Implementing the TILT Internal Language

Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone.

December, 2000

CMU-CS-00-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The TILT compiler for Standard ML represents programs internally using a predicative lambda calculus based on Girard's F_ω. At the kind level, this language is notable for containing singleton kinds and dependent product and function kinds. Previous work [SH99] established the decidability of type equivalence for this language.

This paper presents a typechecking algorithm for the full TILT internal language and discusses some of the more interesting features of the language. The particular use of intensional type analysis to handle arrays of unboxed floating point numbers is described. An extended calculus is also introduced which permits unlabelled singletons at higher kind, in order to allow for more compact program representation. The extended calculus is related to the restricted calculus via a transformation that eliminates the unlabelled singletons, and the decidability of the typechecking algorithms for both the original and extended calculus is shown.

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

Keywords: singleton kinds, typed compilation, intensional polymorphism, type theory

1 Introduction

1.1 Background

The past years have seen a great deal of interest in the idea of "typed compilation": that is, maintaining type information throughout the compilation process. This type information can be exploited by the compiler internally to allow for optimized data representations and to do tag-free garbage collection, as well as providing the compiler with a basis for internal correctness checks. This work was pioneered in the TIL compiler at CMU [TMC$^+$96]. Other recent work has also suggested the possibility of maintaining type information through to the machine code as a form of certification [MWCG97].

The TIL compiler clearly demonstrated that typed compilation was both feasible and desirable. However, TIL compiled only the core language of Standard ML: the powerful modular features that are one of the most important elements of SML were not dealt with. The TIL Two (TILT) compiler was aimed at addressing this shortcoming.

The TILT architecture is based around two typed intermediate languages. The initial elaboration from SML source targets a structures calculus called the HIL (High Intermediate Language). This language is relatively close to SML, and among other things provides the interface language used for separate compilation. After elaboration (and hence typechecking), the HIL is translated to a second typed language called the MIL (Middle Intermediate Language) through a process called phase splitting [HMM90]. The phase splitting process maps each SML structure into separate type and term level records, representing the static and dynamic portions of the structure. Similarly, SML functors are mapped to type and term level functions. In this fashion, modular programs are translated into programs containing only lambda calculus terms.

We will not address the details of phase splitting here, except to note that serving as a target of this translation is the primary motivation for the type theory of the MIL. The MIL must be able to express within a single lambda calculus all of the constructs of both the module language and the core language. Singleton kinds are used to express type definitions in signatures, and dependent product and function kinds serve to express signatures which contain definitions in terms of previous fields.

The MIL is also the language in which almost all of the optimization passes are done. This constrains the design of the MIL, since it must be possible to express the results all of the desired optimizations in a typed fashion. In particular, it is important that the necessary primitives for data representation optimizations be present at this level.

1.2 Overview

This paper gives a detailed overview of the MIL largely as implemented in the TILT compiler. The major omission is that closure conversion and the typing of closures is not treated here.

In [SH99], Stone and Harper present an algorithm for deciding type equivalence in a lambda calculus with singleton kinds. Section 2 of this paper describes the extension of this calculus to the full MIL language. Design issues motivating the extensions are discussed, and algorithms for typechecking are given along with proofs of termination.

Section 3 addresses a major practical shortcoming of the MIL: the inability to represent kinds compactly. We present an extended calculus called the NIL which addresses these shortcomings by providing unlabelled singletons at higher kind. The MIL algorithms and proofs are extended to the NIL.

The main technical results of the paper are the creation of an algorithm for deciding typechecking in a language with unlabelled singletons at higher kind, and the proofs of the decidability of typechecking in both the core and the extended system.

Appendices A and B contain the full static semantics for the MIL and NIL, respectively.

2 Mil

2.1 Relation to $\lambda_{\leq}^{\Pi\Sigma S}$

The constructor and kind level of the MIL has been studied separately by Harper and Stone [SH99]. That paper presented a core MIL-like language called $\lambda_{\leq}^{\Pi\Sigma S}$and gave an a sound and complete algorithm for determining constructor equivalence.

