K_2 = S(B_2)$, and $(\Delta_1; A_1; T)$ **is** $(\Delta_2; A_2; T)$
     – Or, $K_1 = \Pi\alpha{:}K_1'.K_1''$, $K_2 = \Pi\alpha{:}K_2'.K_2''$, and $\forall \Delta_1' \succeq \Delta_1, \Delta_2' \succeq \Delta_2$ if $(\Delta_1'; B_1; K_1')$ **is** $(\Delta_2'; B_2; K_2')$ then $(\Delta_1'; A_1 B_1; \{\alpha \mapsto B_1\}K_1'')$ **is** $(\Delta_2'; A_2 B_2; \{\alpha \mapsto B_2\}K_2'')$.
     – Or, $K_1 = \Sigma\alpha{:}K_1'.K_1''$, $K_2 = \Sigma\alpha{:}K_2'.K_2''$, $(\Delta_1; \pi_1 A_1; K_1')$ **is** $(\Delta_2; \pi_1 A_2; K_2')$ and $(\Delta_1; \pi_2 A_1; \{\alpha \mapsto \pi_1 A_1\}K_1'')$ **is** $(\Delta_2; \pi_2 A_2; \{\alpha \mapsto \pi_1 A_2\}K_2'')$

Figure 6: Logical Relations on Constructors

- $(\Delta; \gamma; \Gamma)$ **valid** iff
  1. $\forall \alpha \in \mathrm{dom}(\Gamma)$. $(\Delta; \gamma\alpha; \gamma(\Gamma(\alpha)))$ **valid**.
- $(\Delta_1; \gamma_1; \Gamma_1)$ **is** $(\Delta_2; \gamma_2; \Gamma_2)$ iff
  1. $(\Delta_1; \gamma_1; \Gamma_1)$ **valid** and $(\Delta_2; \gamma_2; \Gamma_2)$ **valid**
  2. And, $\forall \alpha \in \mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$. $(\Delta_1; \gamma_1\alpha; \gamma_1(\Gamma_1\alpha))$ **is** $(\Delta_2; \gamma_2\alpha; \gamma_2(\Gamma_2\alpha))$.

Figure 7: Logical Relations on Substitutions

Lemma 4.5 (Transitivity)

1. *If* $(\Delta_1; K_1)$ **is** $(\Delta_1; L_1)$ *and* $(\Delta_1; L_1)$ **is** $(\Delta_2; K_2)$ *then* $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$.
2. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_1; B_1; L_1)$ *and* $(\Delta_1; B_1; L_1)$ **is** $(\Delta_2; A_2; K_2)$ *then* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$.

Because of this restricted formulation, we cannot use symmetry and transitivity to derive properties such as "if $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ then $(\Delta_1; K_1)$ **is** $(\Delta_1; K_1)$". An important purpose of the validity predicates is to make sure that this property does in fact hold (by building it into the definition of the equivalence logical relations).

Next we show that logical relations are closed under head expansion and reduction. Define $\Gamma \vdash A_1 \simeq A_2$ to mean that A_1 and A_2 have a common weak head reduct. The following lemma then follows by induction on the size of kinds.

Lemma 4.6 (Weak Head Closure)

1. *If* $(\Delta; A; K)$ **valid** $\Delta \vdash A' \simeq A$, *then* $(\Delta; A'; K)$ **valid**.
2. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$, $\Delta_1 \vdash A_1' \simeq A_1$, *and* $\Delta_2 \vdash A_2' \simeq A_2$ *then* $(\Delta_1; A_1'; K_1)$ **is** $(\Delta_2; A_2'; K_2)$.

Following all this preliminary work, we can now show by induction on the size of kinds that equivalence under the logical relations implies equivalence under the algorithm. This requires a stronger induction hypothesis: that under suitable conditions variables (and more generally paths) are logically valid or logically related.

Lemma 4.7 (Main Lemma)

1. *If* $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ *then* $\Delta_1 \vdash K_1 \Leftrightarrow \Delta_2 \vdash K_2$.
2. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$ *then* $\Delta_1 \vdash A_1 : K_1 \Leftrightarrow \Delta_2 \vdash A_2 : K_2$.
3. *If* $(\Delta; K)$ **valid**, $\Delta \vdash p \uparrow K \leftrightarrow \Delta \vdash p \uparrow K$, *then* $(\Delta; p; K)$ **valid**.
4. *If* $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ *and* $\Delta_1 \vdash p_1 \uparrow K_1 \leftrightarrow \Delta_2 \vdash p_2 \uparrow K_2$ *then* $(\Delta_1; p_1; K_1)$ **is** $(\Delta_2; p_2; K_2)$.

Proof:
Due to space considerations, we show here just the Π case in the proof of Parts 2 and 3.

For Part 2, assume $(\Delta_1; A_1; \Pi\alpha{:}K_1'.K_1'')$ **is** $(\Delta_2; A_2; \Pi\alpha{:}K_2'.K_2'')$. Then $(\Delta_1; \Pi\alpha{:}K_1'.K_1'')$ **is** $(\Delta_2; \Pi\alpha{:}K_2'.K_2'')$, so $(\Delta_1; K_1')$ **is** $(\Delta_2; K_2')$. Now $\Delta_1, \alpha{:}K_1' \vdash \alpha \uparrow K_1' \leftrightarrow \Delta_2, \alpha{:}K_2' \vdash \alpha \uparrow K_2'$, so inductively by Part 4 we have $(\Delta_1, \alpha{:}K_1'; \alpha; K_1')$ **is** $(\Delta, \alpha{:}K_1'; \alpha; K_1')$. Then $(\Delta_1, \alpha{:}K_1'; A_1\alpha; K_1'')$ **is** $(\Delta_2, \alpha{:}K_2'; A_2\alpha; K_2'')$. By the inductive hypothesis, $\Delta_1, \alpha{:}K_1' \vdash A_1\alpha : K_1'' \Leftrightarrow \Delta_2, \alpha{:}K_2' \vdash A_2\alpha : K_2''$. Therefore $\Delta_1 \vdash A_1 : \Pi\alpha{:}K_1'.K_1'' \Leftrightarrow \Delta_2 \vdash A_2 : \Pi\alpha{:}K_1'.K_1''$.

For Part 3, assume $(\Delta; K)$ **valid** and $\Delta \vdash p \uparrow K \leftrightarrow \Delta \vdash p \uparrow K$ where $K = \Pi\alpha{:}K'.K''$. Let $\Delta', \Delta'' \succeq \Delta$ and assume $(\Delta'; B'; K')$ **is** $(\Delta''; B''; K')$. Inductively by Part 2, $\Delta' \vdash B' : K' \Leftrightarrow \Delta'' \vdash B'' : K'$. Since the algorithmic relations are closed under context weakening, $\Delta' \vdash pB' \uparrow \{B'{\mapsto}\alpha\}K'' \leftrightarrow \Delta'' \vdash pB'' \uparrow \{B''{\mapsto}\alpha\}K''$. By $(\Delta; K)$ **valid**, $(\Delta'; \{B'{\mapsto}\alpha\}K'')$ **is** $(\Delta''; \{B''{\mapsto}\alpha\}K'')$. Inductively by Part 4, $(\Delta'; pB'; \{B'{\mapsto}\alpha\}K'')$ **is** $(\Delta''; pB''; \{B''{\mapsto}\alpha\}K'')$. Therefore $(\Delta; p; \Pi\alpha{:}K'.K'')$ **valid**.

QED

Finally we come to the Fundamental Theorem of Logical Relations, which relates provable equivalence of two constructors to the logical relations The statement of the theorem is strengthened to involve related substitutions of constructors for variables within constructors and kinds.

Theorem 4.8 (Fundamental Theorem)

1. *If* $\Gamma \vdash K$ *and* $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ *then* $(\Delta_1; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 K)$.
2. *If* $\Gamma \vdash K_1 \leq K_2$ *and* $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ *then* $(\Delta_1; \gamma_1 K_1 \leq \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_1 \leq \gamma_2 K_2)$, $(\Delta_1; \gamma_1 K_1)$ **is** $(\Delta_2; \gamma_2 K_1)$, *and* $(\Delta_1; \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_2)$.
3. *If* $\Gamma \vdash K_1 \equiv K_2$ *and* $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ *then* $(\Delta_1; \gamma_1 K_1)$ **is** $(\Delta_2; \gamma_2 K_2)$, $(\Delta_1; \gamma_1 K_1)$ **is** $(\Delta_2; \gamma_2 K_1)$, *and* $(\Delta_1; \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_2)$.
4. *If* $\Gamma \vdash A : K$ *and* $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ *then* $(\Delta_1; \gamma_1 A; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A; \gamma_2 K)$.
5. *If* $\Gamma \vdash A_1 \equiv A_2 : K$ *and* $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ *then* $(\Delta_1; \gamma_1 A_1; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_1; \gamma_2 K)$, $(\Delta_1; \gamma_1 A_1; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$, *and* $(\Delta_1; \gamma_1 A_2; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$.

Proof: We show here the proof of just one case, for derivations ending in Rule 20.

$$\frac{\Gamma, \alpha{:}K' \vdash A : K''}{\Gamma \vdash \lambda\alpha{:}K'.A : \Pi\alpha{:}K'.K''}$$

By an easy lemma there is a strict subderivation $\Gamma \vdash K'$, so by induction, $(\Delta_1; \gamma_1 K')$ **is** $(\Delta_2; \gamma_2 K')$.
Let $(\Delta_1', \Delta_2') \succeq (\Delta_1, \Delta_2)$ and assume that $(\Delta_1'; B_1; \gamma_1 K')$ **is** $(\Delta_2'; B_2; \gamma_2 K')$. By monotonicity, $(\Delta_1'; \gamma_1[\alpha{\mapsto}B_1]; \Gamma, \alpha{:}K')$ **is** $(\Delta_2'; \gamma_2[\alpha{\mapsto}B_2]; \Gamma, \alpha{:}K')$. By induction, $(\Delta_1'; (\gamma_1[\alpha{\mapsto}B_1])A; (\gamma_1[\alpha{\mapsto}B_1])K'')$ **is** $(\Delta_2'; (\gamma_2[\alpha{\mapsto}B_2])A; (\gamma_2[\alpha{\mapsto}B_2])K'')$. Now $\Delta_1 \vdash (\gamma_1[\alpha{\mapsto}B_1])A \simeq (\gamma_1(\lambda\alpha{:}K'.A))B_1$ and $\Delta_2 \vdash (\gamma_2[\alpha{\mapsto}B_2])A \simeq (\gamma_2(\lambda\alpha{:}K'.A))B_2$. By Lemma 4.6, $(\Delta_1'; (\gamma_1(\lambda\alpha{:}K'.A))B_1; (\gamma_1[\alpha{\mapsto}B_1])K'')$ **is** $(\Delta_2'; (\gamma_2(\lambda\alpha{:}K'.A))B_2; (\gamma_2[\alpha{\mapsto}B_2])K'')$. Similar arguments show that $(\Delta_1; \gamma_1(\lambda\alpha{:}K'.A); \gamma_1(\Pi\alpha{:}K'.K''))$ **valid**, that $(\Delta_2; \gamma_2(\lambda\alpha{:}K'.A); \gamma_2(\Pi\alpha{:}K'.K''))$ **valid**, and that $(\Delta_1; \gamma_1(\Pi\alpha{:}K'.K''))$ **is** $(\Delta_2; \gamma_2(\Pi\alpha{:}K'.K''))$. Therefore $(\Delta_1; \gamma_1(\lambda\alpha{:}K'.A); \gamma_1(\Pi\alpha{:}K'.K''))$ **is** $(\Delta_2; \gamma_2(\lambda\alpha{:}K'.A); \gamma_2(\Pi\alpha{:}K'.K''))$.

QED

A straightforward proof by induction on well-formed contexts shows that the identity substitution is related to itself:

Lemma 4.9
If $\Gamma \vdash ok$ *then for all* $\beta \in \mathrm{dom}(\Gamma)$ *we have* $(\Gamma; \beta; \Gamma\beta)$ **is** $(\Gamma; \beta; \Gamma\beta)$. *That is,* $(\Gamma; \mathrm{id}; \Gamma)$ **is** $(\Gamma; \mathrm{id}; \Gamma)$ *where* id *is the identity function.*

Corollary 4.10 (Completeness)

1. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $\Gamma \vdash K_1 \Leftrightarrow \Gamma \vdash K_2$.
2. *If* $\Gamma \vdash A_1 \equiv A_2 : K$ *then* $\Gamma \vdash A_1 : K \Leftrightarrow \Gamma \vdash A_2 : K$.

Intuitively, the algorithmic constructor equivalence relation can be viewed as simultaneously and independently normalizing the two constructors and comparing the results as it goes along. Then termination for both terms individually implies their simultaneous comparison will also terminate. This can be proved by induction on the algorithmic judgments (i.e., by induction on the steps of the algorithm).

Lemma 4.11

1. *If* $\Gamma_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Gamma_1 \vdash A_1 \uparrow K_1$ *and* $\Gamma_2 \vdash A_2 \uparrow K_2 \leftrightarrow \Gamma_2 \vdash A_2 \uparrow K_2$ *then* $\Gamma_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash A_2 \uparrow K_2$ *is decidable.*
2. *If* $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_1 \vdash A_1 : K_1$ *and* $\Gamma_2 \vdash A_2 : K_2 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$ *then* $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$ *is decidable.*
3. *If* $\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_1 \vdash K_1$ *and* $\Gamma_2 \vdash K_2 \Leftrightarrow \Gamma_2 \vdash K_2$ *then* $\Gamma_1 \vdash K_1 \Leftrightarrow \Gamma_2 \vdash K_2$ *is decidable.*

Then since every well-formed term is declaratively equivalent to itself, completeness yields the following corollary.

Corollary 4.12 (Decidability)

1. *If* $\Gamma \vdash A_1 : K$ *and* $\Gamma \vdash A_2 : K$ *then* $\Gamma \vdash A_1 : K \Leftrightarrow \Gamma \vdash A_2 : K$ *is decidable.*
2. *If* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$ *then* $\Gamma \vdash K_1 \Leftrightarrow \Gamma \vdash K_2$ *is decidable.*

We conclude this section with an application of completeness.

Proposition 4.13 (Consistency)
Let b_1 *and* b_2 *be two distinct constants of kind* T*. Then the judgment "* $\vdash b_1 \equiv b_2 : T$ *" is not provable.*

This inequivalence (and the inequivalence of $\lambda\alpha{:}T.\alpha$ and $\lambda\alpha{:}T.b_i$ at kind $T{\rightarrow}T$ mentioned in Section 2.2) is obvious for algorithmic equivalence, which by completeness transfers to inequivalence in the declarative system.

In proving soundness of the TILT compiler's intermediate language, these sorts of consistency properties are essential. The argument that, for example, the only closed values of type `int` are the integers would fail if the type `int` were provably equal to another base type.

5 A Simpler Algorithm

We have shown that constructor equivalence is decidable by presenting a sound, complete and terminating algorithm. However, as an implementation it inefficiently maintains two typing contexts and two classifying kinds. We would prefer an algorithm more like the declarative rules for equivalence, having only a single typing context and a single classifier. The revised algorithmic relations are shown in Figure 8.

The definition of this simplified algorithm is asymmetric because of essentially arbitrary choices between two provably equivalent kinds for the classifier or the typing context. Because we cannot prove directly that this simplified algorithm satisfies any symmetry or transitivity properties, we cannot simply use the same proof strategy. However, we can show the simplification is complete with respect to the previous algorithm, from which the remaining correctness properties follow easily.

One other small simplification is that in weak head reduction we need not worry about a path having a proper prefix with a definition; for well-formed constructors this can never occur. (This follows from Lemma 3.1 and the definition of kind extraction.)

The precise details can be found in the companion technical report. We simply state the most important results here.

The proofs use a "size" metric on derivations in the six-place algorithmic system. This metric measures the size of the derivation ignoring head reduction or head normalization steps; equivalently, we can define the metric as the number of term or path equivalence rules used in the derivation. Since every judgment has at most one derivation in the six-place system, we can refer unambiguously to the size of a provable algorithmic judgment.

The following lemma follows by induction on the size of the given algorithmic judgment. The remaining proofs are proved in essentially the same way as the corresponding results for the algorithm of Section 3.

Lemma 5.1

1. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1 \equiv K_2$, $\Gamma_1 \vdash A_1 : K_1$, $\Gamma_2 \vdash A_2 : K_2$, *and* $\Gamma_1 \vdash A_1 : K_1 \Leftrightarrow \Gamma_2 \vdash A_2 : K_2$ *then* $\Gamma_1 \vdash A_1 \Leftrightarrow A_2 : K_1$.
2. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1 \equiv K_2$, $\Gamma_1 \vdash A_1 : K_1$, $\Gamma_2 \vdash A_2 : K_2$, *and* $\Gamma_1 \vdash A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \vdash A_2 \uparrow K_2$ *then* $\Gamma_1 \vdash A_1 \leftrightarrow A_2 \uparrow K_1$.

Corollary 5.2 (Completeness)
If $\Gamma \vdash A_1 \equiv A_2 : K$ *then* $\Gamma \vdash A_1 \Leftrightarrow A_2 : K$.

Theorem 5.3 (Soundness)

1. *If* $\Gamma \vdash A_1 : K$, $\Gamma \vdash A_2 : K$, *and* $\Gamma \vdash A_1 \Leftrightarrow A_2 : K$ *then* $\Gamma \vdash A_1 \equiv A_2 : K$.
2. *If* $\Gamma \vdash p_1 : K_1$, $\Gamma \vdash p_2 : K_2$, *and* $\Gamma \vdash p_1 \leftrightarrow p_2 \uparrow K$ *then* $\Gamma \vdash p_1 \equiv p_2 : K$.

Theorem 5.4 (Decidability)

1. *If* $\Gamma \vdash A_1 : K$ *and* $\Gamma \vdash A_2 : K$ *then* $\Gamma \vdash A_1 \Leftrightarrow A_2 : K$ *is decidable.*
2. *If* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$ *then* $\Gamma \vdash K_1 \Leftrightarrow K_2$ *is decidable.*

6 Related Work

Severi and Poll [18] studied confluence and normalization of $\beta\delta$-reduction for a pure type system with definitions (let bindings), where δ is the replacement of an occurrence of a variable with its definition. Their calculus contains no notion of partial definitions and no subtyping or subkinding.

David Aspinall [1] studied a calculus $\lambda_{\leq\{\}}$ with singleton types and β-equivalence between terms. Labelled singletons are primitive notions in this system; in the absence of η-equivalence the encoding of Section 2.3 does not work. He conjectured that equivalence in this system was decidable. Karl Crary [4] studied an extension of $\lambda_{\leq}^{\Pi\Sigma S}$ with subtyping and power kinds and also conjectured that typechecking was decidable.

Singleton kinds should not be confused with the identity types of extensional Martin-Löf type theory [13]. There the type $I(K, A, B)$ represents the type of *proofs* that $A \equiv B : K$. Identity types allow arbitrary equational properties to

Weak head reduction

$\Gamma \vdash E[(\lambda\alpha{:}K.A)A'] \rightsquigarrow E[\{\alpha{\mapsto}A'\}A]$
$\Gamma \vdash E[\pi_1\langle A_1, A_2\rangle] \rightsquigarrow E[A_1]$
$\Gamma \vdash E[\pi_2\langle A_1, A_2\rangle] \rightsquigarrow E[A_2]$
$\Gamma \vdash p \rightsquigarrow B$ if $\Gamma \vdash p \uparrow S(B)$

Algorithmic constructor equivalence

$\Gamma \vdash A_1 \Leftrightarrow A_2 : T$ if $\Gamma \vdash A_1 \Downarrow p_1$, $\Gamma \vdash A_2 \Downarrow p_2$, and $\Gamma \vdash p_1 \leftrightarrow p_2 \uparrow T$
$\Gamma \vdash A_1 \Leftrightarrow A_2 : S(B)$ always
$\Gamma \vdash A_1 \Leftrightarrow A_2 : \Pi\alpha{:}K'.K''$ if $\Gamma, \alpha{:}K' \vdash A_1\alpha \Leftrightarrow A_2\alpha : K''$
$\Gamma \vdash A_1 \Leftrightarrow A_2 : \Sigma\alpha{:}K'.K''$ if $\Gamma \vdash \pi_1 A_1 \Leftrightarrow \pi_1 A_2 : K'$ and $\Gamma \vdash \pi_2 A_1 \Leftrightarrow \pi_2 A_2 : \{\alpha{\mapsto}\pi_1 A_1\}K''$

Algorithmic path equivalence

$\Gamma \vdash b_i \leftrightarrow b_j \uparrow T$ if $i = j$
$\Gamma \vdash \alpha \leftrightarrow \alpha \uparrow \Gamma(\alpha)$
$\Gamma \vdash p_1 A_1 \leftrightarrow p_2 A_2 \uparrow \{\alpha{\mapsto}A_1\}K_2$ if $\Gamma \vdash p_1 \leftrightarrow p_2 \uparrow \Pi\alpha{:}K_1.K_2$ andalso $\Gamma \vdash A_1 \Leftrightarrow A_2 : K_1$
$\Gamma \vdash \pi_1 p_1 \leftrightarrow \pi_1 p_2 \uparrow K_1$ if $\Gamma \vdash p_1 \leftrightarrow p_2 \uparrow \Sigma\alpha{:}K_1.K_2$
$\Gamma \vdash \pi_2 p_1 \leftrightarrow \pi_2 p_2 \uparrow \{\alpha{\mapsto}\pi_1 p_1\}K_2$ if $\Gamma \vdash p_1 \leftrightarrow p_2 \uparrow \Sigma\alpha{:}K_1.K_2$

Algorithmic kind equivalence

$\Gamma \vdash T \Leftrightarrow T$ always
$\Gamma \vdash S(A_1) \Leftrightarrow S(A_2)$ if $\Gamma \vdash A_1 \Leftrightarrow A_2 : T$
$\Gamma \vdash \Pi\alpha{:}K_1.L_1 \Leftrightarrow \Pi\alpha{:}K_2.L_2$ if $\Gamma \vdash K_1 \Leftrightarrow K_2$ and $\Gamma, \alpha{:}K_1 \vdash L_1 \Leftrightarrow L_2$
$\Gamma \vdash \Sigma\alpha{:}K_1.L_1 \Leftrightarrow \Sigma\alpha{:}K_2.L_2$ if $\Gamma \vdash K_1 \Leftrightarrow K_2$ and $\Gamma, \alpha{:}K_1 \vdash L_1 \Leftrightarrow L_2$

Figure 8: A Simplified Algorithm

be expressed, in comparison to the simpler definitional constraints allowed by singletons. The cost of identity types is that typechecking becomes substantially more difficult, if not undecidable.

Our proof was inspired by that of Coquand [3], but as the equivalence considered there was not context-sensitive in any way our algorithm and proof are substantially different. Because of the validity logical relations and the form of the symmetry and transitivity properties for logical equivalence, our attempts to use more traditional Kripke logical relations (with worlds being pairs of contexts) were unsuccessful.

Other researchers have considered lambda calculi with more interesting equivalences. Lillibridge [12] considered a language in which type constructors could be reified as values and where equivalence depends on the typing context. This system contains subtyping but not subkinding, so equivalence does not depend on the classifying kind. He eliminates context-sensitivity by tagging each path with its enclosing typing context and then giving a rewriting strategy for this tagged system.

Curien and Ghelli [5] gave a proof of decidability of term equivalence in $F_{\leq}$ with $\beta\eta$-reduction and a Top type. Because their Top type is both terminal and maximal, equivalence depends on both the typing context and the type at which terms are compared. Context-sensitivity is removed by the insertion of explicit coercions to mark uses of subsumption. They show decidability by giving a rewriting strategy for this calculus with coercions.

A drawback of both Lillibridge's and Curien and Ghelli's approaches is that they require a studying a separate tagged language and proofs that decidability results from this system can be transfered back to the original systems. Also, it does not seem particularly efficient in practice to translate type constructors into a tagged system, normalizing the tagged constructors, and compare normal forms.

7 Conclusion and Future Work

We have confirmed that $\beta\eta$-equivalence for well-formed constructors is decidable in the presence of singleton kinds by providing a sound, complete, and terminating algorithm. This algorithm — with minor extensions such as stopping early when constructors are found to be α-equivalent — is used by the internal typechecker of the TILT compiler.

The form of our algorithm seems ideally suited for equational theories in which equivalence depends on the typing context or the classifier. Examples other than $\lambda_{\leq}^{\Pi\Sigma S}$ include systems containing a terminal Top type (where any two terms of type Top are equivalent), and systems including records and width subtyping (where only the components mentioned in the classifying record type matter in determining the equivalence of two records). Harper and Pfenning [9] have even applied a simplified version of our algorithm to the LF type theory, in part because of this sort of flexibility.

Although the pattern of our logical relations proof is fairly standard, our formulation — in particular, the equivalence relation involving two constructors, two kinds, and two worlds — appears novel, as is the extension to subkinding and singleton kinds. Because our algorithm is defined separately from the equivalence rules (rather than being an rewriting system based on oriented equivalence rules), the correctness proof is not very sensitive to the exact form of the $\lambda_{\leq}^{\Pi\Sigma S}$ typing or equivalence rules.

We expect that our proof should generalize well to extensions of $\lambda_{\leq}^{\Pi\Sigma S}$ such as subtyping and power kinds like those found in Crary's work. The approach may work as well for other variants of the algorithm.

We are currently investigating the addition of singleton

types to the TILT compiler. These seem a promising formalized vehicle for expressing the information needed by cross-module inlining [2, 19] and modeling the structure sharing feature of original Standard ML.

The formalization of this paper does not include an account of type analysis constructs [15] such as the Typecase type constructor. Extending the algorithm to account for these does not seem difficult, but we have not yet checked carefully to see whether our proof can be correspondingly extended.

The way in which our algorithm is guided by the classifying kind bears a superficial resemblance to Goguen's typed operational semantics [6]. It would be interesting to know if these techniques could be adapted to our system. This is not immediately clear, because Goguen applies his technique to a system in which type constructors have unique kinds and where extensionality can be implemented via η-reduction.

8 Acknowledgements

We would like to thank Lars Birkedal for his suggestion that the form of the logical relations should mirror the form of the algorithmic relations and Karl Crary for his detailed critique of our proofs. We also thank Perry Cheng, Mark Lillibridge, Leaf Petersen, Frank Pfenning, John Reynolds, Jon Riecke, and Rick Statman for helpful discussions.

References

[1] David Aspinall. Subtyping with Singleton Types. In *Proc. Computer Science Logic (CSL'94)*, 1995. In Springer LNCS 933.

[2] Matthias Blume and Andrew W. Appel. Lambda-Splitting: A Higher-Order Approach to Cross-Module Optimizations. In *Proc. 1997 International Conference on Functional Programming (ICFP '97)*, pages 112–124, June 1997.

[3] Thierry Coquand. An Algorithm for Testing Conversion in Type Theory. In Gérard Huet and G. Plotkin, editors, *Logical frameworks*, pages 255–277. Cambridge University Press, 1991.

[4] Karl F. Crary. *Type-Theoretic Methodology for Practical Programming Languages.* PhD thesis, Department of Computer Science, Cornell University, 1998.

[5] Pierre-Louis Curien and Giorgio Ghelli. Decidability and Confluence of $\beta\eta\mathrm{top}_{\leq}$ Reduction in $F_{\leq}$. *Information and Computation*, 1/2:57–114, 1994.

[6] Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994. Available as LFCS Technical Report ECS-LFCS-94-304.

[7] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[8] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order Modules and the Phase Distinction. In *17th Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[9] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the lf type theory. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages*, September 1999. Extended version available as CMU Technical Report CMU-CS-99-159.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[11] Xavier Leroy. Applicative Functors and Fully Transparent Higher-Order Modules. In *Proc. 22nd Symposium on Principles of Programming Languages*, pages 142–153, 1995.

[12] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. Available as CMU Technical Report CMU-CS-97-122.

[13] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1994.

[14] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proc. 23rd Symposium on Princples of Programming Languages*, pages 271–283, 1996.

[15] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. Available as CMU Technical Report CMU-CS-95-226.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. Technical Report TR97-1651, Department of Computer Science, Cornell University, 1997.

[17] George C. Necula. Proof-Carrying Code. In *24th Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[18] Paula Severi and Eric Poll. Pure Type Systems with definitions. In *Logical Foundations of Computer Science '94*, number 813 in LNCS, 1994.

[19] Zhong Shao. Typed Cross-Module Compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 141–152, September 1998.

[20] Christopher A. Stone and Robert Harper. Deciding Type Equivalence in a Language with Singleton Kinds. Technical Report CMU-CS-99-155, Department of Computer Science, Carnegie Mellon University, 1999.

[21] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

A Rules for $\lambda^{\Pi\Sigma S}_{\leq}$

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

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

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

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

$$\frac{}{\vdash \bullet \equiv \bullet} \quad (3)$$

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash A_1 \equiv A_2 : T}{\Gamma \vdash S(A_1) \leq S(A_2)} \quad (11)$$

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

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

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

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

$$\frac{\Gamma \vdash A_1 \equiv A_2 : T}{\Gamma \vdash S(A_1) \equiv S(A_2)} \quad (15)$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash A : \Pi\alpha{:}K'.K'' \qquad \Gamma \vdash A' : K'}{\Gamma \vdash AA' : \{\alpha{\mapsto}A'\}K''} \quad (21)$$

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

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

$$\frac{\Gamma \vdash \Sigma\alpha{:}K'.K'' \qquad \Gamma \vdash A_1 : K' \qquad \Gamma \vdash A_2 : \{\alpha{\mapsto}A_1\}K''}{\Gamma \vdash \langle A_1, A_2\rangle : \Sigma\alpha{:}K'.K''} \quad (24)$$

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

$$\frac{\Gamma \vdash \Sigma\alpha{:}K'.K'' \qquad \Gamma \vdash \pi_1 A : K' \qquad \Gamma \vdash \pi_2 A : \{\alpha{\mapsto}\pi_1 A\}K''}{\Gamma \vdash A : \Sigma\alpha{:}K'.K''} \quad (26)$$

$$\frac{\Gamma \vdash A : \Pi\alpha{:}K'.K_1'' \qquad \Gamma, \alpha{:}K' \vdash A\alpha : K''}{\Gamma \vdash A : \Pi\alpha{:}K'.K''} \quad (27)$$

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

Constructor Equivalence $\boxed{\Gamma \vdash A \equiv A' : K}$

$$\frac{\Gamma, \alpha{:}K' \vdash A_1 \equiv A_2 : K'' \qquad \Gamma \vdash A_1' \equiv A_2' : K'}{\Gamma \vdash (\lambda\alpha{:}K'.A_1)A_1' \equiv \{\alpha{\mapsto}A_2'\}A_2 : \{\alpha{\mapsto}A_1'\}K''} \quad (29)$$

$$\frac{\Gamma \vdash A_1 : \Pi\alpha{:}K'.K_1'' \qquad \Gamma \vdash A_2 : \Pi\alpha{:}K'.K_2'' \qquad \Gamma, \alpha{:}K' \vdash A_1\alpha \equiv A_2\alpha : K''}{\Gamma \vdash A_1 \equiv A_2 : \Pi\alpha{:}K'.K''} \quad (30)$$

$$\frac{\Gamma \vdash \Sigma\alpha{:}K'.K'' \qquad \Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 : K' \qquad \Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 : \{\alpha{\mapsto}\pi_1 A_1\}K''}{\Gamma \vdash A_1 \equiv A_2 : \Sigma\alpha{:}K'.K''} \quad (31)$$

$$\frac{\Gamma \vdash A_1 \equiv A_1' : K_1 \qquad \Gamma \vdash A_2 : K_2}{\Gamma \vdash \pi_1\langle A_1, A_2\rangle \equiv A_1' : K_1} \quad (32)$$

$$\frac{\Gamma \vdash A_1 : K_1 \qquad \Gamma \vdash A_2 \equiv A_2' : K_2}{\Gamma \vdash \pi_2\langle A_1, A_2\rangle \equiv A_2' : K_2} \quad (33)$$

$$\frac{\Gamma \vdash A : S(B)}{\Gamma \vdash A \equiv B : T} \quad (34)$$

$$\frac{\Gamma \vdash A \equiv B : T}{\Gamma \vdash A \equiv B : S(A)} \tag{35}$$

$$\frac{\Gamma \vdash A' \equiv A : K}{\Gamma \vdash A \equiv A' : K} \tag{36}$$

$$\frac{\Gamma \vdash A \equiv A' : K \qquad \Gamma \vdash A' \equiv A'' : K}{\Gamma \vdash A \equiv A'' : K} \tag{37}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash b_i \equiv b_i : T} \tag{38}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \alpha \equiv \alpha : \Gamma(\alpha)} \tag{39}$$

$$\frac{\Gamma \vdash K_1' \equiv K_2' \qquad \Gamma, \alpha{:}K_1' \vdash A_1 \equiv A_2 : K''}{\Gamma \vdash \lambda\alpha{:}K_1'.A_1 \equiv \lambda\alpha{:}K_2'.A_2 : \Pi\alpha{:}K'.K''} \tag{40}$$

$$\frac{\Gamma \vdash A \equiv A' : \Pi\alpha{:}K_1.K_2 \qquad \Gamma \vdash A_1 \equiv A_1' : K_1}{\Gamma \vdash AA_1 \equiv A'A_1' : \{\alpha \mapsto A_1\}K_2} \tag{41}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 : K'} \tag{42}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : \Sigma\alpha{:}K'.K''}{\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 : \{\alpha \mapsto \pi_1 A_1\}K''} \tag{43}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{:}K'.K'' \\ \Gamma \vdash A_1' \equiv A_2' : K' \\ \Gamma \vdash A_1'' \equiv A_2'' : \{\alpha \mapsto A_1'\}K''\end{array}}{\Gamma \vdash \langle A_1', A_1''\rangle \equiv \langle A_2', A_2''\rangle : \Sigma\alpha{:}K'.K''} \tag{44}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : K \qquad \Gamma \vdash K \leq K'}{\Gamma \vdash A_1 \equiv A_2 : K'} \tag{45}$$