Singleton Kinds and Singleton Types

Christopher Allan Stone
August 2, 2000
CMU-CS-00-153

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Robert Harper, Chair
Peter Lee
John Reynolds
Jon Riecke (Bell Laboratories, Lucent Technologies)

This research was supported in part by the US Army Research Office under Grant No. DAAH04-94-G-0289 and 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.

Keywords: Singleton Kinds, Singleton Types, Type Theory, Typechecking, Typed Compilation, Lambda Calculus

In memory of my grandfather, Dr. Joseph F. Bradley

Contents

List of Figures

Abstract

In this dissertation I study the properties of *singleton kinds* and *singleton types*. These are extremely precise classifiers for types and values, respectively: the kind of all types equal to [a given type], and the type of all values equal to [a given value]. Singletons are interesting because they provide a very general and modular form of definition, allow fine-grained control of type computations, and allow many equational constraints to be expressed within the type system. This is useful, for example, when modeling the type sharing and type definition constraints appearing in module signatures in the Standard ML language; singletons are used for this purpose in the TILT compiler for Standard ML.

However, the decidability of typechecking in the presence of singletons is not obvious. In order to typecheck a term, one 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 dissertation I present MIL_0, a lambda calculus with singletons that is based upon the representation used by the TILT compiler. I prove important properties of this language, including type soundness and decidability of typechecking. The main technical result is decidability of equivalence for well-formed type constructors. Inspired by Coquand's result for type theory, I prove decidability of constructor equivalence for MIL_0 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 I can then show the correctness of a natural, practical algorithm used by the TILT compiler.

Acknowledgments

This dissertation would not be possible without the constant support of my family, the encouragement of both my former advisor Peter Lee and my current advisor Bob Harper, the help of the members of the FOX Project at Carnegie Mellon (particularly Perry Cheng and Leaf Petersen), the helpful comments of Frank Pfenning, John Reynolds and Rick Statman, the exciting environment created by the members of the Principles of Programming Languages group, and the friendship of all those folks who lived with me at The Church in the past six years (Andrej Bauer, Susan Blanset, Fay Chang, Marko Grobelnik, Mike Harkavy, John Langford, Dunja Mladenic, Chris Paciorek, Adrian Perrig, and Jeff Polakow).

Lars Birkedal originally suggested that a 6-place logical relation might be made to work. Bob Harper, Karl Crary, John Reynolds, and Jon Riecke proofread versions of this work particularly carefully. All remaining errors are, of course, attributable solely to me.

Chapter 1

Introduction

1.1 Definitions and Constraints in Interfaces

Many programming languages allow some form of definitions to appear in program unit interfaces. In the C language, for example, header files frequently contain definitions of type abbreviations. For example,

```
typedef struct {
            int x;
            int y;
        } point_t;
```

defines the type name `point_t` to stand for the type of a record containing two integers named `x` and `y` respectively. Such type definitions in C are effectively macros; the main advantage of using `typedef` rather than the C preprocessor's `#define` is that the the tortuous syntax of C variable declarations (particularly for function pointers) makes simple textual substitution insufficient [KR88].

The Standard ML language [MTHM97] also permits type definitions to appear in module interfaces. The specification

```
structure S : sig
                 type point_t = {x : int, y : int}
              end
```

says that `S` is a module containing just one element: a type named `point_t`. The interface further specifies that this type `S.point_t` is again the type of a record with two integer components named `x` and `y`. Type abbreviations in SML are qualitatively different from `typedef`, however. This SML code is a true specification, and as such must be a specification *of something*; if code is compiled in the presence of this interface then at some later point (e.g., link time) a module satisfying this specification must be supplied. Furthermore, the definition in this signature acts as a form of constraint: any module satisfying this specification must contain a type `point_t` with an equal definition. Supplying a different type leads to a static error, and this is not the behavior of a simple type macro.

The type-theoretic approach to studying programming languages has proved extremely fruitful. By isolating primitive concepts (organized around types), languages can be understood and compared more easily. Such an atomistic approach can lead to the improved design and implementation of programming languages.

Thus the question arises: what primitive language concept corresponds to type definitions in module interfaces? Several studies have effectively taken the entire SML system of modules and interfaces as primitive [HL94, Ler94, Ler95]. However, this is a rather heavyweight notion. In considering a formal calculus with such modules, either the modules are ordinary values and module interfaces just a form of type, or else these are held separate from the rest of the language. In the former case typechecking becomes undecidable [HL94, Lil97]. In the latter case there is a certain redundancy resulting from having structures (collections of types and values) and parameterized modules (functions from modules to modules) separate from ordinary records of values and ordinary functions.

An alternative approach is to focus on the type specification itself, adding to the primitive specifications such as "a type" or "a parameterized type of one argument" specifications of the form "a type equal to [some given type]". This leads to the notion of *singleton kinds*. If types or kinds (kinds are the types of types) intuitively correspond to sets, then singleton kinds are sets containing one element; membership in such a set is therefore a very strong statement. Analogously, one can form *singleton types*, expressing membership in the "collection of values equal to [some given value]".

The goal of this dissertation is to study the addition of singleton types and kinds to a well-understood type system, with particular emphasis on the important properties of type soundness and decidability of typechecking.

The remainder of this chapter explains more carefully the concepts of singleton types and kinds, and shows several examples besides type definitions where singleton kinds and types appear useful in theory and practice. I conclude with a high-level overview of the dissertation.

1.2 The TIL and TILT Compilers

1.2.1 TIL

TIL [TMC+96, Tar96, Mor95] was a prototype compiler for the core subset of the Standard ML language [MTHM97]. It was structured as a series of translations between explicitly-typed intermediate languages, and indeed the very name TIL refers to the Typed Intermediate Languages used by the compiler. Each pass of the compiler (e.g., common subexpression elimination or closure conversion) transformed both the program and its type while preserving well-typedness. This framework has several advantages:

- A wide variety of common compiler implementation errors can be detected during compilation by running a typechecker on the compiler's program representation after each transformation. The location of the type error yields very precise information about which compiler phase introduced the error and which part of the input program triggered the bug. Although the fact that the compiler preserves well-typedness in no way guarantees that it is also meaning-preserving, a very large class of compiler bugs exhibit themselves by creating type errors [Nec98].
- By maintaining full typing information, the compiler is able 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 [HL94, Mor95].
- Typing information can be used to annotate binaries with an easily verifiable certificate (proof) of safety, the absence of certain run-time errors [MWCG97, Nec97].

The results from TIL — in particular the quality of the generated code — were very encouraging [TMC+96]. However, the implementation was inefficient and could only compile small, complete programs written without use of modules; very few interesting programs meet these criteria. To further test the ideas behind TIL, the members of the CMU Fox Project decided to completely re-engineer the compiler to produce TILT (TIL Two). The aim was to produce a more practical compiler based on typed intermediate languages which could handle separate compilation, the complete SML language, and large inputs. The biggest research challenge in scaling up the compiler to the full language was adding support for modules.

1.2.2 Standard ML Modules

Modules in SML are "second-class" entities — there are no conditional module expressions, nor may modules be assigned to mutable variables or be passed to or returned from ordinary functions. The basic form of an SML module is a *structure*, which is a package of types, values, and sub-modules. *Structure signatures*, the interfaces of structures, consist of a corresponding collection of type, value, and module specifications. Value specifications give the type of a value component, and module specifications give the signature of a module component. Type specifications may either be opaque (specifying only the kind of the component) or transparent (exposing the type's definition). For example, consider the following structure specification:

```
structure Set : 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
```

This states that `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. `Set` also contains five value components; from the names, presumably `Set.empty` will be a representation of the empty set, `set.union` computes the union of a pair of sets, and so on.

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 be written yet 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 types are computed and stored by the run-time code. Although it is possible to get rid of type definitions in signatures by replacing all references to these components with their definitions [Sha98] this is not necessarily a good idea in a type-passing implementation; such substitutions could substantially increase the number of type computations performed at run-time.

An alternative method of expressing information about type components in signatures is by *type sharing* specifications; these specify that two particular type components have the same definition.

Figure 1.1 (adapted from [MT91, p. 65]) shows two equivalent definitions for the signature for the front end of a compiler. The first definition states that the front end has two sub-structures: a

```
signature FRONTEND =
   sig
      structure Lexer  : sig
                            type token
                            val lex : string -> token list
                         end
      structure Parser : sig
                            type token
                            type ast
                            val parse : token list -> ast
                         end
      sharing type Lexer.token = Parser.token
   end

signature FRONTEND =
   sig
      structure Lexer  : sig
                            type token
                            val lex : string -> token list
                         end
      structure Parser : sig
                            type token = Lexer.token
                            type ast
                            val parse : token list -> ast
                         end
   end
```

Figure 1.1: Constraints via Type Sharing or Type Definitions

lexer implementation (which takes a string of characters and splits it up into a list of tokens, which presumably would be things like identifiers or language keywords) and a parser implementation (which takes a list of tokens and translates these into an abstract syntax tree, making the program structure apparent). The `Lexer` and `Parser` sub-structures each have their own notion of tokens; only the final line of this signature specifies that these two notions are compatible. As a consequence, it is allowable to compose the two functions `Lexer.lex` and `Parser.parse` together.

Such **`sharing type`** constraints do not add expressiveness to the language because they can always be viewed as syntactic sugar for the definitions of type components [HS00]. The second definition in Figure 1.1 defines an equal signature using a type definition.

Modules may be given less-specific signatures using *subsumption* — the signature of a module may be weakened to a "larger" signature in the sub-signature ordering. The important part of this ordering is that omitting constraints on types makes structure sharing less precise[1]. For example, a structure satisfying the signature

[1] In SML, the subsignature relation also lets structure components be forgotten or reordered; this coercion is definable and hence does not add essential expressiveness [HS00].

```
structure Set : sig
                  type item = int
                  type set = int list
                  type setpair = (int list) * (int list)

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

(which exposes the implementation of sets as lists of integers) would also satisfy the previous specification, while an implementation satisfying either of these specifications would further satisfy the less-demanding specification

```
structure Set : sig
                  type item
                  type set
                  type setpair

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

The Standard ML module system also permits formation of parameterized modules called *functors*; functors are simply a form of function mapping modules to modules. In the official SML module system there is no way to express the interface of a functor; such an interface would specify the signature of the result in terms of the functor argument. However certain compilers like SML/NJ [MT94, CM94] extend the SML language with higher-order functors and functor signatures. The sub-signature relation is then extended to functor signatures in the usual way: contravariantly in the domain and covariantly in the codomain. In any case, an SML compiler must have an *internal* notion of functor signature in order to do typechecking in the presence of functor applications.

1.2.3 Phase-Splitting in TILT

The primary intermediate language of the TIL compiler was based on F_ω, the higher-order polymorphic lambda calculus [Gir72]. One goal of the TILT redesign was to minimize changes to the internal languages, in the hope that this would minimize the work needed to port the TIL optimization and code generation phases.

F_ω contains the type and kind structures alluded to above, but no module system. However, modules and signatures can still be faithfully represented using ideas of Harper, Mitchell, and Moggi [HMM90, Sha98]. Their key insight was that every module can be uniformly transformed away via a process called *phase-splitting* into two pieces: a type part and a value part. For example structures, which are aggregates of both types and values, become two collections: one of types and one of values. The more interesting observation is that that functors can be split in the same way. Functors map types and values in one structure to types and values in another structure. However, types in the result can only depend on types (not values!) in the argument. This means that a functors can be split into its behavior on types (which can be expressed as a function mapping records of types to records of types) and its behavior on values (expressed as a polymorphic function in F_ω).

Signatures then split in a parallel fashion. Structure signatures, for example, split into a *kind* describing collection of types and a *type* describing a collection of values. For example, the structure

```
struct
  type t = int
  val  n = 3
  val  succ = fn (n:int) => n+1
end
```

splits into two parts: a collection of types (in this case, a one-element collection)

```
{t = int}
```

and a collection of two values

```
{n = 3, succ = fn (n:int) => n+1}.
```

The signature

```
sig
  type t
  val  n    : int
  val  succ : int -> int
end
```

correspondingly splits into two parts: the kind of a single-element collection of types

```
{t :: TYPE}
```

and the type of a collection of two values

```
{n : int, succ : int -> int}.
```

F_ω suffices for these and many other examples. However, a difficulty arises in the specification for sets:

```
structure Set : 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
```

This should split into a specification for a collection `Set_types` of three types and a collection `Set_values` of five values, but what kind should `Set_types` have? It is clear translating the above SML code into the specifications

```
Set_types :: {item :: TYPE, set :: TYPE, setpair :: TYPE}
Set_values : {empty : Set_types.set, ...}
```

(where I have elided the types for the remaining components of `Set_values`) loses important information about the definitions of `item` and `setpair`. If `Set_types.item` is no longer recorded as equal to `int`, then code may suddenly fail to typecheck.

One possibility is to substitute away all such type definitions. Because of the subsignature relation this is not so trivial an operation as it might appear, but there is no essential difficulty [Sha98]. However, in the TILT compiler types correspond to run-time values, and the effect of such a substitution is to duplicate run-time computations. Our goal was to avoid such duplication.

1.3 Dependent and Singleton Kinds

The choice made in TILT was to extend the kind structure with *dependent* and *singleton kinds*. The singleton kind $\mathbf{S}(A :: K)$ is the kind of "all type constructors of kind K which are equal to A. That is, the defining property is that the type constructor A has kind $\mathbf{S}(B :: K)$ if and only if A and B are equal type constructors of kind K. Since the type constructors form a small lambda calculus, I consider equality of types to be based on the usual $\beta\eta$-equivalence of lambda terms[2]. Note that in the presence of singletons assumptions about the kinds of type variables can affect the provable equalities, *and* the equational theory of types affects what types can be shown to have which kinds.

The kinds in TILT were further extended with *dependencies*. First, in kinds of collections of types, the kind of each component may depend upon the contents of earlier components. With this extension, it becomes easy to phase-split the `Set` specification:

$$\texttt{Set\_types :: \{item ::}\ \mathbf{S}(\texttt{int} :: \texttt{TYPE})\texttt{, set :: TYPE, setpair ::}\ \mathbf{S}(\texttt{set*set} :: \texttt{TYPE})\}$$
$$\texttt{Set\_values : \{empty : Set\_types.set, ...\}}$$

Singleton kinds are used here to expose the definitions of `item` and `setpair`. Further, the definition of `setpair` involves a dependency: its kind depends on the contents of the `set` component.

Similarly, in the kinds of functions mapping type constructors to type constructors, the kind of the result is allowed to depend on the argument given to the function. This is used to express the dependencies of types returned from a functor on the functor's argument.

The final extension in the TILT kind structure is a subkinding relation, a preorder $K_1 \leq K_2$ which holds when K_1 is a more-precise (less general) kind than K_2. This relationship is induced by the relation $\mathbf{S}(A :: K) \leq K$; that is, all "types of kind K equivalent to A" are also "types of kind K". Subkinding is used to model the SML sub-signature relation.

1.4 Dependent and Singleton Types

The extensions to the kind level can be applied at the level of types as well. This leads to singleton types of the form $\mathbf{S}(e : \tau)$, the type of "all values of type τ equal to e", as well as dependent

[2] The simpler β-equivalence might suffice in practice, but having both β and η leads to a more expressive and more interesting language. It is also not clear that using this stronger equivalence relation would substantially simplify the metatheoretic results I study in this thesis. (See the proofs for decidability of term equivalence.)

```
sig
   structure BinaryTree : sig
      structure Key : sig
        type t
        val  lesseq : t * t -> bool
      end
      type value
      type tree
      val insert : Key.t * value * tree -> tree
       ... other binary tree operations ...
   end

   structure PriorityQueue : sig
      structure Key : sig
        type t
        val  lesseq : t * t -> bool
      end
      type value
      type pqueue
      val insert : Key.t * value * pqueue -> pqueue
       ... other priority queue operations ...
   end

   sharing BinaryTree.Key = PriorityQueue.Key
end
```

Figure 1.2: Structure Sharing

function and record types, and subtyping

The designer of a system of singleton types must choose a reasonable notion of equality; in the presence of side-effecting program terms this is not obvious. Ideally equality would be observable equivalence: two expressions would be equal if and only if they are indistinguishable in any program context. However, for any interesting term language this relation is not decidable. (For example, checking contextual equivalence with a non-terminating expression in this language is equivalent to the halting problem.) Because typechecking in the presence of singleton types requires determining equivalence of terms, this would immediately lead to a system where there is no algorithm to check the well-formedness of programs.

I choose to study a simple equivalence: a congruence based on projection rules for pairs, extended by singleton types. To avoid problems with side effects, I restrict singleton types to contain only values, and I extend the congruence with the principle that a value v_1 has type $\mathbf{S}(v_2 : \tau)$ if and only if v_1 and v_2 are equivalent and of type τ. (In the presence of recursion there is a non-terminating expression of type τ for any well-formed τ. Hence there is a non-terminating expression e of type $\mathbf{S}(3 : \mathsf{int})$. But since 3 and e are clearly not observably equivalent, they should not be provably equal; hence the restriction to values.)

What use are such singletons? Consider the SML code in Figure 1.2. The interface shown here

```
sig
  structure T : sig
                   val n : int
                end
  structure U : sig
                   val m : int
                end
  sharing T = U
end
```

Figure 1.3: Pointless Structure Sharing

specifies two sub-modules `BinaryTree` and `PriorityQueue` that implement abstract data types for binary trees and priority queues respectively. Each sub-module has its own notion of how keys are represented (the type `Key.t`) and ordered (the relation `Key.lesseq`). In current versions of Standard ML, `sharing` constraints are simply an abbreviation for `sharing type` constraints between the opaque type components common to both structures. Since there is only one such component, the constraint is exactly equal to the constraint

`sharing type BinaryTree.Key.t = PriorityQueue.Key.t.`

This then allows the same key value to be used in a binary tree and in a priority queue. (Note however, that the values stored in binary trees and the values stored in priority queues need not be of the same type; there is no constraint requiring `BinaryTree.value` to be the same type as `PriorityQueue.value`.) This constraint can be modeled as before with singleton kinds by specifying

$$\texttt{PriorityQueue.Key.t} :: \mathbf{S}(\texttt{BinaryTree.Key.t} :: \mathtt{TYPE}).$$

In the original 1990 definition of Standard ML [MTH90], however, the `sharing` constraint in Figure 1.2 actually requires the structures `BinaryTree.Key` and `PriorityQueue.Key` be the *same* structure. As a consequence, not only must the representation type for keys be equal, but the two `lesseq` orderings will be equal. In SML '90 then, whether a given module satisfies this interface or not (a question of typechecking) depends on the *values* of the `Key` substructures.

To model the spirit of this sharing constraint, I can use singleton types. Let t stand for the type `PriorityQueue.Key.t`. Then I can model the constraint by using singleton kinds as previously mentioned and further requiring

$$\texttt{BinaryTree.Key.lesseq} : \mathbf{S}(\texttt{PriorityQueue.Key.lesseq}{:}\, t \;\texttt{*}\; t \;\texttt{->}\; \texttt{bool}).$$

This does not require that the two `Key` structures be exactly the same structure, but does require that corresponding components of the two structures are equal. Because one cannot do assignment directly to components of a structure, however, there is no run-time behavior that can distinguish two componentwise-equal structures; this leads to a more permissive type system while not permitting any changes in run-time behavior.

Not all instances of SML '90 structure sharing can be modeled with singleton types. For example, the signature in Figure 1.3 requires that the `T` and `U` substructures be different views of the same underlying structure. It makes no sense to model this with a dependent record type such as

$$\{\texttt{T : \{n : int\}, U : } \mathbf{S}(\texttt{T} : \{\texttt{m : int}\})\}$$

because this would be ill-formed; T does not have type {m : int}. However, since the sharing constraint in Figure 1.3 does not actually place any restriction on the values of the n and m components, the practical utility of such a specification seems extremely minimal.

1.5 Other Uses for Singletons

1.5.1 Closed-Scope Definitions

In many λ-calculi "let-bindings" or "closed-scope definitions" are treated as syntactic sugar. For example,

```
let x:int = 3 in (x+1)
```

would be encoded as the function application

$$(\lambda \texttt{x:int. x+1})(3).$$

However, this sort of transformation is not always legal. In F_ω, for example, one cannot generally equate

$$\texttt{let t::TYPE = int*int in } e$$

where e is some expression with

$$(\Lambda \texttt{t::TYPE. } e)(\texttt{int*int})$$

because in the former case we know that $\texttt{t} = \texttt{int*int}$ while typechecking e, while in the latter case e must be typecheckable knowing only that t is *some* type.

The alternative definition

$$[\texttt{int*int/t}]e$$

(that is, the result of replacing t with int*int everywhere in e) will preserve meaning and well-typedness, but involves arbitrary duplication of types.

Some authors have therefore considered let-bindings (and generally, the notion of variables-with-definitions) appears as a primitive. For example, the pure type system of Severi and Poll [SP94] adds a new let-binding primitive written $x{=}a{:}A \ \mathit{in}\ b$, and the definitions of variables are maintained during typechecking.

In a language with singleton kinds, however, let-bindings of types become definable via functions:

$$\texttt{let t::TYPE = int*int in } e$$

becomes

$$(\Lambda \texttt{t::}\mathbf{S}(\texttt{int*int :: TYPE}).\ e)(\texttt{int*int}).$$

This time the typechecker knows while typechecking e that $\texttt{t} = \texttt{int*int}$ because this is apparent from the kind of t.

1.5.2 TILT Program Transformations

The encoding of let in the previous section is primarily a theoretic curiosity. However, similar transformations do come up in practice; there are several places in the TILT compiler where it could be beneficial to take types computed within a function body and turn these into new type arguments to be passed into the function at run-time. This comes up in loop invariant removal, in uncurrying, and in closure conversion [MMH96]. An example will make this clearer; consider the following code, written in an approximation of the compiler's internal representation:

```
let
   function F(α::TYPE, y:α) = G(α×α, (y,y))
in
   ... F(int, 3) ... F(int, 4) ... F(int, 5) ...
end
```

This code presupposes a polymorphic function G taking a type and an argument of this type. The polymorphic function F also takes a type α and a value y of this type; it creates the pair (y,y) and its type $\alpha{\times}\alpha$, and then passes these to G. Elsewhere in the code, F is called several times.

Now on each call, F constructs the type $\alpha{\times}\alpha$ in order to be passed this G. In a type-passing implementation like TILT, this corresponds to actual instructions executed at run-time. Since F is repeatedly being given the same type argument int, it would be preferable to compute int×int just once; this could be performed by having the caller pass int×int as a new function argument. Such a transformation leads to the following code:

```
let
   function F(α::TYPE, β::TYPE, y:α) = G(β, (y,y))
   type t = int×int
in
   ... F(int, t, 3) ... F(int, t, 4) ... F(int, t, 5) ...
end
```

Operationally, this new code is correct. Unfortunately, it no longer typechecks; in a standard typed lambda calculus there is no way to perform this particular transformation while preserving well-typedness.

The problem with the above code is that according to the specification of the arguments, F could be called with *any* two types. Therefore, there is no reason why the pair (x,x) should have type β. The intent is that every call to F should pass a type α and the type $\alpha{\times}\alpha$, but if this is not a constraint being checked by the type system it is unsafe to assume this will always be true.

The TILT compiler is based on the principle of type-preserving transformations; we forbid transformations leading to ill-typed programs. What is needed is a way to constrain the new type variable so that the compiler knows it will be given the type $\alpha{\times}\alpha$. Equally importantly, the compiler should be able to check that every application of F obeys this constraint.

Singleton kinds provide exactly the mechanism required to transform type expressions into function arguments while preserving well-typedness. The code becomes

```
let
   function F(α::TYPE, β::S(α×α :: TYPE), y:α) = G(β, (y,y))
   type t = int×int
in
   ... F(int, t, 3) ... F(int, t, 4) ... F(int, t, 5) ...
end
```

This typechecks because we have introduced the appropriate constraint into the type system; the body of the function F will typecheck if we can show that the type constructor β is equivalent to the type of (y, y), namely $\alpha{\times}\alpha$. But $\beta{::}\mathbf{S}(\alpha{\times}\alpha :: \mathtt{TYPE})$ implies that $\beta \equiv \alpha{\times}\alpha :: \mathtt{TYPE}$, as required.

Note that an apparently simpler solution to this problem would be to compile F in curried fashion:

```
let
   function F(α::TYPE) =
      let
         type β = α×α
         function F'(x:α) = G(β, (y,y))
      in
         F'
      end
   F_int = F(int)
in
   ... F_int(3) ... F_int(4) ... F_int(5) ...
end
```

Here F now just takes a single argument, a type α. It computes $\alpha \times \alpha$ and returns a function which expects an argument x of type α. The caller can apply F to int once (computing int×int once) and then apply the resulting function repeatedly. This does typecheck without singletons, and might seem to solve the problem. However, this transformation introduces higher-order functions, which are implemented via a transformation called closure conversion. The closure-conversion transformation involves taking every function and turning its free variables into arguments; in particular, β will become an argument of the function F', and we have exactly the same typechecking problem as we started out with [MMH96].

1.5.3 Cross-Module Inlining

While language features such as abstraction, modularity, polymorphism and higher-order functions have important software engineering benefits, they often impose a run-time cost. Using abstract types or polymorphism can mean that data layouts are not known until run-time. Uses of modularity and higher-order functions can substantially increase the number of function calls, which can be particularly costly on modern processors.

If pieces of a program are compiled and optimized completely separately ("true" separate compilation) it is hard to avoid the costs of abstraction. At the other end of the spectrum, a compiler can do whole-program optimization and generate substantially better code. Unfortunately, the analysis required is usually unusably slow for large inputs and requires source code for the entire program (including libraries). However, in many cases it suffices to do incremental compilation, in which each file is compiled after all of its imports. This allows the compiler to use information gathered while compiling the imports in order to do a better job of compiling the current file. The compiler writer must then decide what information the compiler should collect and store and how to represent it.

For separate compilation in a statically typed framework, a minimal requirement is that the compiler must know the type of all external references. This leads to such mechanisms as header files in C, where the interface of a compilation unit gives the types of its exported components. This also leaves open the possibility of checking that a compilation unit matches the claimed interface.

An elegant and systematic method of handling incremental compilation is to use the same mechanism — where the interface of each unit contains typing information for all exports — but to have the compiler generate the interface directly from the code. This combines cleanly with separate compilation; the programmer can write interfaces for some pieces of the program and have the compiler generate the remainder.

Of course the compiler can determine more information than just simple types when given the source code. A very important optimization for incremental compilation is cross-module inlining. This transformation replaces references to imported values, types, and functions with their actual implementations. In order to achieve this, the interface must express this information, namely to include the implementations of abstract types, values of variables, definitions of functions, and so on. Thus interfaces change from specifying that "`x` is an integer constant" to "`x` is an integer constant equal to 3" and from "`succ` is a function mapping floats to floats" to "`succ` is equal to the function which maps a float f into $f{+}1.0$". In order to maintain the elegance of interfaces containing only type information, this optimization requires a more expressive type system in which such information can be expressed.

Inlining is the process of replacing a reference to a value with the value itself. In my system of singleton types, if $v : S(v' : \tau)$ then the compiler may replace any use of v (in a context expecting a value of type τ) with v'. Singletons can be directly applied to traditional cross-module inlining. Suppose we want to be able to take a definition such as the following (for the successor function on integers)

$$\mathtt{succ} = \lambda x{:}\mathtt{int}.x{+}1$$

and allow other modules to replace `succ` by this function (if it seems locally beneficial). This can be achieved by specializing the type of `succ` in the interface; instead of saying

$$\mathtt{succ} : \mathtt{int} \rightharpoonup \mathtt{int}$$

it can instead say

$$\mathtt{succ} : \mathbf{S}(\lambda x{:}\mathtt{int}.x{+}1 : \mathtt{int} \rightharpoonup \mathtt{int}).$$

Conversely if the compiler sees that an import such as `succ` has a singleton type, it is justified in replacing this reference with the actual definition.

The restriction that well-formed singletons can contain only values suffices for most inlining purposes because the most important case is inlining of function definitions, and functions are values. It is possible that a less conservative approximation might be useful so that we can inline, for example, polymorphic instantiations and partial applications of curried functions. This should be possible by replacing this restriction to values with a restriction to a set of "valuable terms", terms whose evaluation is guaranteed to terminate without side-effects or reference to mutable storage [HS00].

Values in singletons need not be closed, but they must be well-formed and hence cannot refer to items not exported in the interface. In practice, this may require extending interfaces with extra components.

Note that the approach to inlining using singletons is subtly different from C++ `inline` functions in header files, or of the lambda-splitting of Blume and Appel [BA97]. There the functions to be inlined are essentially definitions prepended to the program unit being compiled. Whenever the compiler decides not to inline uses of these functions, it must compile a new local version of the code to call. In contrast, singleton types and kinds used for inlining purposes are specifications of an imported piece of code, which may be referred to if inlining does not appear useful. (Of course, since the compiler has the definition it could also *choose* to create a local copy of the code to call, as yet another alternative to inlining the function's code.)

A more interesting problem is the case where the compiler wants to inline an import which may not have been written yet. This can only occur, of course, if the compiler has some reason to believe it can correctly "predict" what the import's eventual implementation will be. An example of this arises in TILT due to Standard ML datatypes.

The `datatype` mechanism is one of the most successful features of Standard ML. Datatypes combine notions of enumerations, tagged unions, and recursive types into a common framework. A single datatype definition such as

```
datatype tree = Leaf of int | Node of tree*tree
```

automatically generates

- An abstract type `tree`.
- The functions `Leaf` of type `int->tree` and `Node` of type `tree*tree -> tree` for creating new trees;
- Support for discrimination and decomposition for values of type `tree` via pattern-matching;
- A structural equality for trees.

This can be easily modeled as a structure containing one (abstract) type component and several value components. Similarly, a datatype specification signature would correspond to the signature of the appropriate structure [HS00, HS97].

The disadvantage of this elegant encoding is efficiency. Datatype constructors and pattern-matching are used heavily in SML code; making every such use into a function call is unacceptably inefficient. Similarly, although datatypes are officially abstract and must be typechecked as such in the source code, it is often possible to determine from a datatype's description the underlying implementation type for this datatype[3]. Taking advantage of this knowledge would enable more efficient code generation.

Blume [Blu97] suggests that this problem can be overcome by aggressive cross-module inlining. As the functions corresponding to datatype constructors and pattern-matching are generally small pieces of code, they will automatically be exported by the defining compilation unit and inlined into client compilation units. This approach seems logical and should work quite well — but only where it applies. A deficiency is that it does not help when doing separate compilation or compiling SML functors (parameterized modules) which take datatypes as arguments. In these cases no datatype implementation has been specified yet, so there is nothing to inline.

However, if the compiler can predict which types and code will be later supplied as the functor argument, then we are justified in inlining these types and code into the functor body and ignoring the actual argument when it is later applied. There is no typechecking problem involved in this transformation, but for correctness purposes it might be convenient to have a way of formalizing this prediction and a way of checking that the prediction was correct. Singleton types and kinds provide a natural way to record such a prediction: the functor's arguments can be annotated with singleton types and kinds for the datatype components, and inlining can then proceed as discussed above.

Note that because specializing the functor argument to require a particular datatype implementation gives the functor a strictly less-general type, functor applications which were previously valid may no longer typecheck. This is actually an advantage because a typechecking failure occurs when the predicted code does not match the actual implementation; since both parts are automatically generated by the compiler, a typechecking failure here must mean that the compiler is in error.

There is nothing original about inlining datatypes, separately compiled or not. Any reasonable ML compiler *must* do this for efficiency. However, this often occurs in an ad-hoc fashion. With singleton types and kinds a compiler can systematically maintain the datatypes-as-structures encoding throughout the entire compiler, without any loss of efficiency.

[3]In general this may require a non-trivial equational theory for recursive types, however [CHC+98].

1.6 Dissertation Summary

In Chapter 2, I introduce the MIL_0 calculus, a formalization of the key features of the TILT intermediate representation. This language is an predicative variant of the familiar lambda-calculus F_ω, extended with pairs, recursion, and singleton types and kinds. I show that the addition of singletons leads to a calculus with very interesting equational properties; most notably, whether two type constructors are provably equivalent depends strongly on both the typing context and on the kind at which the type constructors are compared.

Chapter 3 contains proofs for many standard properties of the MIL_0 calculus, such as preservation of well-typedness under substitutions and the admissibility of useful typing rules. In particular, although the definition of MIL_0 includes only a very restricted form of singleton kind, general singleton kinds are definable.

Chapter 4 gives algorithms for deciding the kind and constructor-level judgments (e.g., given a well-formed context and a type constructor A, determine whether there is a kind K such that A is well-formed with kind K). This includes an algorithm for constructor equivalence inspired by Coquand's approach to $\beta\eta$-equivalence for a type theory with Π types and one universe [Coq91]. Coquand worked with an algorithm which directly decides equivalence, rather than defining a confluent and strongly-normalizing reduction relation. In contrast to Coquand's system, MIL_0 type constructors cannot be compared by shape alone; equivalence depends on both the typing context and the classifier. Where Coquand maintains a set of bound variables, my algorithm maintains a full typing context. Similarly, he uses shapes of the items being compared to guide the algorithm where my algorithm uses the classifying kind. (For example, where Coquand would check whether either constructor is a lambda-abstraction, this algorithm checks whether the constructors are being compared at a function kind.) I show the algorithms are sound with respect to the language definition.

In Chapter 5 I prove the completeness and termination of the algorithms in the previous chapter. This reduces to proving the completeness and termination of the constructor equivalence algorithm. Unfortunately I 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 are immediately evident in Coquand's case.) Instead, I analyze a related but less efficient algorithm which restores symmetry and transitivity by maintaining redundant information. The proof that this revised algorithm is complete and terminating for all well-formed inputs was inspired by Coquand's use of Kripke logical relations, but the details differ substantially. My proof uses a novel form of Kripke logical relation employing two worlds, rather than one. The correctness of the revised algorithm can then be used to show the correctness of the original, simpler constructor equivalence algorithm. This yields the implementation used by the TILT compiler.

I then repeat the development for types and terms. Chapter 6 gives algorithms for deciding the type and term-level judgments; I show these algorithms are also sound with respect to the corresponding judgments in the MIL_0 definition. The proof of Chapter 7 for the completeness and termination of the term and type algorithms proceeds essentially along the same lines as the proofs in Chapter 5. The simpler notion of equivalence for term-level functions makes some parts of these proofs easier, but others are complicated by the fact that type equivalence is less trivial than kind equivalence.

Chapter 8 shows the MIL_0 type system to be sound with respect to its operational semantics. The proof is very straightforward, but depends critically on using the soundness and completeness of the constructor equivalence algorithm to show consistency properties of constructor equivalence.

In Chapter 9 I show how to extend these proofs when the MIL language is extended with intensional polymorphism (i.e., with run-time constructor analysis constructs) [HM95, Mor95]. This involves surprisingly little change to the previous development.

Finally, Chapter 10 surveys the related literature and concludes with a collection of conjectures and possibilities for future work.

Chapter 2

The MIL_0 calculus

2.1 Overview

The TILT compiler uses as its main internal representation of programs a typed language called the "Mid-level Intermediate Language", or MIL. This is a relatively high-level language; it includes first-class functions, assignment, and exception handling, with no explicit reference to memory layout or allocation/deallocation. However, it contains no notion of a module system.

More formally MIL is a variant of F_ω, the higher-order polymorphic lambda calculus [Gir72]. The language has four levels:

- The *terms* or *expressions* of the language. These include constants, recursive functions, applications, pairs, records, assignments, exceptions, etc.
- The *types*, which classify terms. A term is well-formed if and only if it has a type.
- The *type constructors*, or simply *constructors*.[1] This level contains items corresponding to certain types (these constructors might be considered "the names of types" or "types as data") as well as functions and pairs, forming a small λ-calculus in itself.
- The *kinds*, which serve as types for the language of constructors.

The distinction between types and the corresponding type constructors is made because MIL is a *predicative* language. In an impredicative language, polymorphic types involve quantification over all types, including the polymorphic types themselves. Although one can make sense of this circularity [Gir72], it substantially complicates the metatheory of the language and hence has been avoided here.

In this chapter, I formally define MIL_0, a simplified calculus which captures most of the essential features of the full MIL. The primary differences are:

- The term language has been substantially pared down to contain only recursive functions, pairs, and polymorphism. Assignment and exceptions have been omitted, so that the only remaining side-effect is nontermination. In the full MIL, functions can take any fixed number of constructor and term arguments, and polymorphic recursion is allowed. (When compiling a source language like SML which does not allow polymorphic recursion [Myc84], however, the utility of this last feature is limited.) For simplicity, MIL_0 separates term abstractions and polymorphic abstractions, and disallows polymorphic recursion.

[1]This terminology conflicts with the common usage of "constructor" in ML to refer to the term constructors defined by datatypes. However, context will always make clear which sense of constructor is meant.

- MIL function types have been similarly split into universally-quantified types for polymorphic expressions and ordinary (dependent) function types for term-level functions. MIL contains several varieties of function type (the types of potentially open functions, closed functions, or closures, each of which may be partial or total). Only potentially open, partial functions are modeled here.
- Constructor functions in MIL are multiargument, while MIL_0 constructor functions must be curried to get the same effect.
- For clarity, all constructor analysis constructs used by TILT (e.g., typecase or typerec [HM95]) have been omitted from MIL_0. Such features are essentially orthogonal to my main topic, the effects of adding singletons to the calculus. However, the methods of this dissertation can be applied even in the presence of constructor analysis. In chapter 9 I sketch the (minor) changes to the development required.
- The MIL as actually implemented uses a relatively strong equivalence for recursive type constructors. (Specifically, two recursive type constructors are considered equivalent if their unrollings are equivalent [CHC+98].) This extension is omitted from MIL_0.

For the most part, extending the theory of this chapter to handle the full MIL should not present any fundamental difficulty. The proofs do become more technically involved (for example, when going from pairs to n-ary labeled records) but the essential arguments do not change. Note that since this is an explicitly-typed framework, adding polymorphic recursion creates no challenges.

The one case where the methods do not extend is when considering an interesting equational theory for recursive types. (I see no way to create an *obviously* symmetric and transitive algorithm in the presence of recursive types.) There is an obvious extension of my algorithms that appears to work in practice; the FLINT compiler uses a very similar algorithm.

This is not simply an issue of adding singletons; in the literature there appears to be little study of algorithms for equating recursive types when there are interesting equations beyond those induced by recursive types. (The only instance I have found is the work of Palsberg and Zhao on type isomorphisms in the presence of recursive types [PZ00].) For example, no one has looked at the decidability of typechecking for F_ω (where there is β-equivalence at the type level) extended with recursive types.

As an alternative to extending the theory to the full MIL, the language itself could be simplified. An alternative MIL could use use a much simpler equational theory for recursive types, at the cost of requiring explicit type coercions (i.e., isorecursive types rather than equirecursive types [CHC+98]). There are no problems in extending the theory of MIL_0 in this fashion.

This chapter contains a definition of MIL_0 split into two parts: compile-time and run-time aspects. §2.2 contains the context-free syntax of the language and the context-sensitive rules for determining whether phrases in the language are well-formed, and §2.3 contains a number of admissible rules which follow from this definition. Then §2.4 explains the meanings of complete programs by defining a notion of evaluation.

2.2 Syntax and Static Semantics of MIL_0

The abstract syntax of MIL_0 is shown in Figure 2.1. As usual, I work modulo renaming of bound variables (i.e., modulo α-equivalence). The meaning of each construct is explained in tandem with the static semantics.

Typing Contexts	$\Gamma, \Delta ::=$	$\bullet$	Empty context
	$\mid$	$\Gamma, \alpha{::}K$	
	$\mid$	$\Gamma, x{:}\tau$	
Kinds	$K, L ::=$	$\mathbf{T}$	Kind of names of types
	$\mid$	$\mathbf{S}(A)$	Singleton kind
	$\mid$	$\Pi\alpha{::}K'.K''$	Dependent function kind
	$\mid$	$\Sigma\alpha{::}K'.K''$	Dependent pair kind
Base Constructors	$b ::=$	$\mathsf{Int} \mid \mathsf{Boxedfloat} \mid \ldots$	Names of base types
Constructor Constants	$c ::=$	b	
	$\mid$	$\times$	Pair-type constructor
	$\mid$	$\rightharpoonup$	Function-type constructor
Type Constructors	$A, B ::=$	c	
	$\mid$	$\alpha, \beta, \ldots$	Variables
	$\mid$	$\lambda\alpha{::}K'.A$	Function
	$\mid$	$A\,A'$	Application
	$\mid$	$\langle A', A''\rangle$	Pair of constructors
	$\mid$	$\pi_i A$	Projection
Types	$\tau, \sigma ::=$	$\mathit{Ty}(A)$	Inclusion of type constructors
	$\mid$	$\mathbf{S}(v : \tau)$	Singleton type
	$\mid$	$\forall\alpha{::}K.\tau$	Polymorphic type
	$\mid$	$(x{:}\tau')\rightharpoonup\tau''$	Dependent function type
	$\mid$	$(x{:}\tau')\times\tau''$	Dependent pair type
Values	$v, w ::=$	n	Integer constants
	$\mid$	$x, f, \ldots$	Variables
	$\mid$	$\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e$	Recursive function
	$\mid$	$\Lambda(\alpha{::}K){:}\tau.e$	Polymorphic abstraction
	$\mid$	$\pi_i v$	Projection
	$\mid$	$\langle v_1, v_2\rangle$	Pair
Terms	$e, d ::=$	v	
	$\mid$	$v\,v'$	Application
	$\mid$	$v\,A$	Polymorphic instantiation
	$\mid$	$\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}$	Local variable definition

Figure 2.1: Syntax of the MIL_0 Calculus

$\Gamma \vdash \text{ok}$	Well-formed context
$\vdash \Gamma_1 \equiv \Gamma_2$	Context equivalence
$\Gamma \vdash K$	Well-formed kind
$\Gamma \vdash K_1 \leq K_2$	Subkinding
$\Gamma \vdash K_1 \equiv K_2$	Kind equivalence
$\Gamma \vdash A :: K$	Well-formed constructor
$\Gamma \vdash A_1 \equiv A_2 :: K$	Constructor equivalence
$\Gamma \vdash \tau$	Well-formed type
$\Gamma \vdash \tau_1 \leq \tau_2$	Subtyping
$\Gamma \vdash \tau_1 \equiv \tau_2$	Type equivalence
$\Gamma \vdash e : \tau$	Well-formed term
$\Gamma \vdash e_1 \equiv e_2 : \tau$	Term equivalence

Figure 2.2: Judgment Forms in the Static Semantics

The static semantics (type system) for MIL_0 is given as a collection of inductively-defined judgments. Figure 2.2 lists all the different judgment forms. The purpose of this section is to explain and motivate the choice of judgments.

The definition of the static semantics requires a few preliminary comments. First, the notation $\text{FV}(\mathit{phrase})$ refers to the set of free variables in *phrase*. This is defined Figure 2.3 by induction on syntax.

Secondly, the static semantics uses the notion of capture-avoiding substitution: I use the metavariable γ to stand for an arbitrary mapping from constructor variables to arbitrary constructors and from term variables to term values. The notation $\gamma(\mathit{phrase})$ is used to represent the result of applying γ to all free variables in the phrase *phrase*. The substitution which sends α to A and leaves all other variables unchanged is written $[A/\alpha]$, and $[v/x]$ is define analogously. If γ is a substitution, then $\gamma[\alpha{\mapsto}A]$ stands for the mapping which sends α to A and behaves like γ for all other variables; the notation $\gamma[x{\mapsto}v]$ is defined analogously.

2.2.1 Typing Contexts

A *typing context* Γ (or simply *context* when this is unambiguous) represents assumptions for the types of free term variables and for the kinds of free constructor variables. It is represented as a finite sequence of variable/classifier associations. Typing contexts in MIL_0 are intrinsically sequences because of dependencies introduced by singletons: both types and kinds can refer to constructor variables appearing earlier in the context, while types can additionally refer to term variables appearing earlier in the context.

The *context validity* judgment determines when a context is well-formed: every type or term appearing in the context must be well-formed with respect to the preceding segment of the context.

$$\frac{}{\bullet \vdash \text{ok}} \tag{2.1}$$

$$
\begin{array}{ll}
\mathrm{FV}(\mathbf{T}) & := \emptyset \\
\mathrm{FV}(\mathbf{S}(A)) & := \mathrm{FV}(A) \\
\mathrm{FV}(\Pi\alpha{::}K'.K'') & := \mathrm{FV}(K') \cup (\mathrm{FV}(K'') \setminus \{\alpha\}) \\
\mathrm{FV}(\Sigma\alpha{::}K'.K'') & := \mathrm{FV}(K') \cup (\mathrm{FV}(K'') \setminus \{\alpha\}) \\
\\
\mathrm{FV}(A) & := \emptyset \\
\mathrm{FV}(\alpha) & := \{\alpha\} \\
\mathrm{FV}(\lambda\alpha{::}K.A) & := \mathrm{FV}(K) \cup (\mathrm{FV}(A) \setminus \{\alpha\}) \\
\mathrm{FV}(A\,A') & := \mathrm{FV}(A) \cup \mathrm{FV}(A') \\
\mathrm{FV}(\langle A', A''\rangle) & := \mathrm{FV}(A') \cup \mathrm{FV}(A'') \\
\mathrm{FV}(\pi_i A) & := \mathrm{FV}(A) \\
\\
\mathrm{FV}(\mathit{Ty}(A)) & := \mathrm{FV}(A) \\
\mathrm{FV}(\mathbf{S}(v : \tau)) & := \mathrm{FV}(v) \cup \mathrm{FV}(\tau) \\
\mathrm{FV}(\forall\alpha{::}K.\tau) & := \mathrm{FV}(K) \cup (\mathrm{FV}(\tau) \setminus \{\alpha\}) \\
\mathrm{FV}((x{:}\tau')\rightharpoonup\tau'') & := \mathrm{FV}(\tau') \cup (\mathrm{FV}(\tau'') \setminus \{x\}) \\
\mathrm{FV}((x{:}\tau')\times\tau'') & := \mathrm{FV}(\tau') \cup (\mathrm{FV}(\tau'') \setminus \{x\}) \\
\\
\mathrm{FV}(n) & := \emptyset \\
\mathrm{FV}(x) & := \{x\} \\
\mathrm{FV}(\mathsf{fun}\ f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e) & := \mathrm{FV}(\tau') \cup (\mathrm{FV}(\tau'') \setminus \{x\}) \cup (\mathrm{FV}(e) \setminus \{x, f\}) \\
\mathrm{FV}(\Lambda(\alpha{::}K){:}\tau.e) & := \mathrm{FV}(K) \cup (\mathrm{FV}(\tau) \setminus \{\alpha\}) \cup (\mathrm{FV}(e) \setminus \{\alpha\}) \\
\mathrm{FV}(\pi_i v) & := \mathrm{FV}(v) \\
\mathrm{FV}(\langle v', v''\rangle) & := \mathrm{FV}(v') \cup \mathrm{FV}(v'') \\
\\
\mathrm{FV}(v\,v') & := \mathrm{FV}(v) \cup \mathrm{FV}(v') \\
\mathrm{FV}(v\ A) & := \mathrm{FV}(v) \cup \mathrm{FV}(A) \\
\mathrm{FV}(\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}) & := \mathrm{FV}(\tau') \cup \mathrm{FV}(e') \cup (\mathrm{FV}(e) \setminus \{x\}) \cup \mathrm{FV}(\tau)
\end{array}
$$

Figure 2.3: Free Variable Sets

$$\frac{\Gamma \vdash K}{\Gamma, \alpha{::}K \vdash \mathrm{ok}} \quad (\alpha \notin \mathrm{dom}(\Gamma)) \tag{2.2}$$

$$\frac{\Gamma \vdash \tau}{\Gamma, x{:}\tau \vdash \mathrm{ok}} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{2.3}$$

The side-condition in Rules 2.2 and 2.3 ensures that variables are not bound in a context more than once. It follows that well-formed typing contexts can also be viewed as finite functions: $\Gamma(\alpha)$ represents the kind associated with α in Γ, while $\Gamma(x)$ represents the type associated with x in Γ. Similarly, the notation $\mathrm{dom}(\Gamma)$ is used to represent the set of all constructor and term variables bound by Γ. The free variables of a context, $\mathrm{FV}(\Gamma)$, can then be defined inductively as follows:

$$\begin{array}{ll}
\mathrm{FV}(\bullet) & := \emptyset \\
\mathrm{FV}(\Gamma, \alpha{::}K) & := \mathrm{FV}(\Gamma) \cup (\mathrm{FV}(K) \setminus \mathrm{dom}(\Gamma)) \\
\mathrm{FV}(\Gamma, x{:}\tau) & := \mathrm{FV}(\Gamma) \cup (\mathrm{FV}(\tau) \setminus \mathrm{dom}(\Gamma))
\end{array}$$

Because contexts are finite sequences, there is an obvious definition for appending any two contexts. The result of appending Γ_1 and Γ_2 is written Γ_1, Γ_2.

A similar set of inference rules gives a notion of definitional equivalence for two contexts.

$$\frac{}{\vdash \bullet \equiv \bullet} \tag{2.4}$$

$$\frac{\vdash \Gamma_1 \equiv \Gamma_2 \qquad \Gamma_1 \vdash K_1 \equiv K_2}{\vdash \Gamma_1, \alpha{::}K_1 \equiv \Gamma_2, \alpha{::}K_2} \quad (\alpha \notin \mathrm{dom}(\Gamma_1)) \tag{2.5}$$

$$\frac{\vdash \Gamma_1 \equiv \Gamma_2 \qquad \Gamma_1 \vdash \tau_1 \equiv \tau_2}{\vdash \Gamma_1, x{:}\tau_1 \equiv \Gamma_2, x{:}\tau_2} \quad (x \notin \mathrm{dom}(\Gamma_1)) \tag{2.6}$$

It is obvious that any two equivalent contexts bind the same variables in the same order. I show later that if two contexts are equivalent then they are both well-formed and they are interchangeable in any declarative judgment.

2.2.2 Kinds

The *kind validity* judgment specifies when a kind is well-formed with respect to a given typing context. The kind $\mathbf{T}$ is the kind of all "ordinary" type constructors; that is, the kind of type constructors corresponding to some type.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathbf{T}} \tag{2.7}$$

The premise of Rule 2.7 ensures that in any proof of $\Gamma \vdash K$ there is strict subderivation proving $\Gamma \vdash \mathrm{ok}$. A similar property holds for all of the judgments defined in this chapter; I show this in §3.1.

Well-formed MIL_0 singleton kinds are restricted: they may only contain constructors of kind $\mathbf{T}$. The kind annotation is therefore omitted from the syntax, as it would always be $\mathbf{T}$.

$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash \mathbf{S}(A)} \tag{2.8}$$

However, general singleton kinds $\mathbf{S}(A :: K)$ as described in the introduction are *definable* (see §2.3).

The rules for Π and Σ kinds (dependent function kinds and dependent pair kinds) are essentially standard.

$$\frac{\Gamma, \alpha{::}K' \vdash K''}{\Gamma \vdash \Pi\alpha{::}K'.K''} \tag{2.9}$$

$$\frac{\Gamma, \alpha{::}K' \vdash K''}{\Gamma \vdash \Sigma\alpha{::}K'.K''} \tag{2.10}$$

$\Pi\alpha{::}K'.K''$ is the kind of all functions which map an argument α of kind K' to a result of kind K'', where K'' may depend on α. Similarly, $\Sigma\alpha{::}K'.K''$ is the kind of all pairs of constructors whose first component α has kind K' and whose second component has kind K'', where K'' may refer to α. Both $\Pi\alpha{::}K'.K''$ and $\Sigma\alpha{::}K'.K''$ bind the constructor variable α in K''. I use the usual notation $K'{\times}K''$ for $\Sigma\alpha{::}K'.K''$ and $K'{\to}K''$ for $\Pi\alpha{::}K'.K''$ in those cases where α does not appear free in K''.

Frequently one might see an additional premise $\Gamma \vdash K'$ in these two rules, but as MIL_0 is defined this is already implied by the existing premise.

The *subkinding* judgment $\Gamma \vdash K_1 \leq K_2$ defines a preorder on kinds, which may be intuitively understood to say that K_1 is more precise (exposes more information about a type constructor) than K_2. It will follow that any constructor of kind K_1 will be acceptable in a context requiring a constructor of kind K_2.

Intuitively, since $\mathbf{S}(A)$ represents "the kind of all constructors of kind $\mathbf{T}$ equivalent to A", any constructor of this kind should be acceptable where a constructor of kind $\mathbf{T}$ is expected. Thus the key subkinding rule is:

$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash \mathbf{S}(A) \leq \mathbf{T}} \tag{2.11}$$

The premise of this rule ensures that $\mathbf{S}(A)$ is well-formed.

Subkinding between two singleton kinds coincides with equivalence

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}}{\Gamma \vdash \mathbf{S}(A_1) \leq \mathbf{S}(A_2)} \tag{2.12}$$

because a constructor of kind $\mathbf{T}$ equivalent to A_1 can be equivalent to A_2 if and only if A_1 and A_2 are equivalent to each other.

The following rule is required for subkinding to be reflexive.

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathbf{T} \leq \mathbf{T}} \tag{2.13}$$

The remaining subkinding rules lift the relation to Π and Σ kinds, following the usual co- and contravariance properties. (The first premise in each of the following two rules ensures that $\Gamma \vdash K_1 \leq K_2$ implies $\Gamma \vdash K_1$ and $\Gamma \vdash K_2$.)

$$\frac{\begin{array}{c}\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \\ \Gamma \vdash K_2' \leq K_1' \qquad \Gamma, \alpha{::}K_2' \vdash K_1'' \leq K_2''\end{array}}{\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \leq \Pi\alpha{::}K_2'.K_2''} \tag{2.14}$$

$$\frac{\Gamma \vdash \Sigma\alpha{::}K_2'.K_2'' \qquad \Gamma \vdash K_1' \leq K_2' \qquad \Gamma, \alpha{::}K_1' \vdash K_1'' \leq K_2''}{\Gamma \vdash \Sigma\alpha{::}K_1'.K_1'' \leq \Sigma\alpha{::}K_2'.K_2''} \tag{2.15}$$

Kind equivalence, denoted $\Gamma \vdash K_1 \equiv K_2$, is essentially a symmetrized version of subkinding. I show later that $\Gamma \vdash K_1 \equiv K_2$ if and only if $\Gamma \vdash K_1 \leq K_2$ and $\Gamma \vdash K_2 \leq K_1$, and a reasonable alternative presentation of the system would make this the definition of kind equivalence.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathbf{T} \equiv \mathbf{T}} \tag{2.16}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}}{\Gamma \vdash \mathbf{S}(A_1) \equiv \mathbf{S}(A_2)} \tag{2.17}$$

$$\frac{\Gamma \vdash \Pi\alpha{::}K_2'.K_2'' \qquad \Gamma \vdash K_1' \equiv K_2' \qquad \Gamma, \alpha{::}K_1' \vdash K_1'' \equiv K_2''}{\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \equiv \Pi\alpha{::}K_2'.K_2''} \tag{2.18}$$

$$\frac{\Gamma \vdash \Sigma\alpha{::}K_2'.K_2'' \qquad \Gamma \vdash K_1' \equiv K_2' \qquad \Gamma, \alpha{::}K_1' \vdash K_1'' \equiv K_2''}{\Gamma \vdash \Sigma\alpha{::}K_1'.K_1'' \equiv \Sigma\alpha{::}K_2'.K_2''} \tag{2.19}$$

2.2.3 Type Constructors

The constructors include names for base types, all with kind $\mathbf{T}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash b :: \mathbf{T}} \quad b \in \{\mathsf{Int}, \mathsf{Boxedfloat}, \mathsf{Char}, \ldots\} \tag{2.20}$$

and constants for creating product types and function types:

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \times :: \mathbf{T}{\to}(\mathbf{T}{\to}\mathbf{T})} \tag{2.21}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \rightharpoonup :: \mathbf{T}{\to}(\mathbf{T}{\to}\mathbf{T})} \tag{2.22}$$

Applications of these constants to two arguments will be written in the usual infix manner, $A_1{\times}A_2$ and $A_1{\rightharpoonup}A_2$.

As constructors form a λ-calculus, there are variables, functions mapping constructors to constructors, and applications of such functions.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \alpha :: \Gamma(\alpha)} \quad (\alpha \in \mathrm{dom}(\Gamma)) \tag{2.23}$$

$$\frac{\Gamma, \alpha{::}K' \vdash A :: K''}{\Gamma \vdash \lambda\alpha{::}K'.A :: \Pi\alpha{::}K'.K''} \tag{2.24}$$

$$\frac{\Gamma \vdash A :: K' {\rightarrow} K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash A\,A' :: K''} \tag{2.25}$$

Since the constructors form a dependently-typed λ-calculus, the formulation of Rule 2.25 (which permits only applications of functions with non-dependent types) may appear surprisingly restrictive. However, a consequence of having singleton kinds is that this rule implies the more traditional formulation allowing dependencies, which becomes admissible (see §2.3).

Similarly one can form pairs of constructors, and perform projections from such pairs.

$$\frac{\Gamma \vdash A' :: K' \qquad \Gamma \vdash A'' :: K''}{\Gamma \vdash \langle A', A'' \rangle :: K' {\times} K''} \tag{2.26}$$

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_1 A :: K'} \tag{2.27}$$

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A :: [\pi_1 A/\alpha]K''} \tag{2.28}$$

Next, there is an obvious introduction rule for singletons.

$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash A :: \mathbf{S}(A)} \tag{2.29}$$

The following two rules are somewhat unusual; they can be considered as reflexive instances of extensionality (see Rules 2.41 and 2.42 below).

$$\frac{\Gamma \vdash \pi_1 A :: K' \qquad \Gamma \vdash \pi_2 A :: K''}{\Gamma \vdash A :: K' {\times} K''} \tag{2.30}$$

$$\frac{\begin{array}{c}\Gamma, \alpha{::}K' \vdash A\,\alpha :: K'' \\ \Gamma \vdash A :: \Pi\alpha{::}L'.L'' \qquad \Gamma \vdash K' \equiv L'\end{array}}{\Gamma \vdash A :: \Pi\alpha{::}K'.K''} \tag{2.31}$$

Intuitively, Rules 2.30 and 2.31 say that "a constructor has every kind that its eta-expansion does". In most dependently-typed calculi such rules would be admissible and not part of the system's definition. However, here they allow constructors to be given strictly more precise kinds. (They also ensure that kinds are preserved under η-reduction.) For example, assume that $\alpha{::}\mathbf{T}{\times}\mathbf{T}$. In the absence of Rule 2.30, the most precise kind for α which can be shown is:

$$\alpha{::}\mathbf{T}{\times}\mathbf{T} \vdash \alpha :: \mathbf{T}{\times}\mathbf{T}$$

However, using Rule 2.30 one can conclude

$$\alpha{::}\mathbf{T}{\times}\mathbf{T} \vdash \alpha :: \mathbf{S}(\pi_1\alpha){\times}\mathbf{S}(\pi_2\alpha).$$

This says that α has "the kind of pairs whose first component is equal to the first component of α and whose second component is equal to the second component of α". This is a much more precise and informative kind than $\mathbf{T}{\times}\mathbf{T}$. In fact, by extensionality the *only* pair with this kind is α itself, so that this kind can be considered an encoding of $\mathbf{S}(\alpha :: \mathbf{T}{\times}\mathbf{T})$. These rules are therefore critical for encoding singletons of arbitrary constructors (in §2.3).

I believe that last two premises in Rule 2.31 could be replaced by the much simpler side-condition $\alpha \notin \mathrm{FV}(A)$, but I then become unable to show the existence of principal kinds in §4.2. The formulation here makes explicit that Rule 2.31 yields more-precise Π kinds for constructors only by making the codomain more precise, rather than by weakening the domain kind. For the purposes of principal types this could be expressed more directly with the single premise $\Gamma \vdash A :: \Pi\alpha{::}K'.L''$, but the two-premise form here is more convenient in Chapter 3.

Rules analogous to 2.30 and 2.31 have frequently appeared in literature studying Standard ML modules, including the non-standard structure-typing rule of Harper, Mitchell, and Moggi [HMM90], the VALUE rules of Harper and Lillibridge's translucent sums [HL94], the strengthening operation of Leroy's manifest type system [Ler94], the "self" rule of Leroy's applicative functors [Ler95], and the REFL rule of Aspinall [Asp00].

Subkinding is used by the subsumption rule:

$$\frac{\Gamma \vdash A :: K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash A :: K_2} \tag{2.32}$$

Constructor equivalence defines a notion of equality (interchangeability) for type constructors. The judgment $\Gamma \vdash A_1 \equiv A_2 :: K$ expresses the fact that A_1 and A_2 are equivalent constructors of kind K under context Γ. Whether $\Gamma \vdash A_1 \equiv A_2 :: K$ is provable depends not only on A_1 and A_2, but also on the kinds of their free variables (given by Γ) and the kind K at which the two constructors are being compared. Equivalence is highly context-sensitive.

Equivalence is first defined to be a reflexive, symmetric, and transitive relation:

$$\frac{\Gamma \vdash A :: K}{\Gamma \vdash A \equiv A :: K} \tag{2.33}$$

$$\frac{\Gamma \vdash A_2 \equiv A_1 :: K}{\Gamma \vdash A_1 \equiv A_2 :: K} \tag{2.34}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: K \qquad \Gamma \vdash A_2 \equiv A_3 :: K}{\Gamma \vdash A_1 \equiv A_3 :: K} \tag{2.35}$$

Next, the relation is specified to be a congruence: replacing subparts of a constructor with equivalent parts yields an equivalent constructor.

$$\frac{\Gamma \vdash K_1' \equiv K_2' \qquad \Gamma, \alpha{::}K_1' \vdash A_1 \equiv A_2 :: K''}{\Gamma \vdash \lambda\alpha{::}K_1'.A_1 \equiv \lambda\alpha{::}K_2'.A_2 :: \Pi\alpha{::}K_1'.K''} \tag{2.36}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: K' \to K'' \qquad \Gamma \vdash A_1' \equiv A_2' :: K'}{\Gamma \vdash A_1\, A_1' \equiv A_2\, A_2' :: K''} \tag{2.37}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K'} \tag{2.38}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''} \tag{2.39}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1' \equiv A_2' :: K' \\ \Gamma \vdash A_1'' \equiv A_2'' :: K''\end{array}}{\Gamma \vdash \langle A_1', A_1''\rangle \equiv \langle A_2', A_2''\rangle :: K' \times K''} \tag{2.40}$$

There are two extensionality rules: if two functions or two pairs cannot be distinguished by their uses then they are considered equivalent. In particular, two pairs are equivalent if they have equivalent first and second components

$$\frac{\begin{array}{c}\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K' \\ \Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: K''\end{array}}{\Gamma \vdash A_1 \equiv A_2 :: K' \times K''} \tag{2.41}$$

and two functions are equivalent if they return equivalent results for all arguments:

$$\frac{\begin{array}{c}\Gamma, \alpha{::}K' \vdash A_1\,\alpha \equiv A_2\,\alpha :: K'' \\ \Gamma \vdash A_1 :: \Pi\alpha{::}L_1'.L_1'' \qquad \Gamma \vdash K' \equiv L_1' \\ \Gamma \vdash A_2 :: \Pi\alpha{::}L_2'.L_2'' \qquad \Gamma \vdash K' \equiv L_2'\end{array}}{\Gamma \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K'.K''} \tag{2.42}$$

The last four premises in Rule 2.42 ensure that both A_1 and A_2 actually have kind $\Pi\alpha{::}K'.K''$. If Rule 2.31 were simplified as discussed above then this rule could be simplified in analogous fashion with the side condition $\alpha \notin (\mathrm{FV}(A_1) \cup \mathrm{FV}(A_2))$.

As in the well-formedness rules, there is a subsumption rule:

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash A_1 \equiv A_2 :: K_2} \tag{2.43}$$

Interestingly, an easy inductive argument shows that the rules given so far merely define constructor equivalence to be syntactic identity (up to renaming of bound variables). All the rules except for Rule 2.33 would then appear redundant. Adding one more rule makes this equivalence non-trivial, and justifies the presence of each of the above rules:

$$\frac{\Gamma \vdash A :: \mathbf{S}(B)}{\Gamma \vdash A \equiv B :: \mathbf{S}(B)} \tag{2.44}$$

This completes the definition of constructor equivalence. It may be initially surprising that there are no equivalence rules for reducing function applications or projections from pairs (i.e., β-like rules). It turns out that these are admissible in the presence of singleton kinds and Rule 2.44. The details are in §2.3 and §3.3, but I sketch one example here. It is clear that

$$\vdash \langle \mathsf{Int}, \mathsf{Boxedfloat}\rangle :: \mathbf{S}(\mathsf{Int}) \times \mathbf{S}(\mathsf{Boxedfloat})$$

Therefore by Rule 2.27 it follows

$$\vdash \pi_1\langle \mathsf{Int}, \mathsf{Boxedfloat}\rangle :: \mathbf{S}(\mathsf{Int})$$

and by Rule 2.44 and subsumption we have

$$\vdash \pi_1\langle \mathsf{Int}, \mathsf{Boxedfloat}\rangle \equiv \mathsf{Int} :: \mathbf{T}$$

This same argument can be generalized to projections from arbitrary pairs, and in an analogous fashion to applications of λ-abstractions.

Given the β-rules, then, the extensionality rules 2.42 and 2.41 imply that the usual η-rules are admissible as well. It is well-known that η-reduction is not confluent in the presence of terminal (unit) types. As singletons are a generalized form of unit, the same behavior appears here as well. For example:

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

holds, as does

$$\alpha : \mathbf{S}(\mathsf{Int}){\to}\mathbf{T} \vdash \alpha \equiv (\lambda\beta{::}\mathbf{S}(\mathsf{Int}).(\alpha\,\mathsf{Int})) :: \mathbf{S}(\mathsf{Int}){\to}\mathbf{T}$$

All the constructors in these judgments are normal with respect to $\beta\eta$-reduction; compare the right-hand constructor in the last judgment with $\lambda\beta{::}\mathbf{S}(\mathsf{Int}).(\alpha\,\beta)$, the η-expansion of α.

A more obvious consequence of having singletons — and their original motivation — is that they can be used to express definitions for variables. For example, in the following two judgments the context effectively defines α to be Int.

$$\alpha : \mathbf{S}(\mathsf{Int}) \vdash \alpha \equiv \mathsf{Int} :: \mathbf{T}$$
$$\alpha : \mathbf{S}(\mathsf{Int}) \vdash \langle\alpha, \mathsf{Int}\rangle \equiv \langle\mathsf{Int}, \alpha\rangle :: \mathbf{T}{\times}\mathbf{T}$$

But the system is not restricted merely to giving definitions to variables. In the provable judgment

$$\alpha : \mathbf{T}{\times}\mathbf{S}(\mathsf{Int}) \vdash \pi_2\alpha \equiv \mathsf{Int} :: \mathbf{T}$$

the context *partially* defines α; it is known to be a pair and its second component is (equivalent to) Int, but this does not give a definition for α as a whole. Alternatively, this could be thought of as giving $\pi_2\alpha$ the definition Int without giving one to $\pi_1\alpha$.

Similarly, in the provable judgments

$$\alpha : \Sigma\beta{::}\mathbf{T}.\mathbf{S}(\beta) \vdash \pi_1\alpha \equiv \pi_2\alpha :: \mathbf{T}$$
$$\alpha : \Sigma\beta{::}\mathbf{T}.\mathbf{S}(\beta) \vdash \alpha \equiv \langle\pi_1\alpha, \pi_1\alpha\rangle :: \mathbf{T}{\times}\mathbf{T}.$$

the assumption governing α requires that it be a pair whose first component β has kind $\mathbf{T}$ and whose second component is equal to the first; that is, a pair with two equal components of kind $\mathbf{T}$. This gives a definition to $\pi_2\alpha$, namely $\pi_1\alpha$, without further specifying the contents of these two equal components.

Now because of subkinding and subsumption, constructors do not have unique kinds. The equational system presented here has the relatively unusual property (for a system expected to be decidable) that equivalence of two constructors depends on the kind at which they are compared. Two constructors may be equivalent at one kind but not at another; for example, one *cannot* prove

$$\vdash \lambda\alpha{::}\mathbf{T}.\alpha \equiv \lambda\alpha{::}\mathbf{T}.\mathsf{Int}{::}\mathbf{T}{\to}\mathbf{T}.$$

This is fortunate, as the identity function for constructors of kind $\mathbf{T}$ and the function constantly returning Int do have distinct behaviors and ought not be equivalent in a consistent equational theory. However, by subsumption these two functions both have kind $\mathbf{S}(\mathsf{Int}){\to}\mathbf{T}$ and the judgment

$$\vdash \lambda\alpha{::}\mathbf{T}.\alpha \equiv \lambda\alpha{::}\mathbf{T}.\mathsf{Int} :: \mathbf{S}(\mathsf{Int}){\to}\mathbf{T}$$

is provable. The proof uses extensionality and the fact that the two functions provably agree when restricted to an argument of kind $\mathbf{S}(\mathsf{Int})$, i.e., when applied to the argument Int.

The classifying kind at which constructors are compared may depend on the context of their occurrence. For example, it follows from the previous equation and Rule 2.37 that

$$\beta : (\mathbf{S}(\mathsf{Int}){\to}\mathbf{T}){\to}\mathbf{T} \vdash \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \equiv \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) :: \mathbf{T}$$

is provable. The kind of β guarantees that it will only apply its argument to the constructor Int, so it cannot matter whether β is given $\lambda\alpha{::}\mathbf{T}.\alpha$ or $\lambda\alpha{::}\mathbf{T}.\mathsf{Int}$.

In contrast, the following judgment is *not* provable:

$$\beta : (\mathbf{T}{\to}\mathbf{T}){\to}\mathbf{T} \vdash \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \equiv \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) :: \mathbf{T}$$

because the context makes a weaker assumption about β.

2.2.4 Types

The constructors of kind $\mathbf{T}$ correspond to types; there is an explicit inclusion $Ty(\cdot)$ mapping each such constructor to the corresponding type.

$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash Ty(A)} \tag{2.45}$$

I will use int as an abbreviation for the type $Ty(\mathsf{Int})$, $\mathsf{boxedfloat}$ to abbreviate $Ty(\mathsf{Boxedfloat})$, and similarly for the other primitive constructors.

As discussed in the introduction, singleton types are restricted to contain only syntactic values. The representation of labeled singletons via encodings, as is done for kinds in §2.3 below, does not work for terms due to the lack of extensionality principles. Because for inlining purposes I need singletons at non-base type, labeled singletons *types* are made primitive:

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{S}(v : \tau)} \quad (\tau \text{ not a singleton}) \tag{2.46}$$

Rule 2.46 prohibits the type label in a singleton from being yet another singleton type. So, for example,

$$\mathbf{S}((\lambda x{:}\mathsf{int}.3) : \mathsf{int}{\rightharpoonup}\mathbf{S}(3 : \mathsf{int}))$$

is well-formed, but the following type is not:

$$\mathbf{S}((\lambda x{:}\mathsf{int}.3) : \mathbf{S}((\lambda x{:}\mathsf{int}.3) : \mathsf{int}{\rightharpoonup}\mathbf{S}(3 : \mathsf{int}))).$$

The property of a type not being a singleton is preserved under the important operations of substitution and head-normalization. Also, because of predicativity it is clear from the rules below that singleton types are equivalent only to other singleton types; see Theorem 6.2.2. This restriction could be formalized syntactically by defining a grammatical class of non-singleton types, but in this case I have opted for syntactic simplicity.

This restriction is reasonable because a well-formed type $\mathbf{S}(v_1 : \mathbf{S}(v_2 : \tau))$ contains no more information than is already contained in $\mathbf{S}(v_1 : \tau)$ or $\mathbf{S}(v_2 : \tau)$. At first it might appear that a typing assumption $x{:}\mathbf{S}(v_1 : \mathbf{S}(v_2 : \tau))$ would be equivalent to assuming that v_1 and v_2 are equivalent. However, in order to make such an assumption it must be possible to show that $\mathbf{S}(v_1 : \mathbf{S}(v_2 : \tau))$ is well-formed, and in particular that *without the new assumption* one has $v_1 : \mathbf{S}(v_2 : \tau)$, i.e., that v_1 and v_2 are equivalent at type τ. Thus nested singletons impart no useful information.

Allowing directly nested singletons would have the further consequence that the constant $\mathsf{3}$ would naturally have the types $\mathbf{S}(3 : \mathsf{int})$ and $\mathbf{S}(3 : \mathbf{S}(3 : \mathsf{int}))$ and $\mathbf{S}(3 : \mathbf{S}(3 : \mathbf{S}(3 : \mathsf{int})))$, and so on. By the "obvious" subtyping rules these would form an infinite strictly decreasing chain of subtypes, even though none of these types are really more informative than any of the others. (These types all classify exactly the same set of values, namely the set $\{3\}$.) Furthermore there would be no lower bound to this sequence of types: the system would fail to have principal (most specific) types for all terms.

Aspinall [Asp95] addresses this problem by defining all the types in such a chain to be equivalent: $\mathbf{S}(v : \tau) \equiv \mathbf{S}(v : \mathbf{S}(v : \tau))$. By disallowing directly nested singletons, I avoid a need for this rule. This has the advantage of allowing a much simpler inversion principle for equivalence of singleton types: if two singleton types are equivalent then their type labels are equivalent. (This principle is clearly false in Aspinall's system. It also fails for the encoding of labeled singleton kinds, but the proofs use inversion only for the kinds of the official MIL_0 language.)

Because of singleton types, the types classifying functions and binary products are extended to dependent forms:

$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau') \rightharpoonup \tau''} \tag{2.47}$$

$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau') \times \tau''} \tag{2.48}$$

Such types are written $\tau' \rightharpoonup \tau''$ and $\tau' \times \tau''$ when there is no actual dependency.

Finally, MIL_0 contains the types for polymorphic terms, functions whose argument is a constructor.

$$\frac{\Gamma, \alpha{::}K \vdash \tau}{\Gamma \vdash \forall \alpha{::}K.\tau} \tag{2.49}$$

Note that in this predicative system there are no type constructors corresponding to singleton types, truly dependent function or pair types, or to polymorphic types.

Type equivalence is, like constructor equivalence, reflexive, symmetric, transitive, and a congruence.

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \equiv \tau} \tag{2.50}$$

$$\frac{\Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash \tau \equiv \tau'} \tag{2.51}$$

$$\frac{\Gamma \vdash \tau \equiv \tau' \qquad \Gamma \vdash \tau' \equiv \tau''}{\Gamma \vdash \tau \equiv \tau''} \tag{2.52}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1) \equiv \mathit{Ty}(A_2)} \tag{2.53}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : \tau_1 \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \mathbf{S}(v_1 : \tau_1) \equiv \mathbf{S}(v_2 : \tau_2)} \quad (\tau_1, \tau_2 \text{ not a singleton}) \tag{2.54}$$

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash (x{:}\tau_1') \rightharpoonup \tau_1'' \equiv (x{:}\tau_2') \rightharpoonup \tau_2''} \tag{2.55}$$

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash (x{:}\tau_1') \times \tau_1'' \equiv (x{:}\tau_2') \times \tau_2''} \tag{2.56}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma, \alpha{::}K_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \forall \alpha{::}K_1.\tau_1 \equiv \forall \alpha{::}K_2.\tau_2} \tag{2.57}$$

Finally certain constructors correspond to (non-dependent) pair types and (non-dependent, non-polymorphic) function types.

$$\frac{\Gamma \vdash A_1 :: \mathbf{T} \qquad \Gamma \vdash A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1 \times A_2) \equiv \mathit{Ty}(A_1) \times \mathit{Ty}(A_2)} \tag{2.58}$$

$$\frac{\Gamma \vdash A_1 :: \mathbf{T} \qquad \Gamma \vdash A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1 \rightharpoonup A_2) \equiv \mathit{Ty}(A_1) \rightharpoonup \mathit{Ty}(A_2)} \tag{2.59}$$

These rules are necessary for polymorphism to be useful in this predicative type system. For example, consider the polymorphic identity function

$$\mathtt{id} : \forall \alpha{::}\mathbf{T}.\, \mathit{Ty}(\alpha) \rightharpoonup \mathit{Ty}(\alpha).$$

To apply this function to a pair of integers requires polymorphic instantiation (i.e., an application of `id` to a constructor argument). The only reasonable argument here is $\mathsf{Int} \times \mathsf{Int}$, so we have

$$\mathtt{id}\,(\mathsf{Int} \times \mathsf{Int}) : \mathit{Ty}(\mathsf{Int} \times \mathsf{Int}) \rightharpoonup \mathit{Ty}(\mathsf{Int} \times \mathsf{Int}).$$

But by the typing rules below, a pair of integers does not have type $\mathit{Ty}(\mathsf{Int} \times \mathsf{Int})$ but instead has type $\mathit{Ty}(\mathsf{Int}) \times \mathit{Ty}(\mathsf{Int})$, i.e., the type of a pair whose elements are of type $\mathit{Ty}(\mathsf{Int})$. Rule 2.58 is then necessary to permit an application like $(\mathtt{id}\,(\mathsf{Int} \times \mathsf{Int}))\,\langle 3, 4\rangle$ to typecheck.

Subtyping is reflexive and transitive, and is a strictly weaker relation than equivalence.

$$\frac{\Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash \tau \leq \tau'} \tag{2.60}$$

$$\frac{\Gamma \vdash \tau \leq \tau' \qquad \Gamma \vdash \tau' \leq \tau''}{\Gamma \vdash \tau \leq \tau''} \tag{2.61}$$

One can obtain a supertype of a singleton type by either dropping the singleton (as at the kind level), or by weakening the type label.

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{S}(v : \tau) \leq \tau} \quad (\tau \text{ not a singleton}) \tag{2.62}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{S}(v_1 : \tau_1) \\ \Gamma \vdash v_1 \equiv v_2 : \tau_2 \qquad \Gamma \vdash \tau_1 \leq \tau_2\end{array}}{\Gamma \vdash \mathbf{S}(v_1 : \tau_1) \leq \mathbf{S}(v_2 : \tau_2)} \quad (\tau_1, \tau_2 \text{ not a singleton}) \tag{2.63}$$

Subtyping is lifted to functions, pairs, and polymorphic types in the usual co- and contravariant manner.

$$\frac{\begin{array}{c}\Gamma \vdash (x{:}\tau_1')\times\tau_1'' \\ \Gamma \vdash \tau_2' \leq \tau_1' \qquad \Gamma, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''\end{array}}{\Gamma \vdash (x{:}\tau_1')\rightharpoonup\tau_1'' \leq (x{:}\tau_2')\rightharpoonup\tau_2''} \tag{2.64}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x{:}\tau_2')\times\tau_2'' \\ \Gamma \vdash \tau_1' \leq \tau_2' \qquad \Gamma, x{:}\tau_1 \vdash \tau_1'' \leq \tau_2''\end{array}}{\Gamma \vdash (x{:}\tau_1')\times\tau_1'' \leq (x{:}\tau_2')\times\tau_2''} \tag{2.65}$$

$$\frac{\begin{array}{c}\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \\ \Gamma \vdash K_2 \leq K_1 \qquad \Gamma, \alpha{::}K_2 \vdash \tau_1 \leq \tau_2\end{array}}{\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \leq \forall\alpha{::}K_2.\tau_2} \tag{2.66}$$

Because the system is predicative, there is no difficulty arising from the contravariant subkinding for the domains of universally quantified types as can sometimes arise when polymorphism and subtyping are combined [Pie91].

2.2.5 Terms

The well-formedness rules for the term language are mostly standard. The language has been restricted to a "named" form where intermediate quantities are bound to variables [FSDF93]. Note that projections from values are considered to be values: for the system to be useful it is necessary that projections from variables be values so that they may appear in singletons, and we wish terms to remain well-formed under substitutions of values for variables.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash n : \mathsf{int}} \tag{2.67}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash x : \Gamma(x)} \tag{2.68}$$

Function values are potentially recursive. Within the body e of the function $\mathsf{fun}\, f(x{:}\tau){:}\tau'\ \mathsf{is}\ e$ the variable x refers to the function argument and f refers to the function itself; the result type τ' may also depend on x.

$$\frac{\Gamma, f{:}(x{:}\tau')\rightharpoonup\tau'', x{:}\tau' \vdash e : \tau''}{\Gamma \vdash \mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e : (x{:}\tau')\rightharpoonup\tau''} \tag{2.69}$$

When the function $\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e$ is non-recursive (i.e., $f \notin \mathrm{FV}(e)$) then it can be written as $\lambda(x{:}\tau'){:}\tau''.e$, or even $\lambda x{:}\tau'.e$ when the return-type is obvious or irrelevant.

Type abstractions are also annotated with a return-type. This accurately models the full MIL (where the notions of type and term abstractions are merged) and simplifies the correctness proof for my typechecking algorithm.

$$\frac{\Gamma, \alpha{::}K \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha{::}K){:}\tau.e : \forall\alpha{::}K.\tau} \tag{2.70}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2} \tag{2.71}$$

$$\frac{\Gamma \vdash v : (x{:}\tau') \times \tau''}{\Gamma \vdash \pi_1 v : \tau'} \tag{2.72}$$

$$\frac{\Gamma \vdash v : (x{:}\tau') \times \tau''}{\Gamma \vdash \pi_2 v : [\pi_1 v / x]\tau''} \tag{2.73}$$

$$\frac{\Gamma \vdash v : \tau' \rightharpoonup \tau'' \qquad \Gamma \vdash v' : \tau'}{\Gamma \vdash v\, v' : \tau''} \tag{2.74}$$

$$\frac{\Gamma \vdash v : \forall \alpha{::}K.\tau \qquad \Gamma \vdash A :: K}{\Gamma \vdash v\, A : [A/\alpha]\tau} \tag{2.75}$$

Every let-expression be annotated with two types: the type of the locally-defined variable, and the type of the entire let-expression.

$$\frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x{:}\tau' \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash (\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}) : \tau} \tag{2.76}$$

The former annotation is used to simplify the typechecking algorithm; it would be preferable if this were not needed. The latter type is used to ensure easy calculation of principal types for let-expressions. In the TILT compiler, let is used only in specific positions (i.e., the body of a function or the arms of a conditional expression) which for other reasons are already annotated with their types, so the presence of the body annotation in the MIL_0 is reasonable.

Values are given singleton types via the following singleton introduction rule.

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \mathbf{S}(v : \tau)} \quad (\tau \text{ not a singleton}) \tag{2.77}$$

Finally, subtyping is used by the subsumption rule.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2} \tag{2.78}$$

The following definition of *term equivalence* is the strongest equivalence relation (relating fewest terms) that seems useful for the purposes described in the introductory chapter.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \equiv e : \tau} \tag{2.79}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{2.80}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \qquad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{2.81}$$

Again, equivalence is a congruence:

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2'' \qquad \Gamma, f{:}(x{:}\tau_1')\rightharpoonup\tau_1'', x{:}\tau' \vdash e_1 \equiv e_2 : \tau_1''}{\Gamma \vdash \mathsf{fun}\, f(x{:}\tau_1'){:}\tau_1'' \;\mathsf{is}\; e_1 \equiv \mathsf{fun}\, f(x{:}\tau_2'){:}\tau_2'' \;\mathsf{is}\; e_2 : (x{:}\tau_1')\rightharpoonup\tau_1''} \tag{2.82}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma, \alpha{::}K_1 \vdash \tau_1 \equiv \tau_2 \qquad \Gamma, \alpha{::}K_1 \vdash e_1 \equiv e_2 : \tau_1}{\Gamma \vdash \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \equiv \Lambda(\alpha{::}K_2){:}\tau_2.e_2 : \forall\alpha{::}K_1.\tau_1} \tag{2.83}$$

$$\frac{\Gamma \vdash v_1' \equiv v_2' : \tau' \qquad \Gamma \vdash v_1'' \equiv v_2'' : \tau''}{\Gamma \vdash \langle v_1', v_1''\rangle \equiv \langle v_2', v_2''\rangle : \tau'\times\tau''} \tag{2.84}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : (x{:}\tau')\times\tau''}{\Gamma \vdash \pi_1 v_1 \equiv \pi_1 v_2 : \tau'} \tag{2.85}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : (x{:}\tau')\times\tau''}{\Gamma \vdash \pi_2 v_1 \equiv \pi_2 v_2 : [\pi_1 v_1/x]\tau''} \tag{2.86}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : \tau'\rightharpoonup\tau'' \qquad \Gamma \vdash v' \equiv v_2' : \tau'}{\Gamma \vdash v_1\, v_1' \equiv v_2\, v_2' : \tau''} \tag{2.87}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : \forall\alpha{::}K.\tau \qquad \Gamma \vdash A_1 \equiv A_2 :: K}{\Gamma \vdash v_1\, A_1 \equiv v_2\, A_2 : [A_1/\alpha]\tau_1} \tag{2.88}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma \vdash e_1' \equiv e_2' : \tau_1' \\ \Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma, x{:}\tau_1' \vdash e_1 \equiv e_2 : \tau_1\end{array}}{\Gamma \vdash (\mathsf{let}\ x{:}\tau_1'{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \equiv (\mathsf{let}\ x{:}\tau_2'{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end}) : \tau_1} \tag{2.89}$$

As at the constructor level, there is a singleton elimination rule for equivalence.

$$\frac{\Gamma \vdash v_1 : \mathbf{S}(v_2 : \tau)}{\Gamma \vdash v_1 \equiv v_2 : \mathbf{S}(v_2 : \tau)} \tag{2.90}$$

Finally there is a subsumption rule.

$$\frac{\Gamma \vdash e_1 \equiv e_2 : \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e_1 \equiv e_2 : \tau_2} \tag{2.91}$$

2.3 Admissible Rules

This section lists a number of interesting or useful rules which become admissible in the presence of singletons. The proofs of admissibility are deferred until §3.3.

In MIL_0, the kind $\mathbf{S}(A)$ is well-formed if and only if A is of the base kind $\mathbf{T}$. This initially seems restrictive, especially when compared with singleton types which can contain values of any (non-singleton) type. One might expect to find singleton kinds of the form $\mathbf{S}(A :: K)$ representing the kind of all constructors equivalent to A when compared at kind K, for example to encode definitions of constructor-level functions. However, these labeled singletons are *definable* in MIL_0; Figure 2.4 defines these by induction on the size of the kind label.

For example, if β has kind $\mathbf{T}\to\mathbf{T}$, then $\mathbf{S}(\beta :: \mathbf{T}\to\mathbf{T})$ is defined to be $\Pi\alpha{::}\mathbf{T}.\mathbf{S}(\beta\,\alpha)$. This can be interpreted as "the kind of all functions which, when applied, yield the same answer as β does", or "the kind of all functions which agree pointwise with β". By extensionality, any such function

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

Figure 2.4: Encodings of Labeled Singleton Kinds

is provably equivalent to β, and indeed the non-standard kinding rules mentioned in §2.1 are vital in proving that β has this kind.

Since kinds only matter up to equivalence, the definitions in Figure 2.4 are not unique. One could, for example, define $\mathbf{S}(A :: \mathbf{S}(A'))$ to be $\mathbf{S}(A')$, or define $\mathbf{S}(A :: \Sigma\alpha{::}K_1.K_2)$ to be $\Sigma\alpha{::}\mathbf{S}(\pi_1 A :: K_1).\mathbf{S}(\pi_2 A :: K_2)$.

The following rules are admissible, showing that the defined singleton kinds do behave appropriately.

$$\frac{\Gamma \vdash A :: K}{\Gamma \vdash \mathbf{S}(A :: K)} \tag{2.92}$$

$$\frac{\Gamma \vdash A :: K}{\Gamma \vdash A :: \mathbf{S}(A :: K)} \tag{2.93}$$

$$\frac{\Gamma \vdash A :: K}{\Gamma \vdash \mathbf{S}(A :: K) \leq K} \tag{2.94}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash \mathbf{S}(A_1 :: K_1) \leq \mathbf{S}(A_2 :: K_2)} \tag{2.95}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: K}{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_1 :: K)} \tag{2.96}$$

$$\frac{\Gamma \vdash A_2 :: K \qquad \Gamma \vdash A_1 :: \mathbf{S}(A_2 :: K)}{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2 :: K)} \tag{2.97}$$

Note that $\Gamma \vdash \mathbf{S}(A :: K)$ need not imply $\Gamma \vdash A :: K$. (For example, according to Figure 2.4 we have $\mathbf{S}(\mathsf{Boxedfloat} :: \mathbf{S}(\mathsf{Int})) = \mathbf{S}(\mathsf{Boxedfloat})$, and therefore $\vdash \mathbf{S}(\mathsf{Boxedfloat} :: \mathbf{S}(\mathsf{Int}))$ even though $\mathsf{Boxedfloat}$ cannot be shown to have kind $\mathbf{S}(\mathsf{Int})$. This explains the premise $\Gamma \vdash A_2 :: K$ in Rule 2.97.

Next, we have versions of existing rules allowing dependencies where the primitive rules require non-dependent types or kinds. (For example, compare Rules 2.25 and 2.98, or Rules 2.26 and 2.100.)

$$\frac{\Gamma \vdash A :: \Pi\alpha{::}K'.K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash A\,A' :: [A'/\alpha]K''} \tag{2.98}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K'.K'' \qquad \Gamma \vdash A_1' \equiv A_2' :: K'}{\Gamma \vdash A_1\,A_1' \equiv A_2\,A_2' :: [A_1'/\alpha]K''} \tag{2.99}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{::}K'.K'' \\ \Gamma \vdash A' :: K' \qquad \Gamma \vdash A'' :: [A'/\alpha]K''\end{array}}{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''} \tag{2.100}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{::}K'.K'' \\ \Gamma \vdash A'_1 \equiv A'_2 :: K' \\ \Gamma \vdash A''_1 \equiv A''_2 :: [A'_1/\alpha]K''\end{array}}{\Gamma \vdash \langle A'_1, A''_1\rangle \equiv \langle A'_2, A''_2\rangle :: \Sigma\alpha{::}K'.K''} \tag{2.101}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{::}K'.K'' \\ \Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K' \\ \Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''\end{array}}{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''} \tag{2.102}$$

$$\frac{\Gamma \vdash v : (x{:}\tau')\rightharpoonup\tau'' \qquad \Gamma \vdash v' : \tau'}{\Gamma \vdash v\,v' : [v'/x]\tau''} \tag{2.103}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : (x{:}\tau')\rightharpoonup\tau'' \qquad \Gamma \vdash v'_1 \equiv v'_2 : \tau'}{\Gamma \vdash v_1\,v'_1 \equiv v_2\,v'_2 : [v'_1/x]\tau''} \tag{2.104}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x{:}\tau')\times\tau'' \\ \Gamma \vdash v' : \tau' \qquad \Gamma \vdash v'' : [v'/x]\tau''\end{array}}{\Gamma \vdash \langle v', v''\rangle :: (x{:}\tau')\times\tau''} \tag{2.105}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x{:}\tau')\times\tau'' \\ \Gamma \vdash v'_1 \equiv v'_2 : \tau \\ \Gamma \vdash v''_1 \equiv v''_2 : [v'_1/\alpha]\tau''\end{array}}{\Gamma \vdash \langle v'_1, v''_1\rangle \equiv \langle v'_2, v''_2\rangle : (x{:}\tau')\times\tau''} \tag{2.106}$$

Next, a remarkable observation of Aspinall [Asp95] is that the β-rule for function applications can be admissible in the presence of singletons. In MIL_0, which contains pairs, the projection rules become admissible as well.

$$\frac{\Gamma, \alpha{::}K' \vdash A :: K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash (\lambda\alpha{::}K'.A)\; A' \equiv [A'/\alpha]A :: [A'/\alpha]K''} \tag{2.107}$$

$$\frac{\Gamma \vdash A_1 :: K_1 \qquad \Gamma \vdash A_2 :: K_2}{\Gamma \vdash \pi_1\langle A_1, A_2\rangle \equiv A_1 :: K_1} \tag{2.108}$$

$$\frac{\Gamma \vdash A_1 :: K_1 \qquad \Gamma \vdash A_2 :: K_2}{\Gamma \vdash \pi_2\langle A_1, A_2\rangle \equiv A_2 :: K_2} \tag{2.109}$$

β-equivalence for functions is admissible at the constructor level, but not at the term level; this is a consequence of term applications being non-values. (It is easy to prove that β_v-equivalence for terms is not admissible. The defining rules of term equivalence only equate values to values or non-values to non-values; in contrast, β-equivalence can equate applications with values.) The projection rules for term-level pairs remain, however.

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \pi_1\langle v_1, v_2\rangle \equiv v_1 : \tau_1} \tag{2.110}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \pi_2\langle v_1, v_2\rangle \equiv v_2 : \tau_2} \tag{2.111}$$

It is occasionally convenient to have "parallel" versions of these equivalences:

$$\frac{\Gamma, \alpha{::}K' \vdash A_1 \equiv A_2 :: K'' \qquad \Gamma \vdash A_1' \equiv A_2' :: K'}{\Gamma \vdash (\lambda\alpha{::}K'.A_1)\, A_1' \equiv [A_2'/\alpha]A_2 :: [A_1'/\alpha]K''} \tag{2.112}$$

$$\frac{\Gamma \vdash A_1 \equiv A_1' :: K_1 \qquad \Gamma \vdash A_2 :: K_2}{\Gamma \vdash \pi_1\langle A_1, A_2\rangle \equiv A_1' :: K_1} \tag{2.113}$$

$$\frac{\Gamma \vdash A_1 :: K_1 \qquad \Gamma \vdash A_2 \equiv A_2' :: K_2}{\Gamma \vdash \pi_2\langle A_1, A_2\rangle \equiv A_2' :: K_2} \tag{2.114}$$

$$\frac{\Gamma \vdash v_1 \equiv v_1' : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \pi_1\langle v_1, v_2\rangle \equiv v_1' : \tau_1} \tag{2.115}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 \equiv v_2' : \tau_2}{\Gamma \vdash \pi_2\langle v_1, v_2\rangle \equiv v_2' : \tau_2} \tag{2.116}$$

In the presence of both β-equivalence and extensionality, η-rules for functions and pairs become admissible as well.

$$\frac{\Gamma \vdash A :: \Pi\alpha{::}K'.K''}{\Gamma \vdash A \equiv \lambda\alpha{::}K'.(A\,\alpha) :: \Pi\alpha{::}K'.K''} \tag{2.117}$$

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash A \equiv \langle \pi_1 A, \pi_2 A\rangle :: \Sigma\alpha{::}K'.K''} \tag{2.118}$$

Finally, I give variants of the introduction and elimination rules for singleton kinds and types:

$$\frac{\Gamma \vdash A \equiv B :: \mathbf{T}}{\Gamma \vdash A :: \mathbf{S}(B)} \tag{2.119}$$

$$\frac{\Gamma \vdash A \equiv B :: \mathbf{T}}{\Gamma \vdash A \equiv B :: \mathbf{S}(A)} \tag{2.120}$$

$$\frac{\Gamma \vdash A :: \mathbf{S}(B)}{\Gamma \vdash A \equiv B :: \mathbf{T}} \tag{2.121}$$

$$\frac{\Gamma \vdash v \equiv w : \tau}{\Gamma \vdash v : \mathbf{S}(w : \tau)} \tag{2.122}$$

$$\frac{\Gamma \vdash v_1 \equiv v_2 : \tau}{\Gamma \vdash v_1 \equiv v_2 : \mathbf{S}(v_1 : \tau)} \quad (\tau \text{ not a singleton}) \tag{2.123}$$

$$\frac{\Gamma \vdash v_1 : \mathbf{S}(v_2 : \tau)}{\Gamma \vdash v_1 \equiv v_2 : \tau} \tag{2.124}$$

2.4 Dynamic Semantics

I give the operational meaning of a program in terms of a small-step contextual semantics: the dynamic semantics defines the possible execution steps $e_1 \leadsto e_2$ for programs (closed terms), and evaluation of a program corresponds to taking an execution step until no more steps apply repeatedly.

The evaluation strategy used by MIL_0 for both constructors and terms is left-to-right call-by-value. Furthermore, constructors are evaluated as well as ordinary terms. (For MIL_0 as presented this is not actually necessary; this choice was made in preparation for adding constructor analysis constructs such as typecase to the language; type and kind annotations on terms, however, never require evaluation.) This requires a notion of *fully-evaluated* constructors and terms, denoted $\overline{A}$ and $\overline{v}$ respectively

$$
\begin{array}{rcll}
\overline{A} & ::= & c\,\overline{A}_1 \cdots \overline{A}_n & (n \geq 0) \\
 & | & \langle \overline{A}_1, \overline{A}_2 \rangle & \\
 & | & \lambda\alpha{::}K'.A & \\
 & & & \\
\overline{v} & ::= & n & \\
 & | & \mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e & \\
 & | & \Lambda(\alpha{::}K){:}\tau.e & \\
 & | & \langle \overline{v}_1, \overline{v}_2 \rangle &
\end{array}
$$

Since evaluation concerns only closed terms and types, variables and projections are need not be included here.

The operational semantics uses Felleisen's evaluation context formulation [Fel88] of Plotkin's structured operational semantics (SOS) [Plo81]. This involves the definition of a collection of primitive "instructions" (denoted I) and their one-step reducts (denoted R). The relation between instructions and reducts, written $I \leadsto R$ is shown in Figure 2.5.

Evaluation is extended to one-step reduction for arbitrary terms and constructors though the use of constructor-level and term-level *evaluation contexts*, denoted by $\mathcal{U}$ and $\mathcal{C}$ respectively. These are a restricted form of constructor or term containing a single "hole" $\diamond$:

$$
\begin{array}{rcl}
\mathcal{U} & ::= & \diamond \\
 & | & \mathcal{U}\,A \\
 & | & \overline{A}\,\mathcal{U} \\
 & | & \pi_1\mathcal{U} \\
 & | & \pi_2\mathcal{U} \\
 & | & \langle \mathcal{U}, A \rangle \\
 & | & \langle \overline{A}, \mathcal{U} \rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{C} & ::= & \diamond \\
 & | & \mathcal{C}\,e \\
 & | & \overline{v}\,\mathcal{C} \\
 & | & \pi_1\mathcal{C} \\
 & | & \pi_2\mathcal{C} \\
 & | & \mathcal{C}\,A \\
 & | & \overline{v}\,\mathcal{U} \\
 & | & \mathsf{let}\ x{:}\tau'{=}\mathcal{C}\ \mathsf{in}\ e : \tau\ \mathsf{end}
\end{array}
$$

The notations $\mathcal{U}[A]$, $\mathcal{C}[A]$ and $\mathcal{C}[e]$ denote the result of replacing the hole in the evaluation context with the specified constructor or term. (Since the hole never occurs within the scope of bound variables in the evaluation context, there is no possibility of variable capture.) The evaluation contexts represent a "stack" or "continuation" for the expression being currently evaluated; the specific choice of evaluations contexts enforces the call-by-value nature of the language.

Then the full one-step reduction relation is defined as follows:

$$
\begin{array}{lcl}
A \leadsto A' & \iff & A = \mathcal{U}[I] \text{ and } I \leadsto R \text{ and } A' = \mathcal{U}[R] \\
e \leadsto e' & \iff & e = \mathcal{C}[I] \text{ and } I \leadsto R \text{ and } e' = \mathcal{C}[R]
\end{array}
$$

$$
\begin{array}{lcl}
(\lambda\alpha{::}K'.B)\,\overline{A} & \leadsto & [\overline{A}/\alpha]B \\
\pi_1\langle\overline{A}_1,\overline{A}_2\rangle & \leadsto & \overline{A}_1 \\
\pi_2\langle\overline{A}_1,\overline{A}_2\rangle & \leadsto & \overline{A}_2 \\[6pt]
(\mathsf{fun}\,f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e)\,v & \leadsto & [\mathsf{fun}\,f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e/f][v/x]e \\
(\Lambda(\alpha{::}K){:}\tau.e)\,\overline{A} & \leadsto & [\overline{A}/\alpha]e \\
\pi_1\langle\overline{v}_1,\overline{v}_2\rangle & \leadsto & \overline{v}_1 \\
\pi_2\langle\overline{v}_1,\overline{v}_2\rangle & \leadsto & \overline{v}_2 \\
\mathsf{let}\ x{:}\tau'{=}\overline{v}\ \mathsf{in}\ e : \tau\ \mathsf{end} & \leadsto & [\overline{v}/x]e
\end{array}
$$

Figure 2.5: Reductions of Instructions

For example, consider the term

$$\Big((\Lambda(\alpha{::}\mathbf{T}){:}\,Ty(\alpha){\rightharpoonup}\,Ty(\alpha).\mathsf{fun}\,f(x{:}\,Ty(\alpha)){:}\,Ty(\alpha)\ \mathsf{is}\ x)\ ((\lambda\alpha{::}\mathbf{T}.\alpha)\ \mathsf{Int})\Big)\ 3.$$

For the remainder of this example I elide the return-type annotations, yielding

$$\Big((\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ ((\lambda\alpha{::}\mathbf{T}.\alpha)\ \mathsf{Int})\Big)\ 3.$$

This program evaluates to 3 because

$$
\begin{array}{l}
((\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ ((\lambda\alpha{::}\mathbf{T}.\alpha)\ \mathsf{Int}))\ 3 \\
= (((\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ \diamond)\ 3)[((\lambda\alpha{::}\mathbf{T}.\alpha)\ \mathsf{Int}] \\
\leadsto (((\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ \diamond)\ 3)[\mathsf{Int}] \\
= (((\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ \mathsf{Int})\ 3) \\
= (\diamond\, 3)[(\Lambda(\alpha{::}\mathbf{T}).\mathsf{fun}\,f(x{:}\,Ty(\alpha))\ \mathsf{is}\ x)\ \mathsf{Int}] \\
\leadsto (\diamond\, 3)[\mathsf{fun}\,f(x{:}\,Ty(\mathsf{Int}))\ \mathsf{is}\ x] \\
= ((\mathsf{fun}\,f(x{:}\,Ty(\mathsf{Int}))\ \mathsf{is}\ x)\ 3 \\
= \diamond[(\mathsf{fun}\,f(x{:}\,Ty(\mathsf{Int}))\ \mathsf{is}\ x)\ 3] \\
\leadsto \diamond[3] \\
= 3
\end{array}
$$

The proofs of important properties of evaluation, including type soundness (that "well-typed programs don't go wrong"), are delayed until Chapter 8. The soundness proof is completely straightforward and standard except for one key point: one must know that constructor and type equivalence are sufficiently consistent. For example, the term-level application $3\,(4)$ makes no sense dynamically. However, if $\mathsf{int} \equiv \mathsf{int}{\rightharpoonup}\mathsf{int}$ were provable then one could prove the application well-typed:

$$
\dfrac{\dfrac{3 : \mathsf{int} \qquad \dfrac{\mathsf{int} \equiv \mathsf{int}{\rightharpoonup}\mathsf{int}}{\mathsf{int} \leq \mathsf{int}{\rightharpoonup}\mathsf{int}}}{3 : \mathsf{int}{\rightharpoonup}\mathsf{int}} \qquad 4 : \mathsf{int}}{3\,(4) : \mathsf{int}}
$$

It is not immediately obvious that $\mathsf{int} \equiv \mathsf{int} \rightharpoonup \mathsf{int}$ is not provable, perhaps using transitivity and introducing and eliminating constructor definitions. The consistency of equivalence will follow directly from the correctness of the decision algorithm for equivalence, which immediately rejects such all type equations.

Chapter 3

Declarative Properties

In this chapter I study several basic properties of the MIL_0 calculus. The most important of these are *validity* and *functionality*. From these I derive the definability of general singleton kinds, the admissibility of the rules given in §2.3, and a *strengthening* property for constructor variables.

3.1 Preliminaries

Figure 3.1 defines typing-context-free judgment forms $\mathcal{J}$. Given a context Γ one can construct a MIL_0 judgment $\Gamma \vdash \mathcal{J}$. The substitution $\gamma\mathcal{J}$ is defined by applying the substitution to the kinds, constructors, types and terms making up $\mathcal{J}$, while the free variable computation $\mathrm{FV}(\mathcal{J})$ is similarly defined as the union of the free variables of the phrases comprising $\mathcal{J}$.

Proposition 3.1.1 (Subderivations)

1. *Every proof of* $\Gamma \vdash \mathcal{J}$ *contains a subderivation* $\Gamma \vdash ok$.
2. *Every proof of* $\Gamma_1, \alpha{::}K, \Gamma_2 \vdash \mathcal{J}$ *contains a strict subderivation* $\Gamma_1 \vdash K$.
3. *Every proof of* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathcal{J}$ *contains a strict subderivation* $\Gamma_1 \vdash \tau$.

Proof: By induction on derivations. ∎

Proposition 3.1.2
If $\Gamma \vdash \mathcal{J}$ *then* $FV(\mathcal{J}) \subseteq \mathrm{dom}(\Gamma)$.

Proof: By induction on derivations. ∎

Proposition 3.1.3 (Reflexivity)

1. *If* $\Gamma \vdash ok$ *then* $\vdash \Gamma \equiv \Gamma$.
2. *If* $\Gamma \vdash K$ *then* $\Gamma \vdash K \equiv K$.
3. *If* $\Gamma \vdash K$ *then* $\Gamma \vdash K \leq K$.
4. *If* $\Gamma \vdash A :: K$ *then* $\Gamma \vdash A \equiv A :: K$.
5. *If* $\Gamma \vdash \tau$ *then* $\Gamma \vdash \tau \leq \tau$.
6. *If* $\Gamma \vdash \tau$ *then* $\Gamma \vdash \tau \equiv \tau$.
7. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash e \equiv e : \tau$.

$$
\begin{array}{rcl}
\mathcal{J} & ::= & \mathrm{ok} \\
& | & \Gamma_1 \equiv \Gamma_2 \\
& | & K \\
& | & K_1 \leq K_2 \\
& | & K_1 \equiv K_2 \\
& | & A :: K \\
& | & A_1 \equiv A_2 :: K \\
& | & \tau \\
& | & \tau_1 \leq \tau_2 \\
& | & \tau_1 \equiv \tau_2 \\
& | & e : \tau \\
& | & e_1 \equiv e_2 : \tau
\end{array}
$$

Figure 3.1: Context-Free Judgment Forms

Proof: By induction on derivations. ∎

Definition 3.1.4
The relation $\Gamma_1 \subseteq \Gamma_2$ on contexts is defined to hold if neither Γ_1 nor Γ_2 binds types or kinds to the same variable twice, and if the contexts viewed as partial functions give the same result for every constructor or term variable in $\mathrm{dom}(\Gamma_1)$.

Thus if $\Gamma_1 \subseteq \Gamma_2$ then $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$ and Γ_1 appears as a (not necessarily consecutive) subsequence of Γ_2. I will also write $\Gamma_2 \supseteq \Gamma_1$ to mean $\Gamma_1 \subseteq \Gamma_2$.

Proposition 3.1.5 (Weakening)

1. *If $\Gamma_1 \vdash \mathcal{J}$ and $\Gamma_1 \subseteq \Gamma_2$ and $\Gamma_2 \vdash ok$, then $\Gamma_2 \vdash \mathcal{J}$.*
2. *If $\Gamma_1, \alpha{::}K_2, \Gamma_2 \vdash \mathcal{J}$ and $\Gamma_1 \vdash K_1 \leq K_2$ and $\Gamma_1 \vdash K_1$ then $\Gamma_1, \alpha{::}K_1, \Gamma_2 \vdash \mathcal{J}$.*
3. *If $\Gamma_1, \alpha{:}\tau_2, \Gamma_2 \vdash \mathcal{J}$ and $\Gamma_1 \vdash \tau_1 \leq \tau_2$ and $\Gamma_1 \vdash \tau_1$ then $\Gamma_1, \alpha{:}\tau_1, \Gamma_2 \vdash \mathcal{J}$.*

Later I show that the assumption $\Gamma_1 \vdash K_1$ is already implied by $\Gamma_1 \vdash K_1 \leq K_2$, and similarly that $\Gamma_1 \vdash \tau_1$ is implied by $\Gamma_1 \vdash \tau_1 \leq \tau_2$.

Definition 3.1.6 (Sizes of Kinds)
The size *of a kind or a type is a strictly positive integer; it is defined inductively on the structure of kinds:*

$$
\begin{array}{ll}
\mathit{size}(\mathbf{T}) & = 1 \\
\mathit{size}(\mathbf{S}(A)) & = 2 \\
\mathit{size}(\Pi\alpha{::}K'.K'') & = \mathit{size}(K') + \mathit{size}(K'') + 2 \\
\mathit{size}(\Sigma\alpha{::}K'.K'') & = \mathit{size}(K') + \mathit{size}(K'') + 2
\end{array}
$$

The size of a kind depends only on its "shape" and is thus invariant under substitutions. The key properties of this measure are that $\mathit{size}(\mathbf{S}(A)) > \mathit{size}(\mathbf{T})$ and that the size of a Π or Σ is strictly greater than the sizes of (all substitution instances of) its constituent kinds.

Proposition 3.1.7 (Antisymmetry of Subkinding)
$\Gamma \vdash K_1 \leq K_2$ and $\Gamma \vdash K_2 \leq K_1$ if and only if $\Gamma \vdash K_1 \equiv K_2$.

Proof:

- $\Rightarrow$ By induction on $size(K_1) + size(K_2)$, and cases on the possible last steps in the proofs of $\Gamma \vdash K_1 \leq K_2$ and $\Gamma \vdash K_2 \leq K_1$.
  - Case: $K_1 = K_2 = \mathbf{T}$. Trivial, since by Proposition 3.1.1 we have $\Gamma \vdash \text{ok}$.
  - Case: $K_1 = \mathbf{S}(A_1)$ and $K_2 = \mathbf{S}(A_2)$. By inversion of $\Gamma \vdash K_1 \leq K_2$ we have $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$, so $\Gamma \vdash \mathbf{S}(A_1) \equiv \mathbf{S}(A_2)$.
  - Case:

$$\frac{\begin{array}{c}\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \\ \Gamma \vdash K_2' \leq K_1' \\ \Gamma, \alpha{::}K_2' \vdash K_1'' \leq K_2''\end{array}}{\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \leq \Pi\alpha{::}K_2'.K_2''} \quad \text{and} \quad \frac{\begin{array}{c}\Gamma \vdash \Pi\alpha{::}K_2'.K_2'' \\ \Gamma \vdash K_1' \leq K_2' \\ \Gamma, \alpha{::}K_1' \vdash K_2'' \leq K_1''\end{array}}{\Gamma \vdash \Pi\alpha{::}K_2'.K_2'' \leq \Pi\alpha{::}K_1'.K_1''}$$

    1. By the inductive hypothesis, $\Gamma \vdash K_1' \equiv K_2'$.
    2. By Proposition 3.1.1, there is a strict subderivation $\Gamma \vdash K_1'$.
    3. By Proposition 3.1.5, $\Gamma, \alpha{::}K_1' \vdash K_1'' \leq K_2''$.
    4. By the inductive hypothesis, $\Gamma, \alpha{::}K_1' \vdash K_1'' \equiv K_2''$.
    5. Thus $\Gamma \vdash \Pi\alpha{::}K_1'.K_1'' \equiv \Pi\alpha{::}K_2'.K_2''$.
  - The case for Σ-kinds is analogous.
- $\Leftarrow$ By induction on the proof of $\Gamma \vdash K_1 \equiv K_2$, using Proposition 3.1.5.

∎

The subtyping relation is similarly antisymmetric, but the proof is more complex in the presence of the transitivity rule (Rule 2.61). I return to this point in §7.3.

Proposition 3.1.8 (Symmetry and Transitivity of Kind Equivalence)

1. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $\Gamma \vdash K_2 \equiv K_1$
2. *If* $\Gamma \vdash K_1 \equiv K_2$ *and* $\Gamma \vdash K_2 \equiv K_3$ *then* $\Gamma \vdash K_1 \equiv K_3$.

Proof: By induction on derivations. ∎

Proposition 3.1.9 (Transitivity of Subkinding)
If $\Gamma \vdash K_1 \leq K_2$ *and* $\Gamma \vdash K_2 \leq K_3$ *then* $\Gamma \vdash K_1 \leq K_3$.

Proof: By induction on derivations. ∎

Definition 3.1.10
The judgment $\Delta \vdash \gamma : \Gamma$ *holds if and only if the following conditions all hold:*

1. $\Delta \vdash$ *ok*
2. $\forall \alpha \in \text{dom}(\Gamma).\ \Delta \vdash \gamma(\Gamma(\alpha))$
3. $\forall \alpha \in \text{dom}(\Gamma).\ \Delta \vdash \gamma\alpha :: \gamma(\Gamma(\alpha))$
4. $\forall x \in \text{dom}(\Gamma).\ \Delta \vdash \gamma(\Gamma(x))$
5. $\forall x \in \text{dom}(\Gamma).\ \Delta \vdash \gamma x : \gamma(\Gamma(x))$

Proposition 3.1.11 (Substitution)

1. *If* $\Gamma \vdash \mathcal{J}$ *and* $\Delta \vdash \gamma : \Gamma$ *then* $\Delta \vdash \gamma(\mathcal{J})$.
2. *If* $\Gamma_1, \alpha{::}K, \Gamma_2 \vdash ok$ *and* $\Gamma_1 \vdash A :: K$ *then* $\Gamma_1, [A/\alpha]\Gamma_2 \vdash ok$.
3. *If* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash ok$ *and* $\Gamma_1 \vdash v : \tau$ *then* $\Gamma_1, [v/x]\Gamma_2 \vdash ok$.
4. *If* $\Gamma_1, \alpha{::}K, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash A :: K$ *then* $\Gamma_1, [A/\alpha]\Gamma_2 \vdash [A/\alpha]\mathcal{J}$.
5. *If* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash v : \tau$ *then* $\Gamma_1, [v/x]\Gamma_2 \vdash [v/x]\mathcal{J}$.

Proof:

1. By induction on the proof of $\Gamma \vdash \mathcal{J}$.

2–5. By simultaneous induction on the context in the first assumption and by part 1.

∎

3.2 Validity and Functionality

I next show two important features of the calculus. *Validity* is the property that any phrase appearing within a judgment is well-formed (e.g., if $\Gamma \vdash A_1 \equiv A_2 :: K$ then $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash K$ and $\Gamma \vdash A_1 :: K$ and $\Gamma \vdash A_2 :: K$). *Functionality* states that applying equivalent substitutions to related phrases yields related phrases.

The rules have been structured to assume validity for premises and guarantee and preserve validity for conclusions. A simple proof, however, is hindered by the presence of dependencies in types and kinds. The direct approach by induction on derivations fails because of cases such as Rule 2.39:

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''}.$$

Here we need $\Gamma \vdash \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$ but from the inductive hypothesis we get only $\Gamma \vdash \pi_2 A_2 :: [\pi_1 A_2/\alpha]K''$. The desired result would follow, however, if we knew that $\Gamma \vdash [\pi_1 A_2/\alpha]K'' \leq [\pi_1 A_1/\alpha]K''$. Since $\Gamma \vdash \pi_1 A_2 \equiv \pi_1 A_1 :: K'$, the subkinding judgment required follows from functionality.

This suggests one should first prove functionality. The most general form of functionality also cannot be easily proved directly, but the proof does go through for the restricted case of equivalent substitutions being applied to a *single* phrase. This suffices to show validity, and together these allow a simple proof of general functionality.

Definition 3.2.1
The judgment $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *holds if and only if the following conditions all hold:*

1. $\Delta \vdash \gamma_1 : \Gamma$ *and* $\Delta \vdash \gamma_2 : \Gamma$
2. $\forall \alpha \in \mathrm{dom}(\Gamma).\ \Delta \vdash \gamma_1(\Gamma(\alpha)) \equiv \gamma_2(\Gamma(\alpha))$
3. $\forall \alpha \in \mathrm{dom}(\Gamma).\ \Delta \vdash \gamma_1\alpha \equiv \gamma_2\alpha :: \gamma_1(\Gamma(\alpha))$
4. $\forall x \in \mathrm{dom}(\Gamma).\ \Delta \vdash \gamma_1(\Gamma(x)) \equiv \gamma_2(\Gamma(x))$
5. $\forall x \in \mathrm{dom}(\Gamma).\ \Delta \vdash \gamma_1 x \equiv \gamma_2 x : \gamma_1(\Gamma(x))$

Lemma 3.2.2 (Substitution Extension)

1. *If* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$, $\alpha \notin \mathrm{dom}(\Delta)$, $\Delta \vdash \gamma_1 K$, $\Delta \vdash \gamma_2 K$, *and* $\Delta \vdash \gamma_1 K \equiv \gamma_2 K$, *then* $\Delta, \alpha{::}\gamma_1 K \vdash \gamma_1[\alpha \mapsto \alpha] \equiv \gamma_2[\alpha \mapsto \alpha] : (\Gamma, \alpha{::}K)$ *and* $\Delta, \alpha{::}\gamma_2 K \vdash \gamma_1[\alpha \mapsto \alpha] \equiv \gamma_2[\alpha \mapsto \alpha] : (\Gamma, \alpha{::}K)$.
2. *If* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$, $x \notin \mathrm{dom}(\Delta)$, *and* $\Delta \vdash \gamma_1 \tau$, $\Delta \vdash \gamma_2 \tau$, *and* $\Delta \vdash \gamma_1 \tau \equiv \gamma_2 \tau$ *then* $\Delta, x{:}\gamma_1 \tau \vdash \gamma_1[\alpha \mapsto \alpha] \equiv \gamma_2[\alpha \mapsto \alpha] : (\Gamma, x{:}\tau)$ *and* $\Delta, x{:}\gamma_2 \tau \vdash \gamma_1[\alpha \mapsto \alpha] \equiv \gamma_2[\alpha \mapsto \alpha] : (\Gamma, x{:}\tau)$.

Proof: By the definition $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$, Proposition 3.1.5, and the subsumption rules. ∎

Proposition 3.2.3 (Simple Functionality)

1. *If* $\Gamma \vdash K$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 K \equiv \gamma_2 K$.
2. *If* $\Gamma \vdash A :: K$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1 K$.
3. *If* $\Gamma \vdash \tau$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 \tau \equiv \gamma_2 \tau$.
4. *If* $\Gamma \vdash e : \tau$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 e \equiv \gamma_2 e : \gamma_1 \tau$.

Proof: [By induction on the proof of the first premise]

1. • Case: Rule 2.7

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathbf{T}}$$

Since $\Delta \vdash \mathrm{ok}$ we have $\Delta \vdash \mathbf{T} \equiv \mathbf{T}$.

• Case: Rule 2.8

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

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \mathbf{T}$.
(b) By Rule 2.17 then, $\Delta \vdash \mathbf{S}(\gamma_1 A) \equiv \mathbf{S}(\gamma_2 A)$.

• Case: Rule 2.9

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

(a) Without loss of generality, $\alpha \notin \mathrm{dom}(\Delta)$.
(b) By Proposition 3.1.1, there are strict subderivations $\Gamma, \alpha{::}K' \vdash \mathrm{ok}$ and $\Gamma \vdash K'$.
(c) By inversion and Proposition 3.1.2, $\alpha \notin \mathrm{FV}(K')$.
(d) By the inductive hypothesis, $\Delta \vdash \gamma_1 K' \equiv \gamma_2 K'$
(e) and by Proposition 3.1.11, $\Delta \vdash \gamma_1 K'$ and $\Delta \vdash \gamma_2 K'$.
(f) Using Lemma 3.2.2, we have $\Delta, \alpha{::}\gamma_1 K' \vdash \gamma_1[\alpha \mapsto \alpha] \equiv \gamma_2[\alpha \mapsto \alpha] : (\Gamma, \alpha{::}K')$.
(g) By the inductive hypothesis then, we have $\Delta, \alpha{::}\gamma_1 K' \vdash (\gamma_1[\alpha \mapsto \alpha])K'' \equiv (\gamma_2[\alpha \mapsto \alpha])K''$
(h) By substitution, $\Delta \vdash \gamma_1(\Pi\alpha{::}K'.K'')$
(i) Therefore $\Delta \vdash \gamma_1(\Pi\alpha{::}K'.K'') \equiv \gamma_2(\Pi\alpha{::}K'.K'')$.

• Case: Rule 2.10

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

Analogous to the previous case.

2.
- Case: Rule 2.20

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash b :: \mathbf{T}}$$

Then $\Delta \vdash b \equiv b :: \mathbf{T}$ because $\Delta \vdash \text{ok}$.

- Case: Rule 2.21

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \times :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}}$$

Then $\Delta \vdash \times \equiv \times :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$ because $\Delta \vdash \text{ok}$.

- Case: Rule 2.22

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \to :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}}$$

Then $\Delta \vdash \to \equiv \to :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$ because $\Delta \vdash \text{ok}$.

- Case: Rule 2.23

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

Follows directly from the requirements for γ_1 and γ_2.

- Case: Rule 2.24

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

(a) Without loss of generality, $\alpha \notin \text{dom}(\Delta)$.
(b) As in the case for Rule 2.9, we have $\Delta \vdash \gamma_1 K' \equiv \gamma_2 K'$
(c) and $\Delta, \alpha{::}\gamma_1 K' \vdash \gamma_1[\alpha{\mapsto}\alpha] \equiv \gamma_2[\alpha{\mapsto}\alpha] : \Gamma, \alpha{::}K'$.
(d) Thus by the inductive hypothesis,
$\Delta, \alpha{::}\gamma_1 K' \vdash (\gamma_1[\alpha{\mapsto}\alpha])A \equiv (\gamma_2[\alpha{\mapsto}\alpha])A :: (\gamma_1[\alpha{\mapsto}\alpha])K''$.
(e) By Rule 2.36 we have $\Delta \vdash \gamma_1(\lambda\alpha{::}K'.A) \equiv \gamma_2(\lambda\alpha{::}K'.A) :: \gamma_1(\Pi\alpha{::}K'.K'')$.

- Case: Rule 2.25

$$\frac{\Gamma \vdash A :: K'{\to}K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash A\,A' :: K''}$$

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: (\gamma_1 K'){\to}(\gamma_1 K'')$
(b) and $\Delta \vdash \gamma_1 A' \equiv \gamma_2 A' :: \gamma_1 K'$.
(c) Thus by Rule 2.37, $\Delta \vdash \gamma_1(A\,A') \equiv \gamma_2(A\,A') :: \gamma_1 K''$.

- Case: Rule 2.26

$$\frac{\Gamma \vdash A' :: K' \qquad \Gamma \vdash A'' :: K''}{\Gamma \vdash \langle A', A''\rangle :: K'{\times}K''}$$

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A' \equiv \gamma_2 A' :: \gamma_1 K'$
(b) and $\Delta \vdash \gamma_1 A'' \equiv \gamma_2 A'' :: \gamma_1 K''$.
(c) Thus $\Delta \vdash \langle \gamma_1 A', \gamma_1 A''\rangle \equiv \langle \gamma_2 A', \gamma_2 A''\rangle :: \gamma_1 K'{\times}\gamma_1 K''$ by Rule 2.40.

- Case: Rule 2.27

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_1 A :: K'}$$

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1(\Sigma\alpha{::}K'.K'')$.

(b) By Rule 2.38, $\Delta \vdash \gamma_1(\pi_1 A) \equiv \gamma_2(\pi_1 A) :: \gamma_1 K'$.

- Case: Rule 2.28

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A :: [\pi_1 A/\alpha]K''}$$

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1(\Sigma\alpha{::}K'.K'')$.

(b) By Rule 2.39, $\Delta \vdash \pi_2(\gamma_1 A) \equiv \pi_2(\gamma_2 A) :: [\pi_1(\gamma_1 A)/\alpha](\gamma_1[\alpha\mapsto\alpha])K''$.

(c) That is, $\Delta \vdash \pi_2(\gamma_1 A) \equiv \pi_2(\gamma_2 A) :: \gamma_1([\pi_1 A/\alpha]K'')$.

- Case: Rule 2.29

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

(a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \mathbf{T}$.

(b) By substitution, $\Delta \vdash \gamma_1 A :: \mathbf{T}$.

(c) Thus $\Delta \vdash \gamma_1 A :: \mathbf{S}(\gamma_1 A)$,

(d) but $\Delta \vdash \mathbf{S}(\gamma_1 A) \leq \mathbf{S}(\gamma_2 A)$

(e) so $\Delta \vdash \gamma_1 A :: \mathbf{S}(\gamma_2 A)$.

(f) By Rule 2.44, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \mathbf{S}(\gamma_2 A)$

(g) and by subsumption and symmetry, $\Delta \vdash \gamma_2 A \equiv \gamma_1 A :: \mathbf{T}$.

(h) Thus $\Delta \vdash \mathbf{S}(\gamma_2 A) \leq \mathbf{S}(\gamma_1 A)$

(i) and so $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \mathbf{S}(\gamma_1 A)$.

- Case: Rule 2.30

$$\frac{\Gamma \vdash \pi_1 A :: K' \qquad \Gamma \vdash \pi_2 A :: K''}{\Gamma \vdash A :: K'{\times}K''}$$

(a) By the inductive hypothesis, $\Delta \vdash \pi_1(\gamma_1 A) \equiv \pi_1(\gamma_2 A) :: \gamma_1 K'$

(b) and $\Delta \vdash \pi_2(\gamma_1 A) \equiv \pi_2(\gamma_2 A) :: \gamma_1 K''$.

(c) By Rule 2.41, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: (\gamma_1 K')\times(\gamma_1 K'')$.

- Case: Rule 2.31

$$\frac{\begin{array}{c}\Gamma, \alpha{::}K' \vdash A\,\alpha :: K'' \\ \Gamma \vdash A :: \Pi\alpha{::}L'.L'' \qquad \Gamma \vdash K' \equiv L'\end{array}}{\Gamma \vdash A :: \Pi\alpha{::}K'.K''}$$

(a) Without loss of generality, $\alpha \notin \mathrm{dom}(\Delta)$ and $\alpha \notin \mathrm{FV}(A)$.

(b) As in the case for Rule 2.9, $\Delta \vdash \gamma_1 K' \equiv \gamma_2 K'$

(c) and $\Delta, \alpha{::}\gamma_1 K' \vdash \gamma_1[\alpha\mapsto\alpha] \equiv \gamma_2[\alpha\mapsto\alpha] : \Gamma, \alpha{::}K'$.

(d) Thus by the inductive hypothesis,
$\Delta, \alpha{::}\gamma_1 K' \vdash (\gamma_1[\alpha\mapsto\alpha])(A\,\alpha) \equiv (\gamma_2[\alpha\mapsto\alpha])(A\,\alpha) :: (\gamma_1[\alpha\mapsto\alpha])K''$.

(e) That is, $\Delta, \alpha{::}\gamma_1 K' \vdash (\gamma_1 A)\alpha \equiv (\gamma_2 A)\alpha :: \gamma_1[\alpha\mapsto\alpha]K''$.

(f) By Proposition 3.1.11, we have $\Delta \vdash \gamma_1 A :: \gamma_1(\Pi\alpha{::}L'.L'')$ and
$\Delta \vdash \gamma_2 A :: \gamma_2(\Pi\alpha{::}L'.L'')$.

(g) Similarly we have $\Delta \vdash \gamma_1 K' \equiv \gamma_1 L'$ and $\Delta \vdash \gamma_2 K' \equiv \gamma_2 L'$.

(h) so by Proposition 3.1.8, we have $\Delta \vdash \gamma_1 K' \equiv \gamma_2 L'$.

(i) Therefore by Rule 2.42, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1(\Pi\alpha{::}K'.K'')$.

- Case: Rule 2.32

$$\frac{\Gamma \vdash A :: K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash A :: K_2}$$

  (a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1 K_1$.
  (b) By Proposition 3.1.11, $\Delta \vdash \gamma_1 K_1 \leq \gamma_1 K_2$.
  (c) By Rule 2.43, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1 K_2$.

3. - Case: Rule 2.45

$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A)}$$

  (a) By the inductive hypothesis, $\Delta \vdash \gamma_1 A \equiv \gamma_2 A :: \mathbf{T}$.
  (b) Thus $\Delta \vdash \mathit{Ty}(\gamma_1 A) \equiv \mathit{Ty}(\gamma_2 A)$.

- Rule 2.46

$$\frac{\Gamma \vdash v : \tau \qquad \tau \text{ not a singleton}}{\Gamma \vdash \mathbf{S}(v : \tau)}$$

  (a) By the inductive hypothesis, $\Delta \vdash \gamma_1 v \equiv \gamma_2 v : \gamma_1 \tau$
  (b) and $\Delta \vdash \gamma_1 \tau \equiv \gamma_2 \tau$.
  (c) Since neither $\gamma_1 \tau$ nor $\gamma_2 \tau$ can be a singleton (because τ isn't), we have $\Delta \vdash \mathbf{S}(\gamma_1 v : \gamma_1 \tau) \equiv \mathbf{S}(\gamma_2 v : \gamma_2 \tau)$.

- Case: Rule 2.47

$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau') \rightharpoonup \tau''}$$

  Same argument as for Rule 2.9.

- Case: Rule 2.48

$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau') \times \tau''}$$

  Same argument as for Rule 2.10.

- Case: Rule 2.49

$$\frac{\Gamma, \alpha{::}K \vdash \tau}{\Gamma \vdash \forall \alpha{::}K.\tau}$$

  Similar argument to that for Rule 2.9.

4. - Case: Rules 2.67–2.78. Essentially the same proofs as for the corresponding constructor forms.

∎

Proposition 3.2.4 (Validity)

1. *If* $\Gamma \vdash K_1 \leq K_2$ *then* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$.
2. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$.
3. *If* $\Gamma \vdash A :: K$ *then* $\Gamma \vdash K$.
4. *If* $\Gamma \vdash A_1 \equiv A_2 :: K$ *then* $\Gamma \vdash A_1 :: K$, $\Gamma \vdash A_2 :: K$, *and* $\Gamma \vdash K$.
5. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *then* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$.

6. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$.
7. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash \tau$.
8. *If* $\Gamma \vdash e_1 \equiv e_2 : \tau$ *then* $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, *and* $\Gamma \vdash \tau$.

Proof: There are only two interesting cases.

- Case: Rule 2.39.
$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''}$$
  1. By the inductive hypothesis, $\Gamma \vdash A_1 :: \Sigma\alpha{::}K'.K''$,
  2. $\Gamma \vdash A_2 :: \Sigma\alpha{::}K'.K''$,
  3. and $\Gamma \vdash \Sigma\alpha{::}K'.K''$.
  4. By inversion, $\Gamma, \alpha{::}K' \vdash K''$.
  5. Then $\Gamma \vdash \pi_2 A_1 :: [\pi_1 A_1/\alpha]K''$ by Rule 2.28.
  6. By Proposition 3.1.11, we have $\Gamma \vdash [\pi_1 A_1/\alpha]K''$.
  7. Since $\Gamma \vdash \pi_1 A_2 :: K'$ and $\Gamma \vdash \pi_1 A_1 :: K'$ and $\Gamma \vdash \pi_1 A_2 \equiv \pi_1 A_1 :: K'$,
  8. we have $\Gamma \vdash [\pi_1 A_2/\alpha] \equiv [\pi_1 A_1/\alpha] : \Gamma, \alpha{::}K'$.
  9. By Propositions 3.2.3 and 3.1.7 we have $\Gamma \vdash [\pi_1 A_2/\alpha]K'' \leq [\pi_1 A_1/\alpha]K''$.
  10. Thus by subsumption and $\Gamma \vdash \pi_2 A_2 :: [\pi_1 A_2/\alpha]K''$
  11. we have $\Gamma \vdash \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$.
- Case: Rule 2.86. The proof is analogous.

∎

Corollary 3.2.5 (Full Functionality)

1. *If* $\Gamma \vdash A_1 \equiv A_2 :: K$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 A_1 \equiv \gamma_2 A_2 :: \gamma_1 K$.
2. *If* $\Gamma \vdash K_1 \equiv K_2$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 K_1 \equiv \gamma_2 K_2$.
3. *If* $\Gamma \vdash K_1 \leq K_2$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 K_1 \leq \gamma_2 K_2$.
4. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 \tau_1 \equiv \gamma_2 \tau_2$.
5. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 \tau_1 \leq \gamma_2 \tau_2$.
6. *If* $\Gamma \vdash e_1 \equiv e_2 : \tau$ *and* $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$ *then* $\Delta \vdash \gamma_1 e_1 \equiv \gamma_2 e_2 : \gamma_1 \tau$.

Proof:

1. Assume $\Gamma \vdash A_1 \equiv A_2 :: K$ and $\Delta \vdash \gamma_1 \equiv \gamma_2 : \Gamma$. By substitution, $\Delta \vdash \gamma_1 A_1 \equiv \gamma_1 A_2 :: \gamma_1 K$. By validity (Proposition 3.2.4) we have $\Gamma \vdash A_2 :: K$, and so by Proposition 3.2.3, $\Delta \vdash \gamma_1 A_2 \equiv \gamma_2 A_2 :: \gamma_1 K$. By transitivity, $\Delta \vdash \gamma_1 A_1 \equiv \gamma_2 A_2 :: \gamma_1 K$.

2–6. The remaining cases are similar.

∎

Lemma 3.2.6

1. *If* $\Gamma', \alpha{::}K, \Gamma'' \vdash ok$ *and* $\Gamma' \vdash A_1 \equiv A_2 :: K$ *then* $\Gamma', [A_1/\alpha]\Gamma'' \vdash [A_1/\alpha] \equiv [A_2/\alpha] : (\Gamma', \alpha{::}K, \Gamma'')$ *and* $\Gamma', [A_2/\alpha]\Gamma'' \vdash [A_1/\alpha] \equiv [A_2/\alpha] : (\Gamma', \alpha{::}K, \Gamma'')$.
2. *If* $\Gamma', x{:}\tau, \Gamma'' \vdash ok$ *and* $\Gamma' \vdash v_1 \equiv v_2 : \tau$ *then* $\Gamma', [v_1/x]\Gamma'' \vdash [v_1/x] \equiv [A_2/\alpha] : (\Gamma', x{:}\tau, \Gamma'')$ *and* $\Gamma', [v_2/x]\Gamma'' \vdash [v_1/x] \equiv [v_2/x] : (\Gamma', x{:}\tau, \Gamma'')$.

Proof: By induction on the proof of typing context well-formedness and Proposition 3.2.3. ∎

Corollary 3.2.7

1. *If* $\Gamma', \alpha{::}L, \Gamma'' \vdash K_1 \equiv K_2$ *and* $\Gamma' \vdash B_1 \equiv B_2 :: L$ *then* $\Gamma', [B_1/\alpha]\Gamma'' \vdash [B_1/\alpha]K_1 \equiv [B_2/\alpha]K_2$.
2. *If* $\Gamma', \alpha{::}L, \Gamma'' \vdash K_1 \leq K_2$ *and* $\Gamma' \vdash B_1 \equiv B_2 :: L$ *then* $\Gamma', [B_1/\alpha]\Gamma'' \vdash [B_1/\alpha]K_1 \leq [B_2/\alpha]K_2$.
3. *If* $\Gamma', \alpha{::}L, \Gamma'' \vdash \tau_1 \equiv \tau_2$ *and* $\Gamma' \vdash B_1 \equiv B_2 :: L$ *then* $\Gamma', [B_1/\alpha]\Gamma'' \vdash [B_1/\alpha]\tau_1 \equiv [B_2/\alpha]\tau_2$.
4. *If* $\Gamma', \alpha{::}L, \Gamma'' \vdash \tau_1 \leq \tau_2$ *and* $\Gamma' \vdash B_1 \equiv B_2 :: L$ *then* $\Gamma', [B_1/\alpha]\Gamma'' \vdash [B_1/\alpha]\tau_1 \leq [B_2/\alpha]\tau_2$.
5. *If* $\Gamma', \alpha{::}L, \Gamma'' \vdash v_1 \equiv v_2 : \tau$ *and* $\Gamma' \vdash B_1 \equiv B_2 :: L$ *then* $\Gamma', [B_1/\alpha]\Gamma'' \vdash [B_1/\alpha]v_1 \equiv [B_2/\alpha]v_2 : [B_1/\alpha]\tau$.
6. *If* $\Gamma', y{:}\sigma, \Gamma'' \vdash \tau_1 \equiv \tau_2$ *and* $\Gamma' \vdash w_1 \equiv w_2 : \sigma$ *then* $\Gamma', [w_1/y]\Gamma'' \vdash [w_1/y]\tau_1 \equiv [w_2/y]\tau_2$.
7. *If* $\Gamma', y{:}\sigma, \Gamma'' \vdash \tau_1 \leq \tau_2$ *and* $\Gamma' \vdash w_1 \equiv w_2 : \sigma$ *then* $\Gamma', [w_1/y]\Gamma'' \vdash [w_1/y]\tau_1 \leq [w_2/y]\tau_2$.
8. *If* $\Gamma', y{:}\sigma, \Gamma'' \vdash v_1 \equiv v_2 : \tau$ *and* $\Gamma' \vdash w_1 \equiv w_2 : \sigma$ *then* $\Gamma', [w_1/y]\Gamma'' \vdash [w_1/y]v_1 \equiv [w_2/y]v_2 : [w_1/y]\tau$.

The proof of Proposition 3.2.3 depends heavily on the exact formulation of the rules defining MIL_0. In particular, although dependent kinds and types force the rules to be asymmetric, they are all "asymmetric in the same way". For example, if Rule 2.39 were written instead as

$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_2/\alpha]K''}$$

(where the substitution involves $\pi_1 A_2$ instead of $\pi_1 A_1$) then the above case for Rule 2.39 would not go through. A more robust but more technically involved method would be to prove validity and general functionality simultaneously. This requires a logical relations argument because inductively one needs to know, for example, that not only are Π and Σ kinds functional in their free variables, but also that their codomains are functional with respect to the domain variable. Stone and Harper [SH99] use this method for proving validity and functionality for the kind and constructors levels.

Alternatively, functionality could be built into the system. Harper and Pfenning [HP99] take the approach of making functionality into an axiom. However, it appears that the same proof method used here would show their axiom admissible [Har00]. Martin-Löf goes further and makes functionality the defining property of what it means to be a valid judgment-in-context [ML84].

Corollary 3.2.8 (Weakening 2)

1. *If* $\Gamma_1, \alpha{::}K_2, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash K_1 \leq K_2$ *then* $\Gamma_1, \alpha{::}K_1, \Gamma_2 \vdash \mathcal{J}$.
2. *If* $\Gamma_1, x{:}\tau_2, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \tau_1 \leq \tau_2$ *then* $\Gamma_1, x{:}\tau_1, \Gamma_2 \vdash \mathcal{J}$.
3. *If* $\Gamma \vdash \mathcal{J}$ *and* $\vdash \Gamma \equiv \Gamma'$ *then* $\Gamma' \vdash \mathcal{J}$.

3.3 Proofs of Admissibility

I now have enough technical machinery to prove the admissibility of Rules 2.92–2.124.

Proposition 3.3.1
Rules 2.119 and 2.122 are admissible.

Proof: I show the proof for Rule 2.119 only; the other proof is analogous.

1. Assume $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$.
2. By validity $\Gamma \vdash A_1 :: \mathbf{T}$,
3. so $\Gamma \vdash A_1 :: \mathbf{S}(A_1)$ by Rule 2.29.
4. But $\Gamma \vdash \mathbf{S}(A_1) \leq \mathbf{S}(A_2)$,
5. so by subsumption we have $\Gamma \vdash A_1 :: \mathbf{S}(A_2)$.

∎

Lemma 3.3.2
$\gamma(\mathbf{S}(A :: K)) = \mathbf{S}(\gamma A :: \gamma K)$.

Proof: By induction on the size of K, and by cases on the form of K. ∎

Proposition 3.3.3

1. *Rule 2.96 is admissible. That is, if* $\Gamma \vdash A_1 \equiv A_2 :: K$ *then* $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2 :: K)$.
2. *Rules 2.92 and 2.93 are admissible.*
   That is, if $\Gamma \vdash A :: K$ *then* $\Gamma \vdash \mathbf{S}(A :: K)$ *and* $\Gamma \vdash A :: \mathbf{S}(A :: K)$.
3. *Rule 2.97 is admissible.*
   That is, if $\Gamma \vdash A_1 :: \mathbf{S}(A_2 :: K)$ *and* $\Gamma \vdash A_2 :: K$ *then* $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2 :: K)$.
4. *Rule 2.94 is admissible. That is, if* $\Gamma \vdash A :: K$ *then* $\Gamma \vdash \mathbf{S}(A :: K) \leq K$.
5. *Rules 2.98 and 2.99 are admissible.*
   That is, if $\Gamma \vdash A :: \Pi\alpha{::}K'.K''$ *and* $\Gamma \vdash A' :: K'$ *then* $\Gamma \vdash A\,A' :: [A'/\alpha]K''$. *Similarly, if* $\Gamma \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K'.K''$ *and* $\Gamma \vdash A_1' \equiv A_2' :: K'$ *then* $\Gamma \vdash A_1\,A_1' \equiv A_2\,A_2' :: [A_1'/\alpha]K''$.
6. *Rule 2.102 is admissible.*
   That is, if $\Gamma \vdash \Sigma\alpha{::}K'.K''$, $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K'$, *and* $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$ *then* $\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''$.
7. *Rule 2.95 is admissible.*
   That is, if $\Gamma \vdash A_1 \equiv A_2 :: K_1$ *and* $\Gamma \vdash K_1 \leq K_2$ *then* $\Gamma \vdash \mathbf{S}(A_1 :: K_1) \leq \mathbf{S}(A_2 :: K_2)$.

Proof: By simultaneous induction on the size of kinds. (The size of K for parts 1–4, the size of K' for part 5 and part 6, and the size of K_1 for part 7.)

1.
   - Case $K = \mathbf{T}$ and $\mathbf{S}(A_2 :: K) = \mathbf{S}(A_2)$.
     (a) $\Gamma \vdash A_1 :: \mathbf{S}(A_2)$ by Rule 2.119.
     (b) Then $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2)$ by Rule 2.44
   - Case $K = \mathbf{S}(B)$ and $\mathbf{S}(A_2 :: K) = \mathbf{S}(A_2)$.
     (a) $\Gamma \vdash B :: \mathbf{T}$ by validity and inversion, so $\Gamma \vdash \mathbf{S}(B) \leq \mathbf{T}$.

(b) Then $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$ by subsumption,

(c) and $\Gamma \vdash A_1 :: \mathbf{S}(A_2)$.

(d) Thus $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2)$ by Rule 2.44.

- Case $K = \Pi\alpha{::}K'.K''$ and $\mathbf{S}(A_2 :: K) = \Pi\alpha{::}K'.\mathbf{S}(A_2\,\alpha :: K'')$.

  (a) Inductively by part 5, $\Gamma, \alpha{::}K' \vdash A_1\,\alpha \equiv A_2\,\alpha :: K''$.

  (b) By the inductive hypothesis, $\Gamma, \alpha{::}K' \vdash A_1\,\alpha \equiv A_2\,\alpha :: \mathbf{S}(A_2\,\alpha :: K'')$.

  (c) By validity (Proposition 3.2.4) we have $\Gamma \vdash A_1 :: \Pi\alpha{::}K'.K''$ and $\Gamma \vdash A_2 :: \Pi\alpha{::}K'.K''$.

  (d) Therefore by Rule 2.42, $\Gamma \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K'.\mathbf{S}(A_2\,\alpha :: K'')$.

- $K = \Sigma\alpha{::}K'.K''$ and $\mathbf{S}(A_2 :: K) = (\mathbf{S}(\pi_1 A_2 :: K')) \times (\mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K''))$.

  (a) Then $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K'$

  (b) and $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$.

  (c) By functionality and subsumption, $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_2/\alpha]K''$.

  (d) By the inductive hypothesis, $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: \mathbf{S}(\pi_1 A_2 :: K')$

  (e) and $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: \mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K'')$. (Note that $\mathit{size}([\pi_1 A_2/\alpha]K'') = \mathit{size}(K'') < \mathit{size}(K)$.)

  (f) Therefore by Rule 2.41 we have $\Gamma \vdash A_1 \equiv A_2 :: (\mathbf{S}(\pi_1 A_2 :: K')) \times (\mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K''))$.

2. (a) Assume $\Gamma \vdash A :: K$.

   (b) By Rule 2.33, $\Gamma \vdash A \equiv A :: K$.

   (c) By the previous part, $\Gamma \vdash A \equiv A :: \mathbf{S}(A :: K)$.

   (d) By validity, $\Gamma \vdash \mathbf{S}(A :: K)$ and $\Gamma \vdash A :: \mathbf{S}(A :: K)$.

3. - Case $K = \mathbf{T}$ and $\mathbf{S}(A_2 :: K) = \mathbf{S}(A_2)$. By Rule 2.44, $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2)$.
   - Case $K = \mathbf{S}(B)$ and $\mathbf{S}(A_2 :: K) = \mathbf{S}(A_2)$. By Rule 2.44, $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{S}(A_2)$.
   - Case $K = \Pi\alpha{::}K'.K''$ and $\mathbf{S}(A_2 :: K) = \Pi\alpha{::}K'.\mathbf{S}(A_2\,\alpha :: K'')$.

     (a) Inductively by part 5 we have $\Gamma, \alpha{::}K' \vdash A_1\,\alpha :: \mathbf{S}(A_2\,\alpha :: K'')$.

     (b) and $\Gamma, \alpha{::}K' \vdash A_2\,\alpha :: K''$.

     (c) By the inductive hypothesis, $\Gamma, \alpha{::}K' \vdash A_1\,\alpha \equiv A_2\,\alpha :: \mathbf{S}(A_2\,\alpha :: K'')$.

     (d) Therefore by Rule 2.42 we have $\Gamma \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K'.\mathbf{S}(A_2\,\alpha :: K'')$.
   - $K = \Sigma\alpha{::}K'.K_2$ and $\mathbf{S}(A_2 :: K) = (\mathbf{S}(\pi_1 A_2 :: K')) \times (\mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K''))$.

     (a) Then $\Gamma \vdash \pi_1 A_1 :: \mathbf{S}(\pi_1 A_2 :: K')$ and

     (b) $\Gamma \vdash \pi_2 A_1 :: \mathbf{S}(\pi_2 A_2 :: [\pi_1 A_1/\alpha]K'')$.

     (c) $\Gamma \vdash \pi_1 A_2 :: K'$ and $\Gamma \vdash \pi_2 A_2 :: [\pi_1 A_2/\alpha]K'$,

     (d) so by the inductive hypothesis, $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: \mathbf{S}(\pi_1 A_2 :: K')$ and

     (e) $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: \mathbf{S}(\pi_2 A_2 :: [\pi_1 A_1/\alpha]K'')$.

     (f) By Rule 2.41 we have $\Gamma \vdash A_1 \equiv A_2 :: (\mathbf{S}(\pi_1 A_2 :: K')) \times (\mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K''))$.

4. - Case $K = \mathbf{T}$ and $\mathbf{S}(A :: K) = \mathbf{S}(A)$. By Rule 2.11 we have $\Gamma \vdash \mathbf{S}(A :: \mathbf{T}) \leq \mathbf{T}$.
   - Case $K = \mathbf{S}(B)$ and $\mathbf{S}(A :: K) = \mathbf{S}(A)$.

     (a) Then $\Gamma \vdash A \equiv B :: \mathbf{T}$ so

     (b) $\Gamma \vdash \mathbf{S}(A) \leq \mathbf{S}(B)$.

- Case $K = \Pi\alpha{::}K_1.K_2$ and $\mathbf{S}(A :: K) = \Pi\alpha{::}K_1.\mathbf{S}(A\,\alpha :: K_2)$.
  (a) Then $\Gamma \vdash K_1$ and $\Gamma, \alpha{::}K_1 \vdash A\,\alpha :: K_2$.
  (b) By the inductive hypothesis, $\Gamma, \alpha{::}K_1 \vdash \mathbf{S}(A\,\alpha :: K_2) \leq K_2$.
  (c) Therefore, $\Gamma \vdash \Pi\alpha{::}K_1.\mathbf{S}(A\,\alpha :: K_2) \leq \Pi\alpha{::}K_1.K_2$.
- Case $K = \Sigma\alpha{::}K'.K''$ and $\mathbf{S}(A :: K) = (\mathbf{S}(\pi_1 A :: K'))\times(\mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]K''))$.
  (a) Then $\Gamma \vdash \pi_1 A :: K'$
  (b) so by the inductive hypothesis, $\Gamma \vdash \mathbf{S}(\pi_1 A :: K') \leq K'$.
  (c) Furthermore, $\Gamma \vdash \pi_2 A :: [\pi_1 A/\alpha]K''$.
  (d) By the inductive hypothesis, $\Gamma \vdash \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]K'') \leq [\pi_1 A/\alpha]K''$.
  (e) Also, by Proposition 3.1.1 and weakening, $\Gamma, \alpha{::}\mathbf{S}(\pi_1 A :: K') \vdash K'' \leq K''$.
  (f) By part 3 we have $\Gamma, \alpha{::}\mathbf{S}(\pi_1 A :: K') \vdash \alpha \equiv \pi_1 A :: \mathbf{S}(\pi_1 A :: K')$
  (g) so by functionality we have $\Gamma, \alpha{::}\mathbf{S}(\pi_1 A :: K') \vdash [\pi_1 A/\alpha]K'' \leq K''$.
  (h) Therefore, $\Gamma \vdash (\mathbf{S}(\pi_1 A :: K'))\times(\mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]K'')) \leq \Sigma\alpha{::}K'.K''$.

5. (a) Assume $\Gamma \vdash A :: \Pi\alpha{::}K'.K''$ and $\Gamma \vdash A' :: K'$.
   (b) Then by part 4, $\Gamma \vdash \mathbf{S}(A' :: K') \leq K'$.
   (c) By validity and reflexivity we have $\Gamma, \alpha{::}K' \vdash K'' \leq K''$.
   (d) By weakening, $\Gamma, \alpha{::}\mathbf{S}(A' :: K') \vdash K'' \leq K''$.
   (e) Since by part 3 we have $\Gamma, \alpha{::}\mathbf{S}(A' :: K') \vdash \alpha \equiv A' :: \mathbf{S}(A' :: K')$,
   (f) by functionality it follows that $\Gamma, \alpha{::}\mathbf{S}(A' :: K') \vdash K'' \leq [A'/\alpha]K''$.
   (g) Thus $\Gamma \vdash \Pi\alpha{::}K'.K'' \leq \mathbf{S}(A' :: K')\to([A'/\alpha]K'')$.
   (h) By subsumption $\Gamma \vdash A :: \mathbf{S}(A' :: K')\to([A'/\alpha]K'')$,
   (i) so by Rule 2.25 we have $\Gamma \vdash A\,A' :: [A'/\alpha]K''$.

   The proof for Rule 2.99 is exactly analogous.
6. (a) Assume $\Gamma \vdash \Sigma\alpha{::}K'.K''$, $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K'$, and $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$.
   (b) Then by symmetry and part 1, $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: \mathbf{S}(\pi_1 A_1 :: K')$,
   (c) so $\Gamma \vdash A :: \mathbf{S}(\pi_1 A_1 :: K')\times[A_1/\alpha]K''$.
   (d) Now $\Gamma \vdash \mathbf{S}(\pi_1 A_1 :: K') \leq K'$.
   (e) Since $\Gamma, \alpha{::}K' \vdash K''$ by inversion,
   (f) by weakening and reflexivity we have $\Gamma, \alpha{::}\mathbf{S}(\pi_1 A_1 :: K') \vdash K'' \leq K''$.
   (g) By functionality, $\Gamma, \alpha{::}\mathbf{S}(\pi_1 A_1 :: K') \vdash [\pi_1 A_1/\alpha]K'' \leq K''$.
   (h) Thus $\Gamma \vdash \mathbf{S}(\pi_1 A_1 :: K')\times[\pi_1 A_1/\alpha]K'' \leq \Sigma\alpha{::}K'.K''$.
   (i) By subsumption, $\Gamma \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K'.K''$.
7. - Case $K_1 = \mathbf{T}$ or $\mathbf{S}(A_1)$ and $K_2 = \mathbf{T}$ or $\mathbf{S}(A_2)$.
     (a) $\mathbf{S}(A_1 :: K_1) = \mathbf{S}(A_1)$,
     (b) $\mathbf{S}(A_2 :: K_2) = \mathbf{S}(A_2)$,
     (c) and the desired conclusion follows by Rule 2.12.
   - Case $K_1 = \Pi\alpha{::}K_1'.K_1''$ and $K_2 = \Pi\alpha{::}K_2'.K_2''$.
     (a) $\mathbf{S}(A_i :: K_i) = \Pi\alpha{::}K_i'.\mathbf{S}(A_i\,\alpha :: K_i'')$.

(b) By inversion $\Gamma \vdash K_2' \leq K_1'$ and $\Gamma, \alpha{::}K_2' \vdash K_1'' \leq K_2''$.

(c) Now $\Gamma, \alpha{::}K_2' \vdash A_1\,\alpha \equiv A_2\,\alpha :: K_1''$.

(d) By the inductive hypothesis, $\Gamma, \alpha{::}K_2' \vdash \mathbf{S}(A_1\,\alpha :: K_1'') \leq \mathbf{S}(A_2\,\alpha :: K_2'')$.

(e) The conclusion follows by Rule 2.14.

- Case $K_1 = \Sigma\alpha{::}K_1'.K_1''$ and $K_2 = \Sigma\alpha{::}K_2'.K_2''$.

(a) $\mathbf{S}(A_1 :: K_1) = \Sigma\alpha{::}\mathbf{S}(\pi_1 A_1 :: K_1').\mathbf{S}(\pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'')$

(b) and $\mathbf{S}(A_2 :: K_2) = \Sigma\alpha{::}\mathbf{S}(\pi_1 A_2 :: K_2').\mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2'')$.

(c) Now $\Gamma \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K_1'$

(d) and $\Gamma \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K_1''$.

(e) By the inductive hypothesis, $\Gamma \vdash \mathbf{S}(\pi_1 A_1 :: K_1') \leq \mathbf{S}(\pi_1 A_2 :: K_2')$.

(f) Since $\Gamma \vdash [\pi_1 A_1/\alpha]K_1'' \leq [\pi_1 A_2/\alpha]K_2''$,

(g) the inductive hypothesis applies, yielding
$\Gamma \vdash \mathbf{S}(\pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'') \leq \mathbf{S}(\pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2'')$. (Here it is important that the induction is on the size of K_1 and not by induction on the proof $\Gamma \vdash K_1 \leq K_2$.)

(h) The desired result follows by weakening and Rule 2.15.

∎

Proposition 3.3.4
The remaining rules from §2.3 are all admissible

Proof: By cases.

- Case: Rule 2.100.

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\alpha{::}K'.K'' \\ \Gamma \vdash A' :: K' \qquad \Gamma \vdash A'' :: [A'/\alpha]K''\end{array}}{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''}$$

1. Assume $\Gamma \vdash \Sigma\alpha{::}K'.K''$, $\Gamma \vdash A' :: K'$, and $\Gamma \vdash A'' :: [A'/\alpha]K''$.
2. Then $\Gamma \vdash A' :: \mathbf{S}(A' :: K')$,
3. so $\Gamma \vdash \langle A', A''\rangle :: \mathbf{S}(A' :: K')\times[A'/\alpha]K''$.
4. Now $\Gamma \vdash \mathbf{S}(A' :: K') \leq K'$.
5. Since $\Gamma, \alpha{::}K' \vdash K''$ by inversion,
6. by weakening and reflexivity we have $\Gamma, \alpha{::}\mathbf{S}(A' :: K') \vdash K'' \leq K''$.
7. By functionality, $\Gamma, \alpha{::}\mathbf{S}(A' :: K') \vdash [A'/\alpha]K'' \leq K''$.
8. Thus $\Gamma \vdash \mathbf{S}(A' :: K')\times[A'/\alpha]K'' \leq \Sigma\alpha{::}K'.K''$.
9. By subsumption, $\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''$.

- Case: Rule 2.101. Analogous to the proof for Rule 2.100.
- Case: Rules 2.103 and 2.104. Analogous to the proof for Rule 2.98.
- Case: Rules 2.105 and 2.106. Analogous to the proof for Rule 2.100.
- Case: Rule 2.107

$$\frac{\Gamma, \alpha{::}K' \vdash A :: K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash (\lambda\alpha{::}K'.A)\ A' \equiv [A'/\alpha]A :: [A'/\alpha]K''}$$

1. Assume $\Gamma, \alpha{::}K_2 \vdash A :: K$ and $\Gamma \vdash A_2 :: K_2$.
2. Then $\Gamma, \alpha{::}K_2 \vdash A :: \mathbf{S}(A :: K)$,
3. so $\Gamma \vdash \lambda\alpha{::}K_2.A :: \Pi\alpha{::}K_2.\mathbf{S}(A :: K)$.
4. By Rule 2.98 we have $\Gamma \vdash (\lambda\alpha{::}K_2.A)\ A_2 :: \mathbf{S}([A_2/\alpha]A :: [A_2/\alpha]K)$.
5. By substitution, $\Gamma \vdash [A_2/\alpha]A :: [A_2/\alpha]K$.
6. Thus $\Gamma \vdash (\lambda\alpha{::}K_2.A)\ A_2 \equiv [A_2/\alpha]A :: [A_2/\alpha]K$ by Rule 2.97.

- Case: Rule 2.108

$$\frac{\Gamma \vdash A_1 :: K_1 \qquad \Gamma \vdash A_2 :: K_2}{\Gamma \vdash \pi_1\langle A_1, A_2\rangle \equiv A_1 :: K_1}$$

1. Assume $\Gamma \vdash A_1 :: K_1$ and $\Gamma \vdash A_2 :: K_2$.
2. Then $\Gamma \vdash A_1 :: \mathbf{S}(A_1 :: K_1)$,
3. so $\Gamma \vdash \langle A_1, A_2\rangle :: \mathbf{S}(A_1 :: K_1)\times K_2$.
4. Thus $\Gamma \vdash \pi_1\langle A_1, A_2\rangle :: \mathbf{S}(A_1 :: K_1)$
5. and $\Gamma \vdash \pi_1\langle A_1, A_2\rangle \equiv A_1 :: K_1$.

- Case: Rules 2.109–2.111. Analogous proof to Rule 2.108.
- Case: Rule 2.112. By Rule 2.107 and functionality.
- Case: Rules 2.113–2.116. By Rules 2.108–2.111 and subsumption.
- Case: Rules 2.117–2.118. By the β-rules and extensionality.
- Case: Rules 2.120–2.121. By validity and subsumption.
- Case: Rules 2.123–2.124. By validity and subsumption.

∎

3.4 Kind Strengthening

One can drop those constructor variables in the context which are not referred to (directly or indirectly) in a judgment. This follows from the fact that every kind classifies some constructor:

Proposition 3.4.1 (Inhabitation of Kinds)
If $\Gamma \vdash K$ then there exists a constructor A such that $\Gamma \vdash A :: K$.

Proof: By induction on the size of K, and cases on the form of K.

- Case: $K = \mathbf{T}$. Pick $A = \mathsf{Int}$.
- Case: $K = \mathbf{S}(A)$. Then $\Gamma \vdash A :: \mathbf{S}(A)$.
- Case: $K = \Pi\alpha{::}K'.K''$. Then $\Gamma, \alpha{::}K' \vdash K''$ by inversion, so by the inductive hypothesis there exists A'' such that $\Gamma, \alpha{::}K' \vdash A'' :: K''$. Choose $A = \lambda\alpha{::}K'.A''$.
- Case: $K = \Sigma\alpha{::}K'.K''$. Then $\Gamma \vdash K'$ and $\Gamma, \alpha{::}K' \vdash K''$ by inversion. By the inductive hypothesis we may choose $\Gamma \vdash A' :: K'$. By substitution, $\Gamma \vdash [A'/\alpha]K''$, so inductively we may choose $\Gamma \vdash A'' :: [A'/\alpha]K''$. (It is important here that induction proceeds by the size of the kind, and that size is invariant under substitutions.) By the admissible Rule 2.100, $\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''$.

∎

Corollary 3.4.2 (Kind Strengthening)
If $\Gamma_1, \beta{::}L, \Gamma_2 \vdash \mathcal{J}$ *and* $\beta \notin FV(\Gamma_2) \cup FV(\mathcal{J})$ *then* $\Gamma_1, \Gamma_2 \vdash \mathcal{J}$.

Proof:

1. There exists a strict subderivation $\Gamma_1, \beta{::}L, \Gamma_2 \vdash \mathrm{ok}$, which itself contains a subderivation $\Gamma_1 \vdash L$.
2. By Proposition 3.4.1 there exists $\Gamma_1 \vdash B :: L$.
3. By Proposition 3.1.11 we have $\Gamma_1, [B/\beta]\Gamma_2 \vdash [B/\beta]\mathcal{J}$
4. But since β is not free in Γ_2 or $\mathcal{J}$, this judgment is exactly $\Gamma_1, \Gamma_2 \vdash \mathcal{J}$.

∎

This proof strategy is not applicable for dropping unused term variables in the context; in general one does not expect every *type* to be inhabited by values. Therefore the corresponding proof of strengthening for term variables is delayed until §7.4.

Chapter 4

Algorithms for Kind and Constructor Judgments

4.1 Introduction

In this chapter I present algorithms for checking instances of the kind and constructor-level judgments. For each such algorithm, proving correctness requires showing that three properties hold.

- Soundness: if the algorithm verifies the judgment then the corresponding MIL_0 judgment is provable.
- Completeness: if a MIL_0 judgment is provable then the algorithm will verify the judgment.
- Termination: the algorithm always either verifies or rejects a judgment. (That is, the judgment is decidable.)

In this chapter I show soundness for all of the algorithms, but most completeness and termination results are postponed until the next chapter.

4.2 Principal Kinds

Checking the validity of type constructors is simplified by the existence of *principal kinds*. A principal kind of a constructor (with respect to a given typing context) is a most-specific kind of

$$
\begin{array}{ll}
\Gamma \triangleright b_i \Uparrow \mathbf{S}(b_i) & \\
\Gamma \triangleright \alpha \Uparrow \mathbf{S}(\alpha :: \Gamma(\alpha)) & \\
\Gamma \triangleright \times \Uparrow \mathbf{S}(\times :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}) & \\
\Gamma \triangleright \to \Uparrow \mathbf{S}(\to :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}) & \\
\Gamma \triangleright \lambda\alpha{::}K'.A \Uparrow \Pi\alpha{::}K'.K'' & \text{if } \Gamma, \alpha : K' \triangleright A \Uparrow K'' \\
\Gamma \triangleright A\,A' \Uparrow [A'/\alpha]K'' & \text{if } \Gamma \triangleright A \Uparrow \Pi\alpha{::}K'.K'' \\
\Gamma \triangleright \langle A', A'' \rangle \Uparrow K' {\times} K'' & \text{if } \Gamma \triangleright A' \Uparrow K' \text{ and } \Gamma \triangleright A'' \Uparrow K''. \\
\Gamma \triangleright \pi_1 A \Uparrow K' & \text{if } \Gamma \triangleright A \Uparrow \Sigma\alpha{::}K'.K'' \\
\Gamma \triangleright \pi_2 A \Uparrow [\pi_1 A/\alpha]K'' & \text{if } \Gamma \triangleright A \Uparrow \Sigma\alpha{::}K'.K''
\end{array}
$$

Figure 4.1: Algorithm for Principal Kind Synthesis

that constructor. Formally, K is principal for A in Γ if and only if $\Gamma \vdash A :: K$ and whenever $\Gamma \vdash A :: L$ we have $\Gamma \vdash K \leq L$. When they exist, principal kinds are unique up to provable equivalence.

I show that every well-kinded constructor has a principal kind by giving a correct algorithm for explicitly calculating it; see Figure 4.1. This algorithm, like all of the algorithms I will present, is organized as a collection of "algorithmic" inference rules. The rules have been carefully designed so that a derivation $\Gamma \triangleright A \Uparrow K$ corresponds exactly to a run of the principal kind computation algorithm which takes Γ and A as inputs and produces the principal kind K as the result. To this end, the inference rules are deterministic: given Γ and A, there is at most one kind K such that $\Gamma \triangleright A \Uparrow K$. Furthermore, there is at most one rule which could possibly be used to produce such a K — there is exactly one inference rule for each syntactic form that A might have. Thus given Γ and A, a "proof search" for K such that $\Gamma \triangleright A \Uparrow K$ corresponds to a direct calculation of the principal kind.

For example, in the empty typing context the principal kind of $\lambda\alpha{::}\mathbf{T}.\lambda\beta{::}\mathbf{T}.\langle\alpha,\beta\rangle$ is computed as follows:

$$\begin{array}{l}
\triangleright \lambda\alpha{::}\mathbf{T}.\lambda\beta{::}\mathbf{T}.\langle\alpha,\beta\rangle \Uparrow \Pi\alpha{::}\mathbf{T}.\Pi\beta{::}\mathbf{T}.\mathbf{S}(\alpha)\times\mathbf{S}(\beta) \\
\quad\quad \text{because} \quad \alpha{::}\mathbf{T} \triangleright \lambda\beta{::}\mathbf{T}.\langle\alpha,\beta\rangle \Uparrow \Pi\beta{::}\mathbf{T}.\mathbf{S}(\alpha)\times\mathbf{S}(\beta) \\
\quad\quad\quad\quad \text{because} \quad \alpha{::}\mathbf{T}, \beta{::}\mathbf{T} \triangleright \langle\alpha,\beta\rangle \Uparrow \mathbf{S}(\alpha)\times\mathbf{S}(\beta) \\
\quad\quad\quad\quad\quad\quad \text{because} \quad \alpha{::}\mathbf{T}, \beta{::}\mathbf{T} \triangleright \alpha \Uparrow \mathbf{S}(\alpha) \text{ and } \alpha{::}\mathbf{T}, \beta{::}\mathbf{T} \triangleright \beta \Uparrow \mathbf{S}(\beta)
\end{array}$$

The principal type synthesis algorithm is correct, as shown by the following theorem; note that K is independent of L and hence is principal.

Theorem 4.2.1 (Principal Kinds)
If $\Gamma \vdash A :: L$ then there exists K such that $\Gamma \triangleright A \Uparrow K$ and $\Gamma \vdash A :: K$ and $\Gamma \vdash K \leq \mathbf{S}(A :: L)$, so that $\Gamma \vdash K \leq L$.

Proof: By induction on the proof of the assumption and cases on the last rule used.

- Case: Rule 2.20.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash b :: \mathbf{T}}$$

  1. $\Gamma \triangleright b \Uparrow \mathbf{S}(b)$ and $\Gamma \vdash b :: \mathbf{S}(b)$.
  2. $\mathbf{S}(b :: \mathbf{T}) = \mathbf{S}(b)$.
  3. $\Gamma \vdash b \equiv b :: \mathbf{T}$, so $\Gamma \vdash \mathbf{S}(b) \leq \mathbf{S}(b)$.

- Case: Rule 2.23.

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

  1. $\Gamma \triangleright \alpha \Uparrow \mathbf{S}(\alpha :: \Gamma(\alpha))$.
  2. By Rules 2.92 and 2.93, $\Gamma \vdash \mathbf{S}(\alpha :: \Gamma(\alpha))$ and $\Gamma \vdash \alpha :: \mathbf{S}(\alpha :: \Gamma(\alpha))$.
  3. By reflexivity, $\Gamma \vdash \mathbf{S}(\alpha :: \Gamma(\alpha)) \leq \mathbf{S}(\alpha :: \Gamma(\alpha))$.

- Case: Rule 2.24.

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

1. By the inductive hypothesis $\Gamma, \alpha{::}K' \rhd A \Uparrow K''$,
2. $\Gamma, \alpha{::}K' \vdash A :: K''$,
3. and $\Gamma, \alpha{::}K' \vdash K'' \leq \mathbf{S}(A :: L'')$.
4. Then $\Gamma \rhd \lambda\alpha{::}K'.A \Uparrow \Pi\alpha{::}K'.K''$
5. and $\Gamma \vdash (\lambda\alpha{::}K'.A) :: (\Pi\alpha{::}K'.K'')$.
6. Now $\Gamma, \alpha{::}K' \vdash (\lambda\alpha{::}K'.A)\,\alpha \equiv A :: L''$ by weakening and Rule 2.107,
7. so $\Gamma, \alpha{::}K' \vdash \mathbf{S}(A :: L'') \leq \mathbf{S}((\lambda\alpha{::}K'.A)\,\alpha :: L'')$ by Rule 2.95.
8. Since $\mathbf{S}(\lambda\alpha{::}K'.A :: \Pi\alpha{::}K'.L'') = \Pi\alpha{::}K'.\mathbf{S}((\lambda\alpha{::}K'.A)\,\alpha :: L'')$
9. and $\Gamma \vdash K' \leq K'$,
10. we have $\Gamma \vdash \Pi\alpha{::}K'.K'' \leq \mathbf{S}(\lambda\alpha{::}K'.A :: \Pi\alpha{::}K'.L'')$.

- Case: Rule 2.25.

$$\frac{\Gamma \vdash A :: L'{\to}L'' \qquad \Gamma \vdash A' :: L'}{\Gamma \vdash A\,A' :: L''}$$

1. By the inductive hypothesis $\Gamma \rhd A \Uparrow K$
2. $\Gamma \vdash A :: K$
3. and $\Gamma \vdash K \leq \mathbf{S}(A :: L'{\to}L'')$.
4. Now $\mathbf{S}(A :: L'{\to}L'') = \Pi\alpha{::}L'.\mathbf{S}(A\,\alpha :: L'')$ where $\alpha \notin \mathrm{FV}(A) \cup \mathrm{FV}(L'')$.
5. By inversion of subkinding, $K = \Pi\alpha{::}K'.K''$,
6. $\Gamma \vdash L' \leq K'$,
7. and $\Gamma, \alpha{::}L' \vdash K'' \leq \mathbf{S}(A\,\alpha :: L'')$.
8. Then $\Gamma \rhd A\,A' \Uparrow [A'/\alpha]K''$.
9. By subsumption, $\Gamma \vdash A' :: K'$, so
10. $\Gamma \vdash A\,A' :: [A'/\alpha]K''$.
11. Finally, by Lemma 3.3.2 and Proposition 3.1.11 applied to line 7 we have $\Gamma \vdash [A'/\alpha]K'' \leq \mathbf{S}(A\,A' :: L'')$.

- Case: Rule 2.27

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}L'.L''}{\Gamma \vdash \pi_1 A :: L'}$$

1. By the inductive hypothesis, $\Gamma \rhd A \Uparrow K$,
2. $\Gamma \vdash A :: K$,
3. and $\Gamma \vdash K \leq \mathbf{S}(A :: \Sigma\alpha{::}L'.L'')$.
4. Now $\mathbf{S}(A :: \Sigma\alpha{::}L'.L'') = \mathbf{S}(\pi_1 A :: L') {\times} \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$.
5. By inversion of subkinding, $K = \Sigma\alpha{::}K'.K''$,
6. and $\Gamma \vdash K' \leq \mathbf{S}(\pi_1 A :: L')$.
7. Finally, $\Gamma \rhd \pi_1 A \Uparrow K'$
8. and $\Gamma \vdash \pi_1 A :: K'$.

- Case: Rule 2.28

$$\frac{\Gamma \vdash A :: \Sigma\alpha{::}L'.L''}{\Gamma \vdash \pi_2 A :: [\pi_1 A/\alpha]L''}$$

1. By the inductive hypothesis, $\Gamma \triangleright A \Uparrow K$,
2. $\Gamma \vdash A :: K$,
3. and $\Gamma \vdash K \leq \mathbf{S}(A :: \Sigma\alpha{::}L'.L'')$.
4. Now $\mathbf{S}(A :: \Sigma\alpha{::}L'.L'') = \mathbf{S}(\pi_1 A :: L')\times\mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$.
5. By inversion of subkinding, $K = \Sigma\alpha{::}K'.K''$,
6. $\Gamma \vdash K' \leq \mathbf{S}(\pi_1 A :: L')$,
7. and $\Gamma, \alpha{::}K' \vdash K'' \leq \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$.
8. Then $\Gamma \vdash \pi_1 A :: K'$.
9. so by Proposition 3.1.11 applied to line 7, $\Gamma \vdash [\pi_1 A/\alpha]K'' \leq \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$.
10. Finally, $\Gamma \triangleright \pi_2 A \Uparrow [\pi_1 A/\alpha]K''$
11. and $\Gamma \vdash \pi_2 A :: [\pi_1 A/\alpha]K''$.

- Case: Rule 2.26

$$\frac{\Gamma \vdash A' :: L' \qquad \Gamma \vdash A'' :: L''}{\Gamma \vdash \langle A', A''\rangle :: L'\times L''}$$

1. By the inductive hypothesis, $\Gamma \triangleright A' \Uparrow K'$,
2. $\Gamma \vdash A' :: K'$,
3. $\Gamma \vdash K' \leq \mathbf{S}(A' :: L')$,
4. $\Gamma \triangleright A'' \Uparrow K''$,
5. $\Gamma \vdash A'' :: K''$,
6. and $\Gamma \vdash K'' \leq \mathbf{S}(A'' :: L'')$.
7. Then $\Gamma \triangleright \langle A', A''\rangle \Uparrow K'\times K''$,
8. and $\Gamma \vdash \langle A', A''\rangle :: K'\times K''$.
9. Now $\mathbf{S}(\langle A', A''\rangle :: L'\times L'') = \mathbf{S}(\pi_1\langle A', A''\rangle :: L')\times\mathbf{S}(\pi_2\langle A', A''\rangle :: L'')$.
10. By Rule 2.95, $\Gamma \vdash \mathbf{S}(A' :: L') \leq \mathbf{S}(\pi_1\langle A', A''\rangle :: L')$
11. and $\Gamma \vdash \mathbf{S}(A'' :: L'') \leq \mathbf{S}(\pi_2\langle A', A''\rangle :: L'')$.
12. Therefore, $\Gamma \vdash K'\times K'' \leq \mathbf{S}(\langle A', A''\rangle :: L'\times L'')$.

- Case: Rule 2.29

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

By the inductive hypothesis, noting that $\mathbf{S}(A :: \mathbf{S}(A)) = \mathbf{S}(A)$.

- Case: Rule 2.31

$$\frac{\Gamma, \alpha{::}K' \vdash A\,\alpha :: K'' \qquad \Gamma \vdash A :: \Pi\alpha{::}L'.L'' \qquad \Gamma \vdash K' \equiv L'}{\Gamma \vdash A :: \Pi\alpha{::}K'.K''}$$

1. By the inductive hypothesis, $\Gamma \triangleright A \Uparrow K$,
2. $\Gamma \vdash A :: K$,
3. and $\Gamma \vdash K \leq \mathbf{S}(A :: \Pi\alpha{::}L_1'.L_1'')$.
4. Now $\mathbf{S}(A :: \Pi\alpha{::}L_1'.L_1'') = \Pi\alpha{::}L'.\mathbf{S}(A\,\alpha :: L_1'')$
5. so by inversion $K = \Pi\alpha{::}K'.K''$
6. and $\Gamma \vdash L_1' \leq K'$.
7. Since $\Gamma \vdash L' \equiv L_1'$, we have $\Gamma \vdash L' \leq L_1'$ and hence $\Gamma \vdash L' \leq K'$.
8. Also by the inductive hypothesis, $\Gamma, \alpha{::}L' \triangleright A\,\alpha \Uparrow K_2''$,
9. $\Gamma, \alpha{::}L' \vdash A\,\alpha :: K_2''$,
10. and $\Gamma, \alpha{::}L' \vdash K_2'' \leq \mathbf{S}(A\,\alpha :: L'')$.
11. But since the principal kind synthesis algorithm is deterministic and clearly obeys weakening, we have $K_2'' = [\alpha/\alpha]K'' = K''$.
12. Now $\mathbf{S}(A :: \Pi\alpha{::}L'.L'') = \Pi\alpha{::}L'.\mathbf{S}(A\,\alpha :: L'')$.
13. Therefore $\Gamma \vdash \Pi\alpha{::}K'.K'' \leq \mathbf{S}(A :: \Pi\alpha{::}L'.L'')$.

- Case: Rule 2.30.

$$\frac{\Gamma \vdash \pi_1 A :: L' \qquad \Gamma \vdash \pi_2 A :: L''}{\Gamma \vdash A :: L' {\times} L''}$$

1. There is a subderivation $\Gamma \vdash A :: K_1$ for some kind K_1 (see Proposition 4.4.1 below).
2. By the inductive hypothesis, $\Gamma \triangleright \pi_1 A \Uparrow K'$,
3. $\Gamma \vdash \pi_1 A :: K'$,
4. and $\Gamma \vdash K' \leq \mathbf{S}(\pi_1 A :: L')$.
5. Also, $\Gamma \triangleright \pi_2 A \Uparrow K''$,
6. $\Gamma \vdash \pi_2 A :: K''$,
7. and $\Gamma \vdash K'' \leq \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$.
8. Principal kind synthesis never returns a dependent Σ type, so for kind synthesis for $\pi_1 A$ and $\pi_2 A$ to have succeeded it must be that $\Gamma \triangleright A \Uparrow K' {\times} K''$.
9. By the inductive hypothesis, $\Gamma \vdash A :: K' {\times} K''$.
10. Since $\mathbf{S}(A :: \Sigma\alpha{::}L'.L'') = \mathbf{S}(\pi_1 A :: L') {\times} \mathbf{S}(\pi_2 A :: [\pi_1 A/\alpha]L'')$,
11. $\Gamma \vdash K' {\times} K'' \leq \mathbf{S}(A :: \Sigma\alpha{::}L'.L'')$.
12. so by the inductive hypothesis $\Gamma \vdash A :: K$.

- Rule 2.32

$$\frac{\Gamma \vdash A :: L_2 \qquad \Gamma \vdash L_2 \leq L}{\Gamma \vdash A :: L}$$

The desired result follows from the inductive hypothesis and by Rule 2.95 to get $\Gamma \vdash \mathbf{S}(A :: L_2) \leq \mathbf{S}(A :: L)$.

∎

Kind validity

$\Gamma \rhd \mathbf{T}$	
$\Gamma \rhd \mathbf{S}(A)$	if $\Gamma \rhd A \leftleftarrows \mathbf{T}$
$\Gamma \rhd \Pi\alpha{::}K'.K''$	if $\Gamma \rhd K'$ and $\Gamma, \alpha{::}K' \rhd K''$.
$\Gamma \rhd \Sigma\alpha{::}K'.K''$	if $\Gamma \rhd K'$ and $\Gamma, \alpha{::}K' \rhd K''$.

Subkinding

$\Gamma \rhd \mathbf{T} \leq \mathbf{T}$	always
$\Gamma \rhd \mathbf{S}(A) \leq \mathbf{T}$	always
$\Gamma \rhd \mathbf{S}(A_1) \leq \mathbf{S}(A_2)$	if $\Gamma \rhd A_1 \Leftrightarrow A_2 :: \mathbf{T}$.
$\Gamma \rhd \Pi\alpha{::}K_1'.K_1'' \leq \Pi\alpha{::}K_2'.K_2''$	if $\Gamma \rhd K_2' \leq K_1'$ and $\Gamma, \alpha{::}K_2' \rhd K_1'' \leq K_2''$.
$\Gamma \rhd \Sigma\alpha{::}K_1'.K_1'' \leq \Sigma\alpha{::}K_2'.K_2''$	if $\Gamma \rhd K_1' \leq K_2'$ and $\Gamma, \alpha{::}K_1' \rhd K_1'' \leq K_2''$.

Kind equivalence

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

Figure 4.2: Algorithms for Kinds

4.3 Algorithms for Kind and Constructor Judgments

Figure 4.2 gives algorithms for determining kind validity, subkinding, and kind equivalence. Each is specified as a deterministic set of inference rules. The symbol $\rhd$ is used instead of $\vdash$ to distinguish these as algorithmic judgments.

The *kind validity* judgment

$$\Gamma \rhd K$$

models the declarative kind validity judgment $\Gamma \vdash K$. Viewed as an algorithm this takes a well-formed context Γ and a kind K and determines whether there is a proof of $\Gamma \vdash K$. For any conclusion, at most one rule could apply; there is one rule for each syntactic form that K might have.

The *subkinding* judgment

$$\Gamma \rhd K_1 \leq K_2$$

models the declarative subkinding judgment $\Gamma \vdash K_1 \leq K_2$. As an algorithm, given kinds satisfying $\Gamma \vdash K_1$ and $\Gamma \vdash K_2$ it determines whether there is a proof $\Gamma \vdash K_1 \leq K_2$.

Similarly, the *kind equivalence* judgment

$$\Gamma \rhd K_1 \Leftrightarrow K_2$$

models declarative equivalence; given two kinds satisfying $\Gamma \vdash K_1$ and $\Gamma \vdash K_2$ it determines whether there is a proof $\Gamma \vdash K_1 \equiv K_2$.

Figure 4.3 shows the algorithms for determining the well-formedness of constructors. The *kind synthesis* judgment

$$\Gamma \rhd A \rightrightarrows K$$

Kind synthesis

$\Gamma \triangleright \mathsf{Int} \rightrightarrows \mathbf{S}(\mathsf{Int})$	
$\Gamma \triangleright \times \rightrightarrows \mathbf{S}(\times :: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T})$	
$\Gamma \triangleright \to \rightrightarrows \mathbf{S}(\to:: \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T})$	
$\Gamma \triangleright \alpha \rightrightarrows \mathbf{S}(\alpha :: \Gamma(\alpha))$	if $\alpha \in \mathrm{dom}(\Gamma)$.
$\Gamma \triangleright \lambda\alpha{::}K'.A \rightrightarrows \Pi\alpha{::}K'.K''$	if $\Gamma \triangleright K'$ and $\Gamma, \alpha{::}K' \triangleright A \rightrightarrows K''$.
$\Gamma \triangleright A\,A' \rightrightarrows [A'/\alpha]K''$	if $\Gamma \triangleright A \rightrightarrows \Pi\alpha{::}K'.K''$ and $\Gamma \triangleright A \leftleftarrows K'$.
$\Gamma \triangleright \langle A', A''\rangle \rightrightarrows K'{\times}K''$	if $\Gamma \triangleright A' \rightrightarrows K'$ and $\Gamma \triangleright A'' \rightrightarrows K''$.
$\Gamma \triangleright \pi_1 A \rightrightarrows K'$	if $\Gamma \triangleright A \rightrightarrows \Sigma\alpha{::}K'.K''$
$\Gamma \triangleright \pi_2 A \rightrightarrows [\pi_1 A/\alpha]K''$	if $\Gamma \triangleright A \rightrightarrows \Sigma\alpha{::}K'.K''$

Kind checking

$\Gamma \triangleright A \leftleftarrows K$	if $\Gamma \triangleright A \rightrightarrows L$ and $\Gamma \triangleright L \leq K$.

Figure 4.3: Algorithms for Constructor Validity

combines constructor validity checking with principal kind synthesis. As an algorithm, given a well-formed context Γ and a constructor A it returns a principal kind K of A if A is well-formed (i.e., if it can be given any kind at all) and fails otherwise.

Because all well-formed constructors have principal kinds, it is easy to define a *kind checking* judgment

$$\Gamma \triangleright A \leftleftarrows K.$$

which directly models the constructor validity checking. Given a context and kind satisfying $\Gamma \vdash K$ and constructor A, this algorithm determines whether $\Gamma \vdash A :: K$ holds.

The judgments involved in constructor equivalence are shown in Figure 4.4. Following Coquand [Coq91] equivalence is determined in a direct fashion rather than by independently normalizing the two constructors and comparing normal forms (but see §5.5).

My algorithm is more involved than Coquand's because of the context and kind-dependence of equivalence. The algorithmic constructor equivalence rules are divided into a kind-directed part and a structure-directed part, while Coquand needs only structural comparison. Weak head normalization is extended to include looking for definitions in the context. I have also extended the algorithm in a natural fashion to handle Σ kinds, pairing, and projection.

The algorithm uses the notion of an *elimination context*; this is a series of applications to and projections from "$\diamond$", which is called the context's hole. If $\mathcal{E}$ is such a context, then $\mathcal{E}[A]$ represents the constructor resulting by replacing the hole in $\mathcal{E}$ with A. If a constructor is either of the form $\mathcal{E}[\alpha]$ or of the form $\mathcal{E}[c]$ then this will be called a *path* and denoted by p. (Recall that c ranges over constant type constructors.)

$$\begin{array}{rcl} \mathcal{E} & ::= & \diamond \\ & | & \mathcal{E}\ A \\ & | & \pi_1\mathcal{E} \\ & | & \pi_2\mathcal{E} \end{array}$$

The *kind extraction* relation is written

$$\Gamma \triangleright p \uparrow K.$$

Kind Extraction

$\Gamma \triangleright b \uparrow \mathbf{T}$	
$\Gamma \triangleright \times \uparrow \mathbf{T} \to (\mathbf{T} \to \mathbf{T})$	
$\Gamma \triangleright \rightharpoonup \uparrow \mathbf{T} \to (\mathbf{T} \to \mathbf{T})$	
$\Gamma \triangleright \alpha \uparrow \Gamma(\alpha)$	
$\Gamma \triangleright \pi_1 p \uparrow K'$	if $\Gamma \triangleright p \uparrow \Sigma\beta{::}K'.K''$
$\Gamma \triangleright \pi_2 p \uparrow [\pi_1 p/\beta]K''$	if $\Gamma \triangleright p \uparrow \Sigma\beta{::}K'.K''$
$\Gamma \triangleright p\, A \uparrow [A/\beta]K''$	if $\Gamma \triangleright p \uparrow \Pi\beta{::}K'.K''$

Weak head reduction

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

Weak head normalization

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

Algorithmic constructor equivalence

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

Algorithmic path equivalence

$\Gamma \triangleright b \leftrightarrow b \uparrow \mathbf{T}$	
$\Gamma \triangleright \times \leftrightarrow \times \uparrow \mathbf{T} \to (\mathbf{T} \to \mathbf{T})$	
$\Gamma \triangleright \rightharpoonup \leftrightarrow \rightharpoonup \uparrow \mathbf{T} \to (\mathbf{T} \to \mathbf{T})$	
$\Gamma \triangleright \alpha \leftrightarrow \alpha \uparrow \Gamma(\alpha)$	
$\Gamma \triangleright p_1\, A_1 \leftrightarrow p_2\, A_2 \uparrow [A_1/\alpha]K''$	if $\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow \Pi\alpha{::}K'.K''$ and $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K'$
$\Gamma \triangleright \pi_1 p_1 \leftrightarrow \pi_1 p_2 \uparrow K'$	if $\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow \Sigma\alpha{::}K'.K''$
$\Gamma \triangleright \pi_2 p_1 \leftrightarrow \pi_2 p_2 \uparrow [\pi_1 p_1/\alpha]K''$	if $\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow \Sigma\alpha{::}K'.K''$

Figure 4.4: Kind and Constructor Equivalence Algorithms

Given a well-formed context Γ and p which is well-formed in this context, kind extraction 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 $\mathbf{S}(B)$; in this case B is said to be the definition of the path.

The extracted kind is not always the most precise kind. For example, $\alpha{::}\mathbf{T} \triangleright \alpha \uparrow \mathbf{T}$ but the principal kind of α in this context would be $\mathbf{S}(\alpha)$. Intuitively the extracted kind is the most precise kind which can be assigned without the singleton introduction rule, or Rules 2.30 and 2.31 which can be viewed as extending singleton introduction to higher kinds. This suffices to make $\mathbf{S}(p :: K)$ principal for p if K is its extracted kind.

The *weak head reduction* relation

$$\Gamma \triangleright A \leadsto B$$

takes Γ and A and returns the result of applying one step of head β-reduction if A has such a redex. If the head of A is a path with a definition reduction then the definition is returned. Otherwise, there is no weak head reduct.

The *weak head normalization* relation

$$\Gamma \triangleright A \Downarrow B$$

takes Γ and A and 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

$$\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$$

models the declarative judgment $\Gamma \vdash A_1 \equiv A_2 :: K$ on well-formed constructors. As an algorithm this is defined by induction/recursion on the kind at which the two constructors are being compared. At Π and Σ kinds the algorithm uses extensionality to reduce the problem to comparisons of constructors at kinds whose size is strictly smaller. When comparing two constructors at a singleton kind the algorithm can immediately report success because we only care about inputs where $\Gamma \vdash A_1 :: K$ and $\Gamma \vdash A_2 :: K$; if $K = \mathbf{S}(B)$ then $A_1 \equiv B \equiv A_2$ automatically. Finally, if we are comparing two constructors of kind $\mathbf{T}$ then the algorithm must do some real work. This consists of head-normalizing the two constructors, which (if the process terminates) yields two paths without definitions. Then the paths are compared component-wise.

This component-wise comparison is specified by the algorithmic path equivalence relation

$$\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow K.$$

Given two well-formed *head-normal* paths $\Gamma \vdash p_1 :: K_1$ and $\Gamma \vdash p_2 :: K_2$, this should succeed yielding K if and only if $\Gamma \vdash p_1 \equiv p_2 :: K$ and K is the extracted kind of p_1 with respect to Γ. The only question that arises when writing down these rules is in the case for comparing two applications. If the two function parts are recursively found to be equal, the two arguments must then be compared. Since the two arguments need not be in normal form, they must be compared using the $\Leftrightarrow$ judgment; in this case we must decide at which *kind* the two arguments should be compared.

The right answer is the domain kind of the extracted kind of the function parts, which (by Lemma 4.4.2) below is the same as the domain kind of the principal kind of the function parts. Assume we want to compare $p_1\,A_1$ and $p_2\,A_2$ using the typing context Γ, and that the principal kind of p_1 (and p_2, since they have been verified equivalent) is $\Pi\alpha{::}K'.K''$. Then this is the *least* kind at which the two paths are provably equal, and hence by contravariance the domain kind is *greatest*. By comparing A_1 and A_2 at kind K', then, we have the best chance of proving them equal. (Two constructors equivalent at a subtype will be equivalent at a supertype, but not vice versa.) Thus to find as many equivalences as possible K' is intuitively the correct kind for the algorithm to compare function arguments. Since the extracted kind agrees with the principal kind in negative positions, and it suffices to look at the domain of the extracted function kind rather than computing the full principal kind.

As an example, let $\Gamma = \beta{::}(\mathbf{S}(\mathsf{Int})\to\mathbf{T})\to\mathbf{T}$. Then:

$$
\begin{array}{l}
\Gamma \rhd \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \Leftrightarrow \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) :: \mathbf{T} \\
\quad \textbf{because} \quad \Gamma \rhd \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \Downarrow \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \\
\quad \textbf{and} \quad \Gamma \rhd \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) \Downarrow \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) \\
\quad \textbf{and} \quad \Gamma \rhd \beta\,(\lambda\alpha{::}\mathbf{T}.\alpha) \leftrightarrow \beta\,(\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) \uparrow \mathbf{T} \\
\qquad \textbf{because} \quad \Gamma \rhd \beta \leftrightarrow \beta \uparrow (\mathbf{S}(\mathsf{Int})\to\mathbf{T})\to\mathbf{T} \\
\qquad \textbf{and} \quad \Gamma \rhd (\lambda\alpha{::}\mathbf{T}.\alpha) \Leftrightarrow (\lambda\alpha{::}\mathbf{T}.\mathsf{Int}) :: \boxed{\mathbf{S}(\mathsf{Int})\to\mathbf{T}} \\
\qquad\quad \textbf{because} \quad \Gamma, \alpha{::}\mathbf{S}(\mathsf{Int}) \rhd (\lambda\alpha{::}\mathbf{T}.\alpha)\,\alpha \Leftrightarrow (\lambda\alpha{::}\mathbf{T}.\mathsf{Int})\,\alpha :: \mathbf{T} \\
\qquad\qquad \textbf{because} \quad \Gamma, \alpha{::}\mathbf{S}(\mathsf{Int}) \rhd (\lambda\alpha{::}\mathbf{T}.\alpha)\,\alpha \Downarrow \mathsf{Int} \\
\qquad\qquad \textbf{and} \quad \Gamma, \alpha{::}\mathbf{S}(\mathsf{Int}) \rhd (\lambda\alpha{::}\mathbf{T}.\mathsf{Int})\,\alpha \Downarrow \mathsf{Int} \\
\qquad\qquad \textbf{and} \quad \Gamma, \alpha{::}\mathbf{S}(\mathsf{Int}) \rhd \mathsf{Int} \leftrightarrow \mathsf{Int} \uparrow \mathbf{T}.
\end{array}
$$

4.4 Soundness of the Algorithmic Judgments

In order to show soundness of the constructor equivalence algorithm I first 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. (Compare the statement of Theorem 4.2.1 above and of Lemma 4.4.2 below.)

Proposition 4.4.1
If $\Gamma \vdash \mathcal{E}[A] :: L$ then there is a subderivation of the form $\Gamma \vdash A :: K$.

Proof: By induction on the kinding derivation. If $\mathcal{E} = \diamond$ then the result follows trivially; otherwise, the result follows by the inductive hypothesis. ∎

Lemma 4.4.2
If $\Gamma \vdash p :: K$ then there exists L such that $\Gamma \rhd p \uparrow L$, $\Gamma \vdash p :: L$, and $\Gamma \vdash \mathbf{S}(p :: L) \leq K$.

Proof: By induction on the proof of the hypothesis.

- Case: Rule 2.20. $p = b$.
  1. Then $\Gamma \rhd b \uparrow \mathbf{T}$ and $\mathbf{S}(b :: \mathbf{T}) = \mathbf{S}(b)$.
  2. By Rule 2.20, $\Gamma \vdash b :: \mathbf{T}$
  3. and by Rule 2.11, $\Gamma \vdash \mathbf{S}(b) \leq \mathbf{T}$.
- Case: Rules 2.21 and 2.22. Similar to previous case, using admissible rule 2.94.
- Case: Rule 2.23. $p = \alpha$.

1. Then $\Gamma \rhd \alpha \uparrow \Gamma(\alpha)$.
2. By Rule 2.23 $\Gamma \vdash \alpha :: \Gamma(\alpha)$,
3. and by Rule 2.94, $\Gamma \vdash \mathbf{S}(\alpha :: \Gamma(\alpha)) \leq \Gamma(\alpha)$.

- Case: Rule 2.25.

$$\frac{\Gamma \vdash p :: K' \to K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash p\, A' :: K''}$$

1. By the inductive hypothesis, $\Gamma \rhd p \uparrow \Pi\alpha{::}L'.L''$,
2. $\Gamma \vdash p :: \Pi\alpha{::}L'.L''$, and
3. $\Gamma \vdash \mathbf{S}(p :: \Pi\alpha{::}L'.L'') \leq K' \to K''$.
4. Then $\Gamma \rhd p\, A' \uparrow [A'/\alpha]L''$.
5. Since $\mathbf{S}(p :: \Pi\alpha{::}L'.L'') = \Pi\alpha{::}L'.\mathbf{S}(p\,\alpha :: L'')$,
6. we have by inversion of Rule 2.14 that $\Gamma \vdash K' \leq L'$ and $\Gamma, \alpha{::}K' \vdash \mathbf{S}(p\,\alpha :: L'') \leq K''$ where $\alpha \notin \mathrm{FV}(K'')$ and $\alpha \notin \mathrm{dom}(\Gamma)$.
7. By subsumption, $\Gamma \vdash A' :: L'$
8. and hence $\Gamma \vdash p\, A' :: [A'/\alpha]L''$ by Rule 2.98.
9. Finally, by substitution we have $\Gamma \vdash \mathbf{S}(p\, A' :: [A'/\alpha]L'') \leq K''$.

- Case: Rule 2.27.

$$\frac{\Gamma \vdash p :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_1 p :: K'}$$

1. By the inductive hypothesis, $\Gamma \rhd p \uparrow L$,
2. $\Gamma \vdash p :: L$, and
3. $\Gamma \vdash \mathbf{S}(p :: L) \leq \Sigma\alpha{::}K'.K'$.
4. By inversion $\mathbf{S}(p :: L)$ must be a Σ kind, and so $L' = \Sigma\alpha{::}L'.L''$ for some L' and L''.
5. Then $\Gamma \rhd \pi_1 p \uparrow L'$,
6. and by Rule 2.27, $\Gamma \vdash \pi_1 p :: L'$.
7. Since $\mathbf{S}(p :: \Sigma\alpha{::}L'.L'') = \mathbf{S}(\pi_1 p :: L') \times \mathbf{S}(\pi_2 p :: [\pi_1 p/\alpha]L'')$,
8. by inversion of rule 2.15 we have $\Gamma \vdash \mathbf{S}(\pi_1 p :: L') \leq K'$.

- Case: Rule 2.28.

$$\frac{\Gamma \vdash p :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2 p :: [\pi_1 p/\alpha]K'}$$

1. As in the previous case, $\Gamma \rhd p \uparrow \Sigma\alpha{::}L'.L''$,
2. $\Gamma \vdash p :: \Sigma\alpha{::}L'.L''$, and
3. $\Gamma \vdash \mathbf{S}(p :: \Sigma\alpha{::}L'.L'') \leq \Sigma\alpha{::}K'.K''$.
4. Then $\Gamma \rhd \pi_2 p \uparrow [\pi_1 p/\alpha]L''$,
5. and $\Gamma \vdash \pi_2 p :: [\pi_1 p/\alpha]L''$ by Rule 2.28.
6. Since $\mathbf{S}(p :: \Sigma\alpha{::}L'.L'') = \mathbf{S}(\pi_1 p :: L') \times \mathbf{S}(\pi_2 p :: [\pi_1 p/\alpha]L'')$,
7. by inversion of Rule 2.15 $\Gamma, \alpha{::}\mathbf{S}(\pi_1 p :: L') \vdash \mathbf{S}(\pi_2 p :: [\pi_1 p/\alpha]L'') \leq K''$.
8. Then $\Gamma \vdash \pi_1 p :: \mathbf{S}(\pi_1 p' :: L')$

9. so by Proposition 3.1.11 we have $\Gamma \vdash \mathbf{S}(\pi_2 p :: [\pi_1 p/\alpha]L'') \leq [\pi_1 p/\alpha]K''$.

- Case: Rule 2.29
$$\frac{\Gamma \vdash p :: \mathbf{T}}{\Gamma \vdash p :: \mathbf{S}(p)}$$
  1. By the inductive hypothesis, $\Gamma \rhd p \uparrow L$,
  2. $\Gamma \vdash p :: L$,
  3. and $\Gamma \vdash \mathbf{S}(p :: L) \leq \mathbf{T}$.
  4. Thus L is either $\mathbf{T}$ or a singleton, and $\mathbf{S}(p :: L) = \mathbf{S}(p)$.
  5. and by reflexivity, $\Gamma \vdash \mathbf{S}(p) \leq \mathbf{S}(p)$.

- Case: Rule 2.30.
$$\frac{\Gamma \vdash \pi_1 p :: K' \qquad \Gamma \vdash \pi_2 p :: K''}{\Gamma \vdash p :: K' {\times} K''}$$
  1. By Proposition 4.4.1 and the inductive hypothesis, $\Gamma \rhd p \uparrow \Sigma\alpha{::}L'.L''$,
  2. $\Gamma \vdash p :: \Sigma\alpha{::}L'.L''$,
  3. $\Gamma \rhd \pi_1 p \uparrow L'$,
  4. $\Gamma \vdash \pi_1 p :: L'$,
  5. $\Gamma \vdash \mathbf{S}(\pi_1 p :: L') \leq K'$,
  6. $\Gamma \rhd \pi_2 p \uparrow [\pi_1 p/\alpha]L''$,
  7. $\Gamma \vdash \pi_2 p :: [\pi_1 p/\alpha]L''$,
  8. and $\Gamma \vdash \mathbf{S}(\pi_2 p :: [\pi_1 p/\alpha]L'') \leq K''$.
  9. Thus $\Gamma \vdash \mathbf{S}(p :: \Sigma\alpha{::}L'.L'') \leq K' {\times} K''$

- Case: Rule 2.31.
$$\frac{\begin{array}{c}\Gamma, \alpha{::}K' \vdash p\,\alpha :: K'' \\ \Gamma \vdash p :: \Pi\alpha{::}L'.L'' \qquad \Gamma \vdash K' \equiv L'\end{array}}{\Gamma \vdash p :: \Pi\alpha{::}K'.K''}$$
  1. By the inductive hypothesis, $\Gamma \rhd p \uparrow \Pi\alpha{::}L'.L''$,
  2. $\Gamma \vdash p :: \Pi\alpha{::}L'.L''$,
  3. and $\Gamma \vdash (\Pi\alpha{::}L'.\mathbf{S}(p\,\alpha :: L'')) \leq \Pi\alpha{::}K'.K_1''$.
  4. By inversion, $\Gamma \vdash K' \leq L'$.
  5. By the inductive hypothesis, and determinacy and weakening of the kind extraction algorithm, $\Gamma, \alpha{::}K' \rhd p\,\alpha \uparrow L''$
  6. and $\Gamma, \alpha{::}K' \vdash \mathbf{S}(p\,\alpha :: L'') \leq K''$.
  7. Therefore, $\Gamma \vdash \Pi\alpha{::}L'.\mathbf{S}(p\,\alpha :: L'') \leq \Pi\alpha{::}K'.K''$.

- Case: Rule 2.32.
$$\frac{\Gamma \vdash p :: K_1 \qquad \Gamma \vdash K_1 \leq K_2}{\Gamma \vdash p :: K_2}$$
  1. By the inductive hypothesis, $\Gamma \rhd p \uparrow L$,
  2. $\Gamma \vdash p :: L$,

3. and $\Gamma \vdash \mathbf{S}(p :: L) \leq K_1$.
4. By transitivity, $\Gamma \vdash \mathbf{S}(p :: L) \leq K_2$.

∎

Corollary 4.4.3
If $\Gamma \vdash \mathcal{E}[p] :: K$ and $\Gamma \triangleright p \uparrow \mathbf{S}(A)$ then $\Gamma \vdash \mathcal{E}[p] \equiv \mathcal{E}[A] :: K$.

Proof:

1. By Lemma 4.4.2, $\Gamma \triangleright \mathcal{E}[p] \uparrow L$,
2. $\Gamma \vdash \mathcal{E}[p] :: L$,
3. and $\Gamma \vdash \mathbf{S}(\mathcal{E}[p] :: L) \leq K$.
4. By the determinacy of kind extraction, this can be reconciled with $\Gamma \triangleright p \uparrow \mathbf{S}(A)$ only if $\mathcal{E} = \diamond$ and $L = \mathbf{S}(A)$.
5. Thus $\Gamma \vdash p \equiv A :: \mathbf{T}$.
6. and $\mathbf{S}(\mathcal{E}[p] :: L) = \mathbf{S}(p)$.
7. By inversion of subkinding, either $K = \mathbf{T}$ or $K = \mathbf{S}(A')$ with $\Gamma \vdash p \equiv A' :: \mathbf{T}$.
8. In either case, $\Gamma \vdash p \equiv A :: K$.
9. That is, $\Gamma \vdash \mathcal{E}[p] \equiv \mathcal{E}[A] :: K$ as desired.

∎

Proposition 4.4.4
If $\Gamma \vdash \lambda\alpha{::}K'.A :: L$ then $\Gamma, \alpha{::}K' \vdash A :: K''$ for some kind K''.

Proof: By induction on derivations. For proofs ending with Rule 2.24 the desired result is given directly; for Rules 2.31 and 2.32, the result follows directly by the inductive hypothesis. ∎

Proposition 4.4.5
If $\Gamma \vdash \mathcal{E}[(\lambda\alpha{::}L.A)\ A'] :: K$ then $\Gamma \vdash \mathcal{E}[(\lambda\alpha{::}L.A)\ A'] \equiv \mathcal{E}[[A'/\alpha]A] :: K$

Proof: By induction on the given derivation.

- Case:
$$\frac{\Gamma \vdash \lambda\alpha{::}L'.A :: \Pi\alpha{::}K'.K'' \qquad \Gamma \vdash A' :: K'}{\Gamma \vdash (\lambda\alpha{::}L'.A)\ A' :: [A'/\alpha]K''}$$
  where $\mathcal{E} = \diamond$.

  1. Using Proposition 4.4.4 and the correctness of principal kind synthesis we have $\Gamma, \alpha{::}L' \triangleright A \Uparrow L''$,
  2. $\Gamma, \alpha{::}L' \vdash A :: L''$,
  3. $\Gamma \triangleright \lambda\alpha{::}L'.A \Uparrow \Pi\alpha{::}L'.L''$,
  4. $\Gamma \vdash \lambda\alpha{::}L'.A :: \Pi\alpha{::}L'.L''$,
  5. and $\Gamma \vdash \Pi\alpha{::}L'.L'' \leq \Pi\alpha{::}K'.K''$.

6. By inversion, $\Gamma \vdash K' \leq L'$
7. and $\Gamma, \alpha{::}K' \vdash L'' \leq K''$.
8. By subsumption, $\Gamma \vdash A' :: L'$.
9. Thus $\Gamma \vdash (\lambda\alpha{::}L.A)\, A' \equiv [A'/\alpha]A :: [A'/\alpha]L''$ by Rule 2.107.
10. By substitution $\Gamma \vdash [A'/\alpha]L'' \leq [A'/\alpha]K''$.
11. Therefore by subsumption we have $\Gamma \vdash (\lambda\alpha{::}L.A)\, A' \equiv [A'/\alpha]A :: [A'/\alpha]K''$

- All other cases follow by structural rules and reflexivity of declarative equivalence.

∎

Proposition 4.4.6

1. If $\Gamma \vdash \mathcal{E}[\pi_1\langle A', A''\rangle] :: K$ then $\Gamma \vdash \mathcal{E}[\pi_1\langle A', A''\rangle] \equiv \mathcal{E}[A'] :: K$.

2. If $\Gamma \vdash \mathcal{E}[\pi_2\langle A', A''\rangle] :: K$ then $\Gamma \vdash \mathcal{E}[\pi_2\langle A', A''\rangle] \equiv \mathcal{E}[A''] :: K$.

3. If $\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''$ then $\Gamma \vdash A' :: K'$ and $\Gamma \vdash A'' :: [A'/\alpha]K''$.

Proof:

1. - Case:

$$\frac{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_1\langle A', A''\rangle :: K'}$$

   where $\mathcal{E} = \diamond$.

   (a) Inductively by Part 3, $\Gamma \vdash A' :: K'$
   (b) and $\Gamma \vdash A'' :: [A'/\alpha]K''$.
   (c) The desired result follows by Rule 2.108.

   - The remaining cases follow by structural rules and reflexivity.

2. - Case:

$$\frac{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''}{\Gamma \vdash \pi_2\langle A', A''\rangle :: [\pi_1\langle A', A''\rangle/\alpha]K''}$$

   where $\mathcal{E} = \diamond$.

   (a) Inductively by Part 3, $\Gamma \vdash A' :: K'$
   (b) and $\Gamma \vdash A'' :: [A'/\alpha]K''$.
   (c) By Rule 2.109, $\Gamma \vdash \pi_2\langle A', A''\rangle \equiv A'' :: [A'/\alpha]K''$.
   (d) As in Part 1, $\Gamma \vdash \mathcal{E}[\pi_1\langle A', A''\rangle] \equiv \mathcal{E}[A'] :: K$.
   (e) By validity and inversion, $\Gamma, \alpha{::}K' \vdash K''$
   (f) so by functionality, $\Gamma \vdash [\pi_1\langle A', A''\rangle/\alpha]K'' \equiv [A'/\alpha]K''$.
   (g) Thus by subsumption we have $\Gamma \vdash \pi_2\langle A', A''\rangle :: [\pi_1\langle A', A''\rangle/\alpha]K''$.

   - The remaining cases follow by structural rules and reflexivity.

3. - Case:

$$\frac{\Gamma \vdash A_1 :: K' \qquad \Gamma \vdash A_2 :: K''}{\Gamma \vdash \langle A_1, A_2\rangle :: K'{\times}K''}$$

   Obvious.

- Case:

$$\frac{\Gamma \vdash \Sigma\alpha{::}K'.K'' \quad \Gamma \vdash \pi_1\langle A', A''\rangle :: K' \quad \Gamma \vdash \pi_2\langle A', A''\rangle :: [\pi_1\langle A', A''\rangle/\alpha]K''}{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''}$$

  (a) Inductively by part 1, $\Gamma \vdash \pi_1\langle A', A''\rangle \equiv A' :: K'$.
  (b) Inductively by part 2, $\Gamma \vdash \pi_2\langle A', A''\rangle \equiv A'' :: [\pi_1\langle A', A''\rangle/\alpha]K''$.
  (c) By inversion and functionality, $\Gamma \vdash [\pi_1\langle A', A''\rangle/\alpha]K'' \equiv [A'/\alpha]K''$.
  (d) Thus by validity, subsumption and Proposition 3.1.1, $\Gamma \vdash A' :: K'$
  (e) and $\Gamma \vdash A'' :: [A'/\alpha]K''$.

- Case:

$$\frac{\Gamma \vdash \langle A', A''\rangle :: K_1 \quad \Gamma \vdash K_1 \leq \Sigma\alpha{::}K'.K''}{\Gamma \vdash \langle A', A''\rangle :: \Sigma\alpha{::}K'.K''}$$

  (a) By inversion, $K_1 = \Sigma\alpha{::}K_1'.K_1''$,
  (b) $\Gamma \vdash K_1' \leq K'$,
  (c) and $\Gamma, \alpha{::}K_1' \vdash K_1'' \leq K''$.
  (d) By the inductive hypothesis, $\Gamma \vdash A' :: K_1'$
  (e) and $\Gamma \vdash A'' :: [A'/\alpha]K_1''$.
  (f) By substitution, $\Gamma \vdash [A'/\alpha]K_1'' \leq [A'/\alpha]K''$.
  (g) Then the desired results follow by subsumption.

∎

Corollary 4.4.7
If $\Gamma \vdash A :: K$ and $\Gamma \rhd A \Downarrow B$ then $\Gamma \vdash A \equiv B :: K$.

Proof: By transitivity and reflexivity of declarative equivalence, it suffices to show that if $\Gamma \vdash A :: K$ and $\Gamma \rhd A \leadsto B$ then $\Gamma \vdash A \equiv B :: K$. But all possibilities for the reduction step are covered by Corollary 4.4.3, Proposition 4.4.5, and Proposition 4.4.6. ∎

Proposition 4.4.8
If $\Gamma \vdash \mathcal{E}[A\,A'] :: L$ then there exists a kind $K'{\to}K''$ such that $\Gamma \vdash A :: K'{\to}K''$ and $\Gamma \vdash A' :: K'$.

Proof: By induction on typing derivations. If $\mathcal{E} = \diamond$ and the proof concludes with a use of the application rule 2.25 then the result follows by inversion; in all other cases, the result follows by the inductive hypothesis. ∎

Theorem 4.4.9 (Soundness)
1. *If $\Gamma \vdash A_1 :: K$, $\Gamma \vdash A_2 :: K$, and $\Gamma \rhd A_1 \Leftrightarrow A_2 :: K$ then $\Gamma \vdash A_1 \equiv A_2 :: K$.*
2. *If $\Gamma \vdash p_1 :: K_1$, $\Gamma \vdash p_2 :: K_2$, and $\Gamma \rhd p_1 \leftrightarrow p_2 \uparrow K$ then $\Gamma \vdash p_1 \equiv p_2 :: K$.*
3. *If $\Gamma \vdash K_1$, $\Gamma \vdash K_2$, and $\Gamma \rhd K_1 \leq K_2$ then $\Gamma \vdash K_1 \leq K_2$.*
4. *If $\Gamma \vdash K_1$, $\Gamma \vdash K_2$, and $\Gamma \rhd K_1 \Leftrightarrow K_2$ then $\Gamma \vdash K_1 \equiv K_2$.*
5. *If $\Gamma \vdash ok$ and $\Gamma \rhd K$ then $\Gamma \vdash K$.*

6. *If* $\Gamma \vdash ok$ *and* $\Gamma \rhd A \rightrightarrows K$ *then* $\Gamma \vdash A :: K$ *and* $\Gamma \rhd A \Uparrow K$.
7. *If* $\Gamma \vdash K$ *and* $\Gamma \rhd A \leftleftarrows K$ *then* $\Gamma \vdash A :: K$.

Proof: By (simultaneous) induction on proofs of the algorithmic judgments (i.e., by induction on the execution of the algorithms). ∎

Chapter 5

Completeness and Decidability for Constructors and Kinds

5.1 Introduction

Correctness of the algorithms for constructor and kind judgment can easily be seen to reduce to correctness of the algorithm for constructor equivalence. Since the algorithms of the previous chapter are sound, it suffices to prove completeness of the constructor equivalence algorithm (i.e., if $\Gamma \vdash A_1 \equiv A_2 :: K$ then $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$) and that this algorithm will terminate with an answer for all well-formed inputs.

It is instructive to see why the direct approach of proving completeness by induction on the derivation of $\Gamma \vdash A_1 \equiv A_2 :: K$ fails. We immediately run into trouble with such rules as Rule 2.37:

$$\frac{\Gamma \vdash A \equiv A' :: K' \to K'' \qquad \Gamma \vdash A_1 \equiv A_1' :: K'}{\Gamma \vdash A_1\, A_1' \equiv A_2\, A_2' :: K''}$$

Here we would have by the induction hypothesis that $\Gamma \triangleright A \Leftrightarrow A' :: K' \to K''$ and $\Gamma \triangleright A_1 \Leftrightarrow A_1' :: K'$. However, there appears to be no way to show directly that these imply $\Gamma \triangleright A_1\, A_1' \Leftrightarrow A_2\, A_2' :: K''$ because the algorithm proceeds via head-normalization rather than comparing the applications component-wise.

Similarly, in Rule 2.44

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

there is no way to apply the induction hypothesis and hence no way to show the conclusion.

Coquand [Coq91] proves the completeness of an equivalence algorithm for a lambda calculus with Π types using a form of *Kripke logical relations.* The key idea is to prove completeness by defining a relation (here called *logical equivalence*) which not only implies algorithmic equivalence, but also satisfies stronger properties. For example, if two functions are logically related then their application to logically-related arguments yields logically-related applications. By proving inductively that declarative equivalence implies logical equivalence, we have strengthened the induction hypothesis enough to allow cases such as Rule 2.37 and 2.44 to go through.

I have substantially extended this approach to handle singleton kinds, as well as pairs and subkinding. However, one essential obstacle remains: declarative equivalence is transitive and symmetric, which requires showing that logical equivalence is transitive and symmetric. Since logical equivalence is defined in terms of the equivalence algorithm, this requires showing that algorithmic equivalence is both symmetric and transitive. Surprisingly, this is not at all obvious.

The difficulty is that the presentation of the algorithm is inherently asymmetric. Because of dependencies in the kinds, at various points one must make a choice between one of two provably equal kinds. For example, verifying

$$\Gamma \triangleright A_1 \Leftrightarrow A_2 :: \Sigma\alpha{::}K'.K''$$

requires checking that

$$\Gamma \triangleright \pi_1 A_1 \Leftrightarrow \pi_1 A_2 :: K'$$

and *either*

$$\Gamma \triangleright \pi_2 A_1 \Leftrightarrow \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$$

or

$$\Gamma \triangleright \pi_2 A_1 \Leftrightarrow \pi_2 A_2 :: [\pi_1 A_2/\alpha]K''.$$

(Similar alternatives also appear in the definitions of path equivalence and kind equivalence as well.) Although the kinds $[\pi_1 A_1/\alpha]K''$ and $[\pi_1 A_2/\alpha]K''$ will be provably equivalent, each choice leads to different definitions in the context and may cause head-normalization to take an entirely different path. If the algorithm is correct then it *should* end up with the same answer in either case, but I am unable to give a direct proof that this is true.

The algorithm could be forced to be more symmetric by adding conditions, e.g., by specifying that

$$\Gamma \triangleright A_1 \Leftrightarrow A_2 :: \Sigma\alpha{::}K'.K''$$

requires

$$\Gamma \triangleright \pi_1 A_1 \Leftrightarrow \pi_1 A_2 :: K'$$

and

$$\Gamma \triangleright \pi_2 A_1 \Leftrightarrow \pi_2 A_2 :: [\pi_1 A_1/\alpha]K''$$

and

$$\Gamma \triangleright \pi_2 A_1 \Leftrightarrow \pi_2 A_2 :: [\pi_1 A_2/\alpha]K'',$$

but the problem of showing transitivity remains.

In §5.2 I give a revised form for the constructor and kind equivalence algorithms, designed specifically to make both transitivity and symmetry obvious. This leads to a nonstandard form of Kripke-style logical relation, described in §5.3; using this I show the revised equivalence algorithms are terminating and complete with respect to MIL_0 equivalence. Finally, since the revised algorithm requires redundant bookkeeping, I show in §5.4 that the correctness of the revised algorithm implies the completeness and termination of the equivalence algorithm presented in the previous chapter, which forms the basis of the TILT implementation. It follows that all kind and constructor-level judgments are decidable.

5.2 A Symmetric and Transitive Algorithm

5.2.1 Definition

The way to build transitivity into constructor and kind equivalence is to maintain *two* provably equal typing contexts and *two* (provably equal) classifying kinds. Then the form of algorithmic constructor equivalence becomes

$$\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2.$$

Although the expectation is that the algorithm will only be applied when $\Gamma_1 \vdash A_1 :: K_1$ and $\Gamma_2 \vdash A_2 :: K_2$, this is not a comparison of judgments but merely suggestive notation for a 6-place relation. The algorithm takes these 6 inputs and returns success or failure (or fails to terminate).

The advantage of this formulation is that arbitrary choices disappear. For example, the comparison

$$\Gamma_1 \triangleright A_1 :: \Sigma\alpha{::}K_1'.K_2'' \Leftrightarrow \Gamma_2 \triangleright A_2 :: \Sigma\alpha{::}K_2'.K_2''$$

between two pairs of constructors checks

$$\Gamma_1 \triangleright \pi_1 A_1 :: K_1' \Leftrightarrow \Gamma_2 \triangleright \pi_1 A_2 :: K_2'$$

and

$$\Gamma_1 \triangleright \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Gamma_2 \triangleright \pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2''.$$

Both of the possible substitutions are used, in a symmetric fashion.

Similarly the algorithmic path equivalence relation takes the form

$$\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow K_2,$$

and algorithmic kind equivalence becomes

$$\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2.$$

The full definitions of the revised algorithm are shown in Figure 5.1. (The kind extraction, weak head reduction, and weak head normalization judgments are unchanged.) It is simple to show that these definitions have the required behavior:

Lemma 5.2.1 (Algorithmic Symmetry and Transitivity)

1. *If* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$ *then* $\Gamma_2 \triangleright A_2 :: K_2 \Leftrightarrow \Gamma_1 \triangleright A_1 :: K_1$.
2. *If* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$ *and* $\Gamma_2 \triangleright A_2 :: K_2 \Leftrightarrow \Gamma_3 \triangleright A_3 :: K_3$ *then* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_3 \triangleright A_3 :: K_3$.
3. *If* $\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow K_2$ *then* $\Gamma_2 \triangleright p_2 \uparrow K_2 \leftrightarrow \Gamma_1 \triangleright p_1 \uparrow K_1$.
4. *If* $\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow K_2$ *and* $\Gamma_2 \triangleright p_2 \uparrow K_2 \leftrightarrow \Gamma_3 \triangleright p_3 \uparrow K_3$ *then* $\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_3 \triangleright p_3 \uparrow K_3$.
5. *If* $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ *then* $\Gamma_2 \triangleright K_2 \Leftrightarrow \Gamma_1 \triangleright K_1$.
6. *If* $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ *and* $\Gamma_2 \triangleright K_2 \Leftrightarrow \Gamma_3 \triangleright K_3$ *then* $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_3 \triangleright K_3$.

Proof: By induction on derivations of the algorithmic judgments (i.e., by induction on the execution of the algorithms). ∎

I have made two changes to the constructor equivalence algorithm beyond those necessary to maintain symmetry and transitivity.

- When comparing two constructors with singleton kinds, the algorithm compares the two constructors at kind **T** rather than short-circuiting with immediate success.

- When comparing two constructors with Π kinds, the algorithm also compares the domain kinds of the two Π kinds.

Intuitively these additions are redundant, but they are useful when proving the existence of normal forms of constructors (see §5.5). If this algorithm is sound, complete, and terminating, then it will remain so when these redundant extensions are omitted. However, the converse is less obvious; a priori these extra tests might cause the algorithm to become nonterminating on some inputs. Hence proving the correctness of the algorithm as shown in Figure 5.1 is a stronger result.

5.2.2 Soundness

As before, path equivalence computes extracted kinds of paths, but here it extracts the kinds of *both* paths:

Lemma 5.2.2
If $\Gamma_1 \triangleright A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright A_2 \uparrow K_2$ *then* $\Gamma_1 \triangleright A_1 \uparrow K_1$ *and* $\Gamma_2 \triangleright A_2 \uparrow K_2$.

Then proof of soundness for the revised algorithms is very similar to the proof for the original algorithmic equivalence:

Theorem 5.2.3 (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* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$ *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* $\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow K_2$ *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* $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ *then* $\Gamma_1 \vdash K_1 \equiv K_2$.

Proof: Parts 1 and 2 follow by simultaneous induction on the algorithmic judgments and by cases on the last step in the algorithmic derivation. I omit the proof of part 3, which follows from part 1 and induction.

1. - Case: $\Gamma_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \triangleright A_2 :: \mathbf{T}$ because $\Gamma_1 \triangleright A_1 \Downarrow p_1$, $\Gamma_2 \triangleright A_2 \Downarrow p_2$, and $\Gamma_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \mathbf{T}$.
     (a) By Corollary 4.4.7, $\Gamma_1 \vdash A_1 \equiv p_1 :: \mathbf{T}$
     (b) and $\Gamma_2 \vdash A_2 \equiv p_2 :: \mathbf{T}$.
     (c) By Corollary 3.2.8 $\Gamma_1 \vdash A_2 \equiv p_2 :: \mathbf{T}$.
     (d) By Validity, $\Gamma_1 \vdash p_1 :: \mathbf{T}$
     (e) and $\Gamma_2 \vdash p_2 :: \mathbf{T}$.
     (f) By the inductive hypothesis, $\Gamma_1 \vdash p_1 \equiv p_2 :: \mathbf{T}$.
     (g) By symmetry and transitivity of equivalence therefore, $\Gamma_1 \vdash A_1 \equiv A_2 :: \mathbf{T}$.
   - Case: $\Gamma_1 \triangleright A_1 :: \mathbf{S}(B_1) \Leftrightarrow \Gamma_2 \triangleright A_2 :: \mathbf{S}(B_2)$ because $\Gamma_1 \triangleright A_1 \Downarrow p_1$, $\Gamma_2 \triangleright A_2 \Downarrow p_2$, and $\Gamma_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \mathbf{T}$.
     (a) As in the previous case, $\Gamma_1 \vdash A_1 \equiv A_2 :: \mathbf{T}$.
     (b) Then $\Gamma_1 \vdash A_1 \equiv A_2 :: \mathbf{S}(A_1)$
     (c) but $\Gamma_1 \vdash A_1 \equiv B_1 :: \mathbf{T}$ by inversion of kind equivalence,
     (d) so $\Gamma_1 \vdash A_1 \equiv A_2 :: \mathbf{S}(B_1)$ by subsumption.

Algorithmic constructor equivalence

$\Gamma_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \rhd A_2 :: \mathbf{T}$	if $\Gamma_1 \rhd A_1 \Downarrow p_1$ and $\Gamma_2 \rhd A_2 \Downarrow p_2$ and $\Gamma_1 \rhd p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \mathbf{T}$
$\Gamma_1 \rhd A_1 :: \mathbf{S}(B_1) \Leftrightarrow \Gamma_2 \rhd A_2 :: \mathbf{S}(B_2)$	if $\Gamma_1 \rhd A_1 \Downarrow p_1$ and $\Gamma_2 \rhd A_2 \Downarrow p_2$ and $\Gamma_1 \rhd p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \mathbf{T}$
$\Gamma_1 \rhd A_1 :: \Pi\alpha{::}K_1'.K_1'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Pi\alpha{::}K_2'.K_2''$	if $\Gamma_1, \alpha{::}K_1' \rhd A_1\,\alpha :: K_1'' \Leftrightarrow \Gamma_2, \alpha{::}K_2' \rhd A_2\,\alpha :: K_2''$ and $\Gamma_1 \rhd K_1' \Leftrightarrow \Gamma_2 \rhd K_2'$
$\Gamma_1 \rhd A_1 :: \Sigma\alpha{::}K_1'.K_2'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Sigma\alpha{::}K_2'.K_2''$	if $\Gamma_1 \rhd \pi_1 A_1 :: K_1' \Leftrightarrow \Gamma_2 \rhd \pi_1 A_2 :: K_2'$, and $\Gamma_1 \rhd \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Gamma_2 \rhd \pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2''$

Algorithmic path equivalence

$\Gamma_1 \rhd b \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd b \uparrow \mathbf{T}$	always
$\Gamma_1 \rhd \times \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T} \leftrightarrow \Gamma_2 \rhd \times \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$	always
$\Gamma_1 \rhd \to \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T} \leftrightarrow \Gamma_2 \rhd \to \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$	always
$\Gamma_1 \rhd \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \rhd \alpha \uparrow \Gamma_2(\alpha)$	always
$\Gamma_1 \vdash p_1\,A_1 \uparrow [A_1/\alpha]K_1'' \leftrightarrow \Gamma_2 \vdash p_2\,A_2 \uparrow [A_2/\alpha]K_2''$	if $\Gamma_1 \rhd p_1 \uparrow \Pi\alpha{::}K_1'.K_1'' \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \Pi\alpha{::}K_2'.K_2''$, and $\Gamma_1 \rhd A_1 :: K_1' \Leftrightarrow \Gamma_2 \rhd A_2 :: K_2'$.
$\Gamma_1 \rhd \pi_1 p_1 \uparrow K_1' \leftrightarrow \Gamma_2 \rhd \pi_1 p_2 \uparrow K_2'$	if $\Gamma_1 \rhd p_1 \uparrow \Sigma\alpha{::}K_1'.K_1'' \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \Sigma\alpha{::}K_2'.K_2''$.
$\Gamma_1 \vdash \pi_2 p_1 \uparrow [\pi_1 p_1/\alpha]K_1'' \leftrightarrow \Gamma_2 \vdash \pi_2 p_2 \uparrow [\pi_1 p_2/\alpha]K_2''$	if $\Gamma_1 \rhd p_1 \uparrow \Sigma\alpha{::}K_1'.K_1'' \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \Sigma\alpha{::}K_2'.K_2''$

Algorithmic kind equivalence

$\Gamma_1 \rhd \mathbf{T} \Leftrightarrow \Gamma_2 \rhd \mathbf{T}$	always
$\Gamma_1 \rhd \mathbf{S}(A_1) \Leftrightarrow \Gamma_2 \rhd \mathbf{S}(A_2)$	if $\Gamma_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \rhd A_2 :: \mathbf{T}$
$\Gamma_1 \rhd \Pi\alpha{::}K_1'.K_1'' \Leftrightarrow \Gamma_2 \rhd \Pi\alpha{::}K_2'.K_2''$	if $\Gamma_1 \rhd K_1 \Leftrightarrow \Gamma_2 \rhd K_2$ and $\Gamma_1, \alpha{::}K_1' \rhd K_1'' \Leftrightarrow \Gamma_2, \alpha{::}K_2' \rhd K_2''$
$\Gamma_1 \rhd \Sigma\alpha{::}K_1'.K_1'' \Leftrightarrow \Gamma_2 \rhd \Sigma\alpha{::}K_2'.K_2''$	if $\Gamma_1 \rhd K_1' \Leftrightarrow \Gamma_2 \rhd K_2'$ and $\Gamma_1, \alpha{::}K_1' \rhd K_1'' \Leftrightarrow \Gamma_2, \alpha{::}K_2' \rhd K_2''$

Figure 5.1: Revised Equivalence Algorithm

- Case: $\Gamma_1 \rhd A_1 :: \Pi\alpha{::}K_1'.K_1'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Pi\alpha{::}K_2'.K_2''$ because $\Gamma_1, \alpha{::}K_1' \rhd A_1\,\alpha :: K_1'' \Leftrightarrow \Gamma_2, \alpha{::}K_2' \rhd A_2\,\alpha :: K_2''$ and $\Gamma_1 \rhd K_1' \Leftrightarrow \Gamma_2 \rhd K_2'$.
  (a) Since $\vdash \Gamma_1, \alpha{::}K_1' \equiv \Gamma_2, \alpha{::}K_2'$,
  (b) $\Gamma_1, \alpha{::}K_1' \vdash A_1\,\alpha :: K_1''$,
  (c) $\Gamma_2, \alpha{::}K_2' \vdash A_2\,\alpha :: K_2''$,
  (d) and $\Gamma_1, \alpha{::}K_1' \vdash K_1'' \equiv K_2''$,
  (e) the inductive hypothesis applies, yielding $\Gamma_1, \alpha{::}K_1' \vdash A_1\,\alpha \equiv A_2\,\alpha :: K_1''$.
  (f) Thus by Rule 2.42, $\Gamma_1 \vdash A_1 \equiv A_2 :: \Pi\alpha{::}K_1'.K_1''$.
- $\Gamma_1 \rhd A_1 :: \Sigma\alpha{::}K_1'.K_1'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Sigma\alpha{::}K_2''.K_2''$ because $\Gamma_1 \rhd \pi_1 A_1 :: K_1' \Leftrightarrow \Gamma_2 \rhd \pi_1 A_2 :: K_2'$, and $\Gamma_1 \rhd \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Gamma_2 \rhd \pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2''$.
  (a) Since $\Gamma_1 \vdash \pi_1 A_1 :: K_1'$
  (b) $\Gamma_2 \vdash \pi_1 A_2 :: K_2'$,
  (c) and by inversion $\Gamma_1 \vdash K_1' \equiv K_2'$,
  (d) by the inductive hypothesis we have $\Gamma_1 \vdash \pi_1 A_1 \equiv \pi_1 A_2 :: K_1'$.
  (e) By functionality, $\Gamma_1 \vdash [\pi_1 A_1/\alpha]K_1'' \equiv [\pi_1 A_2/\alpha]K_2''$.
  (f) Then $\Gamma_1 \vdash \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1''$
  (g) and $\Gamma_2 \vdash \pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2''$.
  (h) By the inductive hypothesis, $\Gamma_1 \vdash \pi_2 A_1 \equiv \pi_2 A_2 :: [\pi_1 A_1/\alpha]K_1''$.
  (i) By Corollary 3.2.8 and Rule 2.41, $\Gamma_1 \vdash A_1 \equiv A_2 :: \Sigma\alpha{::}K_1'.K_1''$.

2.
- Case: $\Gamma_1 \rhd b_i \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd b_i \uparrow \mathbf{T}$.
  By Proposition 3.1.1, $\Gamma_1 \vdash \text{ok}$. Thus by Rule 2.33, $\Gamma_1 \vdash b_i \equiv b_i :: \mathbf{T}$.
- Case: $\Gamma_1 \rhd \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \rhd \alpha \uparrow \Gamma_2(\alpha)$.
  By Validity and Rule 2.33, $\Gamma_1 \vdash \alpha \equiv \alpha :: \Gamma_1(\alpha)$.
- Case: $\Gamma_1 \rhd p_1\,A_1 \uparrow [A_1/\alpha]L_1'' \leftrightarrow \Gamma_2 \rhd p_2\,A_2 \uparrow [A_2/\alpha]L_2''$ because $\Gamma_1 \rhd p_1 \uparrow \Pi\alpha{::}L_1'.L_1'' \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \Pi\alpha{::}L_2'.L_2''$ and $\Gamma_1 \rhd A_1 :: L_1' \Leftrightarrow \Gamma_2 \rhd A_2 :: L_2'$.
  (a) By Proposition 4.4.8, $\Gamma_1 \vdash p_1 :: K_1' \to K_1''$,
  (b) $\Gamma_1 \vdash A_1 :: K_1'$,
  (c) $\Gamma_2 \vdash p_2 :: K_2' \to K_2''$,
  (d) and $\Gamma_2 \vdash A_2 :: K_2'$.
  (e) By the inductive hypothesis, $\Gamma_1 \vdash \Pi\alpha{::}L_1'.L_1'' \equiv \Pi\alpha{::}L_2'.L_2''$.
  (f) and $\Gamma_1 \vdash p_1 \equiv p_2 :: \Pi\alpha{::}L_1'.L_1''$.
  (g) By Lemma 4.4.2, $\Gamma_1 \vdash \mathbf{S}(p_1 :: \Pi\alpha{::}L_1'.L_1'') \leq K_1' \to K_1''$
  (h) and $\Gamma_2 \vdash \mathbf{S}(p_2 :: \Pi\alpha{::}L_2'.L_2'') \leq K_2' \to K_2''$.
  (i) Thus $\Gamma_1 \vdash K_1' \leq L_1'$
  (j) and $\Gamma_2 \vdash K_2' \leq L_2'$.
  (k) By subsumption then, $\Gamma_1 \vdash A_1 :: L_1'$
  (l) and $\Gamma_2 \vdash A_2 :: L_2'$.
  (m) The induction hypothesis applies, and so $\Gamma_1 \vdash A_1 \equiv A_2 :: L_1'$.
  (n) Thus $\Gamma_1 \vdash p_1\,A_1 \equiv p_2\,A_2 :: [A_1/\alpha]L_1''$
  (o) and by functionality $\Gamma_1 \vdash [A_1/\alpha]L_1'' \equiv [A_2/\alpha]L_2''$.

- Case: $\Gamma_1 \triangleright \pi_1 p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright \pi_1 p_2 \uparrow K_2$ because $\Gamma_1 \triangleright p_1 \uparrow \Sigma\alpha{::}K_1.L_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \Sigma\alpha{::}K_2.L_2$
  (a) By Proposition 4.4.1 the inductive hypothesis applies,
  (b) so $\Gamma_1 \vdash \Sigma\alpha{::}K_1.L_1 \equiv \Sigma\alpha{::}K_2.L_2$
  (c) and $\Gamma_1 \vdash p_1 \equiv p_2 :: \Sigma\alpha{::}K_1.L_1$.
  (d) Thus $\Gamma_1 \vdash \pi_1 p_1 \equiv \pi_1 p_2 :: K_1$
  (e) and by inversion, $\Gamma_1 \vdash K_1 \equiv K_2$.
- Case: $\Gamma_1 \triangleright \pi_2 p_1 \uparrow [\pi_1 p_1/\alpha]L_1 \leftrightarrow \Gamma_2 \triangleright \pi_2 p_2 \uparrow [\pi_1 p_2/\alpha]L_2$ because $\Gamma_1 \triangleright p_1 \uparrow \Sigma\alpha{::}K_1.L_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \Sigma\alpha{::}K_2.L_2$.
  (a) By Proposition 4.4.1 the inductive hypothesis applies,
  (b) so $\Gamma_1 \vdash \Sigma\alpha{::}K_1.L_1 \equiv \Sigma\alpha{::}K_2.L_2$
  (c) and $\Gamma_1 \vdash p_1 \equiv p_2 :: \Sigma\alpha{::}K_1.L_1$.
  (d) Thus $\Gamma_1 \vdash \pi_2 p_1 \equiv \pi_2 p_2 :: [\pi_1 p_1/\alpha]L_1$.
  (e) $\Gamma_1 \vdash \pi_1 p_1 \equiv \pi_1 p_2 :: K_1$
  (f) So by functionality, $\Gamma_1 \vdash [\pi_1 p_1/\alpha]L_1 \equiv [\pi_1 p_2/\alpha]L_2$

∎

5.3 Completeness of the Revised Algorithms

To show the completeness and termination for the algorithm I use a modified Kripke-style logical relations argument. The strategy for proving completeness of the algorithm is

1. Define the logical relations;
2. Show that logically-related constructors are related by the algorithm;
3. Show that provably-equivalent constructors are logically related.

From completeness it follows that the algorithm terminates for all well-formed inputs.

I use Δ to denote a Kripke world. Worlds are contexts containing no duplicate bound variables; the partial order $\subseteq$ on worlds is simply the weakening ordering given in Definition 3.1.4. The logical relations I use are shown in Figures 5.2, 5.3, and 5.4.

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, I have logically-defined binary equivalence relations between (logically valid) types and terms. 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 two *possibly 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 I impose the algorithmic equivalence as the definition of the logical relation. At higher kinds I use a Kripke-style logical relations interpretation of Π and Σ: functions are related if in all pairs of future worlds related arguments yield related results, and pairs are related if their first and second components are related.

- $(\Delta; K)$ **valid** iff

  1. – $K = \mathbf{T}$
     – Or, $K = \mathbf{S}(A)$ and $(\Delta; A; \mathbf{T})$ **valid**
     – Or, $K = \Pi\alpha{::}K'.K''$ and $(\Delta; K')$ **valid** and $\forall \Delta' \supseteq \Delta, \Delta'' \supseteq \Delta$ if $(\Delta'; A_1; K')$ **is** $(\Delta''; A_2; K')$ then $(\Delta'; [A_1/\alpha]K'')$ **is** $(\Delta''; [A_2/\alpha]K'')$
     – Or, $K = \Sigma\alpha{::}K'.K''$ and $(\Delta; K')$ **valid** and $\forall \Delta' \supseteq \Delta, \Delta'' \supseteq \Delta$ if $(\Delta'; A_1; K')$ **is** $(\Delta''; A_2; K')$ then $(\Delta'; [A_1/\alpha]K'')$ **is** $(\Delta''; [A_2/\alpha]K'')$

- $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ iff

  1. $(\Delta_1; K_1)$ **valid** and $(\Delta_2; K_2)$ **valid**.
  2. And,
     – $K_1 = \mathbf{T}$ and $K_2 = \mathbf{T}$
     – Or, $K_1 = \mathbf{S}(A_1)$ and $K_2 = \mathbf{S}(A_2)$ and $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$
     – 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 $\forall \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_2'; A_2; K_2')$ then $(\Delta_1'; [A_1/\alpha]K_1'')$ **is** $(\Delta_2'; [A_2/\alpha]K_2'')$
     – 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 $\forall \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ if $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_2'; A_2; K_2')$ then $(\Delta_1'; [A_1/\alpha]K_1'')$ **is** $(\Delta_2'; [A_2/\alpha]K_2'')$

- $(\Delta_1; K_1 \leq L_1)$ **is** $(\Delta_2; K_2 \leq L_2)$ iff

  1. $\forall \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ 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 5.2: Logical Relations for Kinds

With these definitions in hand I construct 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:

$$\frac{\begin{array}{c}(\Delta_1; A_1; K_1)\ \mathbf{is}\ (\Delta_2; A_2; K_2)\\ (\Delta_1; K_1 \leq L_1)\ \mathbf{is}\ (\Delta_2; K_2 \leq L_2)\end{array}}{(\Delta_1; A_1; L_1)\ \mathbf{is}\ (\Delta_2; A_2; L_2)}$$

Finally, validity and equivalence relations for substitutions are defined pointwise.

The first property to be checked is that the logical relations are monotone (preserved when passing to future worlds), which corresponds to the weakening property for the algorithmic relations.

Lemma 5.3.1 (Algorithmic Weakening)

1. *If* $\Gamma \triangleright A \rightsquigarrow B$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \triangleright A \rightsquigarrow B$
2. *If* $\Gamma \triangleright A \Downarrow p$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \triangleright A \Downarrow p$.
3. *If* $\Gamma \triangleright A \uparrow K$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \triangleright A \uparrow K$.
4. *If* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$, $\Gamma_1' \supseteq \Gamma_1$, *and* $\Gamma_2' \supseteq \Gamma_2$, *then* $\Gamma_1' \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2' \triangleright A_2 :: K_2$.
5. *If* $\Gamma_1 \triangleright A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright A_2 \uparrow K_2$, $\Gamma_1' \supseteq \Gamma_1$, *and* $\Gamma_2' \supseteq \Gamma_2$, *then* $\Gamma_1' \triangleright A_1 \uparrow K_1 \leftrightarrow \Gamma_2' \triangleright A_2 \uparrow K_2$.

- $(\Delta; A; K)$ **valid** iff

  1. $(\Delta; K)$ **valid**
  2. And,
     - $K = \mathbf{T}$ and $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright A :: \mathbf{T}$.
     - Or, $K = \mathbf{S}(B)$ and $(\Delta; A; \mathbf{T})$ **is** $(\Delta; B; \mathbf{T})$.
     - Or, $K = \Pi\alpha{::}K'.K''$, and $\forall \Delta' \supseteq \Delta, \Delta'' \supseteq \Delta$ if $(\Delta'; B_1; K')$ **is** $(\Delta''; B_2; K')$ then $(\Delta'; A\, B_1; [B_1/\alpha]K'')$ **is** $(\Delta''; A\, B_2; [B_2/\alpha]K'')$.
     - Or, $K = \Sigma\alpha{::}K'.K''$, $(\Delta; \pi_1 A; K')$ **valid** and $(\Delta; \pi_2 A; [\pi_1 A/\alpha]K'')$ **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,
     - $K_1 = K_2 = \mathbf{T}$ and $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
     - Or, $K_1 = \mathbf{S}(B_1)$, $K_2 = \mathbf{S}(B_2)$, and $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$
     - Or, $K_1 = \Pi\alpha{::}K_1'.K_1''$, $K_2 = \Pi\alpha{::}K_2'.K_2''$, and $\forall \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ if $(\Delta_1'; B_1; K_1')$ **is** $(\Delta_2'; B_2; K_2')$ then $(\Delta_1'; A_1\, B_1; [B_1/\alpha]K_1'')$ **is** $(\Delta_2'; A_2\, B_2; [B_2/\alpha]K_2'')$.
     - 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 $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$

Figure 5.3: Logical Relations for 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 5.4: Logical Relations for Substitutions

6. *If* $\Gamma_1 \rhd K_1 \Leftrightarrow \Gamma_2 \rhd K_2$, $\Gamma_1' \supseteq \Gamma_1$, *and* $\Gamma_2' \supseteq \Gamma_2$, *then* $\Gamma_1' \rhd K_1 \Leftrightarrow \Gamma_2' \rhd K_2$.

Proof: By induction on algorithmic derivations. ∎

Lemma 5.3.2 (Monotonicity)

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

Proof:

1–5. By induction on the size of kinds.

6–7. By the previous parts.

∎

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

Lemma 5.3.3 (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** $(\Delta; \gamma; \Gamma)$.

Proof: The "if" direction is immediate from the definitions of the logical relations, so we only show the "only if" direction.

1. By induction on the size of K. Assume $(\Delta; K)$ **valid**.
   - Case: $K = \mathbf{T}$. Follows by definition of $(\Delta; \mathbf{T})$ **is** $(\Delta; \mathbf{T})$.
   - Case: $K = \mathbf{S}(B)$.
     (a) $(\Delta; B; \mathbf{T})$ **valid**.
     (b) $\Delta \rhd B :: \mathbf{T} \Leftrightarrow \Delta \rhd B :: \mathbf{T}$.
     (c) Then $(\Delta; B; \mathbf{T})$ **valid**
     (d) and $(\Delta; B; \mathbf{T})$ **is** $(\Delta; B; \mathbf{T})$.
     (e) Therefore $(\Delta; \mathbf{S}(B))$ **is** $(\Delta; \mathbf{S}(B))$.
   - Case: $K = \Pi\alpha{::}K'.K''$.
     (a) By $(\Delta; \Pi\alpha{::}K'.K'')$ **valid** we have $(\Delta; K')$ **valid**.
     (b) By the inductive hypothesis, $(\Delta; K')$ **is** $(\Delta; K')$.
     (c) Let $(\Delta', \Delta'') \supseteq (\Delta, \Delta)$
     (d) and assume $(\Delta'; A_1; K')$ **is** $(\Delta''; A_2; K')$.

(e) By $(\Delta; \Pi\alpha{::}K'.K'')$ **valid** we have $(\Delta'; [A_1/\alpha]K'')$ **is** $(\Delta''; [A_2/\alpha]K'')$.

(f) Therefore $(\Delta; \Pi\alpha{::}K'.K'')$ **is** $(\Delta; \Pi\alpha{::}K'.K'')$.

- Case: $K = \Sigma\alpha{::}K'.K''$.

  Same proof as for Π case.

2. By induction on the size of A. Assume $(\Delta; A; K)$ **valid**. Then $(\Delta; K)$ **valid** so that by part 1, $(\Delta; K)$ **is** $(\Delta; K)$.

- Case: $K = \mathbf{T}$.

  (a) $(\Delta; A; \mathbf{T})$ **valid** implies $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright A :: \mathbf{T}$.

  (b) Therefore, $(\Delta; A; \mathbf{T})$ **is** $(\Delta; A; \mathbf{T})$.

- Case: $K = \mathbf{S}(B)$.

  (a) $(\Delta; A; \mathbf{S}(B))$ **valid** implies $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright B :: \mathbf{T}$.

  (b) By Lemma 5.2.1, $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright A :: \mathbf{T}$,

  (c) so $(\Delta; A; \mathbf{T})$ **valid**

  (d) and $(\Delta; A; \mathbf{T})$ **is** $(\Delta; A; \mathbf{T})$.

  (e) Therefore $(\Delta; A; \mathbf{S}(B))$ **is** $(\Delta; A; \mathbf{S}(B))$.

- Case: $K = \Pi\alpha{::}K'.K''$.

  (a) Let $\Delta', \Delta'' \supseteq \Delta$ and assume $(\Delta'; B_1; K')$ **is** $(\Delta''; B_2; K')$.

  (b) Then $(\Delta'; A\,B_1; [B_1/\alpha]K'')$ **is** $(\Delta''; A\,B_2; [B_2/\alpha]K'')$.

  (c) Therefore $(\Delta; A; \Pi\alpha{::}K'.K'')$ **is** $(\Delta; A; \Pi\alpha{::}K'.K'')$.

- Case: $K = \Sigma\alpha{::}K'.K''$.

  (a) Then $(\Delta; \pi_1 A; K')$ **valid**

  (b) and $(\Delta; \pi_2 A; [\pi_1 A/\alpha]K'')$ **valid**.

  (c) By the inductive hypothesis, $(\Delta; \pi_1 A; K')$ **is** $(\Delta; \pi_1 A; K')$

  (d) and $(\Delta; \pi_2 A; [\pi_1 A/\alpha]K'')$ **is** $(\Delta; \pi_2 A; [\pi_1 A/\alpha]K'')$.

  (e) Therefore $(\Delta; A; \Sigma\alpha{::}K'.K'')$ **is** $(\Delta; A; \Sigma\alpha{::}K'.K'')$.

3. (a) Assume $(\Delta; \gamma; \Gamma)$ **valid**.

   (b) Let $x \in \mathrm{dom}(\Gamma)$ be given.

   (c) Then $(\Delta; \gamma x; \gamma(\Gamma x))$ **valid**.

   (d) By part 2, $(\Delta; \gamma x; \gamma(\Gamma x))$ **is** $(\Delta; \gamma x; \gamma(\Gamma x))$.

   (e) Therefore $(\Delta; \gamma; \Gamma)$ **is** $(\Delta; \gamma; \Gamma)$.

∎

I next give a technical lemma which relates logical equivalence of kinds to logical subkinding. An easy corollary of this lemma is the following rule:

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

Lemma 5.3.4
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** $(\Delta_2; K_2 \leq L_2)$.

Proof: By induction on the sizes of kinds.
Assume $(\Delta_1; L_1)$ **is** $(\Delta_2; L_2)$, $(\Delta_1; K_1)$ **is** $(\Delta_1; L_1)$, and $(\Delta_2; K_2)$ **is** $(\Delta_2; L_2)$.
Let $(\Delta_1', \Delta_2') \supseteq (\Delta_1, \Delta_2)$ and assume $(\Delta_1'; A_1; K_1)$ **is** $(\Delta_2'; A_2; K_2)$. Then $(\Delta_1'; K_1)$ **is** $(\Delta_2'; K_2)$.

- Case $K_1 = K_2 = L_1 = L_2 = \mathbf{T}$. $(\Delta_1'; A_1; \mathbf{T})$ **is** $(\Delta_2'; A_2; \mathbf{T})$ by assumption.
- Case $K_1 = \mathbf{S}(B_1)$, $K_2 = \mathbf{S}(B_2)$, $L_1 = \mathbf{S}(C_1)$, and $L_2 = \mathbf{S}(C_2)$.
  1. By weakening, $\Delta_1' \triangleright B_1 :: \mathbf{T} \Leftrightarrow \Delta_1' \triangleright C_1 :: \mathbf{T}$
  2. and $\Delta_2' \triangleright B_2 :: \mathbf{T} \Leftrightarrow \Delta_2' \triangleright C_2 :: \mathbf{T}$
  3. and $\Delta_1' \triangleright C_1 :: \mathbf{T} \Leftrightarrow \Delta_2' \triangleright C_2 :: \mathbf{T}$.
  4. Similarly, $\Delta_1' \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_1' \triangleright B_1 :: \mathbf{T}$,
  5. $\Delta_2' \triangleright A_2 :: \mathbf{T} \Leftrightarrow \Delta_2' \triangleright B_2 :: \mathbf{T}$, and
  6. and $\Delta_1' \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2' \triangleright A_2 :: \mathbf{T}$.
  7. Thus by transitivity, $\Delta_1' \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_1' \triangleright C_1 :: \mathbf{T}$
  8. and $\Delta_2' \triangleright A_2 :: \mathbf{T} \Leftrightarrow \Delta_2' \triangleright C_2 :: \mathbf{T}$.
  9. Therefore $(\Delta_1'; A_1; \mathbf{S}(C_1))$ **valid**,
  10. $(\Delta_2'; A_2; \mathbf{S}(C_2))$ **valid**,
  11. and $(\Delta_1'; A_1; \mathbf{S}(C_1))$ **is** $(\Delta_2'; A_2; \mathbf{S}(C_2))$.
- Case: $K_1 = \Pi\alpha{::}K_1'.K_1''$, $K_2 = \Pi\alpha{::}K_2'.K_2''$, $L_1 = \Pi\alpha{::}L_1'.L_1''$, and $L_2 = \Pi\alpha{::}L_2'.L_2''$.
  1. Let $(\Delta_1'', \Delta_2'') \supseteq (\Delta_1', \Delta_2')$ and assume $(\Delta_1''; B_1; L_1')$ **is** $(\Delta_2''; B_2; L_2')$.
  2. By monotonicity, $(\Delta_1''; K_1')$ **is** $(\Delta_2''; K_2')$,
  3. $(\Delta_1''; L_1')$ **is** $(\Delta_2''; L_2')$,
  4. $(\Delta_1''; K_1')$ **is** $(\Delta_1''; L_1')$, and
  5. $(\Delta_2''; K_2')$ **is** $(\Delta_2''; L_2')$.
  6. By reflexivity and the inductive hypothesis, $(\Delta_1''; L_1' \leq K_1')$ **is** $(\Delta_2''; L_2' \leq K_2')$, $(\Delta_1''; L_1' \leq K_1')$ **is** $(\Delta_1''; L_1' \leq L_1')$, and $(\Delta_2''; L_2' \leq K_2')$ **is** $(\Delta_2''; L_2' \leq L_2')$.
  7. Thus $(\Delta_1''; B_1; K_1')$ **is** $(\Delta_2''; B_2; K_2')$.
  8. Since $(\Delta_1''; B_1; L_1')$ **is** $(\Delta_1''; B_1; L_1')$ and $(\Delta_2''; B_2; L_2')$ **is** $(\Delta_2''; B_2; L_2')$,
  9. we have $(\Delta_1''; B_1; K_1')$ **is** $(\Delta_1''; B_1; L_1')$,
  10. and $(\Delta_2''; B_2; K_2')$ **is** $(\Delta_2''; B_2; L_2')$.
  11. So, $(\Delta_1''; A_1\,B_1; [B_1/\alpha]K_1'')$ **is** $(\Delta_2''; A_2\,B_2; [B_2/\alpha]K_2'')$,
  12. $(\Delta_1''; [B_1/\alpha]K_1'')$ **is** $(\Delta_1''; [B_1/\alpha]L_1'')$,
  13. $(\Delta_1''; [B_1/\alpha]L_1'')$ **is** $(\Delta_2''; [B_2/\alpha]L_2'')$,
  14. and $(\Delta_2''; [B_2/\alpha]K_2'')$ **is** $(\Delta_2''; [B_2/\alpha]L_2'')$.
  15. By the inductive hypothesis, $(\Delta_1''; [B_1/\alpha]K_1'' \leq [B_1/\alpha]L_1'')$ **is** $(\Delta_2''; [B_2/\alpha]K_2'' \leq [B_2/\alpha]L_2'')$.

16. Thus $(\Delta_1''; A_1\, B_1; [B_1/\alpha]L_1'')$ **is** $(\Delta_2''; A_2\, B_2; [B_2/\alpha]L_2'')$.
17. Similar arguments show that $(\Delta_1'; A_1; \Pi\alpha{::}L_1'.L_1'')$ **valid** and $(\Delta_2'; A_2; \Pi\alpha{::}L_2'.L_2'')$ **valid**.
18. Therefore $(\Delta_1'; A_1; \Pi\alpha{::}L_1'.L_1'')$ **is** $(\Delta_2'; A_2; \Pi\alpha{::}L_2'.L_2'')$.

- Case: $K_1 = \Sigma\alpha{::}K_1'.K_1''$, $K_2 = \Sigma\alpha{::}K_2'.K_2''$, $L_1 = \Sigma\alpha{::}L_1'.L_1''$, and $L_2 = \Sigma\alpha{::}L_2'.L_2''$.
    1. $(\Delta_1'; \pi_1 A_1; K_1')$ **is** $(\Delta_2'; \pi_1 A_2; K_2')$.
    2. Also, $(\Delta_1'; K_1')$ **is** $(\Delta_2'; K_2')$,
    3. $(\Delta_1'; L_1')$ **is** $(\Delta_2'; L_2')$,
    4. $(\Delta_1'; K_1')$ **is** $(\Delta_1'; L_1')$,
    5. and $(\Delta_2'; K_2')$ **is** $(\Delta_2'; L_2')$.
    6. By the inductive hypothesis, $(\Delta_1'; K_1' \leq L_1')$ **is** $(\Delta_2'; K_2' \leq L_2')$,
    7. so $(\Delta_1'; \pi_1 A_1; L_1')$ **is** $(\Delta_2'; \pi_1 A_2; L_2')$.
    8. By similar considerations, $(\Delta_1'; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_1'; [\pi_1 A_1/\alpha]L_1'')$,
    9. $(\Delta_2'; [\pi_2 A_2/\alpha]K_2'')$ **is** $(\Delta_2'; [\pi_1 A_2/\alpha]L_1'')$,
    10. and $(\Delta_1'; [\pi_1 A_1/\alpha]L_1'')$ **is** $(\Delta_2'; [\pi_1 A_2/\alpha]L_2'')$.
    11. By the inductive hypothesis,
        $(\Delta_1'; [\pi_1 A_1/\alpha]K_1'' \leq [\pi_1 A_1/\alpha]L_1'')$ **is** $(\Delta_2'; [\pi_1 A_2/\alpha]K_2'' \leq [\pi_1 A_2/\alpha]L_2'')$.
    12. Since $(\Delta_1'; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_2'; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$,
    13. we have $(\Delta_1'; \pi_2 A_1; [\pi_1 A_1/\alpha]L_1'')$ **is** $(\Delta_2'; \pi_2 A_2; [\pi_1 A_2/\alpha]L_2'')$.
    14. Therefore $(\Delta_1'; A_1; \Sigma\alpha{::}L_1'.L_1'')$ **is** $(\Delta_2'; A_2; \Sigma\alpha{::}L_2'.L_2'')$.

∎

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

Lemma 5.3.5 (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** $(\Delta_1; \gamma_1; \Gamma_1)$.

Proof: Parts 1 and 2 are proved simultaneously by induction on the size of kinds. Part 3 then follows directly.

1. Assume $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$. Then $(\Delta_1; K_1)$ **valid** and $(\Delta_2; K_2)$ **valid**.
    - Case: $K_1 = K_2 = \mathbf{T}$. Trivial.
    - Case: $K_1 = \mathbf{S}(A_1)$, $K_2 = \mathbf{S}(A_2)$.
        (a) $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.
        (b) Inductively by part 2, $(\Delta_2; A_2; \mathbf{T})$ **is** $(\Delta_1; A_1; \mathbf{T})$.
        (c) Therefore $(\Delta_2; \mathbf{S}(A_2))$ **is** $(\Delta_1; \mathbf{S}(A_1))$.
    - Case: $K_1 = \Pi\alpha{::}K_1'.K_1''$ and $K_2 = \Pi\alpha{::}K_2'.K_2''$.
        (a) $(\Delta_1; K_1')$ **is** $(\Delta_2; K_2')$ by $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$.

(b) Inductively, $(\Delta_2; K_2')$ **is** $(\Delta_1; K_1')$.
(c) Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume $(\Delta_2'; A_2; K_2')$ **is** $(\Delta_1'; A_1; K_1')$.
(d) Inductively by part 2, $(\Delta_1'; A_1; K_1')$ **is** $(\Delta_2'; A_2; K_2')$.
(e) By $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$ again, $(\Delta_1'; [A_1/\alpha]K_1'')$ **is** $(\Delta_2'; [A_2/\alpha]K_2'')$
(f) By the inductive hypothesis again, $(\Delta_2'; [A_2/\alpha]K_2'')$ **is** $(\Delta_1'; [A_1/\alpha]K_1'')$.
(g) Therefore, $(\Delta_2; \Pi\alpha{::}K_2'.K_2'')$ **is** $(\Delta_1; \Pi\alpha{::}K_1'.K_1'')$.

- Case: $K_1 = \Sigma\alpha{::}K_1'.K_1''$ and $K_2 = \Sigma\alpha{::}K_2'.K_2''$. Same proof as for Π types.

2. Assume $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$. Then $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$, $(\Delta_1; A_1; K_1)$ **valid**, and $(\Delta_2; A_2; K_2)$ **valid**.

   By part 1, $(\Delta_2; K_2)$ **is** $(\Delta_1; K_1)$.

- Case $K_1 = K_2 = \mathbf{T}$.
  (a) $\Delta_1 \triangleright A_1 :: K_1 \Leftrightarrow \Delta_2 \triangleright A_2 :: K_2$
  (b) By Lemma 5.2.1, $\Delta_2 \triangleright A_2 :: K_2 \Leftrightarrow \Delta_1 \triangleright A_1 :: K_1$.
  (c) Therefore $(\Delta_2; A_2; \mathbf{T})$ **is** $(\Delta_1; A_1; \mathbf{T})$.
- Case $K_1 = \mathbf{S}(B_1)$ and $K_2 = \mathbf{S}(B_2)$.
  (a) $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.
  (b) By the inductive hypothesis, $(\Delta_2; A_2; \mathbf{T})$ **is** $(\Delta_1; A_1; \mathbf{T})$.
  (c) Therefore $(\Delta_2; A_2; \mathbf{S}(B_1))$ **is** $(\Delta_1; A_1; \mathbf{S}(B_2))$.
- Case $K_1 = \Pi\alpha{::}K_1'.K_1''$ and $K_2 = \Pi\alpha{::}K_2'.K_2''$.
  (a) Let $\Delta_2' \supseteq \Delta_2$ and $\Delta_1' \supseteq \Delta_1$ and assume $(\Delta_2'; B_2; K_2')$ **is** $(\Delta_1'; B_1; K_1')$.
  (b) By the inductive hypothesis, $(\Delta_1'; B_1; K_1')$ **is** $(\Delta_2'; B_2; K_2')$.
  (c) Thus $(\Delta_1'; A_1\,B_1; [B_1/\alpha]K_1'')$ **is** $(\Delta_2'; A_2\,B_2; [B_2/\alpha]K_2'')$.
  (d) By the inductive hypothesis, $(\Delta_2'; A_2\,B_2; [B_2/\alpha]K_2'')$ **is** $(\Delta_1'; A_1\,B_1; [B_1/\alpha]K_1'')$.
  (e) Therefore $(\Delta_2; A_2; \Pi\alpha{::}K_2'.K_2'')$ **is** $(\Delta_1; A_1; \Pi\alpha{::}K_1'.K_1'')$.
- Case $K_1 = \Sigma\alpha{::}K_1'.K_1''$ and $K_2 = \Sigma\alpha{::}K_2'.K_2''$.
  (a) Then $(\Delta_1; \pi_1 A_1; K_1')$ **is** $(\Delta_2; \pi_1 A_2; K_2')$
  (b) and $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$.
  (c) By the inductive hypothesis, $(\Delta_2; \pi_1 A_2; K_2')$ **is** $(\Delta_1; \pi_1 A_1; K_1')$
  (d) and $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$ **is** $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$.
  (e) Therefore $(\Delta_2; A_2; \Sigma\alpha{::}K_2'.K_2'')$ **is** $(\Delta_1; A_1; \Sigma\alpha{::}K_1'.K_1'')$.

∎

In contrast, the logical relation cannot be easily shown to obey the same transitivity property as the algorithmic relations; it does hold at the base kind but does not lift to function kinds. I 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*.

Lemma 5.3.6 (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** $(\Delta_2; A_2; K_2)$.

Proof:

1. Assume $(\Delta_1; K_1)$ **is** $(\Delta_1; L_1)$ and $(\Delta_1; L_1)$ **is** $(\Delta_2; K_2)$. First, $(\Delta_1; K_1)$ **valid** and $(\Delta_2; K_2)$ **valid**.

   - Case: $K_1 = L_1 = K_2 = \mathbf{T}$.
     $(\Delta_1; \mathbf{T})$ **is** $(\Delta_2; \mathbf{T})$ always.
   - Case: $K_1 = \mathbf{S}(A_1)$, $L_1 = \mathbf{S}(B_1)$, and $K_2 = \mathbf{S}(A_2)$.
     (a) Then $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_1 \triangleright B_1 :: \mathbf{T}$
     (b) and $\Delta_1 \triangleright B_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
     (c) By Lemma 5.2.1, $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
     (d) Therefore $(\Delta_1; \mathbf{S}(A_1))$ **is** $(\Delta_2; \mathbf{S}(A_2))$.
   - Case: $K_1 = \Pi\alpha{::}K'_1.K''_1$, $L_1 = \Pi\alpha{::}L'_1.L''_1$, and $K_2 = \Pi\alpha{::}K'_2.K''_2$.
     (a) $(\Delta_1; K'_1)$ **is** $(\Delta_1; L'_1)$ and $(\Delta_1; L'_1)$ **is** $(\Delta_2; K'_2)$.
     (b) By induction, $(\Delta_1; K'_1)$ **is** $(\Delta_2; K'_2)$.
     (c) Let $(\Delta'_1, \Delta'_2) \supseteq (\Delta_1, \Delta_2)$
     (d) and assume $(\Delta'_1; A_1; K'_1)$ **is** $(\Delta'_2; A_2; K'_2)$.
     (e) By Lemma 5.3.3, $(\Delta_1; K'_1)$ **is** $(\Delta_1; K'_1)$.
     (f) By monotonicity and Lemma 5.3.4, $(\Delta'_1; K'_1 \leq K'_1)$ **is** $(\Delta'_1; K'_1 \leq L'_1)$.
     (g) Since $(\Delta'_1; A_1; K'_1)$ **is** $(\Delta'_1; A_1; K'_1)$,
     (h) we have $(\Delta'_1; A_1; K'_1)$ **is** $(\Delta'_1; A_1; L'_1)$.
     (i) Thus $(\Delta'_1; [A_1/\alpha]K''_1)$ **is** $(\Delta'_1; [A_1/\alpha]L''_1)$.
     (j) Similarly, $(\Delta'_1; K'_1 \leq L'_1)$ **is** $(\Delta'_2; K'_2 \leq K'_2)$.
     (k) Then $(\Delta'_1; A_1; L'_1)$ **is** $(\Delta'_2; A_2; K'_2)$.
     (l) So, $(\Delta'_1; [A_1/\alpha]L''_1)$ **is** $(\Delta'_2; [A_2/\alpha]K''_2)$.
     (m) By induction, $(\Delta'_1; [A_1/\alpha]K''_1)$ **is** $(\Delta'_2; [A_2/\alpha]K''_2)$.
     (n) Therefore $(\Delta_1; \Pi\alpha{::}K'_1.K''_1)$ **is** $(\Delta_2; \Pi\alpha{::}K'_2.K''_2)$.
   - Case: $K_1 = \Sigma\alpha{::}K'_1.K''_1$, $L_1 = \Sigma\alpha{::}L'_1.L''_1$, and $K_2 = \Sigma\alpha{::}K'_2.K''_2$.
     Same proof as for Π types.

2. Assume $(\Delta_1; A_1; K_1)$ **is** $(\Delta_1; B_1; L_1)$ and $(\Delta_1; B_1; L_1)$ **is** $(\Delta_2; A_2; K_2)$. Then $(\Delta_1; A_1; K_1)$ **valid**, $(\Delta_2; A_2; K_2)$ **valid**, $(\Delta_1; K_1)$ **is** $(\Delta_1; L_1)$, and $(\Delta_1; L_1)$ **is** $(\Delta_2; K_2)$. By part 1, $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$.

   - Case: $K_1 = L_1 = K_2 = \mathbf{T}$.
     (a) $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_1 \triangleright B_1 :: \mathbf{T}$
     (b) and $\Delta_1 \triangleright B_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_1 :: \mathbf{T}$.
     (c) By Lemma 5.2.1, $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
     (d) Therefore $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.
   - Case: $K_1 = \mathbf{S}(A'_1)$, $L_1 = \mathbf{S}(B'_1)$, and $K_2 = \mathbf{S}(A'_2)$.

(a) $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_1; B_1; \mathbf{T})$

(b) and $(\Delta_1; B_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.

(c) By the inductive hypothesis, $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.

(d) Therefore $(\Delta_1; A_1; \mathbf{S}(A_1'))$ **is** $(\Delta_2; A_2; \mathbf{S}(A_2'))$.

- Case: $K_1 = \Pi\alpha{::}K_1'.K_1''$, $L_1 = \Pi\alpha{::}L_1'.L_1''$, and $K_2 = \Pi\alpha{::}K_2'.K_2''$.

  (a) Let $(\Delta_1', \Delta_2') \supseteq (\Delta_1, \Delta_2)$

  (b) and assume $(\Delta_1'; A_1'; K_1')$ **is** $(\Delta_2'; A_2'; K_2')$.

  (c) Then by monotonicity $(\Delta_1'; K_1')$ **is** $(\Delta_1'; L_1')$ and $(\Delta_1'; L_1')$ **is** $(\Delta_2'; K_2')$.

  (d) By Lemma 5.3.4, $(\Delta_1'; K_1' \leq K_1')$ **is** $(\Delta_1'; K_1' \leq L_1')$.

  (e) By Lemma 5.3.3, $(\Delta_1'; A_1'; K_1')$ **is** $(\Delta_1'; A_1'; K_1')$,

  (f) so $(\Delta_1'; A_1'; K_1')$ **is** $(\Delta_1'; A_1'; L_1')$.

  (g) Thus $(\Delta_1'; A_1\, A_1'; [A_1'/\alpha]K_1'')$ **is** $(\Delta_1'; B_1\, A_1'; [A_1'/\alpha]L_1'')$.

  (h) Similarly, $(\Delta_1'; K_1' \leq L_1')$ **is** $(\Delta_2'; K_2' \leq K_2')$,

  (i) so $(\Delta_1'; A_1'; L_1')$ **is** $(\Delta_2'; A_2'; K_2')$.

  (j) Thus, $(\Delta_1'; B_1\, A_1'; [A_1'/\alpha]L_1'')$ **is** $(\Delta_2'; A_2\, A_2'; [A_2'/\alpha]K_2'')$.

  (k) By the inductive hypothesis, $(\Delta_1'; A_1\, A_1'; [A_1'/\alpha]K_1'')$ **is** $(\Delta_2'; A_2\, A_2'; [A_2'/\alpha]K_2'')$.

  (l) Therefore, $(\Delta_1; A_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; A_2; \Pi\alpha{::}K_2'.K_2'')$.

- Case: $K_1 = \Sigma\alpha{::}K_1'.K_1''$, $L_1 = \Sigma\alpha{::}L_1'.L_1''$, and $K_2 = \Sigma\alpha{::}K_2'.K_2''$.

  (a) $(\Delta_1; \pi_1 A_1; K_1')$ **is** $(\Delta_1; \pi_1 B_1; L_1')$

  (b) and $(\Delta_1; \pi_1 B_1; L_1')$ **is** $(\Delta_2; \pi_1 A_2; K_2')$.

  (c) By the inductive hypothesis, $(\Delta_1; \pi_1 A_1; K_1')$ **is** $(\Delta_2; \pi_1 A_2; K_2')$.

  (d) Similarly, $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_1; \pi_2 B_1; [\pi_1 B_1/\alpha]L_1'')$

  (e) and $(\Delta_1; \pi_2 B_1; [\pi_1 B_1/\alpha]L_1'')$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$.

  (f) By the inductive hypothesis, $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$.

  (g) Therefore, $(\Delta_1; A_1; \Sigma\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; A_2; \Sigma\alpha{::}K_2'.K_2'')$.

∎

Because of this restricted formulation, I 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).

Definition 5.3.7
The judgment $\Gamma \triangleright A_1 \simeq A_2$ holds if and only if A_1 and A_2 have a common weak head reduct under typing context Γ; that is, if and only if there exists B such that $\Gamma \triangleright A_1 \leadsto^* B$ and $\Gamma \triangleright A_2 \leadsto^* B$.

Note that this definition does not require that either constructor have a weak head normal form, though if either constructor has one then they share the same one. The following lemma then shows that logical term equivalence and validity are preserved under weak head expansion and reduction.

Lemma 5.3.8 (Weak Head Closure)

1. *If $\Gamma \triangleright A \leadsto B$ then $\Gamma \triangleright \mathcal{E}[A] \leadsto \mathcal{E}[B]$*
2. *If $\Gamma \triangleright A_1 \simeq A_2$ then $\Gamma \triangleright \mathcal{E}[A_1] \simeq \mathcal{E}[A_2]$.*

3. *If* $(\Delta; A; K)$ **valid** *and* $\Delta \triangleright A' \simeq A$*, then* $(\Delta; A'; K)$ **valid***.*
4. *If* $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$*,* $\Delta_1 \triangleright A'_1 \simeq A_1$*, and* $\Delta_2 \triangleright A'_2 \simeq A_2$ *then* $(\Delta_1; A'_1; K_1)$ **is** $(\Delta_2; A'_2; K_2)$*.*

Proof:

1. Obvious by definition of $\Gamma \triangleright A \rightsquigarrow B$.
2. By repeated application of part 1.
3. Proved simultaneously with the following part by induction on the size of K. Assume $(\Delta; A; K)$ **valid** and $\Delta \triangleright A' \simeq A$. Note that $(\Delta; K)$ **valid**.
   - Case: $K = \mathbf{T}$.
     (a) $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright A :: \mathbf{T}$.
     (b) By the definition of the algorithm and determinacy of weak head reduction, $\Delta \triangleright A' :: \mathbf{T} \Leftrightarrow \Delta \triangleright A' :: \mathbf{T}$.
     (c) Therefore $(\Delta; A'; \mathbf{T})$ **valid**.
   - Case: $K = \mathbf{S}(B)$
     (a) Then $\Delta \triangleright A :: \mathbf{T} \Leftrightarrow \Delta \triangleright B :: \mathbf{T}$
     (b) so by the definition of the algorithm and determinacy of weak head reduction $\Delta \triangleright A' :: \mathbf{T} \Leftrightarrow \Delta \triangleright B :: \mathbf{T}$
     (c) which yields $(\Delta; A'; \mathbf{S}(B))$ **valid**
   - Case: $K = \Pi\alpha{::}K'.K''$.
     (a) Let $\Delta', \Delta'' \supseteq \Delta$ and assume that $(\Delta'; B_1; K')$ **is** $(\Delta''; B_2; K')$.
     (b) Then $(\Delta'; A\,B_1; [B_1/\alpha]K'')$ **is** $(\Delta''; A\,B_2; [B_2/\alpha]K'')$,
     (c) By part 2 and an obvious context weakening property, $\Delta' \triangleright A\,B_1 \simeq A'\,B_1$
     (d) and $\Delta'' \triangleright A\,B_2 \simeq A'\,B_2$.
     (e) By the inductive hypothesis, $(\Delta'; A'\,B_1; [B_1/\alpha]K'')$ **is** $(\Delta''; A'\,B_2; [B_2/\alpha]K'')$.
     (f) Therefore, $(\Delta; A'; \Pi\alpha{::}K'.K'')$ **valid**.
   - Case: $K = \Sigma\alpha{::}K'.K''$.
     (a) Then $(\Delta; \pi_1 A; K')$ **valid**
     (b) and by part 2, $\Delta \triangleright \pi_1 A' \simeq \pi_1 A$.
     (c) By the inductive hypothesis, $(\Delta_1; \pi_1 A'_1; K'_1)$ **valid**.
     (d) By reflexivity $(\Delta_1; \pi_1 A'_1; K'_1)$ **is** $(\Delta_1; \pi_1 A'_1; K'_1)$.
     (e) and inductively by part 4, $(\Delta; \pi_1 A; K')$ **is** $(\Delta; \pi_1 A'; K')$.
     (f) Similarly, $(\Delta_1; \pi_2 A; [\pi_1 A/\alpha]K'')$ **valid**,
     (g) and $\Delta \triangleright \pi_2 A' \simeq \pi_2 A$,
     (h) so by the inductive hypothesis again, $(\Delta; \pi_2 A'; [\pi_1 A/\alpha]K'')$ **valid**.
     (i) But $(\Delta; [\pi_1 A/\alpha]K'')$ **is** $(\Delta; [\pi_1 A'/\alpha]K'')$,
     (j) so by reflexivity and Lemma 5.3.4, $(\Delta; [\pi_1 A/\alpha]K'' \leq [\pi_1 A'/\alpha]K'')$ **is** $(\Delta; [\pi_1 A/\alpha]K'' \leq [\pi_1 A'/\alpha]K'')$.
     (k) so $(\Delta; \pi_2 A'; [\pi_1 A'/\alpha]K'')$ **valid**.
     (l) Therefore, $(\Delta; A'; \Sigma\alpha{::}K'.K'')$ **valid**.

4. Assume $(\Delta_1; A_1; K_1)$ **is** $(\Delta_2; A_2; K_2)$, $\Delta_1 \triangleright A'_1 \simeq A_1$, and $\Delta_2 \triangleright A'_2 \simeq A_2$. First, note that $(\Delta_1; A_1; K_1)$ **valid**, $(\Delta_2; A_2; K_2)$ **valid**, and $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$. By the argument in part 3, $(\Delta_1; A'_1; K_1)$ **valid** and $(\Delta_2; A'_2; K_2)$ **valid**.

- Case: $K_1 = K_2 = \mathbf{T}$.
  (a) $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
  (b) By the definition of the algorithm, $\Delta_1 \triangleright A'_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A'_2 :: \mathbf{T}$.
  (c) Therefore $(\Delta_1; A'_1; \mathbf{T})$ **is** $(\Delta_2; A'_2; \mathbf{T})$.
- Case: $K_1 = \mathbf{S}(B_1)$ and $K_2 = \mathbf{S}(B_2)$.
  (a) Then $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$
  (b) so $\Delta_1 \triangleright A'_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A'_2 :: \mathbf{T}$
  (c) which yields $(\Delta_1; A'_1; \mathbf{S}(B_1))$ **is** $(\Delta_2; A'_2; \mathbf{S}(B_2))$.
- Case: $K_1 = \Pi\alpha{::}K'_1.K''_1$ and $K_2 = \Pi\alpha{::}K'_2.K''_2$.
  (a) Let $\Delta'_1 \supseteq \Delta_1$ and $\Delta'_2 \supseteq \Delta_2$ and assume that $(\Delta'_1; B_1; K'_1)$ **is** $(\Delta'_2; B_2; K'_2)$.
  (b) Then $(\Delta'_1; A_1\, B_1; [B_1/\alpha]K''_1)$ **is** $(\Delta'_2; A_2\, B_2; [B_2/\alpha]K''_2)$,
  (c) By part 2 and an obvious weakening property, $\Delta'_1 \triangleright A_1\, B_1 \simeq A'_1\, B_1$
  (d) and $\Delta'_2 \triangleright A_2\, B_2 \simeq A'_2\, B_2$.
  (e) By the inductive hypothesis $(\Delta'_1; A'_1\, B_1; [B_1/\alpha]K''_1)$ **is** $(\Delta'_2; A'_2\, B_2; [B_2/\alpha]K''_2)$.
  (f) Therefore, $(\Delta_1; A'_1; \Pi\alpha{::}K'_1.K''_1)$ **is** $(\Delta_2; A'_2; \Pi\alpha{::}K'_2.K''_2)$.
- Case: $K_1 = \Sigma\alpha{::}K'_1.K''_1$ and $K_2 = \Sigma\alpha{::}K'_2.K''_2$.
  (a) Then $(\Delta_1; \pi_1 A_1; K'_1)$ **is** $(\Delta_2; \pi_1 A_2; K'_2)$,
  (b) $(\Delta_1; \pi_1 A_1; K'_1)$ **is** $(\Delta_1; \pi_1 A_1; K'_1)$,
  (c) $(\Delta_2; \pi_1 A_2; K'_2)$ **is** $(\Delta_2; \pi_1 A_2; K'_2)$,
  (d) and by part 2, $\Delta_1 \triangleright \pi_1 A'_1 \simeq \pi_1 A_1$,
  (e) and $\Delta_2 \triangleright \pi_1 A'_2 \simeq \pi_1 A_2$.
  (f) By the inductive hypothesis, $(\Delta_1; \pi_1 A'_1; K'_1)$ **is** $(\Delta_2; \pi_1 A'_2; K'_2)$,
  (g) $(\Delta_1; \pi_1 A_1; K'_1)$ **is** $(\Delta_1; \pi_1 A'_1; K'_1)$,
  (h) and $(\Delta_2; \pi_1 A_2; K'_2)$ **is** $(\Delta_2; \pi_1 A'_2; K'_2)$.
  (i) Similarly, $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K''_1)$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K''_2)$,
  (j) $\Delta_1 \triangleright \pi_2 A'_1 \simeq \pi_2 A_1$,
  (k) and $\Delta_2 \triangleright \pi_2 A'_2 \simeq \pi_2 A_2$.
  (l) By the inductive hypothesis again,
  $(\Delta_1; \pi_2 A'_1; [\pi_1 A_1/\alpha]K''_1)$ **is** $(\Delta_2; \pi_2 A'_2; [\pi_1 A_2/\alpha]K''_2)$.
  (m) But $(\Delta_1; K_1)$ **is** $(\Delta_1; K_1)$ and $(\Delta_2; K_2)$ **is** $(\Delta_2; K_2)$,
  (n) so $(\Delta_1; [\pi_1 A_1/\alpha]K''_1)$ **is** $(\Delta_1; [\pi_1 A'_1/\alpha]K''_1)$,
  (o) $(\Delta_2; [\pi_1 A_2/\alpha]K''_2)$ **is** $(\Delta_2; [\pi_1 A'_2/\alpha]K''_2)$,
  (p) and $(\Delta_1; [\pi_1 A'_1/\alpha]K''_1)$ **is** $(\Delta_2; [\pi_1 A'_2/\alpha]K''_2)$.
  (q) By Lemma 5.3.4,
  $(\Delta_1; [\pi_1 A_1/\alpha]K''_1 \leq [\pi_1 A'_1/\alpha]K''_1)$ **is** $(\Delta_2; [\pi_1 A_1/\alpha]K''_2 \leq [\pi_1 A'_1/\alpha]K''_2)$.
  (r) so $(\Delta_1; \pi_2 A'_1; [\pi_1 A'_1/\alpha]K''_1)$ **is** $(\Delta_2; \pi_2 A'_2; [\pi_1 A'_2/\alpha]K''_2)$.
  (s) Therefore, $(\Delta_1; A'_1; \Sigma\alpha{::}K'_1.K''_1)$ **is** $(\Delta_2; A'_2; \Sigma\alpha{::}K'_2.K''_2)$.

$\blacksquare$

Following all this preliminary work, I can now show that equivalence under the logical relations implies equivalence under the algorithm. This requires a strengthened induction hypothesis: that under suitable conditions variables (and more generally paths) are logically valid/equivalent.

Lemma 5.3.9

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

Proof: By simultaneous induction on the size of the kinds involved.
For part 4, note that in all cases $\Delta_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Delta_1 \triangleright p_1 \uparrow K_1$ and $\Delta_2 \triangleright p_2 \uparrow K_2 \leftrightarrow \Delta_2 \triangleright p_2 \uparrow K_2$ by symmetry and transitivity of the algorithm, $(\Delta_1; K_1)$ **valid**, and $(\Delta_2; K_2)$ **valid**. Hence by part 3, $(\Delta_1; p_1; K_1)$ **valid** and $(\Delta_2; p_2; K_2)$ **valid**.

- Case: $K = K_1 = K_2 = \mathbf{T}$.
  1. $\Delta_1 \triangleright \mathbf{T} \Leftrightarrow \Delta_2 \triangleright \mathbf{T}$ by the definition of the algorithm.
  2. (a) Assume $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.
     (b) By the definition of this relation, $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
  3. (a) Assume $(\Delta; \mathbf{T})$ **valid** and
     (b) $\Delta \triangleright p \uparrow \mathbf{T} \leftrightarrow \Delta \triangleright p \uparrow \mathbf{T}$.
     (c) By Lemma 5.2.2, $\Delta \triangleright p \uparrow \mathbf{T}$.
     (d) Then $\Delta \triangleright p \Downarrow p$ because p is a path without a definition.
     (e) so $\Delta \triangleright p :: \mathbf{T} \Leftrightarrow \Delta \triangleright p :: \mathbf{T}$.
     (f) Therefore $(\Delta; p; \mathbf{T})$ **valid**.
  4. (a) Assume $\Delta_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Delta_2 \triangleright p_2 \uparrow \mathbf{T}$
     (b) and $(\Delta_1; \mathbf{T})$ **is** $(\Delta_2; \mathbf{T})$.
     (c) By Lemma 5.2.2, $\Delta_1 \triangleright p_1 \uparrow \mathbf{T}$ and $\Delta_2 \triangleright p_2 \uparrow \mathbf{T}$.
     (d) Thus $\Delta_1 \triangleright p_1 \Downarrow p_1$ and $\Delta_2 \triangleright p_2 \Downarrow p_2$.
     (e) so $\Delta_1 \triangleright p_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright p_2 :: \mathbf{T}$.
     (f) Therefore $(\Delta_1; p_1; \mathbf{T})$ **is** $(\Delta_2; p_2; \mathbf{T})$.

- Case: $K = \mathbf{S}(B)$, $K_1 = \mathbf{S}(B_1)$, and $K_2 = \mathbf{S}(B_2)$.
  1. (a) Assume $(\Delta_1; K_1)$ **is** $(\Delta_2; K_2)$.
     (b) Then by definition $(\Delta_1; B_1; \mathbf{T})$ **is** $(\Delta_2; B_2; \mathbf{T})$,
     (c) so $\Delta_1 \triangleright B_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright B_2 :: \mathbf{T}$.
     (d) Therefore, $\Delta_1 \triangleright \mathbf{S}(B_1) \Leftrightarrow \Delta_2 \triangleright \mathbf{S}(B_2)$.
  2. (a) Then $(\Delta_1; A_1; \mathbf{T})$ **is** $(\Delta_2; A_2; \mathbf{T})$.
     (b) Thus $\Delta_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{T}$.
     (c) By the definition of the algorithm then, $\Delta_1 \triangleright A_1 :: \mathbf{S}(B_1) \Leftrightarrow \Delta_2 \triangleright A_2 :: \mathbf{S}(B_2)$

3. (a) Assume $(\Delta; \mathbf{S}(B))$ **valid**,
   (b) and $\Delta \triangleright p \uparrow \mathbf{S}(B) \leftrightarrow \Delta \triangleright p \uparrow \mathbf{S}(B)$.
   (c) By Lemma 5.2.2, $\Delta \triangleright p \uparrow \mathbf{S}(B)$.
   (d) Then $\Delta \triangleright p \rightsquigarrow B$ so $\Delta \triangleright p \simeq B$.
   (e) By $(\Delta; \mathbf{S}(B))$ **valid**, $\Delta \triangleright B :: \mathbf{T} \Leftrightarrow \Delta \triangleright B :: \mathbf{T}$.
   (f) By the definition of the algorithm, $\Delta \triangleright p :: \mathbf{T} \Leftrightarrow \Delta \triangleright B :: \mathbf{T}$.
   (g) Therefore $(\Delta; p; \mathbf{S}(B))$ **valid**.
4. (a) Assume $(\Delta_1; \mathbf{S}(B_1))$ **is** $(\Delta_2; \mathbf{S}(B_2))$,
   (b) and $\Delta_1 \triangleright p_1 \uparrow \mathbf{S}(B_1) \leftrightarrow \Delta_2 \triangleright p_2 \uparrow \mathbf{S}(B_1)$.
   (c) By definition of the logical relations, $\Delta_1 \triangleright B_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright B_2 :: \mathbf{T}$.
   (d) By Lemma 5.2.2, $\Delta_1 \triangleright p_1 \uparrow \mathbf{S}(B_1)$ and $\Delta_2 \triangleright p_2 \uparrow \mathbf{S}(B_2)$.
   (e) That is, $\Delta_1 \triangleright p_1 \rightsquigarrow B_1$ and $\Delta_2 \triangleright p_2 \rightsquigarrow B_1$.
   (f) Hence $\Delta_1 \triangleright p_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \triangleright p_2 :: \mathbf{T}$.
   (g) Therefore $(\Delta_1; p_1; \mathbf{S}(B_1))$ **is** $(\Delta_2; p_2; \mathbf{S}(B_1))$.

- Case: $K = \Pi\alpha{::}K'.K''$, $K_1 = \Pi\alpha{::}K_1'.K_1''$, and $K_2 = \Pi\alpha{::}K_2'.K_2''$.

1. (a) Assume $(\Delta_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; \Pi\alpha{::}K_2'.K_2'')$.
   (b) Then $(\Delta_1; K_1')$ **is** $(\Delta_2; K_2')$.
   (c) By the inductive hypothesis we have $\Delta_1 \triangleright K_1' \Leftrightarrow \Delta_2 \triangleright K_2'$.
   (d) Now $\Delta_1, \alpha{::}K_1' \triangleright \alpha \uparrow K_1' \leftrightarrow \Delta_2, \alpha{::}K_2' \triangleright \alpha \uparrow K_2'$.
   (e) Inductively by part 4, $(\Delta_1, \alpha{::}K_1'; \alpha; K_1')$ **is** $(\Delta_2, \alpha{::}K_2'; \alpha; K_2')$.
   (f) Thus $(\Delta_1, \alpha{::}K_1'; K_1'')$ **is** $(\Delta_2, \alpha{::}K_2'; K_2'')$
   (g) By the inductive hypothesis, $\Delta_1, \alpha{::}K_1' \triangleright K_1'' \Leftrightarrow \Delta_2, \alpha{::}K_2' \triangleright K_2''$.
   (h) Therefore $\Delta_1 \triangleright \Pi\alpha{::}K_1'.K_1'' \Leftrightarrow \Delta_2 \triangleright \Pi\alpha{::}K_2'.K_2''$.
2. (a) Assume $(\Delta_1; A_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; A_2; \Pi\alpha{::}K_2'.K_2'')$.
   (b) Then $(\Delta_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; \Pi\alpha{::}K_2'.K_2'')$
   (c) so as above, inductively by part 4 we have $(\Delta_1, \alpha{::}K_1'; \alpha; K_1')$ **is** $(\Delta_2, \alpha{::}K_2'; \alpha; K_2')$.
   (d) Then $(\Delta_1, \alpha{::}K_1'; A_1\,\alpha; K_1'')$ **is** $(\Delta_2, \alpha{::}K_2'; A_2\,\alpha; K_2'')$.
   (e) By the inductive hypothesis again, $\Delta_1, \alpha{::}K_1' \triangleright A_1\,\alpha :: K_1'' \Leftrightarrow \Delta_2, \alpha{::}K_2' \triangleright A_2\,\alpha :: K_2''$.
   (f) Therefore $\Delta_1 \triangleright A_1 :: \Pi\alpha{::}K_1'.K_1'' \Leftrightarrow \Delta_2 \triangleright A_2 :: \Pi\alpha{::}K_1'.K_1''$.
3. (a) Assume $(\Delta; K)$ **valid**
   (b) and $\Delta \triangleright p \uparrow K \leftrightarrow \Delta \triangleright p \uparrow K$.
   (c) Let $\Delta', \Delta'' \supseteq \Delta$
   (d) and assume $(\Delta'; B'; K')$ **is** $(\Delta''; B''; K')$.
   (e) Inductively by part 2, $\Delta' \triangleright B' :: K' \Leftrightarrow \Delta'' \triangleright B'' :: K'$.
   (f) Thus using Weakening, $\Delta' \triangleright p\,B' \uparrow [B'/\alpha]K'' \leftrightarrow \Delta'' \triangleright p\,B'' \uparrow [B''/\alpha]K''$.
   (g) By $(\Delta; K)$ **valid**, $(\Delta'; [B'/\alpha]K'')$ **is** $(\Delta''; [B''/\alpha]K'')$.
   (h) Inductively by part 4, $(\Delta'; p\,B'; [B'/\alpha]K'')$ **is** $(\Delta''; p\,B''; [B''/\alpha]K'')$.
   (i) Therefore $(\Delta; p; \Pi\alpha{::}K'.K'')$ **valid**.
4. (a) Assume $(\Delta_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; \Pi\alpha{::}K_2'.K_2'')$,
   (b) and $\Delta_1 \triangleright p_1 \uparrow \Pi\alpha{::}K_1'.K_1'' \leftrightarrow \Delta_2 \triangleright p_2 \uparrow \Pi\alpha{::}K_2'.K_2''$.

(c) Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume that $(\Delta_1'; B_1; K_1')$ **is** $(\Delta_2'; B_2; K_2')$.
(d) Then $(\Delta_1'; [B_1/\alpha]K_1'')$ **is** $(\Delta_2'; [B_2/\alpha]K_2'')$.
(e) Inductively by part 2, $\Delta_1' \triangleright B_1 :: K_1' \Leftrightarrow \Delta_2' \triangleright B_2 :: K_2'$,
(f) and by Weakening, $\Delta_1' \triangleright p_1 \uparrow \Pi\alpha{::}K_1'.K_1'' \leftrightarrow \Delta_2' \triangleright p_2 \uparrow \Pi\alpha{::}K_2'.K_2''$,
(g) so we have $\Delta_1' \triangleright p_1\, B_1 \uparrow [B_1/\alpha]K_1'' \leftrightarrow \Delta_2' \triangleright p_2\, B_2 \uparrow [B_2/\alpha]K_2''$.
(h) By the inductive hypothesis, $(\Delta_1'; p_1\, B_1; [B_1/\alpha]K_1'')$ **is** $(\Delta_2'; p_2\, B_2; [B_2/\alpha]K_2'')$.
(i) Therefore $(\Delta_1; p_1; \Pi\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; p_2; \Pi\alpha{::}K_2'.K_2'')$.

- Case: $K = \Sigma\alpha{::}K'.K''$, $K_1 = \Sigma\alpha{::}K_1'.K_1''$ and $K_2 = \Sigma\alpha{::}K_2'.K_2''$.

1. The corresponding argument for the Π case also applies here.
2. (a) Assume $(\Delta_1; A_1; \Sigma\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; A_2; \Sigma\alpha{::}K_2'.K_2'')$.
(b) Then $(\Delta_1; \pi_1 A_1; K_1')$ **is** $(\Delta_2; \pi_1 A_2; K_2')$.
(c) and $(\Delta_1; \pi_2 A_1; [\pi_1 A_1/\alpha]K_1'')$ **is** $(\Delta_2; \pi_2 A_2; [\pi_1 A_2/\alpha]K_2'')$.
(d) By the inductive hypothesis, $\Delta_1 \triangleright \pi_1 A_1 :: K_1' \Leftrightarrow \Delta_2 \triangleright \pi_1 A_2 :: K_2'$
(e) and $\Delta_1 \triangleright \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Delta_2 \triangleright \pi_2 A_2 :: [\pi_1 A_2/\alpha]K_2''$.
(f) Therefore $\Delta_1 \triangleright A_1 :: \Sigma\alpha{::}K_1'.K_1'' \Leftrightarrow \Delta_2 \triangleright A_2 :: \Sigma\alpha{::}K_2'.K_2''$.
3. (a) Assume $(\Delta; K)$ **valid**,
(b) and $\Delta \triangleright p \uparrow K \leftrightarrow \Delta \triangleright p \uparrow K$.
(c) By definition of the algorithm, $\Delta \triangleright \pi_1 p \uparrow K' \leftrightarrow \Delta \triangleright \pi_1 p \uparrow K'$
(d) and $\Delta \triangleright \pi_2 p \uparrow [\pi_1 p/\alpha]K'' \leftrightarrow \Delta \triangleright \pi_2 p \uparrow [\pi_1 p/\alpha]K''$.
(e) By the induction hypothesis, $(\Delta; \pi_1 p; K')$ **valid**.
(f) By Lemma 5.3.3, $(\Delta; \pi_1 p; K')$ **is** $(\Delta; \pi_1 p; K')$.
(g) By $(\Delta; K)$ **valid**, $(\Delta; [\pi_1 p/\alpha]K'')$ **is** $(\Delta; [\pi_1 p/\alpha]K'')$.
(h) Thus $(\Delta; [\pi_1 p/\alpha]K'')$ **valid**.
(i) By the induction hypothesis again, $(\Delta; \pi_2 p; [\pi_1 p/\alpha]K'')$ **valid**.
(j) Therefore, $(\Delta; p; \Sigma\alpha{::}K'.K'')$ **valid**.
4. (a) Assume $(\Delta_1; \Sigma\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; \Sigma\alpha{::}K_2'.K_2'')$,
(b) and $\Delta_1 \triangleright p_1 \uparrow \Sigma\alpha{::}K_1'.K_1'' \leftrightarrow \Delta_2 \triangleright p_2 \uparrow \Sigma\alpha{::}K_2'.K_2''$.
(c) Then $\Delta_1 \triangleright \pi_1 p_1 \uparrow K_1' \leftrightarrow \Delta_2 \triangleright \pi_1 p_2 \uparrow K_2'$
(d) and $\Delta_1 \triangleright \pi_2 p_1 \uparrow [\pi_1 p_1/\alpha]K_1'' \leftrightarrow \Delta_2 \triangleright \pi_2 p_2 \uparrow [\pi_1 p_2/\alpha]K_2''$.
(e) The inductive hypothesis applies, yielding $(\Delta_1; \pi_1 p_1; K_1')$ **is** $(\Delta_2; \pi_1 p_2; K_2')$
(f) and $(\Delta_1; \pi_2 p_1; [\pi_1 p_1/\alpha]K_1'')$ **is** $(\Delta_2; \pi_2 p_2; [\pi_1 p_2/\alpha]K_2'')$.
(g) Therefore $(\Delta_1; p_1; \Sigma\alpha{::}K_1'.K_1'')$ **is** $(\Delta_2; p_2; \Sigma\alpha{::}K_2'.K_2'')$.

∎

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 allow related substitutions, in order for the induction to go through.

Theorem 5.3.10 (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: By simultaneous induction on the hypothesized derivation.
Note that in all cases, $(\Delta_1;\gamma_1;\Gamma)$ **is** $(\Delta_1;\gamma_1;\Gamma)$ and $(\Delta_2;\gamma_2;\Gamma)$ **is** $(\Delta_2;\gamma_2;\Gamma)$.

Kind Well-formedness Rules: $\Gamma \vdash K$.

- Case: Rule 2.7.
  1. $\gamma_1\mathbf{T} = \gamma_2\mathbf{T} = \mathbf{T}$.
  2. $(\Delta_1;\mathbf{T})$ **is** $(\Delta_2;\mathbf{T})$.
- Case: Rule 2.8.
  1. By the inductive hypothesis, $(\Delta_1;\gamma_1 A;\mathbf{T})$ **is** $(\Delta_2;\gamma_2 A;\mathbf{T})$.
  2. Therefore $(\Delta_1;\mathbf{S}(\gamma_1 A))$ **is** $(\Delta_2;\mathbf{S}(\gamma_2 A))$.
- Case: Rule 2.9.
  1. By Proposition 3.1.1, there is a strict subderivation $\Gamma, \alpha{::}K' \vdash \mathrm{ok}$
  2. and by inversion a strict subderivation $\Gamma \vdash K'$.
  3. By the inductive hypothesis, $(\Delta_1;\gamma_1 K')$ **is** $(\Delta_2;\gamma_2 K')$.
  4. Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume that $(\Delta_1';A_1;\gamma_1 K')$ **is** $(\Delta_2';A_2;\gamma_2 K')$.
  5. Then by monotonicity $(\Delta_1';\gamma_1[\alpha{\mapsto}A_1];\Gamma,\alpha{::}K')$ **is** $(\Delta_2';\gamma_2[\alpha{\mapsto}A_2];\Gamma,\alpha{::}K')$.
  6. By the inductive hypothesis, $(\Delta_1';(\gamma_1[\alpha{\mapsto}A_1])K'')$ **is** $(\Delta_2';(\gamma_2[\alpha{\mapsto}A_2])K'')$.
  7. That is, $(\Delta_1';[A_1/\alpha]((\gamma_1[\alpha{\mapsto}\alpha])K''))$ **is** $(\Delta_2';[A_2/\alpha]((\gamma_2[\alpha{\mapsto}\alpha])K''))$.
  8. Therefore, $(\Delta_1;\gamma_1(\Pi\alpha{::}K'.K''))$ **is** $(\Delta_2;\gamma_2(\Pi\alpha{::}K'.K''))$.
- Case: Rule 2.10. Just like previous case.

Subkinding Rules: $\Gamma \vdash K_1 \leq K_2$. In all cases, the proofs that $(\Delta_1;\gamma_1 K_1)$ **is** $(\Delta_2;\gamma_2 K_1)$ and $(\Delta_1;\gamma_1 K_2)$ **is** $(\Delta_2;\gamma_2 K_2)$ follow essentially as in the proofs for the well-formedness rules.
Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume $(\Delta_1';B_1;\gamma_1 K_1)$ **is** $(\Delta_2';B_2;\gamma_2 K_1)$.

- Case: Rule 2.11. $K_1 = \mathbf{S}(A)$ and $K_2 = \mathbf{T}$. By monotonicity and the definitions of the logical relations.
- Case: Rule 2.12. $K_1 = \mathbf{S}(A_1)$ and $K_2 = \mathbf{S}(A_2)$, with $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$.
  1. By the inductive hypothesis we have $(\Delta_1';\gamma_1 A_2;\mathbf{T})$ **is** $(\Delta_2';\gamma_2 A_2;\mathbf{T})$,

2. $(\Delta_1'; \gamma_1 A_1; \mathbf{T})$ **is** $(\Delta_1'; \gamma_1 A_2; \mathbf{T})$,
3. and $(\Delta_2'; \gamma_2 A_1; \mathbf{T})$ **is** $(\Delta_2'; \gamma_2 A_2; \mathbf{T})$.
4. Thus $(\Delta_1'; \mathbf{S}(\gamma_1 A_2))$ **is** $(\Delta_2'; \mathbf{S}(\gamma_2 A_2))$,
5. $(\Delta_1'; \mathbf{S}(\gamma_1 A_1))$ **is** $(\Delta_1'; \mathbf{S}(\gamma_1 A_2))$,
6. and $(\Delta_2'; \mathbf{S}(\gamma_2 A_1))$ **is** $(\Delta_2'; \mathbf{S}(\gamma_2 A_2))$.
7. so by Lemma 5.3.4, $(\Delta_1'; \mathbf{S}(\gamma_1 A_1) \leq \mathbf{S}(\gamma_1 A_2))$ **is** $(\Delta_2'; \mathbf{S}(\gamma_2 A_1) \leq \mathbf{S}(\gamma_2 A_2))$.
8. Therefore $(\Delta_1'; B_1; \mathbf{S}(\gamma_1 A_2))$ **is** $(\Delta_2'; B_2; \mathbf{S}(\gamma_2 A_2))$.

- Case: Rule 2.13. $K_1 = K_2 = \mathbf{T}$.

  Trivial, since $\gamma_1 \mathbf{T} = \gamma_2 \mathbf{T} = \mathbf{T}$ and $(\Delta_1; \mathbf{T})$ **is** $(\Delta_2; \mathbf{T})$.

- Case: Rule 2.14. $K_1 = \Pi\alpha{::}K_1'.K_1''$ and $K_2 = \Pi\alpha{::}K_2'.K_2''$ with $\Gamma \vdash K_2' \leq K_1'$ and $\Gamma, \alpha{::}K_2' \vdash K_1'' \leq K_2''$.

  1. Let $\Delta_1'' \supseteq \Delta_1'$ and $\Delta_2'' \supseteq \Delta_2'$ and assume $(\Delta_1''; B_1'; \gamma_1 K_2')$ **is** $(\Delta_2''; B_2'; \gamma_2 K_2')$.
  2. By the inductive hypothesis, $(\Delta_1'; \gamma_1 K_2' \leq \gamma_1 K_1')$ **is** $(\Delta_2'; \gamma_2 K_2' \leq \gamma_2 K_1')$.
  3. so $(\Delta_1''; B_1'; \gamma_1 K_1')$ **is** $(\Delta_2''; B_2'; \gamma_2 K_1')$
  4. and $(\Delta_1''; B_1\, B_1'; (\gamma_1[\alpha{\mapsto}B_1'])K_1'')$ **is** $(\Delta_2''; B_2\, B_2'; (\gamma_2[\alpha{\mapsto}B_2'])K_1'')$.
  5. By monotonicity, $(\Delta_1''; \gamma_1[\alpha{\mapsto}B_1']; \Gamma, \alpha{::}K_2')$ **is** $(\Delta_2''; \gamma_2[\alpha{\mapsto}B_2']; \Gamma, \alpha{::}K_2')$.
  6. By the inductive hypothesis again,
     $(\Delta_1''; (\gamma_1[\alpha{\mapsto}B_1'])K_1'' \leq (\gamma_1[\alpha{\mapsto}B_1'])K_2'')$ **is** $(\Delta_2''; (\gamma_2[\alpha{\mapsto}B_2'])K_1'' \leq (\gamma_2[\alpha{\mapsto}B_2'])K_2'')$,
  7. so $(\Delta_1''; B_1\, B_1'; (\gamma_1[\alpha{\mapsto}B_1'])K_2'')$ **is** $(\Delta_2''; B_2\, B_2'; (\gamma_2[\alpha{\mapsto}B_2'])K_2'')$.
  8. Thus $(\Delta_1'; B_1; \gamma_1(\Pi\alpha{::}K_2'.K_2''))$ **is** $(\Delta_2'; B_2; \gamma_2(\Pi\alpha{::}K_2'.K_2''))$.

- Case: Rule 2.15. $K_1 = \Sigma\alpha{::}K_1'.K_1''$ and $K_2 = \Sigma\alpha{::}K_2'.K_2''$ with $\Gamma \vdash K_1' \leq K_2'$ and $\Gamma, \alpha{::}K_1' \vdash K_1'' \leq K_2''$.

  1. By the definitions of the logical relations, $(\Delta_1'; \pi_1 B_1; \gamma_1 K_1')$ **is** $(\Delta_2'; \pi_1 B_2; \gamma_2 K_1')$.
  2. By the inductive hypothesis, $(\Delta_1'; \gamma_1 K_1' \leq \gamma_1 K_2')$ **is** $(\Delta_2'; \gamma_2 K_1' \leq \gamma_2 K_2')$.
  3. Thus $(\Delta_1'; \pi_1 B_1; \gamma_1 K_2')$ **is** $(\Delta_2'; \pi_1 B_2; \gamma_2 K_2')$.
  4. Now $(\Delta_1'; \gamma_1[\alpha{\mapsto}\pi_1 B_1]; \Gamma, \alpha{::}K_1')$ **is** $(\Delta_2'; \gamma_2[\alpha{\mapsto}\pi_1 B_2]; \Gamma, \alpha{::}K_1')$
  5. so by the inductive hypothesis, $(\Delta_1'; (\gamma_1[\alpha{\mapsto}\pi_1 B_1])K_1'' \leq (\gamma_1[\alpha{\mapsto}\pi_1 B_1])K_2'')$ **is** $(\Delta_2'; (\gamma_2[\alpha{\mapsto}\pi_1 B_2])K_1'' \leq (\gamma_2[\alpha{\mapsto}\pi_1 B_2])K_2'')$.
  6. Since $(\Delta_1'; \pi_2 B_1; (\gamma_1[\alpha{\mapsto}\pi_1 B_1])K_1'')$ **is** $(\Delta_2'; \pi_2 B_2; (\gamma_2[\alpha{\mapsto}\pi_1 B_2])K_1'')$,
  7. $(\Delta_1'; \pi_2 B_1; (\gamma_1[\alpha{\mapsto}\pi_1 B_1])K_2'')$ **is** $(\Delta_2'; \pi_2 B_2; (\gamma_2[\alpha{\mapsto}\pi_1 B_2])K_2'')$.
  8. Therefore, $(\Delta_1'; B_1; \gamma_1(\Sigma\alpha{::}K_2'.K_2''))$ **is** $(\Delta_2'; B_2; \gamma_2(\Sigma\alpha{::}K_2'.K_2''))$.

Kind Equivalence Rules: $\Gamma \vdash K_1 \equiv K_2$.
It suffices to prove that 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** $(\Delta_2; \gamma_2 K_2)$, because we can apply this to get $(\Delta_2; \gamma_2 K_1)$ **is** $(\Delta_2; \gamma_2 K_2)$, so $(\Delta_1; \gamma_1 K_1)$ **is** $(\Delta_2; \gamma_2 K_1)$ follows by symmetry and transitivity. A similar argument yields $(\Delta_1; \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_2)$.
In all cases, the proofs that $(\Delta_1; \gamma_1 K_1)$ **is** $(\Delta_2; \gamma_2 K_1)$ and $(\Delta_1; \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_2)$ follow essentially as in the proofs for the well-formedness rules.

- Case: Rule 2.16. $K_1 = K_2 = \mathbf{T}$. $(\Delta_1; \mathbf{T})$ **is** $(\Delta_2; \mathbf{T})$ by the definition of the logical relation.
- Case: Rule 2.17. $K_1 = \mathbf{S}(A_1)$ and $K_2 = \mathbf{S}(A_2)$ with $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$.

  1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A_1; \mathbf{T})$ **is** $(\Delta_2; \gamma_2 A_2; \mathbf{T})$.
  2. Therefore, $(\Delta_1; \mathbf{S}(\gamma_1 A_1))$ **is** $(\Delta_2; \mathbf{S}(\gamma_2 A_2))$.

- Case: Rule 2.18. $K_1 = \Pi\alpha{::}K_1'.K_1''$ and $K_2 = \Pi\alpha{::}K_2'.K_2''$ with $\Gamma \vdash K_2' \leq K_1'$ and $\Gamma, \alpha{::}K_2' \vdash K_1'' \leq K_2''$.

  1. By the inductive hypothesis, $(\Delta_1; \gamma_1 K_1')$ **is** $(\Delta_2; \gamma_2 K_2')$.
  2. Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$
  3. and assume $(\Delta_1'; A_1; \gamma_1 K_1')$ **is** $(\Delta_2'; A_2; \gamma_2 K_2')$.
  4. By the inductive hypothesis, $(\Delta_1'; \gamma_1 K_1')$ **is** $(\Delta_2'; \gamma_2 K_2')$
  5. and $(\Delta_2'; \gamma_2 K_1')$ **is** $(\Delta_2'; \gamma_2 K_2')$.
  6. By symmetry, $(\Delta_2'; \gamma_2 K_2')$ **is** $(\Delta_2'; \gamma_2 K_1')$,
  7. and by reflexivity $(\Delta_1'; \gamma_1 K_1')$ **is** $(\Delta_1'; \gamma_1 K_1')$.
  8. By Lemma 5.3.4, $(\Delta_1'; \gamma_1 K_1' \leq \gamma_1 K_1')$ **is** $(\Delta_2'; \gamma_2 K_2' \leq \gamma_2 K_1')$,
  9. so $(\Delta_1'; A_1; \gamma_1 K_1')$ **is** $(\Delta_2'; A_2; \gamma_2 K_1')$.
  10. By monotonicity, then, $(\Delta_1'; \gamma_1[\alpha{\mapsto}A_1]; \Gamma, \alpha{::}K_1')$ **is** $(\Delta_2'; \gamma_2[\alpha{\mapsto}A_2]; \Gamma, \alpha{::}K_1')$.
  11. By the inductive hypothesis again, $(\Delta_1'; (\gamma_1[\alpha{\mapsto}A_1])K_1'')$ **is** $(\Delta_2'; (\gamma_2[\alpha{\mapsto}A_2])K_2'')$.
  12. Therefore $(\Delta_1; \gamma_1(\Pi\alpha{::}K_1'.K_1''))$ **is** $(\Delta_2; \gamma_2(\Pi\alpha{::}K_2'.K_2''))$.

- Case: Rule 2.19. Same proof as for previous case.

Constructor Validity Rules: $\Gamma \vdash A :: K$.

- Case: Rule 2.20.

  1. $(\Delta_1; \mathbf{T})$ **is** $(\Delta_2; \mathbf{T})$
  2. $\Delta_1 \triangleright b_i \uparrow \mathbf{T} \leftrightarrow \Delta_2 \triangleright b_i \uparrow \mathbf{T}$.
  3. Thus by Lemma 5.3.9 we have $(\Delta_1; b_i; \mathbf{T})$ **is** $(\Delta_2; b_i; \mathbf{T})$.

- Case: Rule 2.21. Analogous to the previous case.
- Case: Rule 2.22. Analogous to the previous case.
- Case: Rule 2.23.

  By the assumptions for γ_1 and γ_2, we have $(\Delta_1; \gamma_1\alpha; \gamma_1(\Gamma(\alpha)))$ **is** $(\Delta_2; \gamma_2\alpha; \gamma_2(\Gamma(\alpha)))$.
- Case: Rule 2.24.

  1. By Proposition 3.1.1 there is a strict subderivation $\Gamma \vdash K'$.
  2. By the inductive hypothesis, $(\Delta_1; \gamma_1 K')$ **is** $(\Delta_2; \gamma_2 K')$.
  3. Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume $(\Delta_1'; B_1; \gamma_1 K')$ **is** $(\Delta_2'; B_2; \gamma_2 K')$.
  4. Using monotonicity, $(\Delta_1'; \gamma_1[\alpha{\mapsto}B_1]; \Gamma, \alpha{::}K')$ **is** $(\Delta_2'; \gamma_2[\alpha{\mapsto}B_2]; \Gamma, \alpha{::}K')$.
  5. By the inductive hypothesis,
     $(\Delta_1'; (\gamma_1[\alpha{\mapsto}B_1])A; (\gamma_1[\alpha{\mapsto}B_1])K'')$ **is** $(\Delta_2'; (\gamma_2[\alpha{\mapsto}B_2])A; (\gamma_2[\alpha{\mapsto}B_2])K'')$.

6. Now $\Delta_1 \triangleright (\gamma_1[\alpha{\mapsto}B_1])A \simeq (\gamma_1(\lambda\alpha{::}K'.A))B_1$
7. and $\Delta_2 \triangleright (\gamma_2[\alpha{\mapsto}B_2])A \simeq (\gamma_2(\lambda\alpha{::}K'.A))B_2$.
8. By Lemma 5.3.8, $(\Delta_1'; (\gamma_1(\lambda\alpha{::}K'.A))B_1; (\gamma_1[\alpha{\mapsto}B_1])K'')$ **is** $(\Delta_2'; (\gamma_2(\lambda\alpha{::}K'.A))B_2; (\gamma_2[\alpha{\mapsto}B_2])K'')$.
9. Similar arguments analogous to lines 3–8 (and reflexivity) show that $(\Delta_1; \gamma_1(\lambda\alpha{::}K'.A); \gamma_1(\Pi\alpha{::}K'.K''))$ **valid**
10. and $(\Delta_2; \gamma_2(\lambda\alpha{::}K'.A); \gamma_2(\Pi\alpha{::}K'.K''))$ **valid**.
11. Therefore $(\Delta_1; \gamma_1(\lambda\alpha{::}K'.A); \gamma_1(\Pi\alpha{::}K'.K''))$ **is** $(\Delta_2; \gamma_2(\lambda\alpha{::}K'.A); \gamma_2(\Pi\alpha{::}K'.K''))$.

- Case: Rule 2.25

1. By the inductive hypothesis $(\Delta_1; \gamma_1 A; \gamma_1(K'{\to}K''))$ **is** $(\Delta_2; \gamma_2 A; \gamma_2(K'{\to}K''))$
2. and $(\Delta_1; \gamma_1 A'; \gamma_1 K')$ **is** $(\Delta_2; \gamma_2 A'; \gamma_2 K')$.
3. Therefore, $(\Delta_1; \gamma_1(A\,A'); \gamma_1(K''))$ **is** $(\Delta_2; \gamma_2(A\,A'); \gamma_2(K''))$.

- Case: Rule 2.26.

1. By the inductive hypothesis and reflexivity, $(\Delta_1; \gamma_1 A_1; \gamma_1 K')$ **valid**
2. and $(\Delta_1; \gamma_1 A_2; \gamma_1 K'')$ **valid**.
3. Now $\Delta_1 \triangleright \gamma_1 A_1 \simeq \pi_1\langle\gamma_1 A_1, \gamma_1 A_2\rangle$
4. and $\Delta_1 \triangleright \gamma_1 A_2 \simeq \pi_2\langle\gamma_1 A_1, \gamma_1 A_2\rangle$.
5. By Lemma 5.3.8 we have $(\Delta_1; \pi_1\langle\gamma_1 A_1, \gamma_1 A_2\rangle; \gamma_1 K')$ **valid**,
6. $(\Delta_1; \pi_2\langle\gamma_1 A_1, \gamma_1 A_2\rangle; \gamma_1 K'')$ **valid**
7. Therefore, $(\Delta_1; \langle\gamma_1 A_1, \gamma_1 A_2\rangle; \gamma_1(K'{\times}K''))$ **valid**
8. A very similar argument shows that $(\Delta_2; \langle\gamma_2 A_1, \gamma_2 A_2\rangle; \gamma_2(K'{\times}K''))$ **valid**
9. and an analogous argument shows that $(\Delta_1; \langle\gamma_1 A_1, \gamma_1 A_2\rangle; \gamma_1(K'{\times}K''))$ **is** $(\Delta_2; \langle\gamma_2 A_1, \gamma_2 A_2\rangle; \gamma_2(K'{\times}K''))$.

- Case: Rule 2.27.

1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A; \gamma_1(\Sigma\alpha{::}K'.K''))$ **is** $(\Delta_2; \gamma_2 A; \gamma_2(\Sigma\alpha{::}K'.K''))$.
2. Therefore $(\Delta_1; \pi_1\gamma_1 A; \gamma_1 K')$ **is** $(\Delta_2; \pi_1\gamma_2 A; \gamma_2 K')$.

- Case: Rule 2.28.

1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A; \gamma_1(\Sigma\alpha{::}K'.K''))$ **is** $(\Delta_2; \gamma_2 A; \gamma_2(\Sigma\alpha{::}K'.K''))$.
2. Therefore $(\Delta_1; \pi_2\gamma_1 A; \gamma_1([\pi_1 A/\alpha]K''))$ **is** $(\Delta_2; \pi_2\gamma_2 A; \gamma_2([\pi_1 A/\alpha]K''))$.

- Case: Rule 2.29

1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A; \mathbf{T})$ **is** $(\Delta_2; \gamma_2 A; \mathbf{T})$.
2. As in the case for Rule 2.8, $(\Delta_1; \mathbf{S}(\gamma_1 A))$ **is** $(\Delta_2; \mathbf{S}(\gamma_2 A))$.
3. Thus $(\Delta_1; \gamma_1 A; \mathbf{S}(\gamma_1 A))$ **valid**,

4. $(\Delta_2; \gamma_2 A; \mathbf{S}(\gamma_2 A))$ **valid**,
5. and $(\Delta_1; \gamma_1 A; \mathbf{S}(\gamma_1 A))$ **is** $(\Delta_2; \gamma_2 A; \mathbf{S}(\gamma_2 A))$.

- Case: Rule 2.30.

1. By the inductive hypothesis, $(\Delta_1; \pi_1(\gamma_1 A); \gamma_1 K')$ **is** $(\Delta_2; \pi_1(\gamma_2 A); \gamma_2 K')$,
2. and $(\Delta_1; \pi_2(\gamma_1 A); \gamma_1 K'')$ **is** $(\Delta_2; \pi_2(\gamma_2 A); \gamma_2 K'')$.
3. Thus $(\Delta_1; \gamma_1 A; \gamma_1(K' \times K''))$ **valid**,
4. $(\Delta_2; \gamma_2 A; \gamma_2(K' \times K''))$ **valid**,
5. and therefore $(\Delta_1; \gamma_1 A; \gamma_1(K' \times K''))$ **is** $(\Delta_2; \gamma_2 A; \gamma_2(K' \times K''))$,

- Case: Rule 2.31

1. $(\Delta_1; \gamma_1(\Pi\alpha{::}K'.K''))$ **is** $(\Delta_2; \gamma_2(\Pi\alpha{::}K'.K''))$ as in the case for Rule 2.9.
2. Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$
3. and assume $(\Delta_1'; B_1; \gamma_1 K')$ **is** $(\Delta_2'; B_2; \gamma_2 K')$.
4. By monotonicity, $(\Delta_1'; \gamma_1[\alpha \mapsto B_1]; \Gamma, \alpha{::}K')$ **is** $(\Delta_2'; \gamma_2[\alpha \mapsto B_2]; \Gamma, \alpha{::}K')$.
5. By the inductive hypothesis,
$(\Delta_1'; (\gamma_1[\alpha \mapsto B_1])(A\,\alpha); (\gamma_1[\alpha \mapsto B_1])K'')$ **is** $(\Delta_2'; (\gamma_2[\alpha \mapsto B_2])(A\,\alpha); (\gamma_2[\alpha \mapsto B_2])K'')$.
6. That is, $(\Delta_1'; (\gamma_1 A)B_1; (\gamma_1[\alpha \mapsto B_1])K'')$ **is** $(\Delta_2'; (\gamma_2 A)B_2; (\gamma_2[\alpha \mapsto B_2])K'')$.
7. and $(\Delta_1; \gamma_1 A; \gamma_1(\Pi\alpha{::}K'.K''))$ **is** $(\Delta_2; \gamma_2 A; \gamma_2(\Pi\alpha{::}K'.K''))$.

- Case: Rule 2.32

1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A; \gamma_1 K_1)$ **is** $(\Delta_1; \gamma_2 A; \gamma_2 K_1)$
2. and $(\Delta_1; \gamma_1 K_1 \leq \gamma_1 K_2)$ **is** $(\Delta_2; \gamma_2 K_1 \leq \gamma_2 K_2)$.
3. Therefore, $(\Delta_1; \gamma_1 A; \gamma_1 K_2)$ **is** $(\Delta_1; \gamma_2 A; \gamma_2 K_2)$

Constructor Equivalence Rules: $\Gamma \vdash A_1 \equiv A_2 :: K$.
It suffices to prove that 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** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$, because it follows that $(\Delta_2; \gamma_2 A_1; \gamma_2 K)$ **is** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$, so $(\Delta_1; \gamma_1 A_1; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$ by symmetry and transitivity. A similar argument yields $(\Delta_1; \gamma_1 A_2; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_2; \gamma_2 K)$.

- Case: Rule 2.33. By the inductive hypothesis.
- Case: Rule 2.34.

  By the inductive hypothesis and Lemma 5.3.5.
- Case: Rule 2.35.

1. By the inductive hypothesis, $(\Delta_1; \gamma_1 A_1; \gamma_1 K)$ **is** $(\Delta_1; \gamma_1 A_2; \gamma_1 K)$
2. and $(\Delta_1; \gamma_1 A_2; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_3; \gamma_2 K)$.
3. By Lemma 5.3.6, $(\Delta_1; \gamma_1 A_1; \gamma_1 K)$ **is** $(\Delta_2; \gamma_2 A_3; \gamma_2 K)$.

- Case: Rule 2.36.

  Analogous to the proof for rule 2.24.

- Case: Rule 2.37.

  Analogous to the proof for Rule 2.25.
- Case: Rule 2.38.

  Analogous to the proof for Rule 2.27.
- Case: Rule 2.39.

  Analogous to proof for Rule 2.28.
- Case: Rule 2.40.

  Analogous to proof for Rule 2.26.
- Case: Rule 2.41.

  Analogous to the proof for Rule 2.30.
- Case: Rule 2.42.

  Analogous to the proof of Rule 2.31.
- Case: Rule 2.43.

  By the inductive hypothesis and the definition of the logical relations.
- Case: Rule 2.44. By the inductive hypothesis.

∎

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

Lemma 5.3.11
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.*

Proof: By induction on the proof of $\Gamma \vdash \mathrm{ok}$.

- Case: Empty context. Vacuous.
- Case: $\Gamma, \alpha{::}K$.

  1. By Proposition 3.1.1, $\Gamma \vdash K$, and $\Gamma \vdash \mathrm{ok}$.
  2. Also, $\alpha \notin \mathrm{dom}(\Gamma)$.
  3. By the inductive hypothesis, $(\Gamma; \beta; \Gamma(\beta))$ **is** $(\Gamma; \beta; \Gamma(\beta))$ for all $\beta \in \mathrm{dom}(\Gamma)$.
  4. By monotonicity, $(\Gamma, \alpha{::}K; \beta; ((\Gamma, \alpha{::}K)(\beta)))$ **is** $(\Gamma, \alpha{::}K; \beta; ((\Gamma, \alpha{::}K)(\beta)))$ for all $\beta \in \mathrm{dom}(\Gamma)$.
  5. By Theorem 5.3.10, $(\Gamma; K)$ **is** $(\Gamma; K)$
  6. and by monotonicity $(\Gamma, \alpha{::}K; K)$ **is** $(\Gamma, \alpha{::}K; K)$
  7. Now $\Gamma, \alpha{::}K \rhd \alpha \uparrow K \leftrightarrow \Gamma, \alpha{::}K \rhd \alpha \uparrow K$,
  8. so by Lemma 5.3.9, $(\Gamma, \alpha{::}K; \alpha; K)$ **is** $(\Gamma, \alpha{::}K; \alpha; K)$.

∎

This yields the completeness result for the equivalence algorithms:

Corollary 5.3.12 (Completeness)

1. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $(\Gamma; K_1)$ **is** $(\Gamma; K_2)$.
2. *If* $\Gamma \vdash A_1 \equiv A_2 :: K$ *then* $(\Gamma; A_1; K)$ **is** $(\Gamma; A_2; K)$.
3. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $\Gamma \rhd K_1 \Leftrightarrow \Gamma \rhd K_2$.
4. *If* $\Gamma \vdash A_1 \equiv A_2 :: K$ *then* $\Gamma \rhd A_1 :: K \Leftrightarrow \Gamma \rhd A_2 :: K$.

Proof:

1,2 By Lemma 5.3.11, we can apply Theorem 5.3.10 with γ_1 and γ_2 being identity substitutions.

3,4 Follows directly from parts 1 and 2 and Lemma 5.3.9.

∎

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 (see §5.5). Thus 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 5.3.13

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

Proof: By induction on algorithmic derivations. ∎

Then completeness yields the following corollary.

Corollary 5.3.14 (Algorithmic Decidability)

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

Proof: By reflexivity, Corollary 5.3.12, and by Lemma 5.3.13. ∎

I conclude this section with an application of completeness.

Proposition 5.3.15 (Consistency)
Assume c_1 *and* c_2 *are distinct type constructor constants. Then the judgment*

$$\Gamma \vdash \mathcal{E}_1[c_1] \equiv \mathcal{E}_2[c_2] :: K$$

is not provable.

Proof: The MIL_0 constructor constants have either kind $\mathbf{T}$ or $\mathbf{T}\to(\mathbf{T}\to\mathbf{T})$, so any path with a constant at its head cannot have its extracted kind be a singleton kind, and hence must be head-normal. Also, two paths with distinct constants at their heads will not be equivalent according to the algorithmic weak constructor equivalence. Therefore the paths will be algorithmically inequivalent at kind K, which by completeness implies 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, every closed value of type int is an integer constant would fail if the type int were provably equivalent to a function type, a product type, or another base type.

5.4 Completeness and Termination

Finally, I transfer the soundness and completeness results of the previous section back to the original algorithm for constructor equivalence. I use a "size" metric for derivations in the six-place equivalence system. This metric measures the size of the derivation ignoring head reduction, head normalization, and kind equivalence steps; that is, the metric is the number of term or path equivalence rules used directly in the derivation. Since every provable algorithmic judgment has at most one derivation, I can refer unambiguously to the size of a judgment.

The important properties of this metric are summarized in the following two lemmas.

Lemma 5.4.1

1. *If* $\Gamma_1 \rhd A_1 :: K_1 \Leftrightarrow \Gamma_2 \rhd A_2 :: K_2$ *and* $\Gamma_1 \rhd A_1 :: K_1 \Leftrightarrow \Gamma_3 \rhd A_3 :: K_3$ *then the two derivations have equal sizes.*
2. *If* $\Gamma_1 \rhd A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \rhd A_2 \uparrow K_2$ *and* $\Gamma_1 \rhd A_1 \uparrow K_1 \leftrightarrow \Gamma_3 \rhd A_3 \uparrow K_3$ *then the two derivations have equal sizes.*

Proof: [By induction on the hypothesized derivations]

- Assume $\Gamma_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \rhd A_2 :: \mathbf{T}$ and $\Gamma_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Gamma_3 \rhd A_3 :: \mathbf{T}$. Then $\Gamma_1 \rhd A_1 \Downarrow p_1$, $\Gamma_2 \rhd A_2 \Downarrow p_2$, $\Gamma_3 \rhd A_3 \Downarrow p_3$, $\Gamma_1 \rhd p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd p_2 \uparrow \mathbf{T}$, and $\Gamma_1 \rhd p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_3 \rhd p_3 \uparrow \mathbf{T}$. By the inductive hypothesis, these last two algorithmic judgments have equal sizes, so the original equivalences have equal sizes (greater by one).
- Assume $\Gamma_1 \rhd A_1 :: \mathbf{S}(B_1) \Leftrightarrow \Gamma_2 \rhd A_2 :: \mathbf{S}(B_2)$ and $\Gamma_1 \rhd A_1 :: \mathbf{S}(B_1) \Leftrightarrow \Gamma_3 \rhd A_3 :: \mathbf{S}(B_3)$. Then the derivations both have a size of one.
- Assume $\Gamma_1 \rhd A_1 :: \Pi\alpha{::}A_1'.A_1'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Pi\alpha{::}A_2'.A_2''$ and $\Gamma_1 \rhd A_1 :: \Pi\alpha{::}A_1'.A_1'' \Leftrightarrow \Gamma_3 \rhd A_3 :: \Pi\alpha{::}A_3'.A_3''$. Then $\Gamma_1, \alpha{::}K_1' \rhd A_1\,\alpha :: K_1'' \Leftrightarrow \Gamma_2, \alpha{::}K_2' \rhd A_2\,\alpha :: K_2''$ and $\Gamma_1, \alpha{::}K_1' \rhd A_1\,\alpha :: K_1'' \Leftrightarrow \Gamma_3, \alpha{::}K_2' \rhd A_3\,\alpha :: K_3''$. By the inductive hypothesis these derivations have equal sizes and hence the original equivalence judgments have equal sizes (greater by one).
- Assume $\Gamma_1 \rhd A_1 :: \Sigma\alpha{::}A_1'.A_1'' \Leftrightarrow \Gamma_2 \rhd A_2 :: \Sigma\alpha{::}A_2'.A_2''$ and $\Gamma_1 \rhd A_1 :: \Sigma\alpha{::}A_1'.A_1'' \Leftrightarrow \Gamma_3 \rhd A_3 :: \Sigma\alpha{::}A_3'.A_3''$. Then $\Gamma_1 \rhd \pi_1 A_1 :: K_1' \Leftrightarrow \Gamma_2 \rhd \pi_1 A_2 :: K_2'$, $\Gamma_1 \rhd \pi_1 A_1 :: K_1' \Leftrightarrow \Gamma_3 \rhd \pi_1 A_3 :: K_3'$, $\Gamma_1 \rhd \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Gamma_2 \rhd \pi_1 A_2 :: [\pi_1 A_2/\alpha]K_2''$, and $\Gamma_1 \rhd \pi_2 A_1 :: [\pi_1 A_1/\alpha]K_1'' \Leftrightarrow \Gamma_3 \rhd \pi_1 A_3 :: [\pi_1 A_3/\alpha]K_3''$. Using the inductive hypothesis twice, the judgments have equal sizes.
- Assume $\Gamma_1 \rhd b_i \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \rhd b_i \uparrow \mathbf{T}$ and $\Gamma_1 \rhd b_i \uparrow \mathbf{T} \leftrightarrow \Gamma_3 \rhd b_i \uparrow \mathbf{T}$. Both derivations have size one.
- Assume $\Gamma_1 \rhd \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_2 \rhd \alpha \uparrow \Gamma_2(\alpha)$ and $\Gamma_1 \rhd \alpha \uparrow \Gamma_1(\alpha) \leftrightarrow \Gamma_3 \rhd \alpha \uparrow \Gamma_3(\alpha)$. Both derivations have size one.
- The remaining three cases follow directly by the inductive hypothesis.

∎

Lemma 5.4.2

1. *If* $\Gamma_1 \rhd A_1 :: K_1 \Leftrightarrow \Gamma_2 \rhd A_2 :: K_2$ *then the derivation* $\Gamma_2 \rhd A_2 :: K_2 \Leftrightarrow \Gamma_1 \rhd A_1 :: K_1$ *has the same size.*

2. *If* $\Gamma_1 \triangleright A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright A_2 \uparrow K_2$ *then the derivation* $\Gamma_2 \triangleright A_2 \uparrow K_2 \leftrightarrow \Gamma_1 \triangleright A_1 \uparrow K_1$ *has the same size.*

Proof: The two derivations are mirror-images of each other, and hence use the same number of rules of each kind. ∎

I can then show the completeness of the four-place algorithm with respect to the six-place algorithm.

Lemma 5.4.3

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* $\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$ *then* $\Gamma_1 \triangleright A_1 \Leftrightarrow A_2 :: K_1$.
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* $\Gamma_1 \triangleright A_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright A_2 \uparrow K_2$ *then* $\Gamma_1 \triangleright A_1 \leftrightarrow A_2 \uparrow K_1$.

Proof: [By induction on the **size** of the hypothesized algorithmic derivation.]
Assume $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1 \equiv K_2$, $\Gamma_1 \vdash A_1 :: K_1$, and $\Gamma_2 \vdash A_2 :: K_2$.

- Case: $\Gamma_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \triangleright A_2 :: \mathbf{T}$ because $\Gamma_1 \triangleright A_1 \Downarrow p_1$, $\Gamma_2 \triangleright A_2 \Downarrow p_2$, and $\Gamma_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \mathbf{T}$.

  Now by the completeness of the six-place algorithm we have $\Gamma_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Gamma_1 \triangleright A_2 :: \mathbf{T}$, where $\Gamma_1 \triangleright A_2 \Downarrow p_2'$ and $\Gamma_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_1 \triangleright p_2' \uparrow \mathbf{T}$.

  By Lemma 5.4.1, the sizes of the two proofs of algorithmic path equivalence have equal sizes. Since this size is less than the size of the original algorithmic judgment (by one), we may apply the inductive hypothesis to the second derivation to get $\Gamma_1 \triangleright p_1 \leftrightarrow p_2' \uparrow \mathbf{T}$. Therefore, $\Gamma_1 \triangleright A_1 \Leftrightarrow A_2 :: \mathbf{T}$.
- The remaining cases are all either trivial or follow easily from the inductive hypothesis.

∎

Theorem 5.4.4 (Completeness for Constructors and Kinds)

1. *If* $\Gamma \vdash A_1 \equiv A_2 :: K$ *then* $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$.
2. *If* $\Gamma \vdash K$ *then* $\Gamma \triangleright K$.
3. *If* $\Gamma \vdash K_1 \leq K_2$ *then* $\Gamma \triangleright K_1 \leq K_2$.
4. *If* $\Gamma \vdash K_1 \equiv K_2$ *then* $\Gamma \triangleright K_1 \Leftrightarrow K_2$.
5. *If* $\Gamma \vdash A :: K$ *then* $\Gamma \triangleright A \rightrightarrows L$ *and* $\Gamma \triangleright A \Uparrow L$.
6. *If* $\Gamma \vdash A :: K$ *then* $\Gamma \triangleright A \leftleftarrows K$.

Proof:

1. Assume $\Gamma \vdash A_1 \equiv A_2 :: K$. By the completeness of the six-place algorithm, $\Gamma \triangleright A_1 :: K \Leftrightarrow \Gamma \triangleright A_2 :: K$. Then $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$ by Lemma 5.4.3.

2–6. By part 1 and induction on derivations

∎

Lemma 5.4.5
If $\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow K_1$, $\Gamma \vdash p_1 :: K_1$, *and* $\Gamma \vdash p_2 :: L$ *then* $\Gamma \triangleright p_2 \uparrow K_2$ *for some kind* K_2, *and* $\Gamma \vdash K_1 \equiv K_2$.

Lemma 5.4.6

1. *If* $\Gamma \triangleright p_1 \leftrightarrow p_1 \uparrow K_1$, $\Gamma \vdash p_1 :: K_1$, *and* $\Gamma \vdash p_2 :: L$ *then it is decidable whether* $\Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow K_1$ *is provable.*
2. *If* $\Gamma \triangleright A_1 \Leftrightarrow A_1 :: K$, $\Gamma \vdash A_1 :: K$ *and* $\Gamma \vdash A_2 :: K$ *then it is decidable whether* $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$ *is provable.*
3. *If* $\Gamma \triangleright K_1 \Leftrightarrow K_1$, $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$ *then it is decidable whether* $\Gamma \triangleright K_1 \Leftrightarrow K_2$ *is provable.*

Proof:

1–2. By induction on algorithmic derivations.

The sequence of constructor and path comparisons is driven by Γ and either p_1 or A_1 and K. In particular, this is independent of A_2 or p_2. Thus the only possible problem would be for head normalization to fail to terminate, which can be seen to be impossible by completeness of the revised algorithm.

3. By induction on kinds, using part 2.

∎

Theorem 5.4.7 (Decidability for Constructors and Kinds)

1. *If* $\Gamma \vdash A_1 :: K$ *and* $\Gamma \vdash A_2 :: K$ *then* $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: K$ *is decidable.*
2. *If* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$ *then* $\Gamma \triangleright K_1 \Leftrightarrow K_2$ *is decidable.*
3. *If* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$ *then* $\Gamma \triangleright K_1 \leq K_2$ *is decidable.*
4. *If* $\Gamma \vdash K_1$, $\Gamma \vdash K_2$ *then* $\Gamma \triangleright K_1 \Leftrightarrow K_2$ *is decidable.*
5. *If* $\Gamma \vdash$ *ok then* $\Gamma \triangleright K$ *is decidable.*
6. *If* $\Gamma \vdash$ *ok then it is decidable whether* $\Gamma \triangleright A \rightrightarrows K$ *holds for some* K*.*
7. *If* $\Gamma \vdash K$ *then* $\Gamma \triangleright A \leftleftarrows K$ *is decidable.*

Proof:

1–2. Follows from reflexivity of constructor and kind equivalence, Completeness, and Lemma 5.4.6.

3–7. By Parts 1 and 2 and by induction on the sizes of constructors and kinds.

∎

5.5 Normalization

The revised equivalence algorithms in Figure 5.1 are effectively doing the work of normalizing the two constructors or two kinds being compared. However, because the algorithm interleaves this process with comparisons, the normalized constructors and kinds need not be explicitly constructed. This is a beneficial for implementations, but it is still interesting and useful to consider the normalization process in isolation. The corresponding algorithms are shown in Figure 5.5.

Lemma 5.5.1 (Determinacy of Normalization)

1. *If* $\Gamma \triangleright A :: K \Longrightarrow B_1$ *and* $\Gamma \triangleright A :: K \Longrightarrow B_2$ *then* $B_1 = B_2$.
2. *If* $\Gamma \triangleright p \longrightarrow p_1' \uparrow K_1$ *and* $\Gamma \triangleright p \longrightarrow p_2' \uparrow K_2$ *then* $p_1' = p_2'$ *and* $K_1 = K_2$.

Constructor Normalization

$\Gamma \triangleright A :: \mathbf{T} \Longrightarrow A''$	if $\Gamma \triangleright A \Downarrow A'$ and $\Gamma \triangleright A' \longrightarrow A'' \uparrow \mathbf{T}$
$\Gamma \triangleright A :: \mathbf{S}(B) \Longrightarrow A''$	if $\Gamma \triangleright A \Downarrow A'$ and $\Gamma \triangleright A' \longrightarrow A'' \uparrow \mathbf{T}$
$\Gamma \triangleright A :: \Pi\alpha{::}K'.K'' \Longrightarrow \lambda\alpha{::}L'.B$	if $\Gamma \triangleright K' \Longrightarrow L'$ and $\Gamma, \alpha{::}K' \triangleright (A\,\alpha) :: K'' \Longrightarrow B$
$\Gamma \triangleright A :: \Sigma\alpha{::}K'.K'' \Longrightarrow \langle B', B''\rangle$	if $\Gamma \triangleright \pi_1 A :: K' \Longrightarrow B'$ and $\Gamma \triangleright \pi_2 A :: [\pi_1 A/\alpha]K'' \Longrightarrow B''$.

Path Normalization

$\Gamma \triangleright b \longrightarrow b \uparrow \mathbf{T}$	
$\Gamma \triangleright \times \longrightarrow \times \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$	
$\Gamma \triangleright \rightharpoonup \longrightarrow \rightharpoonup \uparrow \mathbf{T}{\to}\mathbf{T}{\to}\mathbf{T}$	
$\Gamma \triangleright \alpha \longrightarrow \alpha \uparrow \Gamma(\alpha)$	
$\Gamma \triangleright p\,A \longrightarrow p'\,A' \uparrow [A/\alpha]K''$	if $\Gamma \triangleright p \longrightarrow p' \uparrow \Pi\alpha{::}K'.K''$ and $\Gamma \triangleright A :: K' \Longrightarrow A'$
$\Gamma \triangleright \pi_1 p \longrightarrow \pi_1 p' \uparrow K'$	if $\Gamma \triangleright p \longrightarrow p' \uparrow \Sigma\alpha{::}K'.K''$
$\Gamma \triangleright \pi_2 p \longrightarrow \pi_2 p' \uparrow [\pi_1 p/\alpha]K'$	if $\Gamma \triangleright p \longrightarrow p' \uparrow \Sigma\alpha{::}K'.K''$

Kind Normalization

$\Gamma \triangleright \mathbf{T} \Longrightarrow \mathbf{T}$	
$\Gamma \triangleright \mathbf{S}(A) \Longrightarrow \mathbf{S}(A')$	if $\Gamma \triangleright A :: \mathbf{T} \Longrightarrow A'$
$\Gamma \triangleright \Pi\alpha{::}K'.K'' \Longrightarrow \Pi\alpha{::}L.L''$	if $\Gamma \triangleright K' \Longrightarrow L'$ and $\Gamma, \alpha{::}K' \triangleright K'' \Longrightarrow L''$
$\Gamma \triangleright \Sigma\alpha{::}K'.K'' \Longrightarrow \Sigma\alpha{::}L.L''$	if $\Gamma \triangleright K' \Longrightarrow L'$ and $\Gamma, \alpha{::}K' \triangleright K'' \Longrightarrow L''$

Figure 5.5: Constructor and Kind Normalization

3. *If $\Gamma \triangleright K \Longrightarrow L_1$ and $\Gamma \triangleright K \Longrightarrow L_2$ then $L_1 = L_2$.*

Proof: By induction on algorithmic derivations. ∎

Lemma 5.5.2 (Soundness of Normalization)

1. *If $\Gamma \vdash A :: K$ and $\Gamma \triangleright A :: K \Longrightarrow B$ then $\Gamma \vdash A \equiv B :: K$.*
2. *If $\Gamma \vdash p :: K$ and $\Gamma \triangleright p \longrightarrow p' \uparrow L$ then $\Gamma \vdash p \equiv p' :: L$.*
3. *If $\Gamma \vdash K$ and $\Gamma \triangleright K \Longrightarrow L$ then $\Gamma \vdash K \equiv L$.*

Proof: By induction on algorithmic derivations. ∎

Theorem 5.5.3
Assume $\vdash \Gamma_1 \equiv \Gamma_2$ and $\Gamma_1 \vdash K_1 \equiv K_2$.

1. *$\Gamma_1 \triangleright A_1 :: K_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: K_2$ if and only if $\Gamma_1 \triangleright A_1 :: K_1 \Longrightarrow B$ and $\Gamma_2 \triangleright A_2 :: K_2 \Longrightarrow B$ for some B.*
2. *$\Gamma_1 \triangleright p_1 \uparrow K_1 \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow K_2$ if and only if $\Gamma_1 \triangleright p_1 \longrightarrow p' \uparrow K_1$ and $\Gamma_1 \triangleright p_1 \longrightarrow p' \uparrow K_2$ for some p', K_1, and K_2.*
3. *$\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ if and only if $\Gamma_1 \triangleright K_1 \Longrightarrow L$ and $\Gamma_2 \triangleright K_2 \Longrightarrow L$ for some L.*

Proof:

$\Rightarrow$ By induction on algorithmic derivations.

$\Leftarrow$ By soundness of normalization, transitivity and symmetry, and completeness of the revised equivalence algorithm.

∎

Corollary 5.5.4 (Normalization of Constructors and Kinds)

1. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash A_1 :: K$ *and* $\Gamma_2 \vdash A_2 :: K$ *then* $\Gamma_1 \vdash A_1 \equiv A_2 :: K$ *if and only if* $\Gamma_1 \triangleright A_1 :: K \Longrightarrow B$ *and* $\Gamma_2 \triangleright A_2 :: K \Longrightarrow B$.
2. *If* $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash K_1$ *and* $\Gamma_1 \vdash K_2$ *then* $\Gamma_1 \vdash K_1 \equiv K_2$ *if and only if* $\Gamma_1 \triangleright K_1 \Longrightarrow L$ *and* $\Gamma_2 \triangleright K_2 \Longrightarrow L$.

Chapter 6

Algorithms for Type and Term Judgments

6.1 Introduction

I now turn to the term and type levels of MIL_0; the development parallels that for constructors and kinds. In this chapter I consider algorithms corresponding to the term and type judgments, proving soundness, and partial completeness and termination results depending on *term* equivalence. Term equivalence is then studied in detail in the following chapter.

6.2 Type Head-Normalization

The kind-equivalence and subkinding relations are very simple and structural, and inversion immediately yields various useful properties such as "if two Π kinds are equivalent then their domain kinds are equivalent and their codomain kinds are equivalent". It is clear from inspection of type equivalence that a universally-quantified type can only be equivalent to another universally-quantified type (and that in this case the domain kinds are equivalent as are the codomain types), and similar properties hold for singleton types. However, the fact that there is no chain of equivalences

$$Ty(A_1)\times Ty(A_2) \equiv Ty(A_1\times A_2) \equiv Ty(B_1\rightharpoonup B_2) \equiv Ty(B_1)\rightharpoonup Ty(B_2)$$

equating a function type with a product type (or a chain equating a product type and $Ty(\mathsf{Int})$, etc.) is a consequence of the consistency properties of constructor equivalence, which were proved in the previous chapter.

It is convenient to extend the head-normalization algorithm for constructors to the head-normalization of types; this algorithm is shown in Figure 6.1. The head-normalization algorithm attempts to turn any type of the form $Ty(A)$ into an equivalent function type or product type, and leaves all other types unchanged. Viewed as an algorithm the judgment $\Gamma \triangleright \tau \Downarrow \sigma$ takes inputs Γ and τ with $\Gamma \vdash \tau$ and produces the type σ. It depends upon a typing context because it uses the constructor head-normalization, which is context-dependent.

Lemma 6.2.1 (Type Head-Normalization)
If $\Gamma \vdash \tau$ then there exists a unique σ such that $\Gamma \triangleright \tau \Downarrow \sigma$. Furthermore, $\Gamma \vdash \tau \equiv \sigma$.

Proof: By induction on the derivation of type well-formedness, using the soundness of weak head-reduction for constructors. ∎

Type head normalization

$\Gamma \triangleright Ty(A) \Downarrow Ty(A_1) \times Ty(A_2)$	if $\Gamma \triangleright A \Downarrow A_1 \times A_2$
$\Gamma \triangleright Ty(A) \Downarrow Ty(A_1) \rightharpoonup Ty(A_2)$	if $\Gamma \triangleright A \Downarrow A_1 \rightharpoonup A_2$
$\Gamma \triangleright \tau \Downarrow \tau$	otherwise

Figure 6.1: Head Normalization Algorithm for Types

Use of head-normalization allows a sufficiently strong induction hypothesis to prove useful inversion properties for type equivalence and for subtyping.

Theorem 6.2.2 (Inversion of Type Equivalence)
Assume $\Gamma \vdash \tau_1 \equiv \tau_2$.

1. $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \rightharpoonup \tau_1''$ *if and only if* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \rightharpoonup \tau_2''$. *Furthermore, in this case* $\Gamma \vdash \tau_1' \equiv \tau_2'$ *and* $\Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2''$.
2. $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \times \tau_1''$ *if and only if* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \times \tau_2''$. *Furthermore, in this case* $\Gamma \vdash \tau_1' \equiv \tau_2'$ *and* $\Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2''$.
3. $\Gamma \triangleright \tau_1 \Downarrow Ty(b)$ *if and only if* $\Gamma \triangleright \tau_2 \Downarrow Ty(b)$.
4. $\tau_1 = \forall\alpha{::}K_1'.\tau_1''$ *if and only if* $\tau_2 = \forall\alpha{::}K_2'.\tau_2''$. *Furthermore, in this case* $\Gamma \vdash K_1' \equiv K_2'$ *and* $\Gamma, \alpha{::}K_1' \vdash \tau_1'' \equiv \tau_2''$.
5. $\tau_1 = \mathbf{S}(v_1 : \tau_1')$ *if and only if* $\tau_2 = \mathbf{S}(v_2 : \tau_2')$. *Furthermore, in this case* $\Gamma \vdash v_1 \equiv v_2 : \tau_1'$ *and* $\Gamma \vdash \tau_1' \equiv \tau_2'$.

Proof: By induction on the proof of $\Gamma \vdash \tau_1 \equiv \tau_2$. ∎

Theorem 6.2.3 (Subtyping Inversion)
Assume $\Gamma \vdash \tau_1 \leq \tau_2$.

1. *If* $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \rightharpoonup \tau_1''$ *then* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \rightharpoonup \tau_2''$. *Furthermore, in this case* $\Gamma \vdash \tau_2' \leq \tau_1'$ *and* $\Gamma, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''$.
2. *If* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \rightharpoonup \tau_2''$ *then* τ_1 *is a singleton type or else* $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \rightharpoonup \tau_1''$ *and* $\Gamma \vdash \tau_2' \leq \tau_1'$ *and* $\Gamma, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''$.
3. *If* $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \times \tau_1''$ *then* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \times \tau_2''$. *Furthermore, in this case* $\Gamma \vdash \tau_1' \leq \tau_2'$ *and* $\Gamma, x{:}\tau_1' \vdash \tau_1'' \leq \tau_2''$.
4. *If* $\Gamma \triangleright \tau_2 \Downarrow (x{:}\tau_2') \times \tau_2''$ *then* τ_1 *is a singleton type or else* $\Gamma \triangleright \tau_1 \Downarrow (x{:}\tau_1') \times \tau_1''$ *and* $\Gamma \vdash \tau_1' \leq \tau_2'$ *and* $\Gamma, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''$.
5. *If* $\Gamma \triangleright \tau_1 \Downarrow Ty(b)$ *then* $\Gamma \triangleright \tau_2 \Downarrow Ty(b)$.
6. *If* $\Gamma \triangleright \tau_2 \Downarrow Ty(b)$ *then* τ_1 *is a singleton type or else* $\Gamma \triangleright \tau_2 \Downarrow Ty(b)$.
7. *If* $\tau_1 = \forall\alpha{::}K_1'.\tau_1''$ *then* $\tau_2 = \forall\alpha{::}K_2'.\tau_2''$ *and* $\Gamma \vdash K_2' \leq K_1'$ *and* $\Gamma, \alpha{::}K_2' \vdash \tau_1'' \leq \tau_2''$.
8. *If* $\tau_2 = \forall\alpha{::}K_2'.\tau_2''$ *then* τ_1 *is a singleton type or else* $\tau_1 = \forall\alpha{::}K_1'.\tau_1''$ *and* $\Gamma \vdash K_2' \leq K_1'$ *and* $\Gamma, \alpha{::}K_2' \vdash \tau_1'' \leq \tau_2''$.
9. *If* $\tau_1 = \mathbf{S}(v_1 : \sigma_1)$ *then either* $\tau_2 = \mathbf{S}(v_2 : \sigma_2)$, $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *and* $\Gamma \vdash v_1 \equiv v_2 : \sigma_1$, *or else* τ_2 *is not a singleton and* $\Gamma \vdash \sigma_1 \leq \tau_2$.
10. *If* $\tau_2 = \mathbf{S}(v_2 : \sigma_2)$ *then* $\tau_1 = \mathbf{S}(v_1 : \sigma_1)$, $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *and* $\Gamma \vdash v_1 \equiv v_2 : \sigma_1$.

Proof: By induction on the proof of $\Gamma \vdash \tau_1 \leq \tau_2$. ∎

Singleton stripping

$(\mathbf{S}(v : \tau))^{\$} := \tau$	
$\tau^{\$} := \tau$	if τ is not a singleton

Principal type synthesis

$\Gamma \triangleright n \Uparrow \mathbf{S}(n : \mathsf{int})$	
$\Gamma \triangleright x \Uparrow \mathbf{S}(x : \Gamma(x)^{\$})$	
$\Gamma \triangleright \mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e \Uparrow$ $\mathbf{S}((\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e) : (x{:}\tau')\rightharpoonup\tau'')$	
$\Gamma \triangleright \Lambda(\alpha{::}K){:}\tau.e \Uparrow \mathbf{S}(\Lambda(\alpha{::}K){:}\tau.e : \forall\alpha{::}K.\tau)$	
$\Gamma \triangleright \langle v_1, v_2\rangle \Uparrow \mathbf{S}(\langle v_1, v_2\rangle : \tau_1{\times}\tau_2)$	if $\Gamma \triangleright v_1 \Uparrow \tau_1$ and $\Gamma \triangleright v_2 \Uparrow \tau_2$.
$\Gamma \triangleright \pi_1 v \Uparrow \mathbf{S}(\pi_1 v : \tau'^{\$})$	if $\Gamma \triangleright v \Uparrow \tau$ and $\Gamma \triangleright \tau^{\$} \Downarrow (x{:}\tau')\times\tau''$.
$\Gamma \triangleright \pi_2 v \Uparrow \mathbf{S}(\pi_2 v : ([\pi_1 v/x]\tau'')^{\$})$	if $\Gamma \triangleright v \Uparrow \tau$ and $\Gamma \triangleright \tau^{\$} \Downarrow (x{:}\tau')\times\tau''$.
$\Gamma \triangleright v\, v' \Uparrow [v'/x]\tau''$	if $\Gamma \triangleright v \Uparrow \tau$ and $\Gamma \triangleright \tau^{\$} \Downarrow (x{:}\tau')\rightharpoonup\tau''$
$\Gamma \triangleright v\, A \Uparrow [A/\alpha]\tau''$	if $\Gamma \triangleright v \Uparrow \tau$ and $\tau^{\$} = \forall\alpha{::}K.\tau''$
$\Gamma \triangleright \mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end} \Uparrow \tau$	

Figure 6.2: Principal Type Synthesis Algorithm

6.3 Principal Types

Just as every well-formed constructor has a most-specific kind, every well-formed term has a most-specific type (up to equivalence). The algorithmic judgment $\Gamma \triangleright e \Uparrow \tau$ determines the principal type τ of the term e under context Γ. This algorithm uses the auxiliary notion of a stripped type; for any type τ, the stripped type $\tau^{\$}$ is the type label of τ if τ is a singleton type, and is τ otherwise. Note that because nested singletons are disallowed, $\tau^{\$}$ can never be a singleton type.

Lemma 6.3.1 (Singleton Stripping)

1. *If* $\Gamma \vdash \tau$ *then* $\Gamma \vdash \tau \leq \tau^{\$}$.
2. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* $\Gamma \vdash \tau_1{}^{\$} \equiv \tau_2{}^{\$}$.
3. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *then* $\Gamma \vdash \tau_1{}^{\$} \leq \tau_2{}^{\$}$.
4. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *then either* τ_2 *is a singleton type or* $\Gamma \vdash \tau_1{}^{\$} \leq \tau_2$.
5. *If* $\Gamma \vdash \tau$ *then* $\tau^{\$}$ *is the minimal non-singleton supertype of* τ.
6. *If* $\Gamma \vdash v : \tau$ *then* $\Gamma \vdash \mathbf{S}(v : \tau^{\$}) \leq \tau$.

Proof: Part 1 follows by reflexivity or by Theorem 6.2.3 and Rule 2.62, depending on whether τ is a singleton type or not. Parts 2–3 are shown by induction on derivations. Part 4 is a restatement of part 3. Finally, parts 5 and 6 follow by case analysis on the form of τ. ∎

Theorem 6.3.2 (Principal Types)

1. *If* $\Gamma \vdash v : \sigma$ *then* $\Gamma \triangleright v \Uparrow \tau$ *and* $\Gamma \vdash v : \tau$ *and* $\Gamma \vdash \tau \leq \mathbf{S}(v : \sigma^{\$})$, *so that* $\Gamma \vdash \tau \leq \sigma$.
2. *If* $\Gamma \vdash e : \sigma$ *then* $\Gamma \triangleright e \Uparrow \tau$ *and* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leq \sigma$.

Proof: By simultaneous induction on the proof of the first premise, and cases on the last typing rule used.

1. • Case: Rule 2.67.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash n : \mathsf{int}}$$

Then $\Gamma \rhd n \Uparrow \mathbf{S}(n : \mathsf{int})$ and $\Gamma \vdash n : \mathbf{S}(n : \mathsf{int})$. By reflexivity, $\Gamma \vdash \mathbf{S}(n : \mathsf{int}) \leq \mathbf{S}(n : \mathsf{int})$.

• Case: Rule 2.68.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash x : \Gamma(x)}$$

(a) $\Gamma \rhd x \Uparrow \mathbf{S}(x : \Gamma(x)^{\$})$.

(b) Since $\Gamma \vdash \Gamma(x)$, by Lemma 6.3.1 we have $\Gamma \vdash \Gamma(x) \leq \Gamma(x)^{\$}$

(c) and hence $\Gamma \vdash x : \Gamma(x)^{\$}$.

(d) By Rule 2.77, $\Gamma \vdash x : \mathbf{S}(x : \Gamma(x)^{\$})$.

(e) Finally by reflexivity, $\Gamma \vdash \mathbf{S}(x : \Gamma(x)^{\$}) \leq \mathbf{S}(x : \Gamma(x)^{\$})$.

• Case: Rule 2.69.

$$\frac{\Gamma, f{:}(x{:}\tau') \rightharpoonup \tau'', x{:}\tau' \vdash e : \tau''}{\Gamma \vdash \mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : (x{:}\tau) \rightharpoonup \tau'}$$

(a) First, $\Gamma \rhd \mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e \Uparrow \mathbf{S}(\mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : (x{:}\tau) \rightharpoonup \tau')$.

(b) By Rule 2.77, $\Gamma \vdash \mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : \mathbf{S}(\mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : (x{:}\tau) \rightharpoonup \tau')$.

(c) Finally, by reflexivity we have
$\Gamma \vdash \mathbf{S}(\mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : (x{:}\tau) \rightharpoonup \tau') \leq \mathbf{S}(\mathsf{fun}\, f(x{:}\tau){:}\tau' \;\mathsf{is}\; e : (x{:}\tau) \rightharpoonup \tau')$.

• Case: Rule 2.70.

$$\frac{\Gamma, \alpha{::}K' \vdash e : \sigma''}{\Gamma \vdash \Lambda(\alpha{::}K'){:}\sigma''.e : \forall \alpha{::}K'.\sigma''}$$

(a) $\Gamma \rhd \Lambda(\alpha{::}K'){:}\sigma''.e \Uparrow \mathbf{S}(\Lambda(\alpha{::}K'){:}\sigma''.e : \forall \alpha{::}K'.\sigma'')$.

(b) By Rule 2.77, $\Gamma \vdash \Lambda(\alpha{::}K'){:}\sigma''.e : \mathbf{S}(\Lambda(\alpha{::}K'){:}\sigma''.e : \forall \alpha{::}K'.\sigma'')$.

(c) Finally, $\Gamma \vdash \forall \alpha{::}K'.\sigma'' \leq \forall \alpha{::}K'.\sigma''$ by reflexivity,

(d) so $\Gamma \vdash \mathbf{S}(\Lambda(\alpha{::}K'){:}\sigma''.e : \forall \alpha{::}K'.\sigma'') \leq \mathbf{S}(\Lambda(\alpha{::}K'){:}\sigma''.e : \forall \alpha{::}K'.\sigma'')$.

• Case: Rule 2.71.

$$\frac{\Gamma \vdash v_1 : \sigma_1 \qquad \Gamma \vdash v_2 : \sigma_2}{\Gamma \vdash \langle v_1, v_2 \rangle : \sigma_1 \times \sigma_2}$$

(a) By the inductive hypothesis $\Gamma \rhd v_1 \Uparrow \tau_1$ and $\Gamma \vdash v_1 : \tau_1$ and $\Gamma \vdash \tau_1 \leq \mathbf{S}(v_1 : {\sigma_1}^{\$})$,

(b) and $\Gamma \rhd v_2 \Uparrow \tau_2$ and $\Gamma \vdash v_2 : \tau_2$ and $\Gamma \vdash \tau_2 \leq \mathbf{S}(v_2 : {\sigma_2}^{\$})$.

(c) Thus $\Gamma \rhd \langle v_1, v_2 \rangle \Uparrow \mathbf{S}(\langle v_1, v_2 \rangle : \tau_1 \times \tau_2)$.

(d) Also, $\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2$,

(e) so by Rule 2.77, $\Gamma \vdash \langle v_1, v_2 \rangle : \mathbf{S}(\langle v_1, v_2 \rangle : \tau_1 \times \tau_2)$.

(f) Finally, $\Gamma \vdash \tau_1 \times \tau_2 \leq \mathbf{S}(v_1 : {\sigma_1}^{\$}) \times \mathbf{S}(v_2 : {\sigma_2}^{\$})$

(g) and $\Gamma \vdash \mathbf{S}(v_1 : {\sigma_1}^{\$}) \times \mathbf{S}(v_2 : {\sigma_2}^{\$}) \leq \sigma_1 \times \sigma_2$,

(h) so $\Gamma \vdash \mathbf{S}(\langle v_1, v_2 \rangle : \tau_1 \times \tau_2) \leq \mathbf{S}(\langle v_1, v_2 \rangle : \sigma_1 \times \sigma_2)$.

• Case: Rule 2.72.

$$\frac{\Gamma \vdash v : (x{:}\sigma') \times \sigma''}{\Gamma \vdash \pi_1 v : \sigma'}$$

(a) By the inductive hypothesis, $\Gamma \rhd v \Uparrow \tau$ and $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau \leq \mathbf{S}(v : (x{:}\sigma')\times\sigma'')$.
(b) By Lemma 6.3.1, $\Gamma \vdash \tau^{\$} \leq (x{:}\sigma')\times\sigma''$
(c) and hence by Theorem 6.2.3 $\Gamma \rhd \tau^{\$} \Downarrow (x{:}\tau')\times\tau''$ with $\Gamma \vdash \tau' \leq \sigma'$.
(d) Thus $\Gamma \rhd \pi_1 v \Uparrow \mathbf{S}(\pi_1 v : \tau'^{\$})$.
(e) By Lemmas 6.3.1 and 6.2.1 and subsumption, $\Gamma \vdash \pi_1 v : \tau'^{\$}$,
(f) so by Rule 2.77 we have $\Gamma \vdash \pi_1 v : \mathbf{S}(\pi_1 v : \tau'^{\$})$.
(g) Finally, $\Gamma \vdash \tau'^{\$} \leq \sigma'^{\$}$ by Lemma 6.3.1,
(h) so $\Gamma \vdash \mathbf{S}(\pi_1 v : \tau'^{\$}) \leq \mathbf{S}(\pi_1 v : \sigma'^{\$})$.

- Case: Rule 2.73. Analogous to previous case.
- Case: Rule 2.77.

$$\frac{\Gamma \vdash v : \sigma}{\Gamma \vdash v : \mathbf{S}(v : \sigma)}(\sigma \text{ not a singleton})$$

(a) By the inductive hypothesis, $\Gamma \rhd v \Uparrow \tau$ and $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau \leq \mathbf{S}(v : \sigma^{\$})$.
(b) It suffices to observe that $\mathbf{S}(v : (\mathbf{S}(v : \sigma^{\$}))^{\$}) = \mathbf{S}(v : \sigma^{\$})$.

- Case: Rule 2.78.

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \Gamma \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}$$

(a) By the inductive hypothesis, $\Gamma \rhd v \Uparrow \tau$ and $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau \leq \mathbf{S}(v : {\sigma_1}^{\$})$.
(b) By Lemma 6.3.1, $\Gamma \vdash {\sigma_1}^{\$} \leq {\sigma_2}^{\$}$,
(c) so by transitivity, $\Gamma \vdash \tau \leq \mathbf{S}(v : {\sigma_2}^{\$})$.

2. - Case: e is a value. Follows by Part 1, Lemma 6.3.1, and transitivity.
   - Case: Rule 2.74.

$$\frac{\Gamma \vdash v : \sigma' \rightharpoonup \sigma'' \qquad \Gamma \vdash v' : \sigma'}{\Gamma \vdash v\, v' : \sigma''}$$

(a) By the inductive hypothesis, $\Gamma \rhd v \Uparrow \tau$ and $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau \leq \sigma' \rightharpoonup \sigma''$.
(b) Similarly, $\Gamma \rhd v' \Uparrow \tau_1$ and $\Gamma \vdash v' : \tau_1$ and $\Gamma \vdash \tau_1 \leq \sigma'$.
(c) By Lemma 6.3.1, $\Gamma \vdash \tau^{\$} \leq \sigma' \rightharpoonup \sigma''$.
(d) By Theorem 6.2.3, $\Gamma \rhd \tau^{\$} \Downarrow (x{:}\tau') \rightharpoonup \tau''$ with $\Gamma \vdash \sigma' \leq \tau'$ and $\Gamma, x{:}\sigma' \vdash \tau'' \leq \sigma''$.
(e) Thus $\Gamma \rhd v\, v' \Uparrow [v'/x]\tau''$.
(f) By Lemmas 6.3.1 and 6.2.1, $\Gamma \vdash v : (x{:}\tau') \rightharpoonup \tau''$.
(g) Also by transitivity, $\Gamma \vdash \tau_1 \leq \tau'$.
(h) Hence $\Gamma \vdash v\, v' : [v'/x]\tau''$.
(i) Finally, by substitution we have $\Gamma \vdash [v'/x]\tau'' \leq [v'/x]\sigma''$.

- Case: Rule 2.75

$$\frac{\Gamma \vdash v : \forall\alpha{::}K'.\sigma'' \qquad \Gamma \vdash A :: K'}{\Gamma \vdash v\, A : [A/\alpha]\sigma''}$$

(a) By the inductive hypothesis, $\Gamma \rhd v \Uparrow \tau$ and $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau \leq \forall\alpha{::}K'.\sigma''$.
(b) By Lemma 6.3.1 $\Gamma \vdash \tau^{\$} \leq \forall\alpha{::}K'.\sigma''$,
(c) so by Theorem 6.2.3 $\tau^{\$} = \forall\alpha{::}L'.\tau''$ with $\Gamma \vdash K' \leq L'$ and $\Gamma, \alpha{::}K' \vdash \tau'' \leq \sigma''$.
(d) Thus $\Gamma \rhd v\, A \Uparrow [A/\alpha]\tau''$.

(e) Then $\Gamma \vdash v : \forall\alpha{::}L'.\tau''$ and $\Gamma \vdash A :: L'$,

(f) so $\Gamma \vdash v\,A : [A/\alpha]\tau''$.

(g) Finally, by substitution we have $\Gamma \vdash [A/\alpha]\tau'' \leq [A/\alpha]\sigma''$.

- Case: Rule 2.76.
$$\frac{\Gamma \vdash e' : \sigma' \qquad \Gamma, x{:}\sigma' \vdash e : \sigma \qquad \Gamma \vdash \sigma}{\Gamma \vdash (\mathsf{let}\ x{:}\sigma'{=}e'\ \mathsf{in}\ e : \sigma\ \mathsf{end}) : \sigma}$$

(a) It is immediate that $\Gamma \rhd (\mathsf{let}\ x{:}\sigma'{=}e'\ \mathsf{in}\ e : \sigma\ \mathsf{end}) \Uparrow \sigma$,

(b) and $\Gamma \vdash (\mathsf{let}\ x{:}\sigma'{=}e'\ \mathsf{in}\ e : \sigma\ \mathsf{end}) : \sigma$ by assumption.

(c) Finally, $\Gamma \vdash \sigma \leq \sigma$ by reflexivity.

- Case: Rule 2.78. As in Part 1.

∎

6.4 Algorithms

The term equivalence again makes use of term-level elimination contexts, again denoted by $\mathcal{E}$. In contrast to the elimination contexts for type constructors, applications are not included; the only paths ($\mathcal{E}[v]$ where v is a constant or variable) of interest are those which are values:

$$\begin{array}{rcl} \mathcal{E} & ::= & \diamond \\ & | & \pi_1\mathcal{E} \\ & | & \pi_2\mathcal{E} \end{array}$$

6.5 Soundness

Proposition 6.5.1 (Inversion of Term Validity)

1. *If $\Gamma \vdash v\,v' : \tau$ then $\Gamma \vdash v : (x{:}\tau') \rightharpoonup \tau''$ and $\Gamma \vdash v' : \tau'$ with $\Gamma \vdash [v'/x]\tau'' \leq \tau$.*
2. *If $\Gamma \vdash v\,A : \tau$ then $\Gamma \vdash v : \forall\alpha{::}K'.\tau''$ and $\Gamma \vdash A :: K'$ with $\Gamma \vdash [A/\alpha]\tau'' \leq \tau$.*
3. *If $\Gamma \vdash \pi_1 v : \tau$ then $\Gamma \vdash v : \tau_1 \times \tau_2$ and $\Gamma \vdash \tau_1 \leq \tau$.*
4. *If $\Gamma \vdash \pi_2 v : \tau$ then $\Gamma \vdash v : \tau_1 \times \tau_2$ and $\Gamma \vdash \tau_2 \leq \tau$.*

Proof: By inversion v must be well-formed, so (the stripped, head-normal version of) its principal type satisfies the desired properties. ∎

Proposition 6.5.2
If $\Gamma \vdash \langle v_1, v_2\rangle : \tau$ then $\Gamma \rhd \tau^{\$} \Downarrow (x{:}\tau') \times \tau''$ and $\Gamma \vdash v_1 : \tau$ and $\Gamma \vdash v_2 : [v_1/x]\tau''$.

Proof: By induction on typing derivations, and cases on the last rule used.

- Case: Rule 2.71.
$$\frac{\Gamma \vdash v_1 : \tau' \qquad \Gamma \vdash v_2 : \tau''}{\Gamma \vdash \langle v_1, v_2\rangle : \tau' \times \tau''}$$

  Trivial.

Type validity

$\Gamma \rhd \mathit{Ty}(A)$	if $\Gamma \rhd A \Leftarrow \mathbf{T}$
$\Gamma \rhd \mathbf{S}(v : \tau)$	if $\Gamma \rhd \tau$ and $\Gamma \rhd v \Leftarrow \tau$.
$\Gamma \rhd (x{:}\tau')\rightharpoonup\tau''$	if $\Gamma \rhd \tau'$ and $\Gamma, x{:}\tau' \rhd \tau''$.
$\Gamma \rhd (x{:}\tau')\times\tau''$	if $\Gamma \rhd \tau'$ and $\Gamma, x{:}\tau' \rhd \tau''$.
$\Gamma \rhd \forall\alpha{::}K.\tau$	if $\Gamma \rhd K$ and $\Gamma, \alpha{::}K \rhd \tau$.

Algorithmic subtyping

$\Gamma \rhd \tau_1 \leq \tau_2$	if $\Gamma \rhd \tau_1 \Downarrow \sigma_1$, $\Gamma \rhd \tau_2 \Downarrow \sigma_2$, and $\Gamma \rhd \sigma_1 \sqsubseteq \sigma_2$

Weak algorithmic subtyping

$\Gamma \rhd \mathit{Ty}(A_1) \sqsubseteq \mathit{Ty}(A_2)$	if $\Gamma \rhd A_1 \Leftrightarrow A_2 :: \mathbf{T}$.
$\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \sqsubseteq \mathbf{S}(v_2 : \tau_2)$	if $\Gamma \rhd \tau_1 \leq \tau_2$ and $\Gamma \rhd v_1 \Leftrightarrow v_2$.
$\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \sqsubseteq \tau_2$	if τ_2 not a singleton and $\Gamma \rhd \tau_1 \leq \tau_2$.
$\Gamma \rhd (x{:}\tau_1')\rightharpoonup\tau_1'' \sqsubseteq (x{:}\tau_2')\rightharpoonup\tau_2''$	if $\Gamma \rhd \tau_2' \leq \tau_1'$ and $\Gamma, x{:}\tau_2' \rhd \tau_1'' \leq \tau_2''$
$\Gamma \rhd (x{:}\tau_1')\times\tau_1'' \sqsubseteq (x{:}\tau_2')\times\tau_2''$	if $\Gamma \rhd \tau_1' \leq \tau_2'$ and $\Gamma, x{:}\tau_1' \rhd \tau_1'' \leq \tau_2''$
$\Gamma \rhd \forall\alpha{::}K_1.\tau_1 \sqsubseteq \forall\alpha{::}K_2.\tau_2$	if $\Gamma \rhd K_2 \leq K_1$ and $\Gamma, \alpha{::}K_2 \rhd \tau_1 \leq \tau_2$.

Algorithmic type equivalence

$\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$	if $\Gamma \rhd \tau_1 \Downarrow \sigma_1$, $\Gamma \rhd \tau_2 \Downarrow \sigma_2$, and $\Gamma \rhd \sigma_1 \leftrightarrow \sigma_2$.

Weak algorithmic type equivalence

$\Gamma \rhd \mathit{Ty}(A_1) \leftrightarrow \mathit{Ty}(A_2)$	if $\Gamma \rhd A_1 \Leftrightarrow A_2 :: \mathbf{T}$
$\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \leftrightarrow \mathbf{S}(v_2 : \tau_2)$	if $\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$ and $\Gamma_1 \rhd v_1 \Leftrightarrow v_2$
$\Gamma \rhd (x{:}\tau_1')\rightharpoonup\tau_1'' \leftrightarrow (x{:}\tau_2)\rightharpoonup\tau_2''$	if $\Gamma_1 \rhd \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma_1, x{:}\tau_1' \rhd \tau_1'' \Leftrightarrow \tau_2''$
$\Gamma \rhd (x{:}\tau_1')\times\tau_1'' \leftrightarrow (x{:}\tau_2)\times\tau_2''$	if $\Gamma_1 \rhd \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma_1, x{:}\tau_1' \rhd \tau_1'' \Leftrightarrow \tau_2''$
$\Gamma \rhd \forall\alpha{::}K_1.\tau_1 \leftrightarrow \forall\alpha{::}K_2.\tau_2$	if $\Gamma \rhd K_1 \Leftrightarrow K_2$ and $\Gamma_1, x{::}K_1 \rhd \tau_1 \Leftrightarrow \tau_2$

Figure 6.3: Algorithms for Types

Type synthesis

$\Gamma \triangleright n \rightrightarrows \mathbf{S}(n : \mathsf{int})$	
$\Gamma \triangleright x \rightrightarrows \mathbf{S}(x : \Gamma(x)^{\$})$	
$\Gamma \triangleright \mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e \rightrightarrows$ $\mathbf{S}((\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e) : (x{:}\tau')\rightharpoonup\tau'')$	if $\Gamma \triangleright \tau'$, $\Gamma, x{:}\tau' \triangleright \tau''$, and $\Gamma, f{:}(x{:}\tau')\rightharpoonup\tau'', x{:}\tau' \triangleright e \leftleftarrows \tau''$
$\Gamma \triangleright \Lambda(\alpha{::}K){:}\tau.e \rightrightarrows \mathbf{S}(\Lambda(\alpha{::}K){:}\tau.e : \forall\alpha{::}K.\tau)$	if $\Gamma \triangleright K$ and $\Gamma, \alpha{::}K \triangleright \tau$ and $\Gamma, \alpha{::}K \triangleright e \leftleftarrows \tau$.
$\Gamma \triangleright \langle v_1, v_2\rangle \rightrightarrows \mathbf{S}(\langle v_1, v_2\rangle : \tau_1\times\tau_2)$	if $\Gamma \triangleright v_1 \rightrightarrows \tau_1$ and $\Gamma \triangleright v_2 \rightrightarrows \tau_2$.
$\Gamma \triangleright \pi_1 v \rightrightarrows \tau'$	if $\Gamma \triangleright v \rightrightarrows \tau$ and $\tau^{\$} = (x{:}\tau')\times\tau''$.
$\Gamma \triangleright \pi_2 v \rightrightarrows [\pi_1 v/x]\tau''$	if $\Gamma \triangleright v \rightrightarrows \tau$ and $\tau^{\$} = (x{:}\tau')\times\tau''$.
$\Gamma \triangleright v\, v' \rightrightarrows [v'/x]\tau''$	if $\Gamma \triangleright v \rightrightarrows \tau$, $\tau^{\$} = (x{:}\tau')\rightharpoonup\tau''$, and $\Gamma \triangleright v' \leftleftarrows \tau'$.
$\Gamma \triangleright v\, A \rightrightarrows [A/\alpha]\tau$	if $\Gamma \triangleright v \rightrightarrows \tau$, $\tau^{\$} = \forall\alpha{::}K.\tau$, and $\Gamma \triangleright A \leftleftarrows K$.
$\Gamma \triangleright \mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end} \rightrightarrows \tau$	if $\Gamma \triangleright \tau'$, $\Gamma \triangleright e' \leftleftarrows \tau'$, $\Gamma \triangleright \tau$, and $\Gamma, x{:}\tau' \triangleright e \leftleftarrows \tau$.

Typechecking

$\Gamma \triangleright e \leftleftarrows \tau$	if $\Gamma \triangleright e \rightrightarrows \sigma$ and $\Gamma \triangleright \sigma \leq \tau$.

Figure 6.4: Algorithms for Term Validity

- Case: Rule 2.77.
$$\frac{\Gamma \vdash \langle v_1, v_2\rangle : \tau}{\Gamma \vdash \langle v_1, v_2\rangle : \mathbf{S}(v : \tau)} \quad (\tau \text{ not a singleton})$$
By the inductive hypothesis.
- Case: Rule 2.78
$$\frac{\Gamma \vdash \langle v_1, v_2\rangle : \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \langle v_1, v_2\rangle : \tau_2}$$
  1. By the inductive hypothesis, $\Gamma \triangleright {\tau_1}^{\$} \Downarrow (x{:}\tau_1')\times\tau_1''$
  2. and $\Gamma \vdash v_1 : \tau_1'$ and $\Gamma \vdash v_2 : [v_1/\alpha]\tau_1''$.
  3. Then by Lemma 6.3.1, $\Gamma \vdash {\tau_1}^{\$} \leq {\tau_2}^{\$}$,
  4. so by Theorem 6.2.3 we have $\Gamma \triangleright {\tau_2}^{\$} \Downarrow (x{:}\tau_2')\times\tau_2''$
  5. and $\Gamma \vdash \tau_1' \leq \tau_2'$ and $\Gamma, x{:}\tau_1' \vdash \tau_1'' \leq \tau_2''$.
  6. Thus by substitution and subsumption, $\Gamma \vdash v_1 : \tau_2'$ and $\Gamma \vdash v_2 : [v_1/x]\tau_2''$.

∎

Lemma 6.5.3
If $\Gamma \vdash v_1 : \tau$ *and* $\Gamma \vdash v_2 : \tau$ *and* $\Gamma \vdash v_1 \equiv v_2 : \tau^{\$}$ *then* $\Gamma \vdash v_1 \equiv v_2 : \tau$.

Proof:

- Case: $\tau = \mathbf{S}(w : \sigma)$.
  1. Then $\tau^{\$} = \sigma$ and $\Gamma \vdash v_1 \equiv v_2 : \sigma$.
  2. By Rule 2.120, $\Gamma \vdash v_1 \equiv v_2 : \mathbf{S}(v_1 : \sigma)$.

Type extraction

$\Gamma \triangleright n \uparrow \mathsf{int}$	
$\Gamma \triangleright x \uparrow \Gamma(x)$	
$\Gamma \triangleright \pi_1 p \uparrow \tau_1$	if $\Gamma \triangleright p \uparrow (y{:}\tau_1)\times\tau_2$
$\Gamma \triangleright \pi_2 p \uparrow [\pi_1 p/y]\tau_2$	if $\Gamma \triangleright p \uparrow (y{:}\tau_1)\times\tau_2$

Term weak head reduction

$\Gamma \triangleright \mathcal{E}[\pi_1\langle v_1, v_2\rangle] \rightsquigarrow \mathcal{E}[v_1]$	
$\Gamma \triangleright \mathcal{E}[\pi_2\langle v_1, v_2\rangle] \rightsquigarrow \mathcal{E}[v_2]$	
$\Gamma \triangleright \mathcal{E}[p] \rightsquigarrow \mathcal{E}[v]$	if $\Gamma \triangleright p \uparrow \mathbf{S}(v : \tau)$

Term weak head normalization

$\Gamma \triangleright e \Downarrow d$	if $\Gamma \triangleright e \rightsquigarrow e'$ and $\Gamma \triangleright e' \Downarrow d$
$\Gamma \triangleright e \Downarrow e$	otherwise

Algorithmic term equivalence

$\Gamma \triangleright e_1 \Leftrightarrow e_2$	if $\Gamma \triangleright e_1 \Downarrow d_1$, $\Gamma \triangleright e_2 \Downarrow d_2$, and $\Gamma \triangleright d_1 \leftrightarrow d_2$

Algorithmic weak term equivalence

$\Gamma \triangleright n \leftrightarrow n$	always
$\Gamma \triangleright x \leftrightarrow x$	always
$\Gamma \triangleright \mathsf{fun}\, f(x{:}\tau_1'){:}\tau_1''\ \mathsf{is}\ e_1 \leftrightarrow$ $\mathsf{fun}\, f(x{:}\tau_2'){:}\tau_2''\ \mathsf{is}\ e_2$	if $\Gamma \triangleright \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma, x{:}\tau_1' \triangleright \tau_1'' \Leftrightarrow \tau_2''$ and $\Gamma, f{:}(x{:}\tau_1')\rightharpoonup\tau_1'', x{:}\tau_1' \triangleright e_1 \Leftrightarrow e_2$.
$\Gamma \triangleright \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \leftrightarrow \Lambda(\alpha{::}K_2){:}\tau_2.e_2$	if $\Gamma \triangleright K_1 \Leftrightarrow K_2$ and $\Gamma, \alpha{::}K_1 \triangleright \tau_1 \Leftrightarrow \tau_2$ and $\Gamma, \alpha{::}K_1 \triangleright e_1 \Leftrightarrow e_2$.
$\Gamma \triangleright \langle v_1', v_1''\rangle \leftrightarrow \langle v_2', v_2''\rangle$	if $\Gamma \triangleright v_1' \Leftrightarrow v_2'$ and $\Gamma \triangleright v_1'' \Leftrightarrow v_2''$.
$\Gamma \triangleright \pi_i v_1 \leftrightarrow \pi_i v_2$	if $\Gamma \triangleright v_1 \leftrightarrow v_2$
$\Gamma \triangleright v_1\, v_1' \leftrightarrow v_2\, v_2'$	if $\Gamma \triangleright v_1 \Leftrightarrow v_2$ and $\Gamma \triangleright v_1' \Leftrightarrow v_2'$.
$\Gamma \triangleright v_1\, A_1 \leftrightarrow v_2\, A_2$	if $\Gamma \triangleright v_1 \Leftrightarrow v_2$, $\Gamma \triangleright v_1 \Downarrow w_1$, $\Gamma \triangleright w_1 \Uparrow \sigma$, $\sigma^{\$} = \forall\alpha{::}L'.\sigma''$, and $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: L'$.
$\Gamma \triangleright (\mathsf{let}\ x{:}\tau_1'{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \leftrightarrow$ $(\mathsf{let}\ x{:}\tau_2'{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end})$	if $\Gamma \triangleright \tau_1' \Leftrightarrow \tau_2'$, $\Gamma \triangleright e_1' \Leftrightarrow e_2'$, $\Gamma, x{:}\tau_1' \triangleright e_1 \Leftrightarrow e_2$, and $\Gamma \triangleright \tau_1 \Leftrightarrow \tau_2$.

Figure 6.5: Algorithms for Term Equivalence

3. But $\Gamma \vdash v_1 : \mathbf{S}(w : \sigma)$, so $\Gamma \vdash v_1 \equiv w : \sigma$
4. and hence $\Gamma \vdash \mathbf{S}(v_1 : \sigma) \equiv \mathbf{S}(w : \sigma)$.
5. By subsumption then, $\Gamma \vdash v_1 \equiv v_2 : \mathbf{S}(w : \sigma)$.
6. That is, $\Gamma \vdash v_1 \equiv v_2 : \tau$.

- Case: $\tau^{\$} = \tau$. Trivial.

∎

Lemma 6.5.4 (Term Weak Head-Normalization)
If $\Gamma \vdash e : \tau$ then there exists at most one e' such that $\Gamma \triangleright e \Downarrow e'$. Furthermore, $\Gamma \vdash e' : \tau$ and $\Gamma \vdash e \equiv e' : \tau$.

Lemma 6.5.5 (Soundness for Path Weak Equivalence)
If $\Gamma \vdash p_1 : \tau_1$ and $\Gamma \vdash p_2 : \tau_2$ and $\Gamma \triangleright p_1 \leftrightarrow p_2$ then $\Gamma \triangleright p_1 \Uparrow \sigma_1$, $\Gamma \triangleright p_2 \Uparrow \sigma_2$, $\Gamma \vdash \sigma_1 \equiv \sigma_2$, and $\Gamma \vdash p_1 \equiv p_2 : \sigma_1$.

Proof: By induction on $\Gamma \triangleright p_1 \leftrightarrow p_2$, and cases on the last step.

- Case: $\Gamma \triangleright n \leftrightarrow n$. Direct.
- Case: $\Gamma \triangleright x \leftrightarrow x$. Direct.
- Case: $\Gamma \triangleright \pi_1 p_1' \leftrightarrow \pi_1 p_2'$ because $\Gamma \triangleright p_1' \leftrightarrow p_2'$.

  1. By inversion, p_1' and p_2' are well-formed.
  2. By the inductive hypothesis, $\Gamma \triangleright p_1' \Uparrow \sigma_1$, $\Gamma \triangleright p_2' \Uparrow \sigma_2$, $\Gamma \vdash \sigma_1 \equiv \sigma_2$, and $\Gamma \vdash p_1' \equiv p_2' : \sigma_1$.
  3. Since $\pi_1 p_1'$ and $\pi_1 p_2'$ are well-formed, $\sigma_1 = \mathbf{S}(p_1' : (x{:}\sigma_1') \times \sigma_1'')$ and $\sigma_2 = \mathbf{S}(p_2' : (x{:}\sigma_2') \times \sigma_2'')$,
  4. and $\Gamma \triangleright \pi_1 p_1' \Uparrow \mathbf{S}(\pi_1 p_1' : \sigma_1')$ and $\Gamma \triangleright \pi_1 p_2' \Uparrow \mathbf{S}(\pi_1 p_2' : \sigma_2')$.
  5. By Theorem 6.2.2, $\Gamma \vdash \sigma_1' \equiv \sigma_2'$.
  6. By subsumption and Rule 2.85, $\Gamma \vdash \pi_1 p_1' \equiv \pi_1 p_2' : \sigma_1'$.
  7. Hence $\Gamma \vdash \pi_1 p_1' \equiv \pi_1 p_2' : \mathbf{S}(\pi_1 p_1' : \sigma_1')$ and $\Gamma \vdash \mathbf{S}(\pi_1 p_1' : \sigma_1') \equiv \mathbf{S}(\pi_1 p_2' : \sigma_2')$.

- Case: $\Gamma \triangleright \pi_2 p_1' \leftrightarrow \pi_2 p_2'$ because $\Gamma \triangleright p_1' \leftrightarrow p_2'$. Analogous to previous case.

∎

Theorem 6.5.6 (Soundness of Equivalence)

1. *If $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ and $\Gamma \triangleright e_1 \Leftrightarrow e_2$ then $\Gamma \vdash e_1 \equiv e_2 : \tau$.*
2. *If $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, $\Gamma \triangleright e_1 \leftrightarrow e_2$, and e_1 and e_2 are head-normal then $\Gamma \vdash e_1 \equiv e_2 : \tau$.*
3. *If $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ and $\Gamma \triangleright \tau_1 \Leftrightarrow \tau_2$ then $\Gamma \vdash \tau_1 \equiv \tau_2$.*
4. *If $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ and $\Gamma \triangleright \tau_1 \leftrightarrow \tau_2$ then $\Gamma \vdash \tau_1 \equiv \tau_2$.*

Proof: By simultaneous induction on algorithmic judgments (i.e., on the execution of the algorithms).

1. By the inductive hypothesis and Lemma 6.5.4.
2.
   - Case: $\Gamma \rhd n \leftrightarrow n$. Follows by reflexivity.
   - Case: $\Gamma \rhd x \leftrightarrow x$. Follows by reflexivity.
   - Case: $\Gamma \rhd \mathsf{fun}\, f(x{:}\sigma_1'){:}\sigma_1''\ \mathsf{is}\ e_1 \leftrightarrow \mathsf{fun}\, f(x{:}\sigma_2'){:}\sigma_2''\ \mathsf{is}\ e_2$.
     (a) Then by inversion $\Gamma \vdash \sigma_1'$, $\Gamma \vdash \sigma_2'$, $\Gamma, x{:}\sigma_1' \vdash \sigma_1''$, $\Gamma, x{:}\sigma_2' \vdash \sigma_2''$, $\Gamma, x{:}\sigma_1' \vdash e_1 : \sigma_1''$, and $\Gamma, x{:}\sigma_2' \vdash e_2 : \sigma_2''$.
     (b) By inversion of the algorithm, $\Gamma \rhd \sigma_1' \Leftrightarrow \sigma_2'$ and $\Gamma, x{:}\sigma_1' \rhd \sigma_1'' \Leftrightarrow \sigma_2''$.
     (c) By the inductive hypothesis, $\Gamma \vdash \sigma_1' \equiv \sigma_2'$.
     (d) Thus $\Gamma, x{:}\sigma_1' \vdash \sigma_2''$ and so by the inductive hypothesis $\Gamma, x{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''$.
     (e) This yields $\Gamma, x{:}\sigma_1' \vdash e_2 : \sigma_1''$, so by the inductive hypothesis $\Gamma, x{:}\sigma_1' \vdash e_1 \equiv e_2 : \sigma_1''$.
     (f) Thus $\Gamma \vdash \mathsf{fun}\, f(x{:}\sigma_1'){:}\sigma_1''\ \mathsf{is}\ e_1 \equiv \mathsf{fun}\, f(x{:}\sigma_2'){:}\sigma_2''\ \mathsf{is}\ e_2 : (x{:}\sigma_1') \rightharpoonup \sigma_1''$.
     (g) Finally, by Theorem 6.3.2 and Lemma 6.2.1 we have $\Gamma \vdash (x{:}\sigma_1') \rightharpoonup \sigma_1'' \leq \tau^{\$}$ and so $\Gamma \vdash \mathsf{fun}\, f(x{:}\sigma_1'){:}\sigma_1''\ \mathsf{is}\ e_1 \equiv \mathsf{fun}\, f(x{:}\sigma_2'){:}\sigma_2''\ \mathsf{is}\ e_2 : \tau^{\$}$.
     (h) By Lemma 6.5.3, we have $\Gamma \vdash \mathsf{fun}\, f(x{:}\sigma_1'){:}\sigma_1''\ \mathsf{is}\ e_1 \equiv \mathsf{fun}\, f(x{:}\sigma_2'){:}\sigma_2''\ \mathsf{is}\ e_2 : \tau$.
   - Case: $\Gamma \rhd \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \leftrightarrow \Lambda(\alpha{::}K_2){:}\tau_2.e_2$ because $\Gamma \rhd K_1 \Leftrightarrow K_2$ and $\Gamma, \alpha{::}K_1 \rhd \tau_1 \Leftrightarrow \tau_2$ and $\Gamma, \alpha{::}K_1 \rhd e_1 \Leftrightarrow e_2$.
     (a) By inversion of typing, $\Gamma \vdash K_1$ and $\Gamma, \alpha{::}K_1 \vdash \tau_1$ and $\Gamma, \alpha{::}K_1 \vdash e_1 : \tau_1$.
     (b) Similarly, $\Gamma \vdash K_2$ and $\Gamma, \alpha{::}K_2 \vdash \tau_2$ and $\Gamma, \alpha{::}K_2 \vdash e_2 : \tau_2$.
     (c) By the inductive hypothesis, $\Gamma \vdash K_1 \equiv K_2$.
     (d) Then $\Gamma, \alpha{::}K_1 \vdash \tau_2$, so by the inductive hypothesis $\Gamma, \alpha{::}K_1 \vdash \tau_1 \equiv \tau_2$.
     (e) Then $\Gamma, \alpha{::}K_1 \vdash e_2 : \tau_1$, so by the inductive hypothesis $\Gamma, \alpha{::}K_1 \vdash e_1 \equiv e_2 : \tau_1$.
     (f) Thus, $\Gamma \vdash \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \equiv \Lambda(\alpha{::}K_2){:}\tau_2.e_2 : \forall\alpha{::}K_1.\tau_1$.
     (g) By Theorem 6.3.2 and Lemma 6.3.1, $\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \leq \tau^{\$}$.
     (h) By subsumption, $\Gamma \vdash \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \equiv \Lambda(\alpha{::}K_2){:}\tau_2.e_2 : \tau^{\$}$.
     (i) Therefore by Lemma 6.5.3, $\Gamma \vdash \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \equiv \Lambda(\alpha{::}K_2){:}\tau_2.e_2 : \tau$.
   - Case: $\Gamma \rhd \langle v_1', v_1''\rangle \leftrightarrow \langle v_2', v_2''\rangle$ because $\Gamma \rhd v_1' \Leftrightarrow v_2'$ and $\Gamma \rhd v_1'' \Leftrightarrow v_2''$.
     (a) By Proposition 6.5.2, $\Gamma \rhd \tau^{\$} \Downarrow (x{:}\tau') \times \tau''$,
     (b) and $\Gamma \vdash v_1' : \tau'$ and $\Gamma \vdash v_2' : \tau'$
     (c) and $\Gamma \vdash v_1'' : [v_1'/x]\tau''$ and $\Gamma \vdash v_2'' : [v_2'/x]\tau''$.
     (d) By the inductive hypothesis, $\Gamma \vdash v_1' \equiv v_2' : \tau'$.
     (e) Thus by functionality and subsumption and $\Gamma \vdash v_2'' : [v_1'/x]\tau''$.
     (f) By the inductive hypothesis, $\Gamma \vdash v_1'' \equiv v_2'' : [v_1'/x]\tau''$.
     (g) By Rule 2.106, $\Gamma \vdash \langle v_1', v_1''\rangle \equiv \langle v_2', v_2''\rangle : (x{:}\tau') \times \tau''$.
     (h) By Lemma 6.2.1 and subsumption, $\Gamma \vdash \langle v_1', v_1''\rangle \equiv \langle v_2', v_2''\rangle : \tau^{\$}$.
     (i) Therefore by Lemma 6.5.3, $\Gamma \vdash \langle v_1', v_1''\rangle \equiv \langle v_2', v_2''\rangle : \tau$.
   - Case: $\Gamma \rhd \pi_1 v_1 \leftrightarrow \pi_1 v_2$ because $\Gamma \rhd v_1 \leftrightarrow v_2$. Since $\pi_1 v_1$ and $\pi_1 v_2$ are head-normal and well-formed they must be paths; the result follows by Lemma 6.5.5.
   - Case: $\Gamma \rhd \pi_2 v_1 \leftrightarrow \pi_2 v_2$ because $\Gamma \rhd v_1 \leftrightarrow v_2$. Since $\pi_2 v_1$ and $\pi_2 v_2$ are head-normal and well-formed they must be paths; the result follows by Lemma 6.5.5.

- Case: $\Gamma \triangleright v_1\, v_1' \leftrightarrow v_2\, v_2'$ because $\Gamma \triangleright v_1 \Leftrightarrow v_2$ and $\Gamma \triangleright v_1' \Leftrightarrow v_2'$.
  (a) Then $\Gamma \triangleright v_1 \Downarrow w_1$ and $\Gamma \triangleright v_2 \Downarrow w_2$ and $\Gamma \triangleright w_1 \leftrightarrow w_2$
  (b) By Proposition 6.5.1, $\Gamma \vdash v_1 : (x{:}\tau_1') {\rightharpoonup} \tau_1''$ and $\Gamma \vdash v_1' : \tau_1'$ and $\Gamma \vdash [v_1'/x]\tau_1'' \leq \tau$.
  (c) Similarly, $\Gamma \vdash v_2 : (x{:}\tau_2') {\rightharpoonup} \tau_2''$ and $\Gamma \vdash v_2' : \tau_2'$ and $\Gamma \vdash [v_2'/x]\tau_2'' \leq \tau$.
  (d) By Lemma 6.5.4, w_1 and w_2 have these function types. Thus w_1 and w_2 are not type abstractions, pairs, or (because they are head-normal) projections from pairs. The only remaining possibilities are that either w_1 and w_2 are both paths, or else they are both term abstractions.
    - SUBCASE: $w_1 = p_1$ and $w_2 = p_2$. By Lemma 6.5.5, there exist σ_1 and σ_2 such that $\Gamma \triangleright w_1 \Uparrow \sigma_1$ and $\Gamma \triangleright w_2 \Uparrow \sigma_2$ and $\Gamma \vdash \sigma_1 \equiv \sigma_2$ and $\Gamma \vdash w_1 \equiv w_2 : \sigma_1$.
    - SUBCASE: $w_1 = \mathsf{fun}\, f(x{:}\sigma_1'){:}\sigma_1''\ \mathsf{is}\ e_1$ and $w_2 = \mathsf{fun}\, f(x{:}\sigma_2'){:}\sigma_2''\ \mathsf{is}\ e_2$.
      * Put $\sigma_1 = \mathbf{S}(w_1 : (x{:}\sigma_1') {\rightharpoonup} \sigma_1'')$ and $\sigma_2 = \mathbf{S}(w_2 : (x{:}\sigma_2') {\rightharpoonup} \sigma_2'')$.
      * Then $\Gamma \triangleright w_1 \Uparrow \sigma_1$ and $\Gamma \triangleright w_2 \Uparrow \sigma_2$.
      * By declarative and algorithmic inversion and the inductive hypothesis, $\Gamma \vdash \sigma_1' \equiv \sigma_2'$ and $\Gamma, x{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''$.
      * By the inductive hypothesis, $\Gamma \vdash w_1 \equiv w_2 : {\sigma_1}^{\$}$,
      * so $\Gamma \vdash \sigma_1 \equiv \sigma_2$ and $\Gamma \vdash w_1 \equiv w_2 : \sigma_1$.
    - Since $\Gamma \vdash w_1 : (x{:}\tau_1') {\rightharpoonup} \tau_1''$, by Theorem 6.3.2 we have $\Gamma \vdash \sigma_1 \leq (x{:}\tau_1') {\rightharpoonup} \tau_1''$.
    - Thus in either of the two cases above, ${\sigma_1}^{\$}$ is of the form $(x{:}\sigma_1') {\rightharpoonup} \sigma_1''$.
    - By Theorem 6.2.3, $\Gamma \vdash \tau_1' \leq \sigma_1'$ and $\Gamma, x{:}\tau_1' \vdash \sigma_1'' \leq \tau_1''$.
    - Thus $\Gamma \vdash v_1' : \sigma_1'$.
    - Similarly, ${\sigma_2}^{\$} = (x{:}\sigma_2') {\rightharpoonup} \sigma_2''$ and $\Gamma \vdash v_2' : \sigma_2'$.
    - By subsumption, $\Gamma \vdash v_2' : \sigma_1'$.
    - By the inductive hypothesis, $\Gamma \vdash v_1' \equiv v_2' : \sigma_1'$.
    - Thus $\Gamma \vdash w_1\, v_1' \equiv w_2\, v_2' : [v_1'/x]\sigma_1''$.
    - By substitution, $\Gamma \vdash [v_1'/x]\sigma_1'' \leq [v_1'/x]\tau_1''$,
    - so $\Gamma \vdash [v_1'/x]\sigma_1'' \leq \tau$ and $\Gamma \vdash w_1\, v_1' \equiv w_2\, v_2' : \tau$.
    - Then $\Gamma \vdash v_1\, v_1' \equiv w_1\, v_1' : \tau$ and $\Gamma \vdash v_2\, v_2' \equiv w_2\, v_2' : \tau$.
    - So by symmetry and transitivity, $\Gamma \vdash v_1\, v_1' \equiv v_2\, v_2' : \tau$.
- Case: $\Gamma \triangleright v_1\, A_1 \leftrightarrow v_2\, A_2$ because $\Gamma \triangleright v_1 \Leftrightarrow v_2$, $\Gamma \triangleright v_1 \Downarrow w_1$, $\Gamma \triangleright w_1 \Uparrow \sigma$, $\sigma^{\$} = \forall\alpha{::}L'.\sigma''$, and $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: L'$.

  Analogous to the previous case; this time the head normal forms of v_1 and v_2 must either be paths or type abstractions. The return-type annotations on type abstractions are vital here (as they are for term abstractions in proof of the previous case) so that the induction hypothesis can be applied; they supply a common type for comparing the functions' bodies.
- Case: $\Gamma \triangleright \mathsf{let}\ x{:}\tau_1'{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end} \leftrightarrow \mathsf{let}\ x{:}\tau_2'{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end}$ because $\Gamma \triangleright \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma \triangleright e_1' \Leftrightarrow e_2'$ and $\Gamma, x{:}\tau_1' \triangleright \tau_1 \Leftrightarrow \tau_2$ and $\Gamma, x{:}\tau_1' \triangleright e_1 \Leftrightarrow e_2$.

  Essentially analogous to the proof for equivalence of two term-level functions.

3. By the inductive hypothesis and Lemma 6.2.1.
4. - Case: $\Gamma \triangleright Ty(A_1) \leftrightarrow Ty(A_2)$ because $\Gamma \triangleright A_1 \Leftrightarrow A_2 :: \mathbf{T}$.
     (a) By inversion of typing, $\Gamma \vdash A_1 :: \mathbf{T}$ and $\Gamma \vdash A_2 :: \mathbf{T}$,

(b) By soundness of constructor equivalence then, $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$.

(c) By Rule 2.53, $\Gamma \vdash \mathit{Ty}(A_1) \equiv \mathit{Ty}(A_2)$.

- Case: $\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \leftrightarrow \mathbf{S}(v_2 : \tau_2)$ because $\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$ and $\Gamma_1 \rhd v_1 \Leftrightarrow v_2$.
  (a) By inversion of typing and the inductive hypothesis, $\Gamma \vdash \tau_1 \equiv \tau_2$.
  (b) Thus $\Gamma \vdash v_1 : \tau_1$ and $\Gamma \vdash v_2 : \tau_1$.
  (c) By the inductive hypothesis, $\Gamma \vdash v_1 \equiv v_2 : \tau_1$.
  (d) By Rule 2.54, $\Gamma \vdash \mathbf{S}(v_1 : \tau_1) \equiv \mathbf{S}(v_2 : \tau_2)$.
- Case: $\Gamma \rhd (x{:}\tau_1')\rightharpoonup\tau_1'' \leftrightarrow (x{:}\tau_2)\rightharpoonup\tau_2''$ because $\Gamma_1 \rhd \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma_1, x{:}\tau_1' \rhd \tau_1'' \Leftrightarrow \tau_2''$.
  By inversion of typing and the inductive hypothesis.
- $\Gamma \rhd (x{:}\tau_1')\times\tau_1'' \leftrightarrow (x{:}\tau_2)\times\tau_2''$ because $\Gamma_1 \rhd \tau_1' \Leftrightarrow \tau_2'$ and $\Gamma_1, x{:}\tau_1' \rhd \tau_1'' \Leftrightarrow \tau_2''$.
  By inversion of typing and the inductive hypothesis.
- $\Gamma \rhd \forall\alpha{::}K_1.\tau_1 \leftrightarrow \forall\alpha{::}K_2.\tau_2$ because $\Gamma \rhd K_1 \Leftrightarrow K_2$ and $\Gamma_1, x{::}K_1 \rhd \tau_1 \Leftrightarrow \tau_2$.
  By inversion of typing, soundness of kind equivalence, and the inductive hypothesis.

∎

The soundness proofs for the remaining algorithmic judgments are then straightforward.

Theorem 6.5.7 (Soundness of Subtyping)

1. *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *and* $\Gamma \rhd \tau_1 \leq \tau_2$ *then* $\Gamma \vdash \tau_1 \leq \tau_2$.
2. *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *and* $\Gamma \rhd \tau_1 \sqsubseteq \tau_2$ *then* $\Gamma \vdash \tau_1 \leq \tau_2$.

Proof: By induction on algorithmic derivations. ∎

Theorem 6.5.8 (Soundness of Typechecking)

1. *If* $\Gamma \vdash ok$ *and* $\Gamma \rhd \tau$ *then* $\Gamma \vdash \tau$.
2. *If* $\Gamma \vdash ok$ *and* $\Gamma \rhd e \rightrightarrows \tau$ *then* $\Gamma \vdash e : \tau$ *and* $\Gamma \rhd e \Uparrow \tau$.
3. *If* $\Gamma \vdash \tau$ *and* $\Gamma \rhd e \leftleftarrows \tau$ *then* $\Gamma \vdash e : \tau$.

Proof: By induction on algorithmic derivations. ∎

Chapter 7

Completeness and Decidability for Types and Terms

7.1 Type and Term Equivalence

The approach for studying type and term equivalence is very similar to that for constructor and kind equivalence. Figures 7.1 and 7.2 show a symmetrized version of the type and term equivalence algorithms. By construction the algorithm is symmetric and transitive:

Lemma 7.1.1 (Algorithmic PER Properties)

1. *If* $\Delta_1 \triangleright v_1 \Leftrightarrow \Delta_2 \triangleright v_2$ *then* $\Delta_2 \triangleright v_2 \Leftrightarrow \Delta_1 \triangleright v_1$.
2. *If* $\Delta_1 \triangleright v_1 \Leftrightarrow \Delta_2 \triangleright v_2$ *and* $\Delta_2 \triangleright v_2 \Leftrightarrow \Delta_3 \triangleright v_3$ *then* $\Delta_1 \triangleright v_1 \Leftrightarrow \Delta_3 \triangleright v_3$.
3. *If* $\Delta_1 \triangleright v_1 \leftrightarrow \Delta_2 \triangleright v_2$ *then* $\Delta_2 \triangleright v_2 \leftrightarrow \Delta_1 \triangleright v_1$.
4. *If* $\Delta_1 \triangleright v_1 \leftrightarrow \Delta_2 \triangleright v_2$ *and* $\Delta_2 \triangleright v_2 \leftrightarrow \Delta_3 \triangleright v_3$ *then* $\Delta_1 \triangleright v_1 \leftrightarrow \Delta_3 \triangleright v_3$.
5. *If* $\Delta_1 \triangleright \tau_1 \Leftrightarrow \Delta_2 \triangleright \tau_2$ *then* $\Delta_2 \triangleright \tau_2 \Leftrightarrow \Delta_1 \triangleright \tau_1$.
6. *If* $\Delta_1 \triangleright \tau_1 \Leftrightarrow \Delta_2 \triangleright \tau_2$ *and* $\Delta_2 \triangleright \tau_2 \Leftrightarrow \Delta_3 \triangleright \tau_3$ *then* $\Delta_1 \triangleright \tau_1 \Leftrightarrow \Delta_3 \triangleright \tau_3$.

The proof of completeness for term equivalence is essentially the same as the completeness proof for constructor equivalence. Although the algorithm is not type-directed, the fact that it must maintain two contexts requires the more complex two-world form of logical relation: see Figures 7.3, 7.4, and 7.5. The main differences from the constructor- and kind-level relations are:

1. Since type equivalence is not purely structural (e.g., $Ty(\mathsf{Int}\times\mathsf{Int}) \equiv Ty(\mathsf{Int})\times Ty(\mathsf{Int})$) the logical relations are defined using head normalization of types.
2. The term-level logical relations are defined only for values, not all expressions.
3. The Π cases of the term-level relations have been simplified, since applications are not values.
4. These logical relations also require that $\vdash \Delta_1 \equiv \Delta_2$ as well as declarative well-formedness or equivalences, as appropriate. This allows the invocation of the correctness results for the constructor algorithms.

It is not immediately obvious that these logical relations are well-defined, because they are not defined simply by induction on types.

Algorithmic type equivalence

$\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$	if $\Gamma_1 \triangleright \tau_1 \Downarrow \sigma_1$, $\Gamma_2 \triangleright \tau_2 \Downarrow \sigma_2$, and $\Gamma_1 \triangleright \sigma_1 \leftrightarrow \Gamma_2 \triangleright \sigma_2$.

Weak algorithmic type equivalence

$\Gamma_1 \triangleright Ty(A_1) \leftrightarrow \Gamma_2 \triangleright Ty(A_2)$	if $\Gamma_1 \triangleright A_1 :: \mathbf{T}_1 \Leftrightarrow \Gamma_2 \triangleright A_2 :: \mathbf{T}_2$
$\Gamma_1 \triangleright \mathbf{S}(v_1 : \tau_1) \leftrightarrow \Gamma_2 \triangleright \mathbf{S}(v_2 : \tau_2)$	if $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$ and $\Gamma_1 \triangleright v_1 \Leftrightarrow \Gamma_2 \triangleright v_2$
$\Gamma_1 \triangleright (x{:}\tau_1){\rightharpoonup}\sigma_1 \leftrightarrow \Gamma_2 \triangleright (x{:}\tau_2){\rightharpoonup}\sigma_2$	if $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$ and $\Gamma_1, x{:}\tau_1 \triangleright \sigma_1 \Leftrightarrow \Gamma_2, x{:}\tau_2 \triangleright \sigma_2$
$\Gamma_1 \triangleright (x{:}\tau_1){\times}\sigma_1 \leftrightarrow \Gamma_2 \triangleright (x{:}\tau_2){\times}\sigma_2$	if $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$ and $\Gamma_1, x{:}\tau_1 \triangleright \sigma_1 \Leftrightarrow \Gamma_2, x{:}\tau_2 \triangleright \sigma_2$
$\Gamma_1 \triangleright \forall\alpha{::}K_1.\tau_1 \leftrightarrow \Gamma_2 \triangleright \forall\alpha{::}K_2.\tau_2$	if $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ and $\Gamma_1, \alpha{::}K_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2, \alpha{::}K_2 \triangleright \tau_2$

Figure 7.1: Revised Type Equivalence Algorithm

Algorithmic term equivalence

$\Gamma_1 \triangleright e_1 \Leftrightarrow \Gamma_2 \triangleright e_2$	if $\Gamma_1 \triangleright e_1 \Downarrow d_1$, $\Gamma_2 \triangleright e_2 \Downarrow d_2$, and $\Gamma_1 \triangleright d_1 \leftrightarrow \Gamma_2 \triangleright d_2$

Algorithmic weak term equivalence

$\Gamma_1 \triangleright n \leftrightarrow \Gamma_2 \triangleright n$	always
$\Gamma_1 \triangleright x \leftrightarrow \Gamma_2 \triangleright x$	always
$\Gamma_1 \triangleright \mathsf{fun}\, f(x{:}\tau_1'){:}\tau_1''\ \mathsf{is}\ e_1 \leftrightarrow$ $\Gamma_2 \triangleright \mathsf{fun}\, f(x{:}\tau_2'){:}\tau_2''\ \mathsf{is}\ e_2$	if $\Gamma_1 \triangleright \tau_1' \Leftrightarrow \Gamma_2 \triangleright \tau_2'$ and $\Gamma, x{:}\tau_1' \triangleright \tau_1'' \Leftrightarrow \Gamma_2, x{:}\tau_2' \triangleright \tau_2''$ and $\Gamma, f{:}(x{:}\tau_1'){\rightharpoonup}\tau_1'', x{:}\tau_1' \triangleright e_1 \Leftrightarrow \Gamma_2, f{:}(x{:}\tau_2'){\rightharpoonup}\tau_2'', x{:}\tau_2' \triangleright e_2$.
$\Gamma_1 \triangleright \lambda x{:}\tau_1'.e_1 \leftrightarrow \Gamma_2 \triangleright \lambda x{:}\tau_2'.e_2$	if $\Gamma_1 \triangleright \tau_1' \Leftrightarrow \Gamma_2 \triangleright \tau_2'$ and $\Gamma_1, x{:}\tau_1' \triangleright e_1 \Leftrightarrow \Gamma_2, x{:}\tau_2' \triangleright e_2$.
$\Gamma_1 \triangleright \Lambda(\alpha{::}K_1){:}\tau_1.e_1 \leftrightarrow \Gamma_2 \triangleright \Lambda(\alpha{::}K_2){:}\tau_2.e_2$	if $\Gamma_1 \triangleright K_1 \Leftrightarrow \Gamma_2 \triangleright K_2$ and $\Gamma_1, \alpha{::}K_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2, \alpha{::}K_2 \triangleright \tau_2$ and $\Gamma_1, \alpha{::}K_1 \triangleright e_1 \Leftrightarrow \Gamma_2, \alpha{::}K_2 \triangleright e_2$.
$\Gamma_1 \triangleright \langle v_1', v_1''\rangle \leftrightarrow \Gamma_2 \triangleright \langle v_2', v_2''\rangle$	if $\Gamma_1 \triangleright v_1' \Leftrightarrow \Gamma_2 \triangleright v_2'$ and $\Gamma_1 \triangleright v_1'' \Leftrightarrow \Gamma_2 \triangleright v_2''$.
$\Gamma_1 \triangleright \pi_i v_1 \leftrightarrow \Gamma_2 \triangleright \pi_i v_2$	if $\Gamma_1 \triangleright v_1 \leftrightarrow \Gamma_2 \triangleright v_2$
$\Gamma_1 \triangleright v_1\, v_1' \leftrightarrow \Gamma_2 \triangleright v_2\, v_2'$	if $\Gamma_1 \triangleright v_1 \Leftrightarrow \Gamma_2 \triangleright v_2$ and $\Gamma_1 \triangleright v_1' \Leftrightarrow \Gamma_2 \triangleright v_2'$.
$\Gamma_1 \triangleright v_1\, A_1 \leftrightarrow \Gamma_2 \triangleright v_2\, A_2$	if $\Gamma_1 \triangleright v_1 \Leftrightarrow \Gamma_2 \triangleright v_2$, $\Gamma_i \triangleright v_i \Downarrow w_i$, $\Gamma_i \triangleright w_i \Uparrow \sigma_i$, ${\sigma_i}^{\$} = \forall\alpha{::}L_i'.\sigma_i''$, and $\Gamma_1 \triangleright A_1 :: L_1' \Leftrightarrow \Gamma_2 \triangleright A_2 :: L_2'$.
$\Gamma_1 \vdash (\mathsf{let}\ x{:}\tau_1'{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \leftrightarrow$ $\Gamma_2 \vdash (\mathsf{let}\ x{:}\tau_2'{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end})$	if $\Gamma_1 \triangleright \tau_1' \Leftrightarrow \Gamma_2 \triangleright \tau_2'$, $\Gamma_1 \triangleright e_1' \Leftrightarrow \Gamma_2 \triangleright e_2'$, $\Gamma_1, x{:}\tau_1' \triangleright e_1 \Leftrightarrow \Gamma_1, x{:}\tau_2' \triangleright e_2$, and $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$.

Figure 7.2: Revised Term Equivalence Algorithm

- $(\Delta; \tau)$ valid iff

  1. $\Delta \vdash \tau$
  2. and

     – $\tau = Ty(A)$ and $\Delta \triangleright \tau \Downarrow \tau$
     – Or, $\tau = \mathbf{S}(v : \sigma)$ and $(\Delta; v; \sigma)$ valid
     – Or, $\Delta \triangleright \tau \Downarrow (x{:}\tau') \rightharpoonup \tau''$, and $(\Delta; \tau')$ valid, and for all $\Delta' \supseteq \Delta$ and $\Delta'' \supseteq \Delta$ if $(\Delta'; v'; \tau')$ is $(\Delta''; w'; \tau')$ then $(\Delta'; [v'/x]\tau'')$ is $(\Delta''; [w'/x]\tau'')$.
     – Or $\Delta \triangleright \tau \Downarrow (x{:}\tau') \times \tau''$, and $(\Delta; \tau')$ valid, and for all $\Delta' \supseteq \Delta$ and $\Delta'' \supseteq \Delta$ if $(\Delta'; v'; \tau')$ is $(\Delta''; w'; \tau')$ then $(\Delta'; [v'/x]\tau'')$ is $(\Delta''; [w'/x]\tau'')$.
     – Or $\tau = \forall \alpha{::}K.\tau''$, and for all $\Delta' \supseteq \Delta$ and $\Delta'' \supseteq \Delta$ if $\vdash \Delta' \equiv \Delta''$ and $\Delta' \vdash A_1 \equiv A_2 :: K$ then $(\Delta'; [A_1/\alpha]\tau'')$ is $(\Delta''; [A_2/\alpha]\tau'')$.

- $(\Delta_1; \tau_1)$ is $(\Delta_2; \tau_2)$ iff

  1. $\vdash \Delta_1 \equiv \Delta_2$ and $\Delta_1 \vdash \tau_1 \equiv \tau_2$
  2. $(\Delta_1; \tau_1)$ valid and $(\Delta_2; \tau_2)$ valid.
  3. – $\tau_i = Ty(A_i)$ and $\Delta_i \triangleright \tau_i \Downarrow \tau_i$
     – Or, $\tau_i = \mathbf{S}(v_i : \sigma_i)$ and $(\Delta_1; v_1; \sigma_1)$ is $(\Delta_2; v_2; \sigma_2)$
     – Or, $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i') \rightharpoonup \tau_i''$, and $(\Delta_1; \tau_1')$ is $(\Delta_2; \tau_2')$, and for all $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ if $(\Delta_1'; v_1'; \tau')$ is $(\Delta_2'; v_2'; \tau')$ then $(\Delta_1'; [v_1'/x]\tau_1'')$ is $(\Delta_2'; [v_2'/x]\tau_2'')$.
     – Or, $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i') \times \tau_i''$, and $(\Delta_1; \tau_1')$ is $(\Delta_2; \tau_2')$, and for all $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ if $(\Delta_1'; v_1'; \tau')$ is $(\Delta_2'; v_2'; \tau')$ then $(\Delta_1'; [v_1'/x]\tau_1'')$ is $(\Delta_2'; [v_2'/x]\tau_2'')$.
     – Or $\tau_i = \forall \alpha{::}K_i.\tau_i''$, and for all $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ if $\vdash \Delta_1' \equiv \Delta_2'$ and $\Delta_1' \vdash A_1 \equiv A_2 :: K_1$ then $(\Delta_1'; [A_1/\alpha]\tau_1'')$ is $(\Delta_2'; [A_2/\alpha]\tau_2'')$.

Figure 7.3: Logical Relations for Types

- $(\Delta; v; \tau)$ valid iff

  1. $(\Delta; \tau)$ valid
  2. $\Delta \vdash v : \tau$
  3. $\Delta \triangleright v \Leftrightarrow \Delta \triangleright v$
  4. – $\tau = Ty(A)$ and $\Delta \triangleright \tau \Downarrow \tau$
     – Or, $\tau = \mathbf{S}(w : \tau')$ and $(\Delta; v; \tau')$ is $(\Delta; w; \tau')$
     – Or, $\Delta \triangleright \tau \Downarrow (x{:}\tau') \rightharpoonup \tau''$
     – Or, $\Delta \triangleright \tau \Downarrow (x{:}\tau') \times \tau''$, $(\Delta; \pi_1 v; \tau')$ valid, and $(\Delta; \pi_2 v; [\pi_1 v/x]\tau'')$ valid.
     – Or, $\tau = \forall\alpha{::}K.\tau'$.

- $(\Delta_1; v_1; \tau_1)$ is $(\Delta_2; v_2; \tau_2)$ iff

  1. $(\Delta_1; \tau_1)$ is $(\Delta_2; \tau_2)$
  2. $(\Delta_1; v_1; \tau_1)$ valid and $(\Delta_1; v_2; \tau_1)$ valid
  3. $\Delta_1 \vdash v_1 \equiv v_2 : \tau_1$
  4. $\Delta_1 \triangleright v_1 \Leftrightarrow \Delta_2 \triangleright v_2$
  5. – $\tau_i = Ty(A_i)$ and $\Delta_i \triangleright \tau_i \Downarrow \tau_i$
     – Or, $\tau_i = \mathbf{S}(w_i : \sigma_i)$ and $(\Delta_1; v_1; \sigma_1)$ is $(\Delta_2; v_2; \sigma_2)$
     – Or, $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i') \rightharpoonup \tau_i''$,
     – Or, $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i') \times \tau_i''$, $(\Delta_1; \pi_1 v_1; \tau_1')$ is $(\Delta_2; \pi_1 v_2; \tau_2')$, and $(\Delta_1; \pi_2 v_1; [\pi_1 v_1/x]\tau_1'')$ is $(\Delta_2; \pi_2 v_2; [\pi_1 v_2/x]\tau_2'')$.
     – Or, $\tau_i = \forall\alpha{::}K_i.\tau_i'$.

Figure 7.4: Logical Relations for Values

- $(\Delta_1; \tau_1 \leq \sigma_1)$ **is** $(\Delta_2; \tau_2 \leq \sigma_2)$ iff
  1. $\forall \Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$, if $(\Delta_1'; v_1; \tau_1)$ is $(\Delta_2'; v_2; \tau_2)$ then $(\Delta_1'; v_1; \sigma_1)$ is $(\Delta_2'; v_2; \sigma_2)$

- $(\Delta; \gamma; \Gamma)$ valid iff
  1. $\Delta \vdash \mathrm{ok}$
  2. $\forall \alpha \in \mathrm{dom}(\Gamma).\ \Delta \vdash \gamma\alpha :: \gamma(\Gamma(\alpha))$
  3. $\forall x \in \mathrm{dom}(\Gamma).\ (\Delta; \gamma x; \gamma(\Gamma(x)))$ valid

- $(\Delta_1; \gamma_1; \Gamma_1)$ is $(\Delta_2; \gamma_2; \Gamma_2)$ iff
  1. $\vdash \Delta_1 \equiv \Delta_2$
  2. $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$
  3. $(\Delta_1; \gamma_1; \Gamma_1)$ valid and $(\Delta_2; \gamma_2; \Gamma_2)$ valid
  4. $\forall \alpha \in \mathrm{dom}(\Gamma).\ \Delta_1 \vdash \gamma_1\alpha \equiv \gamma_2\alpha :: \gamma(\Gamma_1(\alpha))$
  5. $\forall x \in \mathrm{dom}(\Gamma).\ (\Delta_1; \gamma_1 x; \gamma_1(\Gamma_1(x)))$ is $(\Delta_2; \gamma_2 x; \gamma_2(\Gamma_2(x)))$

Figure 7.5: Derived Logical Relations

$$\begin{array}{lcl}
\mathit{size}(\Gamma; \forall\alpha{::}K.\tau) & = & (1,0) + \mathit{size}(\Gamma, \alpha{::}K; \tau) \\
\mathit{size}(\Gamma; \mathbf{S}(v : \tau)) & = & (1,0) + \mathit{size}(\Gamma; \tau) \\
\mathit{size}(\Gamma; (x{:}\tau')\rightharpoonup\tau'') & = & (0,1) + \mathit{size}(\Gamma; \tau') + \mathit{size}(\Gamma, x{:}\tau'; \tau'') \\
\mathit{size}(\Gamma; (x{:}\tau')\times\tau'') & = & (0,1) + \mathit{size}(\Gamma; \tau') + \mathit{size}(\Gamma, x{:}\tau'; \tau'') \\
\mathit{size}(\Gamma; \mathit{Ty}(A)) & = & (0, \text{Number of of } \times\text{'s and } \rightharpoonup\text{'s in } B \text{ where } \Gamma \triangleright A :: \mathbf{T} \Longrightarrow B)
\end{array}$$

Figure 7.6: Size Metric for Types

I therefore define the *size* of a type τ relative to a context Γ to be pair of integers, (If Γ is apparent from context, I will just refer to the size of τ.) The formal definition is given in Figure 7.6; the definition here uses componentwise addition:

$$(m_1, m_2) + (n_1, n_2) = (m_1 + n_1, m_2 + n_2).$$

The first component of the size is the number of $\forall$'s and $\mathbf{S}$'s in the type. The second component is the number of $\times$ and $\rightharpoonup$'s in the type *after all the constructors within $Ty(\cdot)$'s have been normalized.* These sizes are ordered lexicographically:

$$(m_1, m_2) \leq (n_1, n_2) \iff (m_1 < n_1) \vee ((m_1 = n_1) \wedge (m_2 \leq n_2)).$$

The relevant properties of sizes are summarized in the following lemma:

Lemma 7.1.2 (Sizes of Types)

1. *If $\Gamma \vdash \tau_1 \equiv \tau_2$ then $size(\Gamma; \tau_1) = size(\Gamma; \tau_2)$.*
2. *If $\Gamma \vdash \tau_1$ and $\Gamma \triangleright \tau_1 \Downarrow \tau_2$ then τ_1 and τ_2 have equal sizes.*
3. *If $\Gamma \vdash \mathbf{S}(v : \tau)$ then the size of $\mathbf{S}(v : \tau)$ is strictly greater than the size of τ.*
4. *If $\Gamma \vdash (x{:}\tau') \rightharpoonup \tau''$ then the size of $(x{:}\tau') \rightharpoonup \tau''$ is strictly greater than both the size of τ' and the size of $[v/x]\tau''$ for any value satisfying $\Gamma \vdash v : \tau'$.*
5. *If $\Gamma \vdash (x{:}\tau') \times \tau''$ then the size of $(x{:}\tau') \times \tau''$ is strictly greater than both the size of τ' and the size of $[v/x]\tau''$ for any value satisfying $\Gamma \vdash v : \tau'$.*
6. *If $\Gamma \vdash \forall \alpha{::}K.\tau$ then the size of $\forall \alpha{::}K.\tau$ is strictly greater than the size of $[A/\alpha]\tau$ for any constructor satisfying $\Gamma \vdash A : K$.*

Proof:

1. By induction on equivalence derivations and the properties of constructor normalization.
2. By part 1 and Lemma 6.2.1.

3–6. By definition of sizes.

∎

Lemma 7.1.3 (Logical Reflexivity)

1. *If $(\Delta; \tau)$ valid then $(\Delta; \tau)$ is $(\Delta; \tau)$.*
2. *If $(\Delta; v; \tau)$ valid then $(\Delta; v; \tau)$ is $(\Delta; v; \tau)$.*
3. *If $(\Delta; \gamma; \Gamma)$ valid then $(\Delta; \gamma; \Gamma)$ is $(\Delta; \gamma; \Gamma)$.*

Proof: By induction on the size of types

1. In all cases, $\vdash \Delta \equiv \Delta$ and and $\Delta \vdash \tau \equiv \tau$ by declarative reflexivity.
   - Case: $\tau = Ty(A)$ and $\Delta \triangleright \tau \Downarrow \tau$. Trivially $(\Delta; Ty(A))$ is $(\Delta; Ty(A))$.
   - Case: $\tau = \mathbf{S}(v : \sigma)$. By the inductive hypothesis $(\Delta; v; \sigma)$ valid implies $(\Delta; v; \sigma)$ is $(\Delta; v; \sigma)$. Thus $(\Delta; \mathbf{S}(v : \sigma))$ is $(\Delta; \mathbf{S}(v : \sigma))$.
   - Case: $\Delta \triangleright \tau \Downarrow (x{:}\tau') \rightharpoonup \tau''$. Then $(\Delta; \tau')$ valid. By the inductive hypothesis, $(\Delta; \tau')$ is $(\Delta; \tau')$. Let $\Delta_1' \supseteq \Delta$ and $\Delta_2' \supseteq \Delta$ and assume $(\Delta_1'; v_1'; \tau')$ is $(\Delta_2'; v_2'; \tau')$. Then $(\Delta_1'; [v_1'/x]\tau'')$ is $(\Delta_2'; [v_2'/x]\tau'')$. Thus $(\Delta; \tau)$ is $(\Delta; \tau)$.

- Case: $\Delta \rhd \tau \Downarrow (x{:}\tau')\times\tau''$. Same proof as in previous case.
- Case: $\tau = \forall\alpha{::}K.\tau''$. Assume $\Delta_1' \supseteq \Delta_1$, $\Delta_2' \supseteq \Delta_2$, $\vdash \Delta_1' \equiv \Delta_2'$, and $\Delta_1' \vdash A_1 \equiv A_2 :: K_1$. Then $(\Delta_1'; [A_1/\alpha]\tau'')$ is $(\Delta_2'; [A_2/\alpha]\tau'')$. Thus $(\Delta; \forall\alpha{::}K.\tau'')$ is $(\Delta; \forall\alpha{::}K.\tau'')$.

2. In all cases, $(\Delta;\tau)$ is $(\Delta;\tau)$ by the argument of the previous part, $\Delta \vdash v \equiv v : \tau$ by Rule 2.79, and $\Delta \rhd v \Leftrightarrow \Delta \rhd v$ by assumption.

   - Case: $\tau = Ty(A)$ and $\Delta \rhd \tau \Downarrow \tau$. Trivial.
   - Case: $\tau = \mathbf{S}(w : \tau')$. Then $(\Delta; v; \tau')$ is $(\Delta; w; \tau')$ so $(\Delta; v; \tau')$ valid. By the inductive hypothesis $(\Delta; v; \tau')$ is $(\Delta; v; \tau')$. Therefore $(\Delta; v; \mathbf{S}(w : \tau'))$ is $(\Delta; v; \mathbf{S}(w : \tau'))$.
   - Case: $\Delta \rhd \tau \Downarrow (x{:}\tau')\rightharpoonup\tau''$. Trivial.
   - Case: $\Delta \rhd \tau \Downarrow (x{:}\tau')\times\tau''$. Then $(\Delta; \pi_1 v; \tau')$ valid, so by the inductive hypothesis we have $(\Delta; \pi_1 v; \tau')$ is $(\Delta; \pi_1 v; \tau')$ and $(\Delta; \pi_2 v; [\pi_1 v/x]\tau'')$ is $(\Delta; \pi_2 v; [\pi_1 v/x]\tau'')$. Thus $(\Delta; v; \tau)$ is $(\Delta; v; \tau)$.
   - Case: $\tau_i = \forall\alpha{::}K_i.\tau_i'$. Trivial.

3. By declarative reflexivity we have $\vdash \Delta \equiv \Delta$. By reflexivity of constructor equivalence, for all $\alpha \in \mathrm{dom}(\Gamma)$ we have $\Delta \vdash \gamma\alpha \equiv \gamma\alpha :: \gamma(\Gamma(\alpha))$. By part 2, for all $x \in \mathrm{dom}(\Gamma)$ we have $(\Delta; \gamma x; \gamma(\Gamma(x)))$ is $(\Delta; \gamma x; \gamma(\Gamma(x)))$. Thus $(\Delta; \gamma; \Gamma)$ is $(\Delta; \gamma; \Gamma)$.

∎

Lemma 7.1.4 (Logical Symmetry)

1. *If* $(\Delta_1; \tau_1)$ is $(\Delta_2; \tau_2)$ *then* $(\Delta_2; \tau_2)$ is $(\Delta_1; \tau_1)$.
2. *If* $(\Delta_1; \tau_1 \leq \sigma_1)$ **is** $(\Delta_2; \tau_2 \leq \sigma_2)$ *then* $(\Delta_2; \tau_2 \leq \sigma_2)$ **is** $(\Delta_1; \tau_1 \leq \sigma_1)$.
3. *If* $(\Delta_1; v_1; \tau_1)$ is $(\Delta_2; v_2; \tau_2)$ *then* $(\Delta_2; v_2; \tau_2)$ is $(\Delta_1; v_1; \tau_1)$.
4. *If* $(\Delta_1; \gamma_1; \Gamma_1)$ is $(\Delta_2; \gamma_2; \Gamma_2)$ *then* $(\Delta_2; \gamma_2; \Gamma_2)$ is $(\Delta_1; \gamma_1; \Gamma_1)$.

Proof: By induction on the size of types, using context replacement, declarative symmetry, and algorithmic symmetry. ∎

The following two lemmas must be proved simultaneously by induction on the size of types. I have separated their statements for clarity.

Lemma 7.1.5

1. *If* $(\Delta; v; \tau)$ valid *and* $(\Delta; \tau)$ is $(\Delta; \sigma)$ *then* $(\Delta; v; \sigma)$ valid.
2. *If* $(\Delta_1; v_1; \tau_1)$ is $(\Delta_2; v_2; \tau_2)$, $(\Delta_1; \tau_1)$ is $(\Delta_1; \sigma_1)$, *and* $(\Delta_2; \tau_2)$ is $(\Delta_2; \sigma_2)$ *then* $(\Delta_1; v_1; \sigma_1)$ is $(\Delta_2; v_2; \sigma_2)$.

Proof: In all cases, by subsumption we have $\Delta \vdash v : \sigma$.

1.
   - Case: $\tau = Ty(A)$ and $\Delta \rhd \tau \Downarrow \tau$. Then $\sigma = Ty(B)$ and $\Delta \rhd \sigma \Downarrow \sigma$.
   - Case: $\tau = \mathbf{S}(w : \tau')$. Then $\sigma = \mathbf{S}(w' : \sigma')$ where $(\Delta; w; \tau')$ is $(\Delta; w'; \sigma')$. Since $(\Delta; v; \tau')$ is $(\Delta; w; \tau')$, inductively by Logical Transitivity we have $(\Delta; v; \tau')$ is $(\Delta; w'; \sigma')$.
   - Case: $\Delta \rhd \tau \Downarrow (x{:}\tau')\rightharpoonup\tau''$. Then $\Delta \rhd \sigma \Downarrow (x{:}\sigma')\rightharpoonup\sigma''$.

- Case: $\Delta \triangleright \tau \Downarrow (x{:}\tau')\times\tau''$. Then $\Delta \triangleright \sigma \Downarrow (x{:}\sigma')\times\sigma''$. Now $(\Delta;\pi_1 v;\tau')\ \mathsf{valid}$ and $(\Delta;\tau')\ \mathsf{is}\ (\Delta;\sigma')$, so by the inductive hypothesis we have $(\Delta;\pi_1 v;\sigma')\ \mathsf{valid}$. By reflexivity and the inductive hypothesis, $(\Delta;\pi_1 v;\tau')\ \mathsf{is}\ (\Delta;\pi_1 v;\sigma')$, so $(\Delta;[\pi_1 v/x]\tau'')\ \mathsf{is}\ (\Delta;[\pi_1 v/x]\sigma'')$. Since $(\Delta;\pi_2 v;[\pi_1 v/x]\tau'')\ \mathsf{valid}$, by the inductive hypothesis we have $(\Delta;\pi_2 v;[\pi_1 v/x]\sigma'')\ \mathsf{valid}$.
- Case: $\tau = \forall\alpha{::}K.\tau'$. Then $\sigma = \forall\alpha{::}L.\sigma'$.

2. By subsumption, in all cases $\Delta_1 \vdash v_1 \equiv v_2 : \sigma_1$. By the argument in part 1, $(\Delta_1;v_1;\sigma_1)\ \mathsf{valid}$ and $(\Delta_2;v_2;\sigma_2)\ \mathsf{valid}$. Recall that that $(\Delta_1;\tau_1)\ \mathsf{is}\ (\Delta_2;\tau_2)$.

   - Case: $\tau_i = \mathit{Ty}(A_i)$ and $\Delta_i \triangleright \tau_i \Downarrow \tau_i$. Then $\sigma_i = \mathit{Ty}(B_i)$ and $\Delta_i \triangleright \sigma_i \Downarrow \sigma_i$.
   - Case: $\tau_i = \mathbf{S}(v_i : \tau_i')$. Then $\sigma_i = \mathbf{S}(w_i : \sigma_i')$, $(\Delta_1;v_1;\tau_1')\ \mathsf{is}\ (\Delta_2;v_2;\tau_2')$, $(\Delta_1;v_1;\tau_1')\ \mathsf{is}\ (\Delta_1;w_1;\sigma_1')$, and $(\Delta_2;v_2;\tau_2')\ \mathsf{is}\ (\Delta_2;w_2;\sigma_2')$. Thus $(\Delta_1;\tau_1')\ \mathsf{is}\ (\Delta_1;\sigma_1')$ and $(\Delta_2;\tau_2')\ \mathsf{is}\ (\Delta_2;\sigma_2')$. By the inductive hypothesis, $(\Delta_1;v_1;\sigma_1')\ \mathsf{is}\ (\Delta_2;v_2;\sigma_2')$.
   - Case: $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i')\rightharpoonup\tau_i''$. Then $\Delta_i \triangleright \sigma_i \Downarrow (x{:}\sigma_i')\rightharpoonup\sigma_i''$.
   - Case: $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i')\times\tau_i''$. Then $\Delta_i \triangleright \sigma_i \Downarrow (x{:}\sigma_i')\times\sigma_i''$. Now $(\Delta_1;\tau_1')\ \mathsf{is}\ (\Delta_1;\sigma_1')$, $(\Delta_2;\tau_2')\ \mathsf{is}\ (\Delta_2;\sigma_2')$, and $(\Delta_1;\pi_1 v_1;\tau_1')\ \mathsf{is}\ (\Delta_2;\pi_1 v_2;\tau_2')$. By the inductive hypothesis, $(\Delta_1;\pi_1 v_1;\sigma_1')\ \mathsf{is}\ (\Delta_2;\pi_1 v_2;\sigma_2')$. Also by Reflexivity we have $(\Delta_1;\pi_1 v_1;\tau_1')\ \mathsf{is}\ (\Delta_1;\pi_1 v_1;\tau_1')$ and $(\Delta_1;\tau_1')\ \mathsf{is}\ (\Delta_1;\tau_1')$, so by the inductive hypothesis we have $(\Delta_1;\pi_1 v_1;\tau_1')\ \mathsf{is}\ (\Delta_1;\pi_1 v_1;\sigma_1')$. Similarly, $(\Delta_2;\pi_1 v_2;\tau_2')\ \mathsf{is}\ (\Delta_2;\pi_1 v_2;\sigma_2')$. Thus $(\Delta_1;\pi_2 v_1;[\pi_1 v_1/x]\tau_1'')\ \mathsf{is}\ (\Delta_2;\pi_2 v_2;[\pi_1 v_2/x]\tau_2'')$, $(\Delta_1;[\pi_1 v_1/x]\tau_1'')\ \mathsf{is}\ (\Delta_1;[\pi_1 v_1/x]\sigma_1'')$, and $(\Delta_2;[\pi_1 v_2/x]\tau_2'')\ \mathsf{is}\ (\Delta_2;[\pi_1 v_2/x]\sigma_2'')$, so by the inductive hypothesis we have $(\Delta_1;\pi_2 v_1;[\pi_1 v_1/x]\sigma_1'')\ \mathsf{is}\ (\Delta_2;\pi_2 v_2;[\pi_1 v_2/x]\sigma_2'')$.
   - Case: $\tau_i = \forall\alpha{::}K_i.\tau_i'$. Then $\sigma_i = \forall\alpha{::}L_i.\sigma_i'$.

∎

Lemma 7.1.6 (Logical Transitivity)

1. *If* $(\Delta_1;\tau_1)\ \mathsf{is}\ (\Delta_2;\tau_2)$ *and* $(\Delta_2;\tau_2)\ \mathsf{is}\ (\Delta_2;\sigma_2)$ *then* $(\Delta_1;\tau_1)\ \mathsf{is}\ (\Delta_2;\sigma_2)$.
2. *If* $(\Delta_1;v_1;\tau_1)\ \mathsf{is}\ (\Delta_2;v_2;\tau_2)$ *and* $(\Delta_2;v_2;\tau_2)\ \mathsf{is}\ (\Delta_2;w_2;\sigma_2)$ *then* $(\Delta_1;v_1;\tau_1)\ \mathsf{is}\ (\Delta_2;w_2;\sigma_2)$.

Proof: By induction on the size of types.

1. By context replacement and declarative transitivity, $\Delta_1 \vdash \tau_1 \equiv \sigma_2$.

   - Case: $\tau_i = \mathit{Ty}(A_i)$, $\sigma_2 = \mathit{Ty}(B_2)$, $\Delta_i \triangleright \tau_i \Downarrow \tau_i$, and $\Delta_2 \triangleright \sigma_2 \Downarrow \sigma_2$. Trivial.
   - Case: $\tau_i = \mathbf{S}(v_i : \tau_i')$ and $\sigma_2 = \mathbf{S}(w_2 : \sigma_2')$. $(\Delta_1;v_1;\tau_1')\ \mathsf{is}\ (\Delta_2;v_2;\tau_2')$ and $(\Delta_2;v_2;\tau_2')\ \mathsf{is}\ (\Delta_2;w_2;\sigma_2')$. By the inductive hypothesis, $(\Delta_1;v_1;\tau_1')\ \mathsf{is}\ (\Delta_2;w_2;\sigma_2')$.
   - Case: $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i')\rightharpoonup\tau_i''$ and $\Delta_2 \triangleright \sigma_2 \Downarrow (x{:}\sigma_2')\rightharpoonup\sigma_2''$. Then $(\Delta_1;\tau_1')\ \mathsf{is}\ (\Delta_2;\tau_2')$ and $(\Delta_2;\tau_2')\ \mathsf{is}\ (\Delta_2;\sigma_2')$, so by the inductive hypothesis we have $(\Delta_1;\tau_1')\ \mathsf{is}\ (\Delta_2;\sigma_2')$. Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume that $(\Delta_1';v_1';\tau_1')\ \mathsf{is}\ (\Delta_2';v_2';\sigma_2')$. By reflexivity and inductively by Lemma 7.1.5, $(\Delta_1';v_1';\tau_1')\ \mathsf{is}\ (\Delta_2';v_2';\tau_2')$, so $(\Delta_1';[v_1'/x]\tau_1'')\ \mathsf{is}\ (\Delta_2';[v_2'/x]\tau_2'')$. Now by reflexivity, $(\Delta_2';v_2';\sigma_2')\ \mathsf{is}\ (\Delta_2';v_2';\sigma_2')$, so by reflexivity and inductively by Lemma 7.1.5, $(\Delta_2';v_2';\tau_2')\ \mathsf{is}\ (\Delta_2';v_2';\sigma_2')$. Thus $(\Delta_2';[v_2'/x]\tau_2'')\ \mathsf{is}\ (\Delta_2';[v_2'/x]\sigma_2'')$. By the inductive hypothesis, $(\Delta_1';[v_1'/x]\tau_1'')\ \mathsf{is}\ (\Delta_2';[v_2'/x]\sigma_2'')$.
   - Case: $\Delta_i \triangleright \tau_i \Downarrow (x{:}\tau_i')\times\tau_i''$ and $\Delta_2 \triangleright \sigma_2 \Downarrow (x{:}\sigma_2')\times\sigma_2''$. Same as previous case.

- Case: $\tau_i = \forall\alpha{::}K_i.\tau_i'$ and $\sigma_2 = \forall\alpha{::}L_2.\sigma_2'$. Assume $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$, $\vdash \Delta_1' \equiv \Delta_2'$, and $\Delta_1' \vdash A_1 \equiv A_2 :: K_1$. Since $\Delta_1' \vdash K_1 \equiv K_2$ by Theorem 6.2.2, we have $(\Delta_1'; [A_1/\alpha]\tau_1')\ \mathsf{is}\ (\Delta_2'; [A_2/\alpha]\tau_2')$. Also $\vdash \Delta_2' \equiv \Delta_2'$, $\Delta_2' \vdash K_2 \equiv L_2$, and by context replacement, declarative reflexivity, and subsumption we have $\Delta_2' \vdash A_2 \equiv A_2 :: K_2$, so $(\Delta_2'; [A_2/\alpha]\tau_2')\ \mathsf{is}\ (\Delta_2'; [A_2/\alpha]\sigma_2')$. By the inductive hypothesis, $(\Delta_1'; [A_1/\alpha]\tau_1')\ \mathsf{is}\ (\Delta_2'; [A_2/\alpha]\sigma_2')$.

2. Inductively using context replacement, declarative and algorithmic transitivity, and part 1.

∎

Definition 7.1.7
The judgment $\Gamma \rhd v_1 \simeq v_2$ holds if and only if v_1 and v_2 have a common weak head reduct under typing context Γ; that is, if and only if there exists w such that $\Gamma \rhd v_1 \leadsto^* w$ and $\Gamma \rhd v_2 \leadsto^* w$.

Lemma 7.1.8 (Weak Head Closure)

1. *If $\Delta_1 \rhd v_1 \Leftrightarrow \Delta_2 \rhd v_2$, $\Delta_1 \rhd v_1 \simeq w_1$, and $\Delta_2 \rhd v_2 \simeq w_2$, then $\Delta_1 \rhd w_1 \Leftrightarrow \Delta_2 \rhd w_2$.*
2. *If $(\Delta; v; \tau)\ \mathsf{valid}$, $\Delta \rhd v \simeq w$, and $\Delta \vdash w : \tau$ then $(\Delta; w; \tau)\ \mathsf{valid}$.*
3. *If $(\Delta_1; v_1; \tau_1)\ \mathsf{is}\ (\Delta_2; v_2; \tau_2)$, $\Delta_1 \rhd v_1 \simeq w_1$, $\Delta_2 \rhd v_2 \simeq w_2$, and $\Delta_1 \vdash w_1 \equiv w_2 : \tau_1$ then $(\Delta_1; w_1; \tau_1)\ \mathsf{is}\ (\Delta_2; w_2; \tau_2)$.*

Proof:

1. By definition of the algorithm.

2–3. By simultaneous induction on the sizes of types.

∎

Lemma 7.1.9

1. *If $\Delta \rhd p \uparrow \tau$, $\Delta \rhd p \leftrightarrow \Delta \rhd p$, and $\Delta \vdash p : \tau$, then $(\Delta; p; \tau)\ \mathsf{valid}$.*
2. *If $\Delta_1 \rhd p_1 \uparrow \tau_1$, $\Delta_2 \rhd p_2 \uparrow \tau_2$, $\Delta_1 \rhd p_1 \leftrightarrow \Delta_2 \rhd p_2$, $\Delta_1 \vdash p_1 \equiv p_2 : \tau_1$, and $(\Delta_1; \tau_1)\ \mathsf{is}\ (\Delta_2; \tau_2)$ then $(\Delta_1; p_1; \tau_1)\ \mathsf{is}\ (\Delta_2; p_2; \tau_2)$.*

Proof: By induction on algorithmic derivations and weak head closure. ∎

Corollary 7.1.10
If $(\Delta_1; (\Delta_1(x)))\ \mathsf{is}\ (\Delta_2; (\Delta_2(x)))$ then $(\Delta_1; x; (\Delta_1(x)))\ \mathsf{is}\ (\Delta_2; x; (\Delta_2(x)))$.

Proof: By part 2 of Lemma 7.1.9 with $p_1 = p_2 = x$, $\tau_1 = \Delta_1(x)$, and $\tau_2 = \Delta_2(x)$. ∎

Lemma 7.1.11

1. *If $\Delta \vdash \mathit{Ty}(A)$ then $(\Delta; \mathit{Ty}(A))\ \mathsf{valid}$.*
2. *If $\vdash \Delta_1 \equiv \Delta_2$ and $\Delta_1 \vdash \mathit{Ty}(A_1) \equiv \mathit{Ty}(A_2)$ then $(\Delta_1; \mathit{Ty}(A_1))\ \mathsf{is}\ (\Delta_2; \mathit{Ty}(A_2))$.*

Proof: By induction on the size of types. Note that $\mathit{Ty}(A)$ cannot head-normalize to a truly dependent product or function type, or to a polymorphic or singleton type. ∎

Lemma 7.1.12
If $(\Delta_1; \tau_1)\ \mathsf{is}\ (\Delta_2; \tau_2)$ then $\Delta_1 \rhd \tau_1 \Leftrightarrow \Delta_2 \rhd \tau_2$.

Proof: By induction on the sizes of types. ∎

In the following theorem, not that part 6 uses algorithm equivalence because logical equivalence is defined only for values.

Theorem 7.1.13

1. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash \tau$ *then* $(\Delta_1;\gamma_1\tau)$ is $(\Delta_2;\gamma_2\tau)$
2. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* $(\Delta_1;\gamma_1\tau_1)$ is $(\Delta_2;\gamma_2\tau_2)$
3. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash \tau_1 \leq \tau_2$ *then* $(\Delta_1;\gamma_1\tau_1 \leq \gamma_1\tau_2)$ **is** $(\Delta_2;\gamma_2\tau_1 \leq \gamma_2\tau_2)$
4. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash v : \tau$ *then* $(\Delta_1;\gamma_1 v;\gamma_1\tau)$ is $(\Delta_2;\gamma_2 v;\gamma_2\tau)$
5. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash v_1 \equiv v_2 : \tau$ *then* $(\Delta_1;\gamma_1 v_1;\gamma_1\tau_1)$ is $(\Delta_2;\gamma_2 v_2;\gamma_2\tau_2)$
6. *If* $(\Delta_1;\gamma_1;\Gamma)$ is $(\Delta_2;\gamma_2;\Gamma)$ *and* $\Gamma \vdash e_1 \equiv e_2 : \tau$ *then* $\Delta_1 \triangleright \gamma_1 e_1 \Leftrightarrow \Delta_2 \triangleright \gamma_2 e_2$.

Proof: By induction on derivations.

Type Well-formedness Rules: $\Gamma \vdash \tau$. In all cases, by Substitution we have $\Delta_1 \vdash \gamma_1\tau$ and $\Delta_2 \vdash \gamma_2\tau$ and by Functionality we have $\Delta_1 \vdash \gamma_1\tau \equiv \gamma_2\tau$.

- Case: Rule 2.45
$$\frac{\Gamma \vdash A :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A)}$$
By Functionality, $\Delta_1 \vdash \gamma_1 A_1 \equiv \gamma_2 A_2 :: \mathbf{T}$. By Lemma 7.1.11, $(\Delta_1;\, \mathit{Ty}(\gamma_1 A_1))$ is $(\Delta_2;\, \mathit{Ty}(\gamma_2 A_2))$.
- Case: Rule 2.46
$$\frac{\Gamma \vdash v : \tau \qquad \tau \text{ not a singleton}}{\Gamma \vdash \mathbf{S}(v : \tau)}$$
By the inductive hypothesis, $(\Delta_1;\gamma_1 v;\gamma_1\tau)$ is $(\Delta_2;\gamma_2 v;\gamma_2\tau)$. Thus $(\Delta_1;\mathbf{S}(\gamma_1 v : \gamma_1\tau))$ valid, $(\Delta_2;\mathbf{S}(\gamma_2 v : \gamma_2\tau))$ valid, and $(\Delta_1;\mathbf{S}(\gamma_1 v : \gamma_1\tau))$ is $(\Delta_2;\mathbf{S}(\gamma_2 v : \gamma_2\tau))$.
- Case: Rule 2.47
$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau')\rightharpoonup\tau''}$$
Same argument as for Π kinds in Theorem 5.3.10.
- Case: Rule 2.48
$$\frac{\Gamma, x{:}\tau' \vdash \tau''}{\Gamma \vdash (x{:}\tau')\times\tau''}$$
Same argument as for Σ kinds in Theorem 5.3.10.
- Case: Rule 2.49
$$\frac{\Gamma, \alpha{::}K \vdash \tau}{\Gamma \vdash \forall\alpha{::}K.\tau}$$
There is a strict subderivation, $\Gamma \vdash K$, so by substitution and functionality we have $\Delta_1 \vdash \gamma_1 K$, $\Delta_2 \vdash \gamma_2 K$, and $\Delta_1 \vdash \gamma_1 K \equiv \gamma_2 K$. Assume $\Delta_1' \supseteq \Delta_1$ and $\Delta_1'' \supseteq \Delta_1$ and $\Delta_1' \vdash A_1 \equiv A_2 :: \gamma_1 K$. Then $(\Delta_1';\gamma_1[\alpha{\mapsto}A_1];\Gamma,\alpha{::}K)$ is $(\Delta_1'';\gamma_1[\alpha{\mapsto}A_2];\Gamma,\alpha{::}K)$. By the inductive hypothesis, $(\Delta_1';(\gamma_1[\alpha{\mapsto}A_1])\tau)$ is $(\Delta_1'';(\gamma_1[\alpha{\mapsto}A_2])\tau)$. That is, $(\Delta_1';[A_1/\alpha](\gamma_1[\alpha{\mapsto}\alpha]\tau))$ is $(\Delta_1'';[A_2/\alpha](\gamma_1[\alpha{\mapsto}\alpha]\tau))$. Thus $(\Delta_1';\gamma_1(\forall\alpha{::}K.\tau))$ valid. Similar arguments show that $(\Delta_2';\gamma_2(\forall\alpha{::}K.\tau))$ valid and $(\Delta_1';\gamma_1(\forall\alpha{::}K.\tau))$ is $(\Delta_2';\gamma_2(\forall\alpha{::}K.\tau))$.

Type Equivalence: $\Gamma \vdash \tau_1 \equiv \tau_2$. In all cases, by validity and substitution we have $\Delta_1 \vdash \gamma_1\tau_1$ and $\Delta_2 \vdash \gamma_2\tau_2$ and by functionality we have $\Delta_1 \vdash \gamma_1\tau_1 \equiv \gamma_2\tau_2$.

- Case: Rule 2.50.
$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \equiv \tau}$$
By the inductive hypothesis.
- Case: Rule 2.51.
$$\frac{\Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash \tau \equiv \tau'}$$
By symmetry, $(\Delta_2;\gamma_2;\Gamma)$ is $(\Delta_1;\gamma_1;\Gamma)$. By the inductive hypothesis, $(\Delta_2;\gamma_2\tau')$ is $(\Delta_1;\gamma_1\tau)$. By Symmetry again, $(\Delta_1;\gamma_1\tau)$ is $(\Delta_2;\gamma_2\tau')$.
- Case: Rule 2.52.
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma \vdash \tau_2 \equiv \tau_3}{\Gamma \vdash \tau_1 \equiv \tau_3}$$
Same proof as for transitive rule for constructor equivalence in Theorem 5.3.10.
- Case: Rule 2.53.
$$\frac{\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1) \equiv \mathit{Ty}(A_2)}$$
By functionality, $\Delta_1 \vdash \gamma_1 A_1 \equiv \gamma_2 A_2 :: \mathbf{T}$, so by Lemma 7.1.11, $(\Delta_1;\, \mathit{Ty}(\gamma_1 A_1))$ is $(\Delta_2;\, \mathit{Ty}(\gamma_2 A_2))$.
- Case: Rule 2.58.
$$\frac{\Gamma \vdash A_1 :: \mathbf{T} \qquad \Gamma \vdash A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1 \times A_2) \equiv \mathit{Ty}(A_1) \times \mathit{Ty}(A_2)}$$
First, $\Delta_1 \triangleright \gamma_1(\mathit{Ty}(A_1 \times A_2)) \Downarrow \mathit{Ty}(\gamma_1 A_1) \times \mathit{Ty}(\gamma_1 A_2)$ and $\Delta_2 \triangleright \gamma_2(\mathit{Ty}(A_1) \times \mathit{Ty}(A_2)) \Downarrow \mathit{Ty}(\gamma_2 A_1) \times \mathit{Ty}(\gamma_2 A_2)$. By functionality, $\Delta_1 \vdash \gamma_1 A_1 \equiv \gamma_2 A_1 :: \mathbf{T}$ and $\Delta_1 \vdash \gamma_1 A_2 \equiv \gamma_2 A_2 :: \mathbf{T}$. By Lemma 7.1.11, $(\Delta_1;\, \mathit{Ty}(\gamma_1 A_1))$ is $(\Delta_2;\, \mathit{Ty}(\gamma_2 A_1))$ and $(\Delta_1;\, \mathit{Ty}(\gamma_1 A_2))$ is $(\Delta_2;\, \mathit{Ty}(\gamma_2 A_2))$.
- Case: Rule 2.59.
$$\frac{\Gamma \vdash A_1 :: \mathbf{T} \qquad \Gamma \vdash A_2 :: \mathbf{T}}{\Gamma \vdash \mathit{Ty}(A_1 \rightharpoonup A_2) \equiv \mathit{Ty}(A_1) \rightharpoonup \mathit{Ty}(A_2)}$$
Analogous to the proof for Rule 2.58.
- Case: Rule 2.54.
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma \vdash v_1 \equiv v_2 : \tau_1 \qquad \tau_1, \tau_2 \text{ not a singleton}}{\Gamma \vdash \mathbf{S}(v_1 : \tau_1) \equiv \mathbf{S}(v_2 : \tau_2)}$$
By the inductive hypothesis, $(\Delta_1;\gamma_1 v_1;\gamma_1\tau_1)$ is $(\Delta_2;\gamma_2 v_2;\gamma_2\tau_1)$ and $(\Delta_2;\gamma_2\tau_1)$ is $(\Delta_2;\gamma_2\tau_2)$. By Lemma 7.1.5, $(\Delta_1;\gamma_2 v_2;\gamma_2\tau_2)$ valid and $(\Delta_1;\gamma_1 v_1;\gamma_1\tau_1)$ is $(\Delta_2;\gamma_2 v_2;\gamma_2\tau_2)$.
- Case: Rule 2.55.
$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma, x{:}\tau_1' \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash (x{:}\tau_1') \rightharpoonup \tau_1'' \equiv (x{:}\tau_2') \rightharpoonup \tau_2''}$$
As in the proof for Π kinds.

- Case: Rule 2.56.

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \qquad \Gamma, x{:}\tau_1 \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash (x{:}\tau_1')\times\tau_1'' \equiv (x{:}\tau_2')\times\tau_2''}$$

  As in the proof for Σ kinds.

- Case: Rule 2.57.

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma, \alpha{::}K_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \equiv \forall\alpha{::}K_2.\tau_2}$$

  Analogous to the proofs for the previous two rules, also using functionality to show $\Delta_1 \vdash \gamma_1 K_1 \equiv \gamma_2 K_2$.

Subtyping: $\Gamma \vdash \tau_1 \leq \tau_2$. In all cases, by validity and substitution we have $\Delta_1 \vdash \gamma_1\tau_1$, $\Delta_2 \vdash \gamma_2\tau_2$, $\Delta_1 \vdash \gamma_1\tau_1 \leq \gamma_1\tau_2$, and $\Delta_2 \vdash \gamma_2\tau_1 \leq \gamma_2\tau_2$. By functionality we have $\Delta_1 \vdash \gamma_1\tau_1 \leq \gamma_2\tau_2$.

- Case: Rule 2.60

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \tau_1 \leq \tau_2}$$

  Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume $(\Delta_1'; v_1; \gamma_1\tau_1)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_1)$. By the inductive hypothesis, $(\Delta_1'; \gamma_1\tau_1)\ \mathsf{is}\ (\Delta_1'; \gamma_1\tau_2)$ and $(\Delta_2'; \gamma_2\tau_1)\ \mathsf{is}\ (\Delta_2'; \gamma_2\tau_2)$. By Lemma 7.1.5, $(\Delta_1'; v_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_2)$.

- Case: Rule 2.61

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3}$$

  Obvious by inductive hypothesis that $(\Delta_1'; v_1; \gamma_1\tau_1)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_1)$ implies $(\Delta_1'; v_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_2)$ which implies $(\Delta_1'; v_1; \gamma_1\tau_3)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_3)$.

- Case: Rule 2.62.

$$\frac{\Gamma \vdash w : \tau \qquad \tau \text{ not a singleton}}{\Gamma \vdash \mathbf{S}(w : \tau) \leq \tau}$$

  Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ and assume $(\Delta_1'; v_1; \mathbf{S}(\gamma_1 w : \gamma_1\tau))\ \mathsf{is}\ (\Delta_2'; v_2; \mathbf{S}(\gamma_2 w : \gamma_2\tau))$. Then by definition of the logical relation, $(\Delta_1'; v_1; \gamma_1\tau)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau)$.

- Case: Rule 2.63

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{S}(w_1 : \tau_1)\\ \Gamma \vdash w_1 \equiv w_2 : \tau_2 \qquad \Gamma \vdash \tau_1 \leq \tau_2\end{array}}{\Gamma \vdash \mathbf{S}(w_1 : \tau_1) \leq \mathbf{S}(w_2 : \tau_2)}(\tau_1, \tau_2 \text{ not a singleton})$$

  Let $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ be given, and assume $(\Delta_1'; v_1; \mathbf{S}(\gamma_1 w_1 : \gamma_1\tau_1))\ \mathsf{is}\ (\Delta_2'; v_2; \mathbf{S}(\gamma_2 w_1 : \gamma_2\tau_1))$. Then $(\Delta_1'; v_1; \gamma_1\tau_1)\ \mathsf{is}\ (\Delta_1'; \gamma_1 w_1; \gamma_1\tau_1)$ and $(\Delta_2'; v_2; \gamma_2\tau_1)\ \mathsf{is}\ (\Delta_2'; \gamma_2 w_1; \gamma_2\tau_1)$ and $(\Delta_1'; v_1; \gamma_1\tau_1)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_1)$. Using the inductive hypothesis we have $(\Delta_1'; v_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_1'; \gamma_1 w_1; \gamma_1\tau_2)$, and $(\Delta_2'; v_2; \gamma_2\tau_2)\ \mathsf{is}\ (\Delta_2'; \gamma_2 w_1; \gamma_2\tau_2)$, and $(\Delta_1'; v_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_2'; v_2; \gamma_2\tau_2)$. Again by the inductive hypothesis, $(\Delta_1'; \gamma_1 w_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_1'; \gamma_1 w_2; \gamma_1\tau_2)$ and $(\Delta_2'; \gamma_2 w_1; \gamma_2\tau_2)\ \mathsf{is}\ (\Delta_2'; \gamma_2 w_2; \gamma_2\tau_2)$. By transitivity, $(\Delta_1'; v_1; \gamma_1\tau_2)\ \mathsf{is}\ (\Delta_1'; \gamma_1 w_2; \gamma_1\tau_2)$ and $(\Delta_2'; v_2; \gamma_2\tau_2)\ \mathsf{is}\ (\Delta_2'; \gamma_2 w_2; \gamma_2\tau_2)$. Therefore $(\Delta_1'; v_1; \mathbf{S}(\gamma_1 w_2 : \gamma_1\tau_2))\ \mathsf{is}\ (\Delta_2'; v_2; \mathbf{S}(\gamma_2 w_2 : \gamma_2\tau_2))$.

- Case: Rule 2.64.
$$\frac{\Gamma \vdash (x{:}\tau_1')\times\tau_1'' \quad \Gamma \vdash \tau_2' \leq \tau_1' \qquad \Gamma, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''}{\Gamma \vdash (x{:}\tau_1')\rightharpoonup\tau_1'' \leq (x{:}\tau_2')\rightharpoonup\tau_2''}$$
Same proof as for subkinding of Π kinds.

- Case: Rule 2.65.
$$\frac{\Gamma \vdash (x{:}\tau_2')\times\tau_2'' \quad \Gamma \vdash \tau_1' \leq \tau_2' \qquad \Gamma, x{:}\tau_1 \vdash \tau_1'' \leq \tau_2''}{\Gamma \vdash (x{:}\tau_1')\times\tau_1'' \leq (x{:}\tau_2')\times\tau_2''}$$
Same proof as for subkinding of Σ kinds.

- Case: Rule 2.66.
$$\frac{\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \quad \Gamma \vdash K_2 \leq K_1 \qquad \Gamma, \alpha{::}K_2 \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \forall\alpha{::}K_1.\tau_1 \leq \forall\alpha{::}K_2.\tau_2}$$
Analogous to the proof for function types.

Term Validity: $\Gamma \vdash e : \tau$. In all cases, by validity and Substitution we have $\Delta_1 \vdash \gamma_1 e : \gamma_1\tau_1$ and $\Delta_2 \vdash \gamma_2 e : \gamma_2\tau$. By functionality we have $\Delta_1 \vdash \gamma_1 e \equiv \gamma_2 e : \gamma_1\tau$.

- Case: Rule 2.67
$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash n : \mathsf{int}}$$
Recall that $\mathsf{int} = \mathit{Ty}(\mathsf{Int})$. Now $\Delta_i \triangleright \mathsf{int} \Downarrow \mathsf{int}$, and $\Delta_i \triangleright n \Leftrightarrow \Delta_i \triangleright n$, and $\Delta_1 \triangleright n \Leftrightarrow \Delta_2 \triangleright n$. Since $(\Delta_1; \mathsf{int})$ is $(\Delta_2; \mathsf{int})$, we have $(\Delta_i; n; \mathsf{int})$ valid and $(\Delta_1; n; \mathsf{int})$ is $(\Delta_2; n; \mathsf{int})$.

- Case: Rule 2.68
$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash x : \Gamma(x)}$$
By the assumptions for γ_1 and γ_2.

- Case: Rule 2.69
$$\frac{\Gamma, f{:}(x{:}\tau')\rightharpoonup\tau'', x{:}\tau' \vdash e : \tau''}{\Gamma \vdash \mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e : (x{:}\tau')\rightharpoonup\tau''}$$
There are strict subderivations $\Gamma \vdash (x{:}\tau')\rightharpoonup\tau''$ and by inversion, $\Gamma \vdash \tau'$ and $\Gamma, x{:}\tau' \vdash \tau''$. By the inductive hypothesis, $(\Delta_1; \gamma_1\tau')$ is $(\Delta_2; \gamma_2\tau')$ and $(\Delta_1; \gamma_1((x{:}\tau')\rightharpoonup\tau''))$ is $(\Delta_2; \gamma_2((x{:}\tau')\rightharpoonup\tau''))$. Then $(\Delta_1, f{:}\gamma_1((x{:}\tau')\rightharpoonup\tau''), x{:}\gamma_1\tau'; \gamma_1[f\mapsto f][x\mapsto x]; \Gamma, f{:}(x{:}\tau')\rightharpoonup\tau'', x{:}\tau')$ is $(\Delta_2, f{:}\gamma_2((x{:}\tau')\rightharpoonup\tau''), x{:}\gamma_2\tau'; \gamma_2[f\mapsto f][x\mapsto x]; \Gamma, f{:}(x{:}\tau')\rightharpoonup\tau'', x{:}\tau')$. By the inductive hypothesis, $\Delta_1, f{:}\gamma_1((x{:}\tau')\rightharpoonup\tau''), x{:}\gamma_1\tau' \triangleright (\gamma_1[f\mapsto f][x\mapsto x])e \Leftrightarrow \Delta_2, f{:}\gamma_2((x{:}\tau')\rightharpoonup\tau''), x{:}\gamma_2\tau' \triangleright (\gamma_2[f\mapsto f][x\mapsto x])e$. Similarly, by the inductive hypothesis $(\Delta_1, x{:}\gamma_1\tau'; (\gamma_1[\alpha\mapsto\alpha])\tau'')$ is $(\Delta_2, x{:}\gamma_2\tau'; (\gamma_2[\alpha\mapsto\alpha])\tau'')$, so $\Delta_1, x{:}\gamma_1\tau' \triangleright (\gamma_1[\alpha\mapsto\alpha])\tau'' \Leftrightarrow \Delta_2, x{:}\gamma_2\tau' \triangleright (\gamma_2[\alpha\mapsto\alpha])\tau''$. Therefore $\Delta_1 \triangleright \gamma_1(\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e) \Leftrightarrow \Delta_2 \triangleright \gamma_2(\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e)$, so $(\Delta_1; \gamma_1(\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e); \gamma_1((x{:}\tau')\rightharpoonup\tau''))$ is $(\Delta_2; \gamma_2(\mathsf{fun}\, f(x{:}\tau'){:}\tau''\ \mathsf{is}\ e); \gamma_2((x{:}\tau')\rightharpoonup\tau''))$.

- Case: Rule 2.70.

$$\frac{\Gamma, \alpha{::}K \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha{::}K){:}\tau.e : \forall\alpha{::}K.\tau}$$

Analogous to previous case, using
$(\Delta_1, \alpha{::}\gamma_1 K; \gamma_1[\alpha{\mapsto}\alpha]; \Gamma, \alpha{::}K)$ is $(\Delta_2, \alpha{::}\gamma_2 K; \gamma_2[\alpha{\mapsto}\alpha]; \Gamma, \alpha{::}K)$.

- Case: Rule 2.71.

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \langle v_1, v_2\rangle : \tau_1{\times}\tau_2}$$

By the inductive hypothesis, $(\Delta_1; \gamma_1 v_1; \gamma_1\tau_1)$ is $(\Delta_2; \gamma_2 v_1; \gamma_2\tau_1)$ and
$(\Delta_1; \gamma_1 v_2; \gamma_1\tau_2)$ is $(\Delta_2; \gamma_2 v_2; \gamma_2\tau_2)$. By Lemma 7.1.8, we have
$(\Delta_1; \pi_1\langle\gamma_1 v_1, \gamma_1 v_2\rangle; \gamma_1\tau_1)$ is $(\Delta_2; \pi_1\langle\gamma_2 v_1, \gamma_2 v_2\rangle; \gamma_2\tau_1)$. and
$(\Delta_1; \pi_2\langle\gamma_1 v_1, \gamma_1 v_2\rangle; \gamma_1\tau_2)$ is $(\Delta_2; \pi_2\langle\gamma_2 v_1, \gamma_2 v_2\rangle; \gamma_2\tau_2)$.

- Case: Rule 2.72.

$$\frac{\Gamma \vdash v : (x{:}\tau'){\times}\tau''}{\Gamma \vdash \pi_1 v : \tau'}$$

By the inductive hypothesis, $(\Delta_1; \gamma_1 v; \gamma_1((x{:}\tau'){\times}\tau''))$ is $(\Delta_2; \gamma_2 v; \gamma_2((x{:}\tau'){\times}\tau''))$. Thus
$(\Delta_1; \pi_1(\gamma_1 v); \gamma_1\tau')$ is $(\Delta_2; \pi_1(\gamma_2 v); \gamma_2\tau')$.

- Case: Rule 2.73

$$\frac{\Gamma \vdash v : (x{:}\tau'){\times}\tau''}{\Gamma \vdash \pi_2 v : [\pi_1 v/x]\tau''}$$

By the inductive hypothesis, $(\Delta_1; \gamma_1 v; \gamma_1((x{:}\tau'){\times}\tau''))$ is $(\Delta_2; \gamma_2 v; \gamma_2((x{:}\tau'){\times}\tau''))$. Thus
$(\Delta_1; \pi_2(\gamma_1 v); \gamma_1([\pi_1 v/x]\tau''))$ is $(\Delta_2; \pi_2(\gamma_2 v); \gamma_2([\pi_1 v/x]\tau''))$.

- Case: Rule 2.74.

$$\frac{\Gamma \vdash v : \tau'{\rightharpoonup}\tau'' \qquad \Gamma \vdash v' : \tau'}{\Gamma \vdash v\, v' : \tau''}$$

By the inductive hypothesis and definition of the logical relations, $\Delta_1 \triangleright \gamma_1 v \Leftrightarrow \Delta_2 \triangleright \gamma_2 v$ and
$\Delta_1 \triangleright \gamma_1 v' \Leftrightarrow \Delta_2 \triangleright \gamma_2 v'$. Thus $\Delta_1 \triangleright \gamma_1(v\, v') \Leftrightarrow \Delta_2 \triangleright \gamma_2(v\, v')$.

- Case: Rule 2.75

$$\frac{\Gamma \vdash v : \forall\alpha{::}K.\tau \qquad \Gamma \vdash A :: K}{\Gamma \vdash v\, A : [A/\alpha]\tau}$$

By the inductive hypothesis and the definition of the logical relations, $\Delta_1 \triangleright \gamma_1 v \Leftrightarrow \Delta_2 \triangleright \gamma_2 v$.
That is, $\Delta_1 \triangleright \gamma_1 v \Downarrow w_1$ and $\Delta_2 \triangleright \gamma_2 v \Downarrow w_2$ and $\Delta_1 \triangleright w_1 \leftrightarrow \Delta_2 w_2$. By substitution,
$\Delta_1 \vdash \gamma_1 v : \gamma_1(\forall\alpha{::}K.\tau)$, so by soundness of weak head reduction we have
$\Delta_1 \vdash w_1 : \gamma_1(\forall\alpha{::}K.\tau)$. Let $\Delta_1 \vdash w_1 : L_1$. Then $\Delta_1 \vdash {L_1}^{\$} \leq \gamma_1(\forall\alpha{::}K.\tau)$ by Lemma 6.3.1.
By Theorem 6.2.3, ${L_1}^{\$} = \forall\alpha{::}L_1'.\sigma_1''$ with $\Delta_1 \vdash \gamma_1 K \leq L_1'$. Similarly, $\Delta_2 \triangleright w_2 \Uparrow \forall\alpha{::}L_2'.\sigma_2''$
with $\Delta_2 \vdash \gamma_2 K \leq L_2'$. Now either both w_1 and w_2 are paths or they are are both
polymorphic abstractions. In either case, $\Delta_1 \vdash \forall\alpha{::}L_1'.\sigma_1'' \equiv \forall\alpha{::}L_2'.\sigma_2''$. By Theorem 6.2.2,
$\Delta_1 \vdash L_1' \equiv L_2'$. Then $\Delta_1 \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1 K$ by functionality, so $\Delta_1 \vdash \gamma_1 A \equiv \gamma_2 A :: \gamma_1 L_1'$ by
subsumption. Then $\Delta_1 \triangleright \gamma_1 A :: \gamma_1 K \Leftrightarrow \Delta_2 \triangleright \gamma_2 A :: \gamma_2 K$ by the completeness of constructor
equivalence, and therefore $\Delta_1 \triangleright \gamma_1(v\, A) \Leftrightarrow \Delta_2 \triangleright \gamma_2(v\, A)$.

- Case: Rule 2.76
$$\frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x{:}\tau' \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash (\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}) : \tau}$$

  By the inductive hypothesis and the definition of the logical relations, $\Delta_1 \triangleright \gamma_1 e' \Leftrightarrow \Delta_2 \triangleright \gamma_2 e'$. There is a strict subderivation $\Gamma \vdash \tau'$. By the inductive hypothesis $(\Delta_1; \gamma_1\tau')\ \mathsf{is}\ (\Delta_2; \gamma_2\tau')$, so by Lemma 7.1.12 we have $\Delta_1 \triangleright \gamma_1\tau' \Leftrightarrow \Delta_2 \triangleright \gamma_2\tau'$. Similarly, $\Delta_1 \triangleright \gamma_1\tau \Leftrightarrow \Delta_2 \triangleright \gamma_2\tau$. Finally, using Corollary 7.1.10 we have $(\Delta_1, x{:}\gamma_1\tau'; \gamma_1[\alpha{\mapsto}\alpha]; \Gamma, x{:}\tau')\ \mathsf{is}\ (\Delta_2, x{:}\gamma_2\tau'; \gamma_2[\alpha{\mapsto}\alpha]; \Gamma, x{:}\tau')$, so by the inductive hypothesis $\Delta_1, x{:}\gamma_1\tau' \triangleright (\gamma_1[\alpha{\mapsto}\alpha])e \Leftrightarrow \Delta_2, x{:}\gamma_2\tau' \triangleright (\gamma_2[\alpha{\mapsto}\alpha])e$. Therefore $\Delta_1 \triangleright \gamma_1(\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}) \Leftrightarrow \Delta_2 \triangleright \gamma_2(\mathsf{let}\ x{:}\tau'{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end})$.

Term Equivalence: $\Gamma \vdash e_1 \equiv e_2 : \tau$. All these cases are straightforward, similar to cases already proved. ∎

Lemma 7.1.14

1. *If* $\Gamma \vdash$ *ok then* $(\Gamma; \mathrm{id}; \Gamma)\ \mathsf{valid}$ *where* id *is the identity function.*
2. *If* $\Gamma \vdash$ *ok* $(\Gamma; \mathrm{id}; \Gamma)\ \mathsf{is}\ (\Gamma; \mathrm{id}; \Gamma)$ *where* id *is the identity function.*

Proof:

1. By induction on the proof of $\Gamma \vdash$ ok.
   - Case: Empty context. Vacuous.
   - Case: $\Gamma, \alpha{::}K \vdash$ ok because $\Gamma \vdash K$.
     By the inductive hypothesis and monotonicity.
   - Case: $\Gamma, x{:}\tau \vdash$ ok because $\Gamma \vdash \tau$.
     (a) By Proposition 3.1.1, $\Gamma \vdash \tau$, and $\Gamma \vdash$ ok.
     (b) Also, $x \notin \mathrm{dom}(\Gamma)$.
     (c) By the inductive hypothesis, $(\Gamma; y; \Gamma(y))\ \mathsf{valid}$ for all $y \in \mathrm{dom}(\Gamma)$ and $(\Gamma; \alpha; \Gamma(\alpha))\ \mathsf{valid}$ for all $\alpha \in \mathrm{dom}(\Gamma)$.
     (d) By monotonicity, $(\Gamma, x{:}\tau; y; ((\Gamma, x{:}\tau)y))\ \mathsf{valid}$ for all $y \in \mathrm{dom}(\Gamma)$. and $(\Gamma, x{:}\tau; \alpha; ((\Gamma, x{:}\tau)\alpha))\ \mathsf{valid}$ for all $\alpha \in \mathrm{dom}(\Gamma)$.
     (e) By Theorem 7.1.13, $(\Gamma; \tau)\ \mathsf{valid}$
     (f) and by monotonicity $(\Gamma, x{:}\tau; \tau)\ \mathsf{valid}$
     (g) Now by Corollary 7.1.10, $(\Gamma, x{:}\tau; x; \tau)\ \mathsf{valid}$.
     (h) Hence $(\Gamma, x{:}\tau; \mathrm{id}; \Gamma, x{:}\tau)\ \mathsf{valid}$.
2. By part 1 and reflexivity.

∎

This yields a completeness result for the symmetrized algorithm:

Corollary 7.1.15

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* $(\Gamma; \tau_1)\ \mathsf{is}\ (\Gamma; \tau_2)$.
2. *If* $\Gamma \vdash e_1 \equiv e_2 : \tau$ *then* $(\Gamma; e_1; \tau)\ \mathsf{is}\ (\Gamma; e_2; \tau)$.
3. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* $\Gamma \triangleright \tau_1 \Leftrightarrow \Gamma \triangleright \tau_2$.

4. If $\Gamma \vdash e_1 \equiv e_2 : \tau$ then $\Gamma \triangleright e_1 \Leftrightarrow \Gamma \triangleright e_2$.

Proof:

1,2 By Lemma 7.1.14, we can apply the Theorem 7.1.13 with γ_1 and γ_2 being identity substitutions.

3,4 Follows directly from parts 1 and 2 and the definition of the logical relations.

∎

Again, use of a size function for algorithmic equivalence (number of non head-normalization rules used) allows the proof to be transferred to the original equivalence algorithm.

Theorem 7.1.16

1. *If $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash e_1 : \tau$, $\Gamma_2 \vdash e_2 : \tau$, and $\Gamma_1 \triangleright e_1 \Leftrightarrow \Gamma_2 \triangleright e_2$ then $\Gamma_1 \triangleright e_1 \Leftrightarrow e_2$.*
2. *If $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash e_1 : \tau$, $\Gamma_2 \vdash e_2 : \tau$, and $\Gamma_1 \triangleright e_1 \Leftrightarrow \Gamma_2 \triangleright e_2$ then $\Gamma_1 \triangleright e_1 \leftrightarrow e_2$.*
3. *If $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash \tau_1$, $\Gamma_2 \vdash \tau_2$, and $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \Gamma_2 \triangleright \tau_2$ then $\Gamma_1 \triangleright \tau_1 \Leftrightarrow \tau_2$.*
4. *If $\vdash \Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash \tau_1$, $\Gamma_2 \vdash \tau_2$, and $\Gamma_1 \triangleright \tau_1 \leftrightarrow \Gamma_2 \triangleright \tau_2$ then $\Gamma_1 \triangleright \tau_1 \leftrightarrow \tau_2$.*

Corollary 7.1.17 (Completeness for Type and Term Equivalence)

1. *If $\Gamma \vdash e_1 \equiv e_2 : \tau$ then $\Gamma \triangleright e_1 \Leftrightarrow e_2$.*
2. *If $\Gamma \vdash \tau_1 \equiv \tau_2$ then $\Gamma \triangleright \tau_1 \Leftrightarrow \tau_2$.*

Theorem 7.1.18

1. *If $\Gamma \triangleright \tau_1 \Leftrightarrow \tau_1$ and $\Gamma \triangleright \tau_2 \Leftrightarrow \tau_2$ then it is decidable whether $\Gamma \triangleright \tau_1 \Leftrightarrow \tau_2$.*
2. *If $\Gamma \triangleright e_1 \Leftrightarrow e_1$ and $\Gamma \triangleright e_2 \Leftrightarrow e_2$ then it is decidable whether $\Gamma \triangleright e_1 \Leftrightarrow e_2$.*

Corollary 7.1.19 (Decidability of Type and Term Equivalence)

1. *If $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ then it is decidable whether $\Gamma \vdash \tau_1 \equiv \tau_2$.*
2. *If $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ then it is decidable whether $\Gamma \vdash e_1 \equiv e_2 : \tau$.*

Proof: Follows from Theorem 7.1.18 and by soundness and completeness of the equivalence algorithms. ∎

7.2 Completeness and Decidability for Subtyping and Validity

Given completeness for term equivalence, proving completeness of the subtyping algorithm would be straightforward if it were not for transitivity (Rule 2.61). Proving transitivity of the algorithm requires some care because of polymorphic types, and the fact that changes to kinds in the typing context affect type head-normalization.

Reflexivity, in contrast, is direct

Lemma 7.2.1
If $\Gamma \vdash \tau$ then $\Gamma \triangleright \tau \Downarrow \sigma$ and $\Gamma \triangleright \sigma \sqsubseteq \sigma$ (i.e., $\Gamma \triangleright \tau \leq \tau$).

Proof: By induction on the proof of $\Gamma \vdash \tau$, using correctness of the term, kind, and constructor equivalence algorithms. ∎

Proving transitivity requires showing that the algorithm obeys a weakening property: types in the context can be replaced by subtypes, and kinds in the context can be replaced by subkinds. Half of this is straightforward:

Lemma 7.2.2 (Algorithmic Weakening for Term Variables)
Assume $\Gamma' \vdash \sigma_2 \leq \sigma_1$.

1. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash v_1 : \tau$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash v_2 : \tau$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd v_1 \Leftrightarrow v_2$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd v_1 \Leftrightarrow v_2$.
2. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_1$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_2$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd \tau_1 \Leftrightarrow \tau_2$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd \tau_1 \Leftrightarrow \tau_2$.
3. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_1$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_2$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd \tau_1 \sqsubseteq \tau_2$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd \tau_1 \sqsubseteq \tau_2$.
4. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_1$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau_2$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd \tau_1 \leq \tau_2$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd \tau_1 \leq \tau_2$.
5. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash ok$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd \tau$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd \tau$.
6. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash ok$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd e \rightrightarrows \tau$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd e \rightrightarrows \tau$.
7. *If* $\Gamma', x{:}\sigma_1, \Gamma'' \vdash \tau$ *and* $\Gamma', x{:}\sigma_1, \Gamma'' \rhd e \leftleftarrows \tau$ *then* $\Gamma', x{:}\sigma_2, \Gamma'' \rhd e \leftleftarrows \tau$.

Proof:

1,2. By soundness and completeness for type/term equivalence, and Corollary 3.2.8.

3,4. By induction on algorithmic derivations and part 1. (For part 4, note that head-normalization of types is completely unaffected by the type of x.)

5–7. By induction on algorithmic derivations and part 4.

∎

However, modifying kinds in the context affects head-normalization of types, and hence it is harder to show that algorithmic subtyping is preserved when kinds in the context are made more specific.

I solve this problem with a two-step process. First I prove soundness and completeness for the algorithm applied to the subset of types not containing the universal quantifier. I then use this to show the required weakening property, which then allows a proof of full transitivity. The success of this method depends critically on the predicativity of MIL_0.

First, any two related types either both contain a universal quantifier, or neither do.

Proposition 7.2.3

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *then* τ_1 *contains a* $\forall$ *if and only if* τ_2 *contains a* $\forall$.
2. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *then* τ_1 *contains a* $\forall$ *if and only if* τ_2 *contains a* $\forall$.

Proof: By induction on derivations. ∎

Lemma 7.2.4 (Pre-transitivity of Algorithmic Subtyping)
Assume τ_1, τ_2, *and* τ_3 *contain no* $\forall$*'s, and that* $\Gamma \vdash \tau_1$, $\Gamma \vdash \tau_2$, *and* $\Gamma \vdash \tau_3$.

1. *If* $\Gamma \rhd \tau_1 \sqsubseteq \tau_2$ *and* $\Gamma \rhd \tau_2 \sqsubseteq \tau_3$ *then* $\Gamma \rhd \tau_1 \sqsubseteq \tau_3$.
2. *If* $\Gamma \rhd \tau_1 \leq \tau_2$ *and* $\Gamma \rhd \tau_2 \leq \tau_3$ *then* $\Gamma \rhd \tau_1 \leq \tau_3$.

Proof: By simultaneous induction on $size(\Gamma;\tau_1) + size(\Gamma;\tau_2) + size(\Gamma;\tau_3)$.

1.
   - Case: $\Gamma \rhd Ty(A_1) \sqsubseteq Ty(A_2) \sqsubseteq Ty(A_3)$. By transitivity of the constructor equivalence algorithm.
   - Case: $\Gamma \rhd \mathbf{S}(v_1 : \tau_1') \sqsubseteq \mathbf{S}(v_3 : \tau_2') \sqsubseteq \mathbf{S}(v_3 : \tau_3')$. By the inductive hypothesis, $\Gamma \rhd \tau_1' \leq \tau_3'$. By the correctness of algorithmic term equivalence, $\Gamma \rhd v_1 \Leftrightarrow v_3$.

- Case: $\Gamma \triangleright \mathbf{S}(v_1 : \tau_1') \sqsubseteq \mathbf{S}(v_3 : \tau_2') \sqsubseteq \tau_3$, where τ_3 is not a singleton. By the inductive hypothesis, $\Gamma \triangleright \tau_1' \leq \tau_3$.
- Case: $\Gamma \triangleright \mathbf{S}(v_1 : \tau_1') \sqsubseteq \tau_2 \sqsubseteq \tau_3$, where τ_2 and τ_3 are not singletons. By the inductive hypothesis, $\Gamma \triangleright \tau_1' \leq \tau_3$.
- Case: $\Gamma \triangleright (x{:}\tau_1')\rightharpoonup\tau_1'' \sqsubseteq (x{:}\tau_2')\rightharpoonup\tau_2'' \sqsubseteq (x{:}\tau_3')\rightharpoonup\tau_3''$. By the inductive hypothesis, $\Gamma \triangleright \tau_3' \leq \tau_1'$. By Lemma 7.2.2, $\Gamma, x{:}\tau_3' \triangleright \tau_1'' \leq \tau_2''$, so by the inductive hypothesis we have $\Gamma, x{:}\tau_3' \triangleright \tau_1'' \leq \tau_3''$.
- Case: $\Gamma \triangleright (x{:}\tau_1')\times\tau_1'' \sqsubseteq (x{:}\tau_2')\times\tau_2'' \sqsubseteq (x{:}\tau_3')\times\tau_3''$. Analogous to previous case.

2. By part 1.

∎

Lemma 7.2.5
Assume τ_1 and τ_2 contain no $\forall$'s.

1. *If $\tau_1 = Ty(A_1)$, $\tau_2 = Ty(A_2)$, and $\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T}$ then $\Gamma \triangleright \tau_1 \leq \tau_2$.*
2. *If $\Gamma \vdash \tau_1 \equiv \tau_2$ then $\Gamma \triangleright \tau_1 \Downarrow \sigma_1$, $\Gamma \triangleright \tau_2 \Downarrow \sigma_2$, $\Gamma \triangleright \sigma_1 \sqsubseteq \sigma_2$, and $\Gamma \triangleright \sigma_2 \sqsubseteq \sigma_1$ (i.e., $\Gamma \triangleright \tau_1 \leq \tau_2$ and $\Gamma \triangleright \tau_2 \leq \tau_1$).*
3. *If $\Gamma \vdash \tau_1 \leq \tau_2$ then $\Gamma \triangleright \tau_1 \Downarrow \sigma_1$, $\Gamma \triangleright \tau_2 \Downarrow \sigma_2$, and $\Gamma \triangleright \sigma_1 \sqsubseteq \sigma_2$ (i.e., $\Gamma \triangleright \tau_1 \leq \tau_2$).*

Proof:

1. By induction on the common normal form of A_1 and A_2.

2–3. By induction on derivations, and part 1. Note that for the case of transitivity, by Proposition 7.2.3 the mediating term will contain no $\forall$'s and so the inductive hypothesis applies.

∎

Lemma 7.2.6 (Algorithmic Weakening for Constructor Variables)
Assume $\Gamma' \vdash K_2 \leq K_1$.

1. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash v_1 : \tau$, $\Gamma', \alpha{::}K_1, \Gamma'' \vdash v_2 : \tau$, and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright v_1 \Leftrightarrow v_2$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright v_1 \Leftrightarrow v_2$.*
2. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_1$, $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_2$, and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_1 \Leftrightarrow \tau_2$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \Leftrightarrow \tau_2$.*
3. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_1$, $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_2$, and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_1 \sqsubseteq \tau_2$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \sqsubseteq \tau_2$.*
4. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_1$, $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_2$, and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_1 \leq \tau_2$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \leq \tau_2$.*
5. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash ok$ and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau$.*
6. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash ok$ and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright e \rightrightarrows \tau$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright e \rightrightarrows \tau$.*
7. *If $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau$ and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright e \leftleftarrows \tau$ then $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright e \leftleftarrows \tau$.*

Proof:

1,2. By soundness and completeness for type/term equivalence, and Corollary 3.2.8.

3. Proved simultaneously with part 4, by induction on algorithmic derivations.

- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright Ty(A_1) \sqsubseteq Ty(A_2)$. By correctness of the constructor equivalence algorithm and Corollary 3.2.8.
- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \mathbf{S}(v_1 : \tau_1') \sqsubseteq \mathbf{S}(v_2 : \tau_2')$. By the inductive hypothesis $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1' \leq \tau_2'$. By correctness of term equivalence algorithm and Corollary 3.2.8, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright v_1 \Leftrightarrow v_2$.
- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \mathbf{S}(v_1 : \tau_1') \sqsubseteq \tau_2$ where τ_2 is not a singleton. By the inductive hypothesis $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1' \leq \tau_2$.
- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright (x{:}\tau_1') \rightharpoonup \tau_1'' \sqsubseteq (x{:}\tau_2') \rightharpoonup \tau_2''$. By the inductive hypothesis, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_2' \leq \tau_1'$ and $\Gamma', \alpha{::}K_2, \Gamma'', x{:}\tau_2' \triangleright \tau_1'' \leq \tau_2''$.
- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright (x{:}\tau_1') \times \tau_1'' \sqsubseteq (x{:}\tau_2') \times \tau_2''$. Analogous to previous case.
- Case: $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \forall \alpha{::}K_1'.\tau_1'' \sqsubseteq \forall \alpha{::}K_2'.\tau_2''$. By correctness of algorithm subkinding and Corollary 3.2.8, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright K_2' \leq K_1'$ and by the inductive hypothesis, $\Gamma', \alpha{::}K_2, \Gamma'', \alpha{::}K_2' \triangleright \tau_1'' \leq \tau_2''$.

4. - Case: τ_1 and τ_2 contain $\forall$.
     (a) Then neither type is of the form $Ty(A)$,
     (b) so $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_1 \Downarrow \tau_1$, $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_2 \Downarrow \tau_2$, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \Downarrow \tau_1$, and $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_2 \Downarrow \tau_2$.
     (c) By part 3 we have $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \sqsubseteq \tau_2$,
     (d) so $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \leq \tau_2$.
   - Case: neither τ_1 nor τ_2 contains $\forall$.
     (a) By assumption $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_1 \Downarrow \sigma_1$, $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \tau_2 \Downarrow \sigma_2$, and $\Gamma', \alpha{::}K_1, \Gamma'' \triangleright \sigma_1 \sqsubseteq \sigma_2$.
     (b) By part 3, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_1 \sqsubseteq \sigma_2$.
     (c) By Lemma 6.2.1, $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_1 \equiv \sigma_1$ and $\Gamma', \alpha{::}K_1, \Gamma'' \vdash \tau_2 \equiv \sigma_2$.
     (d) By Corollary 3.2.8 and completeness of the type equivalence algorithm $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \Downarrow \sigma_1'$, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_2 \Downarrow \sigma_2'$, $\Gamma', \alpha{::}K_2, \Gamma'' \vdash \tau_1 \equiv \sigma_1'$, and $\Gamma', \alpha{::}K_2, \Gamma'' \vdash \tau_2 \equiv \sigma_2'$.
     (e) By Corollary 3.2.8 and transitivity, $\Gamma', \alpha{::}K_2, \Gamma'' \vdash \sigma_1 \equiv \sigma_1'$ and $\Gamma', \alpha{::}K_2, \Gamma'' \vdash \sigma_2 \equiv \sigma_2'$.
     (f) By Lemma 7.2.5, $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_1' \leq \sigma_1$ and $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_2 \leq \sigma_2'$.
     (g) Since $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_1 \leq \sigma_2$, by Lemma 7.2.4 applied twice we have $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_1' \leq \sigma_2'$.
     (h) But σ_1' and σ_2' are head-normal, so $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \sigma_1' \sqsubseteq \sigma_2'$.
     (i) Therefore $\Gamma', \alpha{::}K_2, \Gamma'' \triangleright \tau_1 \leq \tau_2$.

5–7. By induction on algorithmic derivations and part 4.

∎

Given this weakening property, I can now show the full transitivity result for algorithmic subtyping. I show only one case of the proof, because all the others are exactly the same as in the proof of Lemma 7.2.4.

Lemma 7.2.7 (Transitivity of Algorithmic Subtyping)
Assume $\Gamma \vdash \tau_1$, $\Gamma \vdash \tau_2$, *and* $\Gamma \vdash \tau_3$.

1. *If* $\Gamma \triangleright \tau_1 \sqsubseteq \tau_2$ *and* $\Gamma \triangleright \tau_2 \sqsubseteq \tau_3$ *then* $\Gamma \triangleright \tau_1 \sqsubseteq \tau_3$.
2. *If* $\Gamma \triangleright \tau_1 \leq \tau_2$ *and* $\Gamma \triangleright \tau_2 \leq \tau_3$ *then* $\Gamma \triangleright \tau_1 \leq \tau_3$.

Proof: By induction on $size(\Gamma;\tau_1) + size(\Gamma;\tau_2) + size(\Gamma;\tau_3)$.

- Case: $\Gamma \triangleright \forall\alpha{::}K_1'.\tau_1'' \sqsubseteq \forall\alpha{::}K_2'.\tau_2'' \sqsubseteq \forall\alpha{::}K_3'.\tau_3''$. By the transitivity of the subkinding algorithm, $\Gamma \triangleright K_3' \leq K_1'$. By Lemma 7.2.6 have we $\Gamma, \alpha{::}K_3' \triangleright \tau_1'' \leq \tau_2''$. By the inductive hypothesis, $\Gamma, \alpha{::}K_3' \triangleright \tau_1'' \leq \tau_3''$.

∎

At this point I have shown that the subtyping and kind equivalence algorithms are transitive on well-formed types. At this point, completeness of the remaining type and term algorithms is straightforward.

Theorem 7.2.8 (Completeness for Subtyping and Validity)

1. *If* $\Gamma \vdash \tau$ *then* $\Gamma \triangleright \tau$.
2. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *then* $\Gamma \triangleright \tau_1 \leq \tau_2$.
3. *If* $\Gamma \vdash \tau_1 \leq \tau_2$ *and* τ_1 *and* τ_2 *are head-normal then* $\Gamma \triangleright \tau_1 \sqsubseteq \tau_2$.
4. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \triangleright e \rightrightarrows \sigma$ *and* $\Gamma \triangleright e \Uparrow \sigma$.
5. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \triangleright e \leftleftarrows \tau$.

Proof: By simultaneous induction on the hypothesized derivations, using the completeness of the type and term equivalence algorithms, and transitivity of algorithmic subtyping. ∎

Theorem 7.2.9

1. *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *then it is decidable whether* $\Gamma \triangleright \tau_1 \sqsubseteq \tau_2$
2. *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *then it is decidable whether* $\Gamma \triangleright \tau_1 \leq \tau_2$
3. *If* $\Gamma \vdash ok$ *then it is decidable whether* $\Gamma \triangleright \tau$ *is provable.*
4. *If* $\Gamma \vdash ok$ *then it is decidable whether* $\Gamma \triangleright e \rightrightarrows \tau$ *holds for some* τ.
5. *If* $\Gamma \vdash \tau$ *and* e *is given then it is decidable whether* $\Gamma \triangleright e \leftleftarrows \tau$ *is provable.*

Proof:

1,2. By induction on $size(\Gamma;\tau_1) + size(\Gamma;\tau_2)$, invoking the decidability of term equivalence and of type head-normalization.

3–5. By simultaneous induction on the textual size of τ, e, and e respectively.

∎

Corollary 7.2.10 (Decidability of Subtyping and Validity)

1. *If* $\Gamma \vdash ok$ *then it is decidable whether* $\Gamma \vdash \tau$ *is provable.*
2. *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *then it is decidable whether* $\Gamma \vdash \tau_1 \leq \tau_2$
3. *If* $\Gamma \vdash ok$ *then it is decidable whether* $\Gamma \vdash e : \tau$ *holds for some* τ.
4. *If* $\Gamma \vdash \tau$ *and* e *is given then it is decidable whether* $\Gamma \vdash e : \tau$ *is provable.*

7.3 Antisymmetry of Subtyping

By taking advantage of the algorithmic form of subtyping — which contains no transitivity rule — subtyping can be shown to be antisymmetric.

Lemma 7.3.1
Assume $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$.

1. *If* $\Gamma \rhd \tau_1 \leq \tau_2$ *and* $\Gamma \rhd \tau_2 \leq \tau_1$ *then* $\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$.
2. *If* $\Gamma \rhd \tau_1 \sqsubseteq \tau_2$ *and* $\Gamma \rhd \tau_2 \sqsubseteq \tau_1$ *then* $\Gamma \rhd \tau_1 \leftrightarrow \tau_2$.

Proof: By simultaneous induction on the size of the hypothesized derivations.
Note that by soundness, $\Gamma \vdash \tau_1 \leq \tau_2$ and $\Gamma \vdash \tau_2 \leq \tau_1$.

1. (a) By inversion, $\Gamma \rhd \tau_1 \Downarrow \sigma_1$, $\Gamma \rhd \tau_2 \Downarrow \sigma_2$, $\Gamma \rhd \sigma_1 \sqsubseteq \sigma_2$ and $\Gamma \rhd \sigma_2 \sqsubseteq \sigma_1$.
   (b) By the inductive hypothesis, $\Gamma \rhd \sigma_1 \Leftrightarrow \sigma_2$.
   (c) Thus $\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$.
2. - Case: $\Gamma \rhd \mathit{Ty}(A_1) \sqsubseteq \mathit{Ty}(A_2)$ and $\Gamma \rhd \mathit{Ty}(A_2) \sqsubseteq \mathit{Ty}(A_1)$ because $\Gamma \rhd A_1 \Leftrightarrow A_2 :: \mathbf{T}$ and $\Gamma \rhd A_2 \Leftrightarrow A_1 :: \mathbf{T}$. Then $\Gamma \rhd \mathit{Ty}(A_1) \leftrightarrow \mathit{Ty}(A_2)$.
   - Case: $\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \sqsubseteq \mathbf{S}(v_2 : \tau_2)$ and $\Gamma \rhd \mathbf{S}(v_2 : \tau_2) \sqsubseteq \mathbf{S}(v_1 : \tau_1)$ because $\Gamma \rhd \tau_1 \leq \tau_2$, $\Gamma \rhd v_1 \Leftrightarrow v_2$, $\Gamma \rhd \tau_2 \leq \tau_1$, and $\Gamma \rhd v_2 \Leftrightarrow v_1$.
     By the inductive hypothesis, $\Gamma \rhd \tau_1 \Leftrightarrow \tau_2$, so $\Gamma \rhd \mathbf{S}(v_1 : \tau_1) \Leftrightarrow \mathbf{S}(v_2 : \tau_2)$.
   - Case: $\Gamma \rhd (x{:}\tau_1')\rightharpoonup\tau_1'' \sqsubseteq (x{:}\tau_2')\rightharpoonup\tau_2''$ and $\Gamma \rhd (x{:}\tau_2')\rightharpoonup\tau_2'' \sqsubseteq (x{:}\tau_1')\rightharpoonup\tau_1''$ because $\Gamma \rhd \tau_1' \leq \tau_2'$ and $\Gamma, x{:}\tau_2' \rhd \tau_1'' \leq \tau_2''$ and $\Gamma \rhd \tau_2' \leq \tau_1'$ and $\Gamma, x{:}\tau_1' \rhd \tau_2'' \leq \tau_1''$.
     (a) By the inductive hypothesis, $\Gamma \rhd \tau_1' \Leftrightarrow \tau_2'$.
     (b) By completeness, $\Gamma, x{:}\tau_1' \rhd \tau_1'' \leq \tau_2''$.
     (c) By the inductive hypothesis, $\Gamma, x{:}\tau_1' \rhd \tau_1'' \Leftrightarrow \tau_2''$.
     (d) Thus $\Gamma \rhd (x{:}\tau_1')\rightharpoonup\tau_1'' \Leftrightarrow (x{:}\tau_2')\rightharpoonup\tau_2''$.
   - The remaining two cases are similar.

∎

Proposition 7.3.2 (Antisymmetry of Subtyping)
If $\Gamma \vdash \tau_1 \leq \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau_1$ *then* $\Gamma \vdash \tau_1 \equiv \tau_2$.

Proof: By soundness and completeness of the subtyping algorithms and by Lemma 7.3.1. ∎

7.4 Strengthening for Term Variables

From the correctness of the algorithmic judgments I now derive a strengthening property for term variables. I show that all of the judgments in the definition of MIL_0 are preserved under dropping of apparently-unused typing hypotheses for term variables.

However, recall that in the presence of transitivity rules strengthening cannot be proved directly by induction on derivations. For example, consider an instance of Rule 2.81:

$$\frac{\Gamma_1, y{:}\sigma, \Gamma_2 \vdash e \equiv e' : \tau \qquad \Gamma_1, y{:}\sigma, \Gamma_2 \vdash e' \equiv e'' : \tau}{\Gamma_1, y{:}\sigma, \Gamma_2 \vdash e \equiv e'' : \tau}$$

And assume that y is not used in the conclusion (formally, that $y \notin (\mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(e) \cup \mathrm{FV}(e'') \cup \mathrm{FV}(\tau))$) It does not follow, however, that $y \notin \mathrm{FV}(e')$; a priori, it might be that the equivalence of e and e'' is provable *only* by equating both to a term involving y. Thus the inductive hypothesis cannot be applied to the premises.

Also, the trick used for eliminating unused kind variables in §3.4 is not applicable here, because although every *kind* may be inhabited by a constructor, we cannot expect in general that every *type* is likewise inhabited by a value.[1]

However, the definitions of the algorithmic relations involve no transitivity rules, so here strengthening can be proved directly:

Lemma 7.4.1
If $\Gamma_1, y{:}\sigma, \Gamma_2 \rhd \mathcal{J}$ holds and $y \notin (FV(\Gamma_2) \cup FV(\mathcal{J}))$ then $\Gamma_1, \Gamma_2 \rhd \mathcal{J}$ holds as well.

Proof: By induction on the derivation $\Gamma_1, y{:}\sigma, \Gamma_2 \rhd \mathcal{J}$. ∎

By soundness and completeness of the algorithmic relations, the strengthening property can be transferred to the official MIL_0. This is easy, but not quite immediate. For example, suppose $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \tau_1 \leq \tau_2$ where $y \notin (\mathrm{dom}(\Gamma_2) \cup \mathrm{FV}(\tau_1) \cup \mathrm{FV}(\tau_2))$. By Completeness we have $\Gamma_1, y{:}\sigma, \Gamma_2 \rhd \tau_1 \leq \tau_2$, and by Lemma 7.4.1 we have $\Gamma_1, \Gamma_2 \rhd \tau_1 \leq \tau_2$. However, we cannot simply conclude that $\Gamma_1, \Gamma_2 \vdash \tau_1 \leq \tau_2$; the statement of soundness requires that we previously know $\Gamma_1, \Gamma_2 \vdash \tau_1$ and $\Gamma_1, \Gamma_2 \vdash \tau_2$.

Lemma 7.4.2
If $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash ok$ and $y \notin FV(\Gamma_2)$ then $\Gamma_1, \Gamma_2 \vdash ok$.

Proof: By induction on Γ_2.
First, note that if $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \mathrm{ok}$ then $y \notin \mathrm{FV}(\Gamma_1)$. Then there are three cases for the form of the proof $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \mathrm{ok}$:

- Case: $\Gamma_2 = \bullet$.

$$\frac{\Gamma_1 \vdash \sigma}{\Gamma_1, y{:}\sigma \vdash \mathrm{ok}} \quad y \notin \mathrm{dom}(\Gamma_1)$$

  Then by Proposition 3.1.1, $\Gamma_1 \vdash \mathrm{ok}$.

- Case: $\Gamma_2 = \Gamma_2', \alpha{::}K$.

$$\frac{\Gamma_1, y{:}\sigma, \Gamma_2' \vdash K}{\Gamma_1, y{:}\sigma, \Gamma_2', \alpha{::}K \vdash \mathrm{ok}} \quad (\alpha \notin \mathrm{dom}(\Gamma_1, y{:}\sigma, \Gamma_2'))$$

  1. By Completeness, $\Gamma_1, y{:}\sigma, \Gamma_2' \rhd K$.
  2. By Lemma 7.4.1, $\Gamma_1, \Gamma_2' \rhd K$.
  3. By Proposition 3.1.1 and the inductive hypothesis, $\Gamma_1, \Gamma_2' \vdash \mathrm{ok}$.
  4. By Soundness, $\Gamma_1, \Gamma_2' \vdash K$.
  5. Therefore $\Gamma_1, \Gamma_2', \alpha{::}K \vdash \mathrm{ok}$.

[1] Actually, since all the base types mentioned are inhabited, every type in MIL_0 *is* inhabited by a value. Because this property is not preserved when recursive types are added, I choose not take advantage of it.

- Case: $\Gamma_2 = \Gamma_2', x{:}\tau$.

$$\frac{\Gamma_1, y{:}\sigma, \Gamma_2' \vdash \tau}{\Gamma_1, y{:}\sigma, \Gamma_2', x{:}\tau \vdash \mathrm{ok}} \quad x \notin \mathrm{dom}(\Gamma_1, y{:}\sigma, \Gamma_2')$$

  1. By Completeness, $\Gamma_1, y{:}\sigma, \Gamma_2' \rhd \tau$.
  2. By Lemma 7.4.1, $\Gamma_1, \Gamma_2' \rhd \tau$.
  3. By Proposition 3.1.1 and the inductive hypothesis, $\Gamma_1, \Gamma_2' \vdash \mathrm{ok}$.
  4. By Soundness, $\Gamma_1, \Gamma_2' \vdash \tau$.
  5. Therefore $\Gamma_1, \Gamma_2', x{:}\tau \vdash \mathrm{ok}$.

∎

Theorem 7.4.3 (Strengthening for Term Variables)
If $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \mathcal{J}$ holds and $y \notin (FV(\Gamma_2) \cup FV(\mathcal{J}))$ then $\Gamma_1, \Gamma_2 \vdash \mathcal{J}$ holds as well.

Proof: By Lemmas 7.4.1 and 7.4.2, and soundness and completeness of the algorithmic judgments with respect to the MIL_0 definition. I show two representative cases:

- Case: $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \tau$.

  1. By Completeness, $\Gamma_1, y{:}\sigma, \Gamma_2 \rhd \tau$.
  2. By Lemma 7.4.1, $\Gamma_1, \Gamma_2 \rhd \tau$.
  3. By Proposition 3.1.1 and Lemma 7.4.2, $\Gamma_1, \Gamma_2 \vdash \mathrm{ok}$.
  4. By Soundness, $\Gamma_1, \Gamma_2 \vdash \tau$.

- Case: $\Gamma_1, y{:}\sigma, \Gamma_2 \vdash \tau_1 \leq \tau_2$.

  1. By Completeness, $\Gamma_1, y{:}\sigma, \Gamma_2 \rhd \tau_1 \leq \tau_2$.
  2. By Lemma 7.4.1, $\Gamma_1, \Gamma_2 \rhd \tau_1 \leq \tau_2$.
  3. As in the previous case $\Gamma_1, \Gamma_2 \vdash \mathrm{ok}$ and $\Gamma_1, \Gamma_2 \vdash \tau_1$ and $\Gamma_1, \Gamma_2 \vdash \tau_2$.
  4. By Soundness, $\Gamma_1, \Gamma_2 \vdash \tau_1 \leq \tau_2$.

∎

Chapter 8

Properties of Evaluation

8.1 Determinacy of Evaluation

It is straightforward to show that evaluation in MIL_0 is deterministic.

Proposition 8.1.1

1. *Given A, there is at most one $\mathcal{U}$ and one instruction I such that $A = \mathcal{U}[I]$.*
2. *Given e, there is at most one $\mathcal{C}$ and one instruction I such that $e = \mathcal{C}[I]$.*

Proof: By induction on A and e respectively. ∎

Corollary 8.1.2 (Determinacy of Evaluation)
If $e \leadsto e_1$ and $e \leadsto e_2$ then $e_1 = e_2$.

8.2 Type Soundness

Type soundness is informally the property that "well-typed programs don't go wrong". In a small-step operational semantics, soundness can be expressed as the combination of two principles:

1. *Type Preservation:* If e is well-typed and e can take a step to e', then e' is well-typed.
2. *Progress:* If e is well-typed then either e is a fully-evaluated value and execution is done, or else e can take a step to some e'.

Put together, these guarantee that, when starting with a well-formed program, execution either terminates (yielding a fully-evaluated value) or execution goes on forever. Evaluation of well-typed programs cannot get "stuck" — reach a situation where no execution step applies but evaluation has not terminated. Examples of stuck programs would be $3(4)$ or $\pi_1(\mathsf{fun}\, f(x{:}\mathsf{int}){:}\mathsf{int}\ \mathsf{is}\ x)$.

Lemma 8.2.1

1. *If $\Gamma \vdash I :: K$ and $I \leadsto R$ then $\Gamma \vdash R :: K$.*
2. *If $\Gamma \vdash I : \tau$ and $I \leadsto R$ then $\Gamma \vdash R : \tau$.*

Lemma 8.2.2 (Decomposition and Replacement)

1. *If $\vdash \mathcal{C}[e] : \tau$ then for some σ, $\vdash e : \sigma$, and $\vdash e' : \sigma$ implies $\vdash \mathcal{C}[e'] : \tau$.*
2. *If $\vdash \mathcal{C}[A] : \tau$ then for some L, $\vdash A :: L$, and $\vdash A' :: L$ implies $\vdash \mathcal{C}[A'] : \tau$.*

3. *If* $\vdash \mathcal{U}[A] :: K$ *then for some* L, $\vdash A :: L$, *and* $\vdash A' :: L$ *implies* $\vdash \mathcal{U}[A'] :: K$.

Proof: By induction on derivations. ∎

Corollary 8.2.3 (Type Preservation)
If $\Gamma \vdash e :: \tau$ *and* $e \leadsto e'$ *then* $\Gamma \vdash e' :: \tau$.

Lemma 8.2.4 (Canonical Forms for Constructors)

1. *If* $\vdash \overline{A} :: \Sigma\alpha{::}K'.K''$ *then* $\overline{A} = \langle \overline{A}', \overline{A}'' \rangle$.
2. *If* $\vdash \overline{A} :: \Pi\alpha{::}K'.K''$ *then either* $\overline{A} = \lambda\alpha{::}L.A$ *or else* $\overline{A} = c\,\overline{A}_1 \cdots \overline{A}_n$ *with* $n \geq 0$.

Proof: By induction on the kinding derivation. ∎

Lemma 8.2.5 (Canonical Forms for Terms)
Assume $\vdash \overline{v} : \tau$.

1. *If* $\triangleright \tau^{\$} \Downarrow \mathsf{int}$ *then* $\overline{v} = n$ *for some integer* n.
2. *If* $\triangleright \tau^{\$} \Downarrow (x{:}\tau') \times \tau''$ *then* $\overline{v} = \langle \overline{v}', \overline{v}'' \rangle$ *for some* $\overline{v}'$ *and* $\overline{v}''$.
3. *If* $\triangleright \tau^{\$} \Downarrow (x{:}\tau') \rightharpoonup \tau''$ *then* $\overline{v} = \mathsf{fun}\, f(x{:}\sigma'){:}\sigma''\ \mathsf{is}\ e$ *for some* σ', σ'', *and* e.
4. *If* $\triangleright \tau^{\$} \Downarrow \forall\alpha{::}K.\tau$ *then* $\overline{v} = \Lambda(\alpha{::}L'){:}L''.e$ *for some* L', L'', *and* e.

Proof: By induction on typing derivations, using Theorem 6.2.3 and Lemma 6.3.1. ∎

Theorem 8.2.6 (Progress)

1. *If* $\vdash A :: K$ *then* $A = \overline{A}$ *or* $A \mapsto A'$ *for some* A'.
2. *If* $\vdash e : \tau$ *then* $e = \overline{v}$ *or* $e \mapsto e'$ *for some* e'.

Proof: By simultaneous induction on typing and kinding derivations, and cases on the last inference rule used. I show one representative case:

- Case: Rule 2.25

$$\frac{\Gamma \vdash A_1 :: K' \to K'' \qquad \Gamma \vdash A_2 :: K'}{\Gamma \vdash A_1\, A_2 :: K''}$$

  If A_1 is not a constructor value, then by the inductive hypothesis $A_1 \leadsto A_1'$, so $A_1\, A_2 \leadsto A_1'\, A_2$. Alternatively, if A_1 is a value but A_2 is not, then $A_2 \leadsto A_2'$ and $A_1\, A_2 \leadsto A_1\, A_2'$. Finally, assume A_1 and A_2 are both values. Then by Lemma 8.2.4, $A_1 = c\, v_1' \ldots v_n'$ and so $A_1\, A_2$ is a value, or else $A_1 = \lambda\alpha{::}K.A$ so that $A_1\, A_2 \leadsto [A_2/\alpha]A$.

∎

Chapter 9

Intensional Polymorphism

9.1 Introduction

As discussed earlier, the TIL and TILT compilers use the *intensional type analysis* framework of Harper and Morrisett [HM95, TMC+96, Mor95]. Type constructors correspond to run-time values, and the language includes constructs which permit primitive recursion over constructors of kind $\mathbf{T}$. I model these by adding two new constructs to the language: $\mathsf{Typerec}$ and $\mathsf{typerec}$. The former is a constructor which does run-time analysis of constructors, while the latter is a term which does a similar run-time analysis. There are several applications for such constructs, both in implementing Standard ML (by, for example, using different array representations for values of different types) and elsewhere (e.g., implementing generic pretty-printing or marshaling routines) [HM95, TMC+96, Mor95].

9.2 Language Changes

9.2.1 Grammar

Intensional type analysis adds two constructs to the language: `Typerec` allows primitive recursion over constructors to compute a type constructor, while `typerec` allows primitive recursion over constructors to compute a term value.

$$
\begin{array}{lrcl}
\text{Type Constructors} & A, B & ::= & \cdots \\
 & & | & \mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) \\[1ex]
\text{Terms} & e, d & ::= & \cdots \\
 & & | & \mathsf{typerec}[\alpha.\tau](A; e^{\rightharpoonup}; e^{\mathrm{ow}})
\end{array}
$$

For simplicity, the type analysis constructs considered here make only the distinction between those constructors which are (equivalent to) function type constructors, and the rest (the "otherwise" case). That is, I have restricted `Typerec` to allow the definitions for a function $F :: \Pi\alpha{::}\mathbf{T}.K$ of the form

$$
\begin{array}{lcll}
F(\alpha_1 \rightharpoonup \alpha_2) & = & G(\alpha_1)(\alpha_2)(F(\alpha_1))(F(\alpha_2)) & \\
F(\alpha) & = & H(\alpha) & \text{if } \alpha \text{ is not equivalent to a function type constructor}
\end{array}
$$

where G and H are arbitrary constructor-level functions of the right kind; this function F would be defined in the official syntax as

$$\lambda\beta{::}\mathbf{T}.\mathsf{Typerec}[\alpha.K](\beta; G; H).$$

A similar restriction is made for the term-level typerec.

The most interesting aspects of constructs for intensional polymorphism are distinctions made between different constructors, primitive recursion, and the possibility of a default case. Extending `Typerec` and `typerec` to test for specific base type constructors or the product type constructor would not substantially affect the results of this chapter.

9.2.2 Static Semantics

The following rules must be added:

Well-Formedness

$$\frac{\begin{array}{c}\Gamma \vdash A :: \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash K \\ \Gamma \vdash A^{\rightharpoonup} :: \Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K{\to}[\alpha_2/\alpha]K{\to}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]K \\ \Gamma \vdash A^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K\end{array}}{\Gamma \vdash \mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) :: [A/\alpha]K} \tag{9.1}$$

$$\frac{\begin{array}{c}\Gamma \vdash A :: \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash \tau \\ \Gamma \vdash e^{\rightharpoonup} : \forall\alpha_1{::}\mathbf{T}.\forall\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]\tau{\rightharpoonup}[\alpha_2/\alpha]\tau{\rightharpoonup}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]\tau \\ \Gamma \vdash e^{\mathrm{ow}} : \forall\alpha{::}\mathbf{T}.\tau\end{array}}{\Gamma \vdash \mathsf{typerec}[\alpha.\tau](A; e^{\rightharpoonup}; e^{\mathrm{ow}}) : [A/\alpha]\tau} \tag{9.2}$$

Equivalence

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \equiv A_2 :: \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash K_1 \equiv K_2 \\ \Gamma \vdash A_1^{\rightharpoonup} \equiv A_2^{\rightharpoonup} :: \Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K_1{\to}[\alpha_2/\alpha]K_1{\to}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]K_1 \\ \Gamma \vdash A_1^{\mathrm{ow}} \equiv A_2^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K_1\end{array}}{\Gamma \vdash \mathsf{Typerec}[\alpha.K_1](A_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}) \equiv \mathsf{Typerec}[\alpha.K_2](A_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}) :: [A_1/\alpha]K_1} \tag{9.3}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 :: \mathbf{T} \qquad \Gamma \vdash A_2 :: \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash K \\ \Gamma \vdash A^{\rightharpoonup} :: \Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K{\to}[\alpha_2/\alpha]K{\to}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]K \\ \Gamma \vdash A^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{Typerec}[\alpha.K](A_1{\rightharpoonup}A_2; A^{\rightharpoonup}; A^{\mathrm{ow}}) \equiv \\ A^{\rightharpoonup}\,(A_1)\,(A_2)\,(\mathsf{Typerec}[\alpha.K](A_1; A^{\rightharpoonup}; A^{\mathrm{ow}}))(\mathsf{Typerec}[\alpha.K](A_2; A^{\rightharpoonup}; A^{\mathrm{ow}})) :: [(A_1{\rightharpoonup}A_2)/\alpha]K\end{array}} \tag{9.4}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{E}[c] :: \mathbf{T} \qquad c \text{ is not } \rightharpoonup \qquad \Gamma, \alpha{::}\mathbf{T} \vdash K \\ \Gamma \vdash A^{\rightharpoonup} :: \Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K{\to}[\alpha_2/\alpha]K{\to}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]K \\ \Gamma \vdash A^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K\end{array}}{\Gamma \vdash \mathsf{Typerec}[\alpha.K](\mathcal{E}[c]; A^{\rightharpoonup}; A^{\mathrm{ow}}) \equiv A^{\mathrm{ow}}\,(\overline{A}) :: [\overline{A}/\alpha]K} \tag{9.5}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \equiv A_2 : \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash \tau_1 \equiv \tau_2 \\ \Gamma \vdash e_1^{\rightharpoonup} \equiv e_2^{\rightharpoonup} : \forall\alpha_1{::}\mathbf{T}.\forall\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]\tau_1{\rightharpoonup}[\alpha_2/\alpha]\tau_1{\rightharpoonup}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]\tau_1 \\ \Gamma \vdash e_1^{\mathrm{ow}} \equiv e_2^{\mathrm{ow}} : \forall\alpha{::}\mathbf{T}.\tau_1\end{array}}{\Gamma \vdash \mathsf{typerec}[\alpha.\tau_1](A_1; e_1^{\rightharpoonup}; e_1^{\mathrm{ow}}) \equiv \mathsf{typerec}[\alpha.\tau_2](A_2; e_2^{\rightharpoonup}; e_2^{\mathrm{ow}}) : [A_1/\alpha]\tau_1} \tag{9.6}$$

9.2.3 Dynamic Semantics

The constructor-level and term-level evaluation contexts are each extended with one case:

$$\begin{array}{rcl} \mathcal{U} & ::= & \cdots \\ & | & \mathsf{Typerec}[\alpha.K](\mathcal{U}; A^{\rightharpoonup}; A^{\mathrm{ow}}) \\ \mathcal{C} & ::= & \cdots \\ & | & \mathsf{typerec}[\alpha.\tau](\mathcal{U}; e^{\rightharpoonup}; e^{\mathrm{ow}}) \end{array}$$

and there are four new instruction reduction steps:

$$\begin{array}{lll} \mathsf{Typerec}[\alpha.K](A_1 \rightharpoonup A_2; A^{\rightharpoonup}; A^{\mathrm{ow}}) & \leadsto & A^{\rightharpoonup}\,(A_1)\,(A_2)\,(\mathsf{Typerec}[\alpha.K](A_1; A^{\rightharpoonup}; A^{\mathrm{ow}})) \\ & & \qquad\qquad (\mathsf{Typerec}[\alpha.K](A_2; A^{\rightharpoonup}; A^{\mathrm{ow}})) \\ \mathsf{Typerec}[\alpha.K](\overline{A}; A^{\rightharpoonup}; A^{\mathrm{ow}}) & \leadsto & A^{\mathrm{ow}}\,(\overline{A}), \quad \text{if } \overline{A} \text{ not of the form } A_1 \rightharpoonup A_2 \\ \mathsf{typerec}[\alpha.\tau](A_1 \rightharpoonup A_2; e^{\rightharpoonup}; e^{\mathrm{ow}}) & \leadsto & e^{\rightharpoonup}\,(A_1)\,(A_2)\,(\mathsf{typerec}[\alpha.K](A_1; e^{\rightharpoonup}; e^{\mathrm{ow}})) \\ & & \qquad\qquad (\mathsf{typerec}[\alpha.K](A_2; e^{\rightharpoonup}; e^{\mathrm{ow}})) \\ \mathsf{typerec}[\alpha.\tau](\overline{A}; e^{\rightharpoonup}; e^{\mathrm{ow}}) & \leadsto & e^{\mathrm{ow}}\,(\overline{A}), \quad \text{if } \overline{A} \text{ not of the form } A_1 \rightharpoonup A_2 \end{array}$$

9.3 Declarative Properties

The proofs of Chapter 3 go through without any problems. Those proofs needing modifications merely require extra cases to be added for each of the new static semantic rules; these are straightforward uses of the inductive hypotheses. Preserved properties include substitution, validity, and functionality.

The reduction rule for Typerec is not admissible. However, it is interesting to note that the system comes very close to having an admissible *extensionality* rule for Typerec. Suppose this construct contained no kind annotation, as in the formulation of Harper and Morrisett [HL94]. The well-formedness rule would be little changed:

$$\frac{\begin{array}{c} \Gamma \vdash A :: \mathbf{T} \qquad \Gamma, \alpha{::}\mathbf{T} \vdash K \\ \Gamma \vdash A^{\rightharpoonup} :: \Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K \to [\alpha_2/\alpha]K \to [(\alpha_1 \rightharpoonup \alpha_2)/\alpha]K \\ \Gamma \vdash A^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K \end{array}}{\Gamma \vdash \mathsf{Typerec}(A; A^{\rightharpoonup}; A^{\mathrm{ow}}) :: [A/\alpha]K}.$$

But assume now that $\Gamma \vdash f :: \mathbf{T} \to L$ for some kind L, and $\Gamma \vdash A :: \mathbf{T}$. By taking $K = \mathbf{S}(f(\alpha) :: L)$ in the above rule we can derive

$$\Gamma \vdash \mathsf{Typerec}(A; \lambda\alpha_1{::}\mathbf{T}.\lambda\alpha_2{::}\mathbf{T}.\lambda\_{::}L.\lambda\_{::}L.f(\alpha_1 \rightharpoonup \alpha_2); \lambda\alpha_1{::}\mathbf{T}.f(\alpha_1)) :: \mathbf{S}(f(A) :: L),$$

where I have used _ to denote function arguments which are not used in their body. It follows, then, that

$$\Gamma \vdash f(A) \equiv \mathsf{Typerec}(A; \lambda\alpha_1{::}\mathbf{T}.\lambda\alpha_2{::}\mathbf{T}.\lambda\_{::}L.\lambda\_{::}L.f(\alpha_1 \rightharpoonup \alpha_2); \lambda\alpha_1{::}\mathbf{T}.f(\alpha_1)) :: L.$$

This is exactly analogous to the standard extensionality rule for sum types [Mit96]:

$$f(z) \equiv (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow f(\mathsf{inl}\ x) \mid \mathsf{inr} x \Rightarrow f(\mathsf{inr}\ x)).$$

9.4 Algorithms for Constructors and Kinds

To make the following algorithms readable, for any kind K I will use K^α to stand for the kind

$$\Pi\alpha_1{::}\mathbf{T}.\Pi\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]K{\to}[\alpha_2/\alpha]K{\to}[(\alpha_1{\rightharpoonup}\alpha_2)/\alpha]K.$$

This is the kind of the function-type constructor arm of a $\mathsf{Typerec}$ whose kind annotation is $[\alpha.K]$.

The principal kind for a well-formed $\mathsf{Typerec}$ is easily computed from the kind annotation:

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) \Uparrow \mathbf{S}(\mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) :: [A/\alpha]K),$$

but actually checking that a $\mathsf{Typerec}$ is well-formed requires more work:

$$\begin{array}{ll} \Gamma \triangleright \mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) \rightrightarrows [A/\alpha]K & \text{if } \Gamma, \alpha{::}\mathbf{T} \triangleright K,\ \Gamma \triangleright A \leftleftarrows \mathbf{T}, \\ & \Gamma \triangleright A^{\rightharpoonup} \leftleftarrows K^\alpha, \text{ and } \Gamma \triangleright A^{\mathrm{ow}} \leftleftarrows \Pi\alpha{::}\mathbf{T}.K. \end{array}$$

I extend the notion of a constructor-level path to allow $\mathsf{Typerec}$'s:

$$\begin{array}{rcl} \mathcal{E} & ::= & \cdots \\ & \mid & \mathsf{Typerec}[\alpha.K](\mathcal{E}; A^{\rightharpoonup}; A^{\mathrm{ow}}) \end{array}$$

Then the equivalence algorithm is extended with the following cases:

Kind extraction

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.K](A; A^{\rightharpoonup}; A^{\mathrm{ow}}) \uparrow [A/\alpha]K$$

Weak head reduction

$$\begin{array}{ll} \Gamma \triangleright \mathcal{E}[\mathsf{Typerec}[\alpha.K](A_1{\rightharpoonup}A_2; A^{\rightharpoonup}; A^{\mathrm{ow}})] \leadsto & \\ \quad \mathcal{E}[A^{\rightharpoonup}\ (A_1)\ (A_2)\ (\mathsf{Typerec}[\alpha.K](A_1; A^{\rightharpoonup}; A^{\mathrm{ow}}))(\mathsf{Typerec}[\alpha.K](A_2; A^{\rightharpoonup}; A^{\mathrm{ow}}))] & \\ \Gamma \triangleright \mathcal{E}[\mathsf{Typerec}[\alpha.K](\overline{A}; A^{\rightharpoonup}; A^{\mathrm{ow}})] \leadsto & \\ \quad \mathcal{E}[A^{\mathrm{ow}}\ (\overline{A})[\overline{A}/\alpha]K] & \text{if } \overline{A} \text{ not of the form } A_1{\rightharpoonup}A_2 \end{array}$$

Algorithmic path equivalence

$$\begin{array}{ll} \Gamma \triangleright \mathsf{Typerec}[\alpha.K_1](p_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}) \leftrightarrow & \\ \quad \mathsf{Typerec}[\alpha.K_2](p_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}) \uparrow [p_1/\alpha]K_1 & \text{if } \Gamma, \alpha{::}\mathbf{T} \triangleright K_1 \Leftrightarrow K_2,\ \Gamma \triangleright p_1 \leftrightarrow p_2 \uparrow \mathbf{T}, \\ & \Gamma \triangleright A_1^{\rightharpoonup} \Leftrightarrow A_2^{\rightharpoonup}{::}K^\alpha \\ & \text{and } \Gamma \triangleright A_1^{\mathrm{ow}} \Leftrightarrow A_2^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K. \end{array}$$

It is straightforward to show that soundness is preserved by the above modifications.

9.5 Completeness and Decidability for Constructors and Kinds

The revised version of path equivalence is extended in the obvious fashion:

$$\begin{array}{l} \Gamma_1 \triangleright \mathsf{Typerec}[\alpha.K_1](p_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}) \uparrow [p_1/\alpha]K_1 \leftrightarrow \\ \qquad\qquad \Gamma_2 \triangleright \mathsf{Typerec}[\alpha.K_2](p_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}) \uparrow [p_2/\alpha]K_2 \end{array}$$

$$\begin{array}{ll} \text{if} & \Gamma_1, \alpha{::}\mathbf{T} \triangleright K_1 \Leftrightarrow \Gamma_2, \alpha{::}\mathbf{T} \triangleright K_2, \\ & \Gamma_1 \triangleright p_1 \uparrow \mathbf{T} \leftrightarrow \Gamma_2 \triangleright p_2 \uparrow \mathbf{T}, \\ & \Gamma_1 \triangleright A_1^{\rightharpoonup} :: {K_2}^\alpha \Leftrightarrow \Gamma_2 \triangleright A_2^{\rightharpoonup} :: {K_2}^\alpha, \\ & \textit{and} \\ & \Gamma_1 \triangleright A_1^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K_1 \Leftrightarrow \Gamma_2 \triangleright A_2^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K_2. \end{array}$$

The logical relations, however, need not change. One point to be aware of, however, is that a path $\mathcal{E}[c]$ is no longer guaranteed to be head-normal, because of cases like

$$\mathsf{Typerec}[\alpha.\mathbf{T}](\mathsf{Int}; A^{\rightharpoonup}; A^{\mathrm{ow}}).$$

Thus, for example, parts 3 and 4 of Lemma 5.3.9 must be restricted to the case where either p_1 and p_2 and of the form $\mathcal{E}_i[\alpha]$ or else of the form $\mathcal{E}_i[c]$ and head-normal. In all cases in which this lemma has been invoked, one of these two cases holds. (For the same reason, Proposition 5.3.15 must be restricted to the case in which $\mathcal{E}_1[c_1]$ and $\mathcal{E}_2[c_2]$ are both head-normal.)

With the addition of new kinding and equivalence rules for Typerec, two new cases must be added to the proof of the logical relations theorem (Theorem 5.3.10). These cases follow from the following lemma:

Lemma 9.5.1
If $\Delta_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \rhd A_2 :: \mathbf{T}$, $(\Delta_1; A_1^{\rightharpoonup}; K_2{}^{\alpha})$ **is** $(\Delta_2; A_2^{\rightharpoonup}; K_2{}^{\alpha})$, *and*
$(\Delta_1; A_1^{\mathrm{ow}}; \Pi\alpha{::}\mathbf{T}.K_1)$ **is** $(\Delta_2; A_2^{\mathrm{ow}}; \Pi\alpha{::}\mathbf{T}.K_2)$ *then*
$(\Delta_1; \mathsf{Typerec}[\alpha.K_1](A_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [A_1/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](A_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [A_2/\alpha]K_2)$.

Proof: By induction on $\Delta_1 \rhd A_1 :: \mathbf{T} \Leftrightarrow \Delta_2 \rhd A_2 :: \mathbf{T}$.

- $\Delta_1 \rhd A_1 \Downarrow \mathcal{E}_1[\beta]$ and $\Delta_2 \rhd A_2 \Downarrow \mathcal{E}_2[\beta]$, with $\Delta_1 \rhd \mathcal{E}_1[\beta] \uparrow \mathbf{T} \leftrightarrow \Delta_2 \rhd \mathcal{E}_2[\beta] \uparrow \mathbf{T}$.
  1. Then $\mathsf{Typerec}[\alpha.K_1](\mathcal{E}_1[\beta]; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}})$ and $\mathsf{Typerec}[\alpha.K_1](\mathcal{E}_2[\beta]; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}})$ are head-normal.
  2. The last assumption in the statement of the lemma implies $(\Delta_1; \Pi\alpha{::}\mathbf{T}.K_1)$ **is** $(\Delta_2; \Pi\alpha{::}\mathbf{T}.K_2)$.
  3. By Lemma 5.3.9 parts 1 and 2, we have $\Delta_1 \rhd \mathsf{Typerec}[\alpha.K_1](\mathcal{E}_1[\beta]; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}) \uparrow [\mathcal{E}_1[\beta]/\alpha]K_1 \leftrightarrow \Delta_2 \rhd \mathsf{Typerec}[\alpha.K_2](\mathcal{E}_2[\beta]; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}) \uparrow [\mathcal{E}_2[\beta]/\alpha]K_2$.
  4. By the same lemma we have $(\Delta_1; \mathcal{E}_1[\beta]; \mathbf{T})$ **is** $(\Delta_2; \mathcal{E}_2[\beta]; \mathbf{T})$,
  5. $(\Delta_1; [\mathcal{E}_1[\beta]/\alpha]K_1)$ **is** $(\Delta_2; [\mathcal{E}_2[\beta]/\alpha]K_2)$.
  6. By Lemma 5.3.9 part 4, it then follows that $(\Delta_1; \mathsf{Typerec}[\alpha.K_1](\mathcal{E}_1[\beta]; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [\mathcal{E}_1[\beta]/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](\mathcal{E}_2[\beta]; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [\mathcal{E}_2[\beta]/\alpha]K_2)$.
  7. Using Lemma 5.3.8 and Lemma 5.3.4 it follows that $(\Delta_1; \mathsf{Typerec}[\alpha.K_1](A_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [A_1/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](A_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [A_2/\alpha]K_2)$.
- Case: $\Delta_1 \rhd A_1 \Downarrow \mathcal{E}_1[\rightharpoonup]$ and $\Delta_2 \rhd A_2 \Downarrow \mathcal{E}_2[\rightharpoonup]\Delta_1 \rhd \mathcal{E}_1[\rightharpoonup] \uparrow \mathbf{T} \leftrightarrow \Delta_2 \rhd \mathcal{E}_2[\rightharpoonup] \uparrow \mathbf{T}$.
  1. Since $\Delta_1 \rhd \mathcal{E}_1[\rightharpoonup] \uparrow \mathbf{T}$, it follows that $\Delta_1 \rhd A_1 \Downarrow A_1' \rightharpoonup A_1''$, and similarly that $\Delta_2 \rhd A_2 \Downarrow A_2' \rightharpoonup A_2''$,
  2. and that $\Delta_1 \rhd A_1' :: \mathbf{T} \Leftrightarrow \Delta_2 \rhd A_2' :: \mathbf{T}$ and $\Delta_1 \rhd A_1'' :: \mathbf{T} \Leftrightarrow \Delta_2 \rhd A_2'' :: \mathbf{T}$.
  3. By the inductive hypothesis, then $(\Delta_1; \mathsf{Typerec}[\alpha.K_1](A_1'; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [A_1'/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](A_2'; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [A_2'/\alpha]K_2)$.
  4. and $(\Delta_1; \mathsf{Typerec}[\alpha.K_1](A_1''; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [A_1''/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](A_2''; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [A_2''/\alpha]K_2)$.

5. Therefore,
$(\Delta_1; A_1^{\rightharpoonup}(A_1')(A_1'')(\mathsf{Typerec}[\alpha.K_1](A_1'; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}))(\mathsf{Typerec}[\alpha.K_1](A_1''; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}})); [A_1' \rightharpoonup A_1''/\alpha]K_1)$ **is** $(\Delta_2; A_2^{\rightharpoonup}(A_2')(A_2'')(\mathsf{Typerec}[\alpha.K_2](A_2'; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}))(\mathsf{Typerec}[\alpha.K_2](A_2''; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}})); [A_2' \rightharpoonup A_2''/\alpha]K_2)$.
6. By Lemma 5.3.8 and Lemma 5.3.4, $(\Delta_1; \mathsf{Typerec}[\alpha.K_1](A_1; A_1^{\rightharpoonup}; A_1^{\mathrm{ow}}); [A_1/\alpha]K_1)$ **is** $(\Delta_2; \mathsf{Typerec}[\alpha.K_2](A_2; A_2^{\rightharpoonup}; A_2^{\mathrm{ow}}); [A_2/\alpha]K_2)$.

- $\Delta_1 \triangleright A_1 \Downarrow \mathcal{E}_1[c]$ and $\Delta_2 \triangleright A_2 \Downarrow \mathcal{E}_2[c]$ where c is not $\rightharpoonup$. Analogous to previous case, although there is no need to appeal to the inductive hypothesis for the "otherwise" case.

∎

Then the remaining decidability results for the constructor and kind algorithms go through unchanged. Finally, the normalization algorithm must be extended with a new case:

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.K](p; A^{\rightharpoonup}; A^{\mathrm{ow}}) \longrightarrow \mathsf{Typerec}[\alpha.K'](p'; A^{\rightharpoonup\prime}; A^{\mathrm{ow}\prime}) \uparrow [p/\alpha]K$$

$$\begin{aligned} \text{if } & \Gamma, \alpha{::}\mathbf{T} \triangleright K \Longrightarrow K',\ \Gamma \triangleright p :: \mathbf{T} \Longrightarrow p', \\ & \Gamma \triangleright A^{\rightharpoonup} :: K^{\alpha} \Longrightarrow A^{\rightharpoonup\prime}, \\ & \text{and } \Gamma \triangleright A^{\mathrm{ow}} :: \Pi\alpha{::}\mathbf{T}.K \Longrightarrow A^{\mathrm{ow}\prime}. \end{aligned}$$

9.6 Algorithms for Type and Term Judgments

In analogy with the notation for kinds, for any type τ I write τ^{α} to represent the type

$$\forall\alpha_1{::}\mathbf{T}.\forall\alpha_2{::}\mathbf{T}.[\alpha_1/\alpha]\tau \rightharpoonup [\alpha_2/\alpha]\tau \rightharpoonup [(\alpha_1 \rightharpoonup \alpha_2)/\alpha]\tau.$$

This is the type of the function type-constructor case of a term-level typerec annotated with $[\alpha.\tau]$.

Head-normalization and other properties of types are unaffected by the addition of Typerec and typerec. A new cases must be added to the algorithm for computing principal types

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.\tau](A; e^{\rightharpoonup}; e^{\mathrm{ow}}) \Uparrow [A/\alpha]\tau,$$

to weak term equivalence

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.\tau_1](A_2; e_1^{\rightharpoonup}; e_1^{\mathrm{ow}}) \leftrightarrow \mathsf{Typerec}[\alpha.\tau_2](A_2; e_2^{\rightharpoonup}; e_2^{\mathrm{ow}})$$

$$\begin{aligned} \text{if } & \Gamma, \alpha{::}\mathbf{T} \triangleright \tau_1 \Leftrightarrow \tau_2,\quad \Gamma \triangleright A_1 \Leftrightarrow A_2 :: \mathbf{T}, \\ & \Gamma \triangleright e_1^{\rightharpoonup} \Leftrightarrow e_2^{\rightharpoonup},\quad \text{and}\quad \Gamma \triangleright e_1^{\mathrm{ow}} \Leftrightarrow e_2^{\mathrm{ow}}, \end{aligned}$$

and to type synthesis

$$\Gamma \triangleright \mathsf{Typerec}[\alpha.\tau](A; e^{\rightharpoonup}; e^{\mathrm{ow}}) \rightrightarrows [A/\alpha]\tau$$

$$\begin{aligned} \text{if } & \Gamma, \alpha{::}\mathbf{T} \triangleright \tau,\quad \Gamma \triangleright A \leftleftarrows \mathbf{T}, \\ & \Gamma \triangleright e^{\rightharpoonup} \leftleftarrows \tau^{\alpha},\quad \text{and}\quad \Gamma \triangleright e^{\mathrm{ow}} \leftleftarrows \forall\alpha{::}K.\tau. \end{aligned}$$

9.7 Completeness and Decidability for Types and Terms

The symmetrized weak term equivalence algorithm gets a new case:

$$\Gamma_1 \triangleright \mathsf{Typerec}[\alpha.\tau_1](A_2; e_1^{\rightharpoonup}; e_1^{\mathrm{ow}}) \leftrightarrow \Gamma_2 \triangleright \mathsf{Typerec}[\alpha.\tau_2](A_2; e_2^{\rightharpoonup}; e_2^{\mathrm{ow}})$$

$$\begin{aligned} \text{if } & \Gamma_1, \alpha{::}\mathbf{T} \triangleright \tau_1 \Leftrightarrow \Gamma_2, \alpha{::}\mathbf{T} \triangleright \tau_2,\quad \Gamma_1 \triangleright A_1 :: \mathbf{T} \Leftrightarrow \Gamma_2 \triangleright A_2 :: \mathbf{T}, \\ & \Gamma_1 \triangleright e_1^{\rightharpoonup} \Leftrightarrow \Gamma_2 \triangleright e_2^{\rightharpoonup},\quad \text{and}\quad \Gamma_1 \triangleright e_1^{\mathrm{ow}} \Leftrightarrow \Gamma_2 \triangleright e_2^{\mathrm{ow}}, \end{aligned}$$

Again, the logical relations are unchanged. The new case for the proof that declarative equivalence implies algorithmic equivalence follows directly from the inductive hypothesis. The completeness and decidability results then hold unchanged, as does strengthening for term variables.

9.8 Properties of Evaluation

Even if Proposition 5.3.15 is restricted to head-normal paths as suggested above, one can still prove the Canonical Forms lemmas. Thus it is easy to see that evaluation of well-typed terms never gets "stuck".

Chapter 10

Conclusion

10.1 Summary of Contributions

In this dissertation I have presented the MIL_0 calculus, which models the internal language used by the TILT compiler. The language contains two variants of singletons: singletons with $\beta\eta$-equivalence (instantiated as singleton kinds) and labeled singletons with a weak term equivalence (instantiated as singleton types). The former is particularly simple and elegant, but is unusually context-sensitive.

I have thoroughly studied the equational and proof-theoretic properties of the MIL_0 calculus, and have shown that typechecking is decidable. I have presented algorithms for implementing typechecking; those for constructors and kinds form the basis of the typechecker implementation in the TILT compiler [Pet00].

The equivalence algorithm for type constructors employs an apparently novel kind-directed framework. This is extremely well-suited for cases in which equivalence is dependent upon the classifier. Examples of other such languages include those with terminal types (where all terms of this type are equal), or calculi with records and width subtyping (where equivalence of two records depends only on the equivalence of the subset of fields mentioned in the classifying record type). This approach can even be used in the absence of subtyping, subkinding, or singletons [HP99].

The correctness proofs for my equivalence algorithms employ an unusual variant of Kripke logical relation, in which the relations are indexed by two kinds or types and by two worlds. This permits a very straightforward proof of correctness for the equivalence algorithms. I have found the logical relations approach to proving completeness to be remarkably robust under minor changes to the equational theory; even the addition of type analysis constructs requires few changes.

Crary has used the results of Chapter 5 to show that a language with singleton kinds can be translated into a language without, in a fashion which preserves well-typedness [Cra00]. Intuitively, one can certainly "substitute in" all of the definitions induced by singletons. However, the correctness of afterwards erasing all of singleton kinds is a form of strengthening property. Crary proves this by working with the algorithmic form of constructor equivalence.

10.2 Related Work

10.2.1 Singletons and Definitions in Type Systems

The main previous study of singleton types in the literature is due to Aspinall [Asp95, Asp97]. He studied a calculus $\lambda_{\leq\{\}}$ containing singleton types, dependent function types, and β-equivalence.

Labeled singletons are primitive notions in this system; in the absence of η-equivalence the encoding of §2.3 does not work. He conjectured that term equivalence in $\lambda_{\leq\{\}}$ was decidable, but gave no algorithm.

Crary has also used singleton types and singleton kinds. His thesis [Cra98] includes a system whose kind system extends the one presented here with subtyping and power kinds. He also conjectured that both type equivalence and typechecking were decidable.

Crary has also used an extremely simple form of singleton type (with no elimination rule or subtyping) in order to prove parametricity results [Cra99]. As one example, he shows that any function f of type $\forall\alpha.\alpha\to\alpha$ must act as a the identity because

$$f(\mathbf{S}(v : \tau))(v) : \mathbf{S}(v : \tau)$$

so by soundness of the type system any value returned by this application must be equal to v. Furthermore, evaluation in his system obviously does not depend upon type arguments to functions, so f must act as an identity[1] for every argument of any type. (This argument does not apply to MIL_0 because here singleton types are not type constructors.)

There are other ways to support equational information in a type system besides singleton types. Severi and Poll [SP94] study 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. In this system, the typing context contains both the type for each variable, and an optional definition. This calculus contains no notion of partial definition, no subtyping, and cannot express constraints on function arguments. This approach may be sufficient to represent information needed for cross-module inlining (particularly when based upon the lambda-splitting work of Blume and Appel [BA97, Blu97]), but this cannot model sharing constraints or definitions in a modular framework (where only some parts of a module have known definition).

Type theoretic studies of the SML module system have been studied by Harper and Lillibridge under the name of *translucent sums* [HL94, Lil97] in which modules are first-class values, and by Leroy under the name of *manifest types* [Ler94] in which modules are second-class. These two systems are essentially similar: the calculus includes module constructs, and corresponding signatures; as in Standard ML the type components of signatures may optionally specify definitions. The key difference from MIL_0 is that type definitions are specified at the *type* level, rather than at the *kind* level. Because of this, type equivalence does depend on the typing context but not on the (unique) classifying kind. Typechecking for translucent sums is undecidable (although type equivalence is decidable). No analogous result is known for manifest types; modules may lack most-specific signatures, prohibiting standard methods for typechecking.

A very powerful construct is the I-type of Martin-Löf's extensional type theory [ML84, Hof95]. A term of type $I(e_1, e_2)$ represents a *proof* that e_1 and e_2 are equivalent. This can lead to undecidable typechecking very quickly, as one can use this to add arbitrary equations as assumptions in the typing context.

The language Dylan [Sha96] contains a notion of "singleton type", but these are checked only at run-time (essentially pointer-equality) to resolve dynamic overloading.

10.2.2 Decidability of Equivalence and Typechecking

My approach to implementing and studying constructor equivalence was inspired by work by Coquand for a dependently-typed lambda calculus [Coq91]. However, because his the equivalence was not context-sensitive in any way, both our algorithm and proof are substantially different from

[1] Up to type annotations, which as just stated do not affect evaluation behavior

Coquand's. Because of issues such as the form of the validity logical relations and the particular symmetry and transitivity properties of the 6-place algorithm, our initial attempts to use more traditional Kripke logical relations (with a pair of contexts being a single world) were unsuccessful.

Systems in which equivalence depends upon the typing context were mentioned in §10.2.1. However, there appear to be relatively few decidability results for lambda calculi with typing-context-sensitive or classifier-sensitive equivalences, perhaps because standard techniques of rewriting to normal form are difficult to apply. Many calculi include subtyping but not subkinding; in such cases either only type equivalence is considered (which is independent of subtyping) or else term equivalence is not affected by subtyping and hence can be computed in a context-free manner.

One exception is the work of Curien and Ghelli [CG94], who proved the 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. They eliminate context-sensitivity by inserting explicit coercions to mark uses of subsumption and then give a rewriting strategy for the calculus with coercions. Their proof uses translations between three different typed λ-calculi.

It would be interesting to see if the approach used for MIL_0 could be applied to their source language, avoiding the use of translations. Although adapting my equivalence algorithm seems easy, the fact that they study an impredicative calculus would require an extension of the theory in order to prove the completeness of this algorithm.

Compagnoni and Goguen [CG97] also use a normalization algorithm and Kripke logical relations argument for proving properties (including decidability of subtyping) for the language $\mathcal{F}^\omega_\leq$, a variant of $F^\omega_{<:}$ with higher-order subtyping and the kernel Fun rule [CW85] for quantifier subtyping. However, adapting these methods to include subkinding and η-expansion seems nontrivial.

10.3 Open Questions and Conjectures

I conclude with an overview of several remaining issues which could be the subject of future work in the study of singleton types and kinds.

10.3.1 Removing Type Annotations from let

The primary practical defect of the MIL_0 term language appears to be the required type labels in let-bindings — in particular, the type annotation on the bound variable. Because a local binding is required for every sub-computation, these type annotations can substantially increase the total size of a program. This exacts not only a penalty in the space consumed by the program's representation, but also costs time in manipulating the representation: the typechecker must verify the correctness of these annotations, transformations such as substitutions or optimizations must be applied to all of the annotations, and so on. Furthermore, if one wishes to bind x to the pair $\langle 3, 4\rangle$, one must choose whether to annotate this binding with the simple type $\mathsf{int}\times\mathsf{int}$, or one of its larger but more-precise types: $\mathbf{S}(3 : \mathsf{int})\times\mathbf{S}(4 : \mathsf{int})$ or $\mathbf{S}(\langle 3,4\rangle : \mathsf{int}\times\mathsf{int})$ or even $\mathbf{S}(\langle 3,4\rangle : \mathbf{S}(3 : \mathsf{int})\times\mathbf{S}(4 : \mathsf{int}))$.

This is easy to change in the MIL_0 definition; the mediating type of the bound variable is simply chosen nondeterministically. In this fashion Rule 2.76 becomes

$$\frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x{:}\tau' \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash (\mathsf{let}\ x{=}e'\ \mathsf{in}\ e : \tau\ \mathsf{end}) : \tau}$$

and Rule 2.89 becomes

$$\frac{\begin{array}{c}\Gamma \vdash e_1' \equiv e_2' : \tau' \\ \Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma, x{:}\tau_1' \vdash e_1 \equiv e_2 : \tau_1\end{array}}{\Gamma \vdash (\mathsf{let}\ x{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \equiv (\mathsf{let}\ x{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end}) : \tau_1}.$$

Adapting the algorithm for checking the well-formedness of a let-binding is easy: just replace uses of the annotation with uses of the principal type of the bound expression, which is already being calculated. As the type annotation need no longer be validated, this requires doing strictly less work.

Unfortunately, computing equivalence of two let-bindings without this type annotation is more difficult. It should look something like the following:

$$\begin{array}{ll}\Gamma \triangleright (\mathsf{let}\ x{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \leftrightarrow & \text{if } \Gamma \triangleright e_1' \Leftrightarrow e_2' \text{ and } \Gamma, x{:}\boxed{???} \triangleright e_1 \Leftrightarrow e_2, \text{ and } \Gamma \triangleright \tau_1 \Leftrightarrow \tau_2. \\ \qquad (\mathsf{let}\ x{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end}). & \end{array}$$

But what type x should be given while comparing e_1 and e_2? A problem arises; is entirely possible for e_1' and e_2' to be well-formed and for $\Gamma \triangleright e_1' \Leftrightarrow e_2'$ but for e_1' and e_2' to have *different* principal types. (For example, assume $y{:}\mathbf{S}(\langle 3,4\rangle : \mathsf{int}{\times}\mathsf{int})$ and compare y with $\langle 3,4\rangle$.) If I attempt to avoid this asymmetry by maintaining two contexts and using both principal types, then the contexts maintained by the algorithm no longer remain provably equivalent and properties like soundness become more difficult to show.

However, any two equivalent terms *in weak head-normal form* have equivalent principal types. More generally, any two well-formed terms equivalent under the weak term equivalence relation $\leftrightarrow$ have provably equivalent principal types. This suggests the strategy of using the principal type of the head-normal form of either let-bound expression:

$$\begin{array}{ll}\Gamma \triangleright (\mathsf{let}\ x{=}e_1'\ \mathsf{in}\ e_1 : \tau_1\ \mathsf{end}) \leftrightarrow & \text{if } \Gamma \triangleright e_1' \Leftrightarrow e_2',\ \Gamma \triangleright e_1' \Downarrow d_1',\ \Gamma \triangleright d_1' \Uparrow \tau', \\ \qquad (\mathsf{let}\ x{=}e_2'\ \mathsf{in}\ e_2 : \tau_2\ \mathsf{end}) & \qquad \Gamma, x{:}\tau' \triangleright e_1 \Leftrightarrow e_2, \text{ and } \Gamma \triangleright \tau_1 \Leftrightarrow \tau_2.\end{array}$$

or using both equivalent types in the symmetric form of the algorithm.

It is not too hard to show this modified algorithm is sound. The key insight is that if d_i' is the head-normal form for e_i' (for $i \in \{1,2\}$) then

$$\Gamma \vdash (\mathsf{let}\ x{=}e_i'\ \mathsf{in}\ e_i : \tau_i\ \mathsf{end}) \equiv (\mathsf{let}\ x{=}d_i'\ \mathsf{in}\ e_i : \tau_i\ \mathsf{end}) : \tau_i$$

so that while comparing the bodies the algorithm can assume it was given d_1' and d_2' instead of e_1' and e_2', taking advantage of the equal principal types.

Unfortunately, I cannot prove this algorithm complete. Everything goes through except the final step, proving that declarative equivalence implies logical equivalence. The difficulty is that the type τ' computed by the algorithm need not have a counterpart in the declarative proof of equivalence, so that the inductive hypothesis cannot be applied to τ'.

Conjecture 10.3.1
The algorithm as modified as suggested here is not only sound, but complete and terminating for the language where the type annotations are omitted from local variable bindings.

10.3.2 Unlabeled Singleton Types

Principal types in MIL_0 can be quite large. For example, the principal type of the pair $\langle\langle 2,3\rangle,\langle 4,5\rangle\rangle$ is

$$\mathbf{S}(\langle\langle 2,3\rangle,\langle 4,5\rangle\rangle : \mathbf{S}(\langle 2,3\rangle : \mathbf{S}(2:\mathsf{int})\times\mathbf{S}(3:\mathsf{int}))\times\mathbf{S}(\langle 4,5\rangle : \mathbf{S}(4:\mathsf{int})\times\mathbf{S}(5:\mathsf{int}))).$$

Despite the fact that this type classifies exactly the same values as the simpler type

$$\mathbf{S}(\langle\langle 2,3\rangle,\langle 4,5\rangle\rangle : (\mathsf{int}\times\mathsf{int})\times(\mathsf{int}\times\mathsf{int}))$$

these two types are not provably equivalent. The former is a strict subtype of the latter, and is hence the one which must be synthesized by the typechecking algorithms. Even if type equivalence were strengthened to equate these two types, experience in the TILT compiler with labeled singleton kinds has demonstrated that it is difficult to avoid generating singletons with redundant information in the labels.

Furthermore, term equivalence is weak enough that it does not depend upon the classifying type. In a sense, then, the classifier in a singleton type is not adding useful information. An obvious alternative is the "unlabeled singleton" $\mathbf{S}(v)$ briefly considered by Aspinall. Declaratively one might have such rules as

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \mathbf{S}(v)}$$

and

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{S}(v) \leq \tau}.$$

Finding a plausible typechecking algorithm for such a language has proven surprisingly difficult, however. Principal type synthesis becomes trivial (the principal type for any value v is just $\mathbf{S}(v)$) and useless for the purposes of type-checking. What is needed is the "most-precise type that is not a singleton", which for values is the "second-most-precise type"[2]. I do not yet have a plausible algorithm for when both projections and pairs are values[3].

Leaf Petersen has studied a variant of the MIL_0 kind system which allows unlabeled singleton kinds [Pet00] to decrease the size of program representations. This has been implemented in TILT. His approach is to treat unlabeled singletons as an abbreviation mechanism, and he shows how to translate away all uses of unlabeled singletons.

It is possible that a similar approach may work for singleton types. There are additional difficulties, however. In particular, mixing labeled and unlabeled singletons can cause problems. Assume we have a program context in which x has type $\mathsf{int}\times\mathsf{int}$. Then under the natural translation approach one would expect $\mathbf{S}(x)$ to be equivalent to the labeled singleton type $\mathbf{S}(x : \mathsf{int}\times\mathsf{int})$. However, upon substituting the pair $\langle 2,3\rangle$ the types become $\mathbf{S}(\langle 2,3\rangle)$ and $\mathbf{S}(\langle 2,3\rangle : \mathsf{int}\times\mathsf{int})$. However, the labeled singleton corresponding to the former of these two types is now the more precise type $\mathbf{S}(\langle 2,3\rangle : \mathbf{S}(2:\mathsf{int})\times\mathbf{S}(3:\mathsf{int}))$.

Thus two equivalent types become inequivalent after substitution of a value for a variable. This means that substitution (and hence inlining) is no longer guaranteed to preserve well-formedness of programs. This is not a good property for a compiler representation to have.

[2]Leaf Petersen has suggested this be called the "vice-principal type".

[3]There are some hints, however, that computing types of values by looking at their head-normal forms may be possible.

Conjecture 10.3.2
If labeled singleton types are replaced completely with unlabeled singleton types, then there is still a reasonable algorithm for deciding well-formedness of programs.

The current TILT implementation includes only singleton kinds. I intend to implement singleton types for cross-module inlining, based on the algorithm sketched here.

10.3.3 Recursive Types

Several authors from Amadio and Cardelli on [AC93, Bra97] have studied algorithms for deciding type equivalence for recursive types, which are viewed as representing infinite trees. This can be most simply formalized with two rules: the roll-unroll rule

$$\frac{\Gamma, \alpha{::}\mathbf{T} \vdash A}{\Gamma \vdash \mu\alpha{::}\mathbf{T}.A \equiv [\mu\alpha{::}\mathbf{T}.A/\alpha]A :: \mathbf{T}}$$

and a coinductive principle. Together these rules allow such equivalences as

$$\vdash (\mu\alpha{::}\mathbf{T}.\mathsf{int}\rightharpoonup\alpha) \equiv (\mu\alpha{::}\mathbf{T}.\mathsf{int}\rightharpoonup(\mathsf{int}\rightharpoonup\alpha)) :: \mathbf{T}.$$

For the case of simple types where type equivalence is the congruence induced by these two rules, the standard simple algorithm combines structural comparison of the two types with unrolling whenever a recursive type is reached. To prevent infinite unrolling, a trail of the previously compared types is maintained; by coinductive nature of equivalence, any comparison previously seen can simply be reported successful.

The requirements for the TILT compiler appear to be much simpler; we need only the one rule

$$\frac{\Gamma \vdash [\mu\alpha{::}\mathbf{T}.A_1/\alpha]A_1 \equiv [\mu\alpha{::}\mathbf{T}.A_2/\alpha]A_2 :: \mathbf{T}}{\Gamma \vdash \mu\alpha{::}\mathbf{T}.A_1 \equiv \mu\alpha{::}\mathbf{T}.A_2 :: \mathbf{T}}$$

That is, two recursive types are equal if their unrollings are equal. This is equivalent to the rule

$$\frac{\Gamma, \alpha{::}\mathbf{T} \vdash A}{\Gamma \vdash \mu\alpha{::}\mathbf{T}.A \equiv \mu\alpha{::}\mathbf{T}.[\mu\alpha{::}\mathbf{T}.A/\alpha]A :: \mathbf{T}}$$

called "Shao's Rule" in [CHC$^+$98]. This is a much weaker equational theory; In contrast to the roll-unroll rule above, it equates recursive types only to other recursive types.

There has been no study of algorithms for recursive types where there are other interesting type equations such as β-equivalence (e.g., F_ω extended with recursive types). However there is a seemingly natural extension of the simple algorithm above, which has been implemented in TILT.

1. TILT keeps a trail of the pairs of recursive types previously compared;
2. Whenever weak path equivalence is about to compare two recursive types, it adds them to the trail, unrolls the two types, and runs the general constructor equivalence algorithm on the two results.
3. If a loop is detected, comparison fails. (Recall that we are not requiring equivalence to be coinductive.)

Conjecture 10.3.3
The above algorithm is sound, complete, and terminating for MIL_0 extended with recursive types and Shao's rule.

The difficulty in proving completeness and termination is that because of the trail I see no way to make this algorithm obviously transitive. This is a key step in my theoretical development, and so the approach I use in this dissertation does not appear to extend in any nice fashion.

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[Asp95] David Aspinall. Subtyping with Singleton Types. In *Proc. Computer Science Logic (CSL '94)*, 1995. In Springer LNCS 933.

[Asp97] David Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[Asp00] David Aspinall. Subtyping with Power Types. In *Proc. Computer Science Logic (CSL 2000)*, 2000. To Appear.

[BA97] Matthias Blume and Andrew W. Appel. Lambda-Splitting: A Higher-Order Approach to Cross-Module Optimizations. In *Proc. 1997 ACM International Conference on Functional Programming (ICFP '97)*, pages 112–124, 1997.

[Blu97] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, 1997.

[Bra97] Michael Brandt. Recursive subtyping: Axiomatizations and computational interpretations. Master's thesis, DIKU, University of Copenhagen, August 1997.

[CG94] 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.

[CG97] Adriana Compagnoni and Healfdene Goguen. Typed Operational Semantics for Higher Order Subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, 1997.

[CHC$^+$98] Karl Crary, Robert Harper, Perry Cheng, Leaf Petersen, and Chris Stone. Transparent and Opaque Interpretations of Datatypes. Technical Report CMU-CS-98-177, Department of Computer Science, Carnegie Mellon University, 1998.

[CM94] Pierre Crégut and David B. MacQueen. An implementation of higher-order functors, June 1994.

[Coq91] 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.

[Cra98] Karl F. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, 1998.

[Cra99] Karl Crary. A simple proof technique for certain parametricity results. In *Proc. 1999 ACM International Conference on Functional Programming (ICFP '99)*, pages 82–89, 1999.

[Cra00] Karl Crary. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, School of Computer Science, Carnegie Mellon University, 2000.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 180–190, 1988.

[FSDF93] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *Proc. ACM 1993 Conference on Programming Language Design and Implementation (PLDI '93)*, pages 237–247, 1993.

[Gir72] J. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[Har00] Robert Harper, 2000. Private communication.

[HL94] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proc. 21st ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 123–137, 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling Polymorphism using Intensional Type Analysis. In *Proc. 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 130–141, 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order Modules and the Phase Distinction. In *Proc. 17th ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 341–354, 1990.

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Edinburgh LFCS, 1995. Available as Edinburgh LFCS Technical Report ECS-LFCS-95-327.

[HP99] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. In *Proc. Workshop on Logical Frameworks and Meta-Languages*, 1999. Extended version available as CMU Technical Report CMU-CS-99-159.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, 1997.

[HS00] Robert Harper and Christopher Stone. A Type-Theoretic Interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 109–122, 1994.

[Ler95] Xavier Leroy. Applicative Functors and Fully Transparent Higher-Order Modules. In *Proc. 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 142–153, 1995.

[Lil97] 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.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis-Napoli, 1984.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 271–283, 1996.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. Available as CMU Technical Report CMU-CS-95-226.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MT94] David B. MacQueen and Mads Tofte. A Semantics for Higher-order Functors. In *Proc. 5th European Symposium on Programming*, number 788 in LNCS, pages 409–423, 1994.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[MWCG97] 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.

[Myc84] A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proc. 6th Int. Conf. on Programming*, number 167 in LNCS, pages 217–239, 1984.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, 1997.

[Nec98] George Ciprian Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1998. Available as CMU Technical Report CMU-CS-98-154.

[Pet00] Leaf Petersen, 2000. Unpublished manuscript.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-205.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[PZ00] Jens Palsberg and Tian Zhao. Efficient and Flexible Matching of Recursive Types. In *Proc. 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*, pages 388–400, 2000.

[SH99] 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.

[Sha96] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, 1996.

[Sha98] Zhong Shao. Typed Cross-Module Compilation. In *Proc. 1998 ACM International Conference on Functional Programming (ICFP '98)*, pages 141–152, 1998.

[SP94] Paula Severi and Eric Poll. Pure Type Systems with definitions. In *Logical Foundations of Computer Science '94*, number 813 in LNCS, 1994.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Available as CMU Technical Report CMU-CS-97-108.

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