Kinds	$\kappa ::=$	T	Kind of simple constructors
	$\mid$	$S_T(c)$	Singleton kind
	$\mid$	$\Sigma(\alpha :: \kappa).\kappa$	Dependent function kind
	$\mid$	$\Pi(\alpha :: \kappa).\kappa$	Dependent product kind
Constructors	$c ::=$	b_i	Base types
	$\mid$	α	Variables
	$\mid$	$\lambda\alpha{::}\kappa.c$	Function
	$\mid$	$c\,c$	Application
	$\mid$	$\langle c, c\rangle$	Pair
	$\mid$	$c.i$	Projection
Contexts	$\Delta ::=$	$\bullet$	Empty context
	$\mid$	$\Delta[\alpha{::}\kappa]$	Context extension

Figure 1: $\lambda_{\leq}^{\Pi\Sigma S}$Syntax

The syntax of the $\lambda_{\leq}^{\Pi\Sigma S}$ calculus is given in figure 1. This calculus makes up the core of the MIL language discussed here. The major type theoretic ideas of the MIL are for the most part already present in $\lambda_{\leq}^{\Pi\Sigma S}$. From a practical standpoint however, many essential components are missing from $\lambda_{\leq}^{\Pi\Sigma S}$: in particular, $\lambda_{\leq}^{\Pi\Sigma S}$ does not deal with the term level structure of the language. This section will flesh out the term level extensions necessary, and will discuss their typing properties. The kind level remains unchanged from $\lambda_{\leq}^{\Pi\Sigma S}$ to MIL, but the set of constructors increases.

2.2 Constructors and types

The syntax for the constructor and kind levels of the MIL is given in figure 2. In contrast to $\lambda_{\leq}^{\Pi\Sigma S}$, the MIL language includes base constructors such as *Int* that are used to classify terms. All of these base constructors are standard, with the exception of the use of the known sum type, corresponding to the type of a sum for which the branch inhabited is known.

The MIL also includes an explicit let construct, although technically this is definable in the calculus [SH99]. Let binding provides a means for expressing constructors more compactly, as well as to name and reuse the results of type computations. This serves both to help make compilation faster and to improve runtime performance, since constructors may be needed at runtime. In order to reduce the size of programs, we elide the classifiers on the let bound variables. While this

Kinds	$\kappa ::=$	T	Types
	$\mid$	$S_T(c)$	Singleton kinds
	$\mid$	$\Sigma(\alpha :: \kappa).\kappa$	Dependent pair kinds
	$\mid$	$\Pi(\alpha :: \kappa).\kappa$	Dependent function kinds
Constructors	$c ::=$	α	Constructor variable
	$\mid$	Int	Integers
	$\mid$	$Boxedfloat$	Boxed floating point numbers
	$\mid$	$\mu(\alpha, \beta).(c, c)$	Recursive constructor
	$\mid$	$c \times c$	Pairs
	$\mid$	$c \rightarrow c$	Monomorphic functions
	$\mid$	$c + c$	Sums
	$\mid$	$c +^i c$	Known sums
	$\mid$	$c\ array$	Polymorphic arrays
	$\mid$	$\lambda\alpha{::}\kappa.c$	Function
	$\mid$	$c\,c$	Application
	$\mid$	$\langle c, c \rangle$	Constructor pairing
	$\mid$	$\pi_i\, c$	Projection
	$\mid$	$\textbf{let}\ \alpha = c\ \textbf{in}\ c\ \textbf{end}$	Constructor definition
Types	$\tau ::=$	$T(c)$	Constructor inclusion
	$\mid$	$(\alpha :: \kappa, \tau) \rightarrow \tau$	Polymorphic functions
	$\mid$	$Float$	Unboxed floating point numbers
	$\mid$	$\tau \times \tau$	Pair type
	$\mid$	$\textbf{let}\ \alpha = c\ \textbf{in}\ \tau\ \textbf{end}$	Constructor definition
Contexts	$\Delta ::=$	$\bullet$	Empty context
	$\mid$	$\Delta[x : \tau]$	Constructor extension
	$\mid$	$\Delta[\alpha{::}\kappa]$	Kind extension

The notation $\kappa_1 \times \kappa_2$ indicates $\Sigma(\alpha :: \kappa_1).\kappa_2$ where $\alpha \notin \mathrm{fv}(\kappa_2)$.

Figure 2: MIL Kinds, constructors and contexts

information is easily reconstructed from the definition itself, this imposes some additional work on the compiler.

Also given in figure 2 is the syntax for the type level. Unlike the constructor level which corresponds to the notion of *types as data*, the type level in a predicative system corresponds to the notion of *types as classifiers*. The constructor level is included into the type level via an explicit inclusion $T(c)$. The type level also contains classifiers for polymorphic functions, unboxed floating point numbers, and pairs of terms. The duplication of the the pair type at the type level indicates the possibility of constructing pairs containing arbitrary terms (such as unboxed floats) which is not provided for by the constructor level. For similar reasons a constructor let form is also included in the type level so that constructors (but not types!) can be bound in types.

For presentational purposes, the static semantics of the MIL calculus is initially described using a straightforward declarative approach which is more easily understood. This approach does not correspond naturally to an algorithm, and hence it is will be necessary in subsequent sections to develop an equivalent algorithmic presentation of the static semantics. The complete declarative static semantics for the MIL language is defined in appendix A.1, but for the most part this section will concentrate on the key non-standard elements that make the MIL theory interesting.

Δ ok	Well formed-contexts.
$\Delta \vdash \kappa$	Well-formed kinds.
$\Delta \vdash \kappa_1 \preceq \kappa_2$	Subkinding.
$\Delta \vdash c_1 \equiv c_2 :: \kappa$	Constructor equivalence.
$\Delta \vdash c :: \kappa$	Well-formed constructors.
$\Delta \vdash \tau$	Well-formed types.
$\Delta \vdash e : \tau$	Well-formed terms.

Figure 3: MIL declarative judgements

The judgements used to define the MIL static semantics are described in figure 3. In addition to the expected well-formedness judgements, there is also a sub-kinding judgement. The presence of singleton kinds means that a constructor may have multiple kinds: for example, the judgements $\Delta \vdash Int :: T$ and $\Delta \vdash Int :: S_T(Int)$ are both derivable in the system. The sub-kinding judgment reflects the fact that a singleton kind gives more information than does a simple kind, and hence should be viewed as a subtype. In particular, the key rule from the sub-kinding judgment is the singleton rule:

$$\frac{\Delta \vdash S_T(c)}{\Delta \vdash S_T(c) \preceq T}\ \text{SingletonL}$$

which says that any well-formed singleton kind is a sub-kind of T. The sub-kinding judgment affects constructor well-formedness via a subsumption rule

$$\frac{\Delta \vdash c :: \kappa \quad \Delta \vdash \kappa \preceq \kappa'}{\Delta \vdash c :: \kappa'}\ \text{Subkind}$$

which says that a constructor is well-formed at kind κ if it is well-formed at a subtype of κ.

The main non-standard typing rules are the extensionality rules and the self rule of the constructor well-formedness judgement [HL94]. The self rule is the introduction rule for singleton kinds, and says that any constructor c which is well-formed at kind T is well-formed at kind $S_T(c)$.

$$\frac{\Delta \vdash c :: T}{\Delta \vdash c :: S_T(c)} \;\text{Selfify}$$

Accompanying this rule are the extensionality rules:

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c.1 :: \kappa_1'}{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1').\kappa_2} \;\text{SigmaExt1}$$

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c.2 :: \kappa_2'}{\Delta \vdash c :: \kappa_1 \times \kappa_2'} \;\text{SigmaExt2}$$

$$\frac{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta[\alpha{::}\kappa_1] \vdash c\,\alpha :: \kappa_2'}{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2'} \;\text{PiExt}$$

These rules essentially extend the notion of the self rule to higher kinds via eta-expansion: that is, they allow derivations such as $[\alpha{::}\Pi(\beta :: T).T] \vdash \alpha :: \Pi(\beta :: T).S_T(\alpha\ \beta)$ For a more detailed discussion of these rules see [SH99, HL94].

2.3 Terms

The term level MIL syntax is given in figure 4. In addition to the standard lambda calculus constructs the MIL also provides for expression and constructor let bindings, again with the classifier elided for reasons of program size. Unlike most lambda calculi though, the MIL also includes low level data representation primitives (such as float boxing and unboxing primitives). In addition to serving as the target language of phase-splitting, the MIL also serves as the object of most of the compiler optimization phases, including inlining, common subexpression elimination, and function specialization. These optimizations may expose opportunities for data-layout optimization, such as eliminating redundant boxing and unboxing of floats which can only be performed if the boxing and unboxing operations are present at the MIL level.

For similar reasons, the sum case construct in the MIL is also somewhat non-standard, as can be seen from the sum elimination rule [HS97].

$$\frac{\begin{array}{l}\Delta \vdash e \;:\; T(c_1 + c_2) \\ \Delta[x : T(c_1 +^1 c_2)] \vdash e_1 \;:\; \tau \quad \Delta[x : T(c_1 +^2 c_2)] \vdash e_2 \;:\; \tau\end{array}}{\Delta \vdash \mathbf{case}_\tau\, e\, \mathbf{of}\, \{\mathbf{inl}(x) \Rightarrow e_1, \mathbf{inr}(x) \Rightarrow e_2\} \;:\; \tau} \;\text{Sum elimination}$$

Notice that the case construct does not destructure its argument - rather, it will bind the argument in the appropriate branch to a variable whose type is a known sum indicating the inhabited branch. The known sum projection construct can then be used to project out the value if it is actually required by that particular branch.

$$\frac{\Delta \vdash e \;:\; \tau \quad \Delta \vdash \tau \equiv T(c_1 +^i c_2)}{\Delta \vdash \mathbf{proj}_i(e) \;:\; T(c_i)} \;\text{Known sum elimination}$$

Exps $e ::=$	x	Term variables
$\mid$	n	Integers
$\mid$	f	Floating point numbers
$\mid$	$\mathbf{boxfloat}(e)$	Float boxing
$\mid$	$\mathbf{unboxfloat}(e)$	Float unboxing
$\mid$	$\mathbf{array}_c(e, e)$	Polymorphic array
$\mid$	$\mathbf{sub}[c](e, e)$	Polymorphic subscript
$\mid$	$\mathbf{fsub}(e, e)$	Float subscript
$\mid$	$\langle e, e\rangle$	Polymorphic pairing
$\mid$	$\pi_i[c]\,e$	Polymorphic selection
$\mid$	$\mathbf{rec}\, f = \lambda(\alpha{::}\kappa, x : \tau) : \tau.e$	Recursive function abstraction
$\mid$	$e[c]e$	Application
$\mid$	$\mathbf{inl}_{c,c}e$	Sum injection left
$\mid$	$\mathbf{inr}_{c,c}e$	Sum injection right
$\mid$	$\mathbf{case}_\tau\, e\, \mathbf{of}\, \{\mathbf{inl}(x) \Rightarrow e, \mathbf{inr}(x) \Rightarrow e\}$	Sum case
$\mid$	$\mathbf{proj}_i(e)$	Known sum projection
$\mid$	$\mathbf{roll}_c(e)$	Recursive type introduction
$\mid$	$\mathbf{unroll}(e)$	Recursive type elimination
$\mid$	$\mathbf{let}\, x = e\, \mathbf{in}\, e\, \mathbf{end}$	Expression binding
$\mid$	$\mathbf{let}\, \alpha = c\, \mathbf{in}\, e\, \mathbf{end}$	Constructor binding

Figure 4: MIL expressions

2.3.1 Type analysis

A key optimization that the original TIL compiler implemented was the use of non-uniform data representation. Many implementations of languages with polymorphism require that all values fit into a word. In particular, array elements must always be word-sized, which means that arrays of 64 bit floats (for example) must actually be arrays of pointers to floats. This is highly undesirable, both because of the extra pointer indirections implicit in each lookup and because of the consequent loss of data locality.

TIL pioneered the use of intensional polymorphism to avoid this overhead. By passing types at runtime and allowing code to dispatch on them, unboxed floating point arrays could be used with the appropriate subscript stride chosen at runtime. Different pieces of code could be run based on the runtime type of polymorphic data.

The MIL calculus differs from the λ_i^{ml}calculus of [HM95] in that it does not contain an explicit type analysis construct such as typerec or typecase. This does not mean however that the idea of intensional type analysis has been abandoned: rather, the type analysis has been hidden inside the primitives which need to use it. For example the constructor argument to the polymorphic subscript operator $\mathbf{sub}[c](e, e)$ is actually used at runtime to determine the appropriate stride. This polymorphic subscript in the language without a typecase can be thought of as a derived form in an underlying language with typecase: that is, subscript is a polymorphic function which internally uses typecase to choose the appropriate monomorphic subscript operator.

2.3.2 Floating point numbers

TILT deals with floating point numbers by using two different types, *Boxedfloat* and *Float* corresponding to the types of boxed and unboxed floats, with appropriate term level coercions between them. This allows the optimizer to deal directly with data representation optimizations, even at the relatively high level of the MIL. To prevent unboxed floats from being passed to polymorphic functions or to polymorphic primitives (such as pair injections and projections), the *Float* type is restricted to the type level. The predicativity restriction therefore enforces the uniform representation of polymorphic arguments. In non-polymorphic argument positions on the other hand, the compiler is free to use the unboxed floating point type. This is more efficient because it avoids repeatedly boxing and unboxing arguments, and also since it allows floating point arguments to be passed in floating point registers.

One obvious problem with this is that the type of arrays of unboxed floats cannot be constructed in this system, since the argument to the array constructor must be a constructor (not a type). This would seem to mean that we are unable to implement flattened float arrays. However, by using type analysis in the array constructor as well as the subscript operator, we can avoid at least some of the difficulty. There is nothing that prevents the *Boxedfloat array* type from being implemented using unboxed floats, even though the *Boxedfloat* type itself may be boxed.

The downside of this is that the subscript operation will therefore actually have to do a runtime typecase in order to determine the stride of an array of unknown types. Moreover, even when the type is known, the subscript operation will be forced to rebox the float before returning it, since subscripting into an array of boxed floats returns a value of type *Boxedfloat*. To avoid this problem, we provide a specialized floating point subscript $\mathbf{fsub}(e, i)$ which is well typed only when its argument is a *Boxedfloat array*, but which returns a value of type *Float*. This primitive avoids the problems with using the standard polymorphic subscript in cases where the element type is statically known to be *Boxedfloat*, since it need not dispatch on its constructor argument, and since it does not need to rebox its return value.

2.4 Algorithmic typechecking

In addition to using types for runtime optimization, TILT was also designed with the idea that the type annotations can provide a degree of self-checking within the compiler: just as a programmer profits from the degree of error checking imposed by the typechecker, so should a compiler. With this in mind, a good deal of work went into designing efficient algorithms for typechecking the MIL.

Modulo the constructor equivalence algorithm which is treated separately in [SH99], the complete typechecking algorithm for the MIL is presented in appendix A.2. The algorithm is presented as an alternative set of typing rules which are intended to express the structure of the algorithm: in the few cases where more than one rule might apply the result of a single common premise indicates which rule is applicable. The algorithmic judgements are listed in figure 5. The most noticeable presentational change is that the constructor and term well-formedness rules have been split into synthesis and analysis rules. For the term level, the intension is that the synthesis algorithm corresponds to synthesizing a type for a term: given a well-typed term, the algorithm will return its type. In the case of the analysis algorithm the type is an additional argument: the algorithm checks that the term argument is well formed at that type. The constructor level algorithms work in the same manner, with the additional constraint that the kind returned by the kind synthesis algorithm is principal.

$\Delta \models \kappa$	Well-formed kinds.
$\Delta \models \kappa_1 \preceq \kappa_2$	Subkinding.
$\Delta \models c \Downarrow \kappa$	Kind analysis
$\Delta \models c \Uparrow \kappa$	Kind synthesis
$\Delta \vdash c_1 \equiv c_2 :: \kappa$	Constructor equivalence.
$\Delta \models \tau$	Well formed type
$\Delta \models e \Downarrow \tau$	Type analysis
$\Delta \models e \Uparrow \tau$	Type synthesis
$\Delta \models c \mapsto c'$	Constructor weak head normal form

Figure 5: MIL algorithmic judgements

2.4.1 Selfification

Unlike the declarative system, the algorithmic MIL has no extensionality rules and no explicit self rule. Instead, the base-cases for the kind synthesis algorithm include implicit applications of the self-rule. For the most part this is very straightforward: for example, the rule for the *Int* constructor.

$$\frac{}{\Delta \models Int \Uparrow S_T(Int)} \;\text{Int}$$

In the variable rule however, it is not necessarily possible to apply the self rule directly, since the variable may be bound at a higher kind. For variables, it is necessary to inline implicit applications of the extensionality rules as well. This is done in the form of an auxilliary judgement called selfification: $\models c :: \kappa \doteq \kappa'$.

$$\frac{\models \alpha :: \kappa \doteq \kappa'}{\Delta[\alpha :: \kappa] \models \alpha \Uparrow \kappa'} \;\text{Variable}$$

Selfification takes a constructor and a kind and replaces the abstract components of the kind with singletons containing projections from and applications of the constructor. So for example, $\models \alpha :: \Sigma(\beta :: T).T \doteq \Sigma(\beta :: S_T(\pi_1\, \alpha)).S_T(\pi_2\, \alpha)$. The resulting kind is therefore principal for the variable in question.

It is interesting to note here that there are some apparently arbitrary choices to be made in the manner in which selfification is done that are nonetheless significant from an implementation standpoint. In particular, the singleton rule could be implemented in either of two ways.

$$\frac{}{\models c :: S_T(d) \doteq S_T(c)} \;\text{Singleton 1}$$

$$\frac{}{\models c :: S_T(d) \doteq S_T(d)} \;\text{Singleton 2}$$

From a theoretical standpoint, either choice gives a correct and equivalent kind. From an implementation standpoint however, the first choice which replaces the contents of singletons tends to yield smaller kinds. The reason for this is straightforward: since selfification always starts with a variable as the constructor argument, the new singletons created via selfification with the replacement strategy always contain only paths which are relatively quite small. In practice, the pre-existing contents of the singletons are often quite large, and are almost never smaller than a projection from a variable.

The rule for the dependent pair kind presents a related choice. It is equally possible to retain or eliminate occurrences of the dependent variable in the second kind, since the constructor gives us a definition for this variable.

$$\frac{\models c.1 :: \kappa_1 \doteq \kappa_1' \quad \models c.2 :: \{c.1/\alpha\}\kappa_2 \doteq \kappa_2'}{\models c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \doteq \Sigma(\alpha :: \kappa_1').\kappa_2'}\ \text{Sigma}$$

By choosing to substitute for the free occurrences of the variable, we ensure that selfification never generates dependent pair kinds. This property extends naturally through the rest of the kind synthesis algorithm: it is possible never to generate dependent pair kinds as the result of kind synthesis. This means that the constructor projection rule

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models \pi_2\, c \Uparrow \{\pi_1\, c/\alpha\}\kappa_2}\ \text{Second projection}$$

need not perform substitution. Eliminating this substitution yields significant efficiency gains. This can be further improved by noticing that a side effect of using the replacement strategy for the singleton case is that the only place that the dependent variable can occur is in the argument decoration of function kinds. Therefore, the notion of substitution can be specialized further to avoid the unnecessary traversal of the rest of the kind.

2.5 Termination Proofs

In this section, we show the decidability of the typechecking algorithm for the MIL calculus modulo constructor equivalence. The decidability of the constructor equivalence algorithm is proved separately for the $\lambda_{\leq}^{\Pi\Sigma S}$ calculus in [SH99]. This result extends trivially to the full MIL language. Note that the decidability of the formal system corresponds to termination of the algorithm.

In section 2.5.1 the proof of the decidability of sub-kinding is given, followed in section 2.5.2 by the proof of decidability of the well-formed kind, kind analysis, and kind synthesis judgements. All of the proofs follow essentially the same form:

1. Define a size metric mapping kinds and constructors into the natural numbers (basically textual size)
2. Extend the metric to derivations
3. Show that the judgements only permit derivations which only use smaller sub-derivations as hypotheses.
4. Observe that an infinite derivation contradicts the well-foundedness of the natural numbers

2.5.1 Termination of sub-kinding.

Consider the relation $\prec$ on sub-kinding derivations J defined as follows: $J_1 \prec J_2$ iff J_1 is an immediate sub-derivation of J_2. It suffices to show that the $\prec$ relation is well-founded, since if there are no infinite descending chains in the relation, then clearly there are no infinite derivations (notice that all rules have a finite number of hypotheses). To show that this is the case, we exhibit a mapping SZ which maps derivations to natural numbers, and show that this map is order preserving. For notational simplicity, we write $SZ(\Delta \models \kappa_1 \preceq \kappa_2)$ for $SZ(J)$ where J is a derivation the conclusion of which is $\Delta \models \kappa_1 \preceq \kappa_2$.

Definition 1
$SZ(\Delta \models \kappa_1 \preceq \kappa_2) = sz(\kappa_1) + sz(\kappa_2)$, where

$$sz(\kappa) = \begin{cases} 1 & \text{if } \kappa = T \\ 1 & \text{if } \kappa = S_T(c) \\ sz(\kappa_1) + sz(\kappa_2) & \text{if } \kappa = \Sigma(\alpha :: \kappa_1).\kappa_2 \\ sz(\kappa_1) + sz(\kappa_2) & \text{if } \kappa = \Pi(\alpha :: \kappa_1).\kappa_2 \end{cases}$$

It is fairly easy to see that SZ is a function (lemma 1). This establishes that SZ serves as a metric mapping derivations into the natural numbers. A less obvious result is that SZ preserves the ordering $\prec$ - that is, that the immediate sub-derivations are always smaller according to the metric SZ (lemma 2). Given this lemma, the main result (theorem 1) follows almost immediately.

Lemma 1
SZ is a function.

Proof. It is easy to see that $\forall\kappa\exists! n \ s.t. \ sz(\kappa) = n$ by induction over the structure of κ. The lemma follows immediately. ∎

Lemma 2
SZ is order preserving. That is,

$$J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$$

Proof. The proof proceeds by cases on the last rule used in J_2. See appendix A.3.1 for details. ∎

Theorem 1
The algorithm for checking subkinding always terminates. That is, the algorithmic rules for subkinding do not permit any infinite sequences of rule applications.

Proof. By the previous lemmas, every derivation has as immediate hypotheses only sub-derivations that are strictly smaller according to a well-founded ordering. Therefore, there can be no derivations of infinite depth, since such a derivation would correspond to an infinite descending chain in the well-founded ordering. ∎

2.5.2 Termination of the well-formed kind, kind analysis, and kind synthesis algorithms

The proof of decidability of the well-formed kind, kind analysis and kind synthesis algorithms proceeds in much the same fashion as above. The only significant difference is that the measure function for derivations maps into lexicographically ordered pairs of natural numbers. This arises because of the form of the kind analysis judgement, and is mostly a technicality: it is easy to see that all uses of the single kind analysis rule could be inlined into the other judgements allowing the proof to proceed as before.

We start by defining measure functions which map derivations to pairs of natural numbers ordered lexicographically below. These functions are defined as before in terms of inductively defined functions $sz_\kappa()$ and $sz_c()$, which act as measures on kinds and constructors, respectively.

Definition 2

$$sz_\kappa(\kappa) = \begin{cases} 1 & \text{if } \kappa = T \\ sz_c(c)+1 & \text{if } \kappa = S_T(c) \\ sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2) & \text{if } \kappa = \Sigma(\alpha :: \kappa_1).\kappa_2 \\ sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2) & \text{if } \kappa = \Pi(\alpha :: \kappa_1).\kappa_2 \end{cases}$$

$$sz_c(c) = \begin{cases} 1 & \text{if } c = \alpha, Int, \mathit{Boxedfloat} \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \mu(\alpha,\beta).(c_1,c_2) \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1 \times c_2, c_1 \to c_2, c_1 + c_2 \\ sz_c(c')+1 & \text{if } c = c' \mathit{\ array} \\ sz_c(c') + sz_\kappa(\kappa) & \text{if } c = \lambda\alpha{::}\kappa.c' \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1\, c_2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c =< c_1, c_2 > \\ sz_c(c')+1 & \text{if } c = c'.1, c'.2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \textbf{let } \alpha = c_1 \textbf{ in } c_2 \textbf{ end} \end{cases}$$

$$SZ(J) = \begin{cases} (sz_\kappa(\kappa), 0) & \text{if the conclusion is } \Delta \models \kappa \\ (sz_c(c), 1) & \text{if the conclusion is } \Delta \models c \Downarrow \kappa \\ (sz_c(c), 0) & \text{if the conclusion is } \Delta \models c \Uparrow \kappa \end{cases}$$

The proof then proceeds almost exactly as in the sub-kinding case, except that there is an additional lemma observing that the selfification judgement used by the kind synthesis algorithm is also decidable.

Lemma 3
SZ is a function.

Proof. It suffices to show that $sz_c()$, and $sz_\kappa()$ are well-defined. This follows by induction over the structure of κ and c. ■

Lemma 4
The selfification judgement $\models c :: \kappa_1 \doteq \kappa_2$ is decidable.

Proof. Follows by induction over the structure of κ. ■

Lemma 5
SZ is order preserving. That is,

$$J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$$

where $<$ is the lexicographic ordering on $N \times N$.

Proof. The proof proceeds by cases on the last rule used in J_2. See appendix A.3.2 for details. ■

Theorem 2
The kind synthesis, kind analysis, and kind well-formedness judgements are decidable.

Proof. By lemma 5, any infinite sequence of rule applications corresponds to an infinite descending chain of pairs of natural numbers ordered lexicographically, which contradicts the well-foundedness of $(N \times N, <)$. ■

2.6 Efficiency concerns with the MIL

In the previous sections we define a language sufficiently expressive for our purposes and give algorithms for checking the well-formedness of terms in this language. This language is very close to the original MIL calculus that was first used in the TILT implementation. While sufficient from a theoretical perspective, this turns out to suffer from some practical deficiencies.

An early challenge in the TILT implementation was to keep the size of the compiler intermediate forms manageably small. In some cases relatively small programs increased in size dramatically when translated into the MIL, and larger programs became simply unmanageable. Surprisingly, measurements suggested that a good deal of the program size was due to kinds.

One of the major reasons for this becomes apparent upon closer inspection of the MIL typing rules. Because singleton kinds are restricted to contain only constructors of kind T, constructors of higher kind end up being duplicated in their principal kinds. For example, if c is a large constructor of kind $T \times T$ then principal kind of c is $S_T(\pi_1\, c) \times S_T(\pi_2\, c)$: the kind is more than twice as large as the constructor it classifies. The duplication of constructors in kinds is quite pernicious: since structures and functors turn into constructor records and functions, kinds may contain many copies of entire structures. This becomes especially bad in the case of nested structures, a common ML programming idiom.

2.6.1 Singletons at higher kinds

$$\begin{array}{lcl}
S(c{::}T) & := & S_T(c) \\
S(c{::}S_T(c')) & := & S_T(c) \\
S(c{::}\Pi(\alpha :: \kappa_1).\kappa_2) & := & \Pi(\alpha :: \kappa_1).S(c\,\alpha{::}\kappa_2) \\
S(c{::}\Sigma(\alpha :: \kappa_1).\kappa_2) & := & \Sigma(\alpha :: S(\pi_1\, c{::}\kappa_1)).S(\pi_2\, c{::}\kappa_2)
\end{array}$$

Figure 6: Definability of singletons at higher kind

An obvious solution to the constructor duplication problem is to permit the use of singletons at higher kind. This is not at all difficult so long as the singletons are labeled with the kind of their contents: in fact, as figure 6 shows, this is definable in the original calculus. This allows for kinds of the form $S_T(\pi_1\, c) \times S_T(\pi_2\, c)$ to be replaced with an equivalent kind of the form $S(c{::}T \times T)$, which contains only one copy of the classified constructor.

In practice however, this solution is not sufficient: kinds still account for too much of the space used by the intermediate forms. In this system, the decorations on the singletons themselves now occupy a significant amount of the space saved - the kinds used are generally smaller, but there are more of them. Moreover, it is hard to systematically avoid the creation of kinds of the form $S(c{::}S(c{::}T))$: a perfectly legitimate kind, but not desirable from an efficiency standpoint.

As a result of these observations, it became clear that what was needed was a system containing unlabelled singletons at higher kind: $S(c)$ instead of $S(c{::}\kappa)$. In such a system, the principal kind of a constructor c is always $S(c)$. This kind is both small, and fast to synthesize, but does not provide any useful structural information. An attempt to use this kind (for example, to determine if a projection from a variable of this kind is well-formed) requires additional work. The system with unlabelled singletons introduces a significant measure of type reconstruction into the language in addition to that already introduced by the decision to elide classifiers on let bindings. (In fact, if we view the binding **let** $\alpha :: \kappa = c_1$ **in** c_2 **end** as syntactic sugar for $\lambda\alpha{::}S(c_1{::}\kappa).c_2$ [SH99], then it becomes clear that eliding the classifier on let bindings is merely a special usage of unlabelled singletons: i.e. $\mathbf{let}\,\alpha = c_1\,\mathbf{in}\,c_2\,\mathbf{end}$ corresponds to $\lambda\alpha{::}S(c_1).c_2$.)

Because of this additional burden of type reconstruction, it is not immediately clear that the language with unlabelled singletons is decidable: unlike labelled singletons, there is no simple inductive definition that tells what the corresponding simple singleton kind is. The next section defines a language with unlabelled singletons, presents an algorithm for typechecking, and proves its decidability.

3 NIL (Extended MIL)

The relatively simple core calculus described above is sufficient from the standpoint of serving as a target language for the elaboration phase. However, from the standpoint of efficient implementation, it is somewhat deficient. This section describes the extension of the MIL language to permit unlabelled singletons at higher kinds. For clarity, we use the term NIL to describe this extended calculus.

3.1 Syntax

$$
\begin{array}{rcl}
\underline{k} & ::= & S(c) \mid T \mid S_T(c) \mid \Sigma(\alpha :: \underline{k}).\underline{k} \mid \Pi(\alpha :: \underline{k}).\underline{k} \\
\\
c & ::= & \ldots \mid \lambda\alpha{::}\underline{k}.c \\
t & ::= & T(c) \mid (\alpha :: \underline{k}, x : t) \to t \mid \mathit{Float} \\
 & & \mid t \times t \mid \mathbf{let}\,\alpha = c\,\mathbf{in}\,t\,\mathbf{end} \\
\\
p & ::= & \alpha \mid p.1 \mid p.2 \mid p\,c \\
\\
e & ::= & x \mid \mathbf{let}\,x = e\,\mathbf{in}\,e\,\mathbf{end} \mid \mathbf{let}\,\alpha = c\,\mathbf{in}\,e\,\mathbf{end} \\
 & & \mid \mathbf{rec}\,f = \lambda(\alpha{::}\underline{k}, x : \tau) : \tau.e \\
 & & \mid e[c]e \mid < e, e > \mid e.1 \mid e.2 \mid n \mid r \mid \mathbf{boxfloat}(e) \mid \mathbf{unboxfloat}(e) \\
 & & \mid \mathbf{inl}_{c,c}e \mid \mathbf{inr}_{c,c}e \mid \mathbf{case}_{\tau}\,e\,\mathbf{of}\,\{\mathbf{inl}(x) \Rightarrow e, \mathbf{inr}(x) \Rightarrow e\} \\
 & & \mid \mathbf{roll}_c(e) \mid \mathbf{unroll}(e) \mid \mathbf{proj}_i(e) \\
 & & \mid \mathbf{array}_c(e, e) \mid \mathbf{sub}[c](e, e) \mid \mathbf{fsub}(e, e) \\
\\
\Delta & ::= & \bullet \mid \Delta[x : \tau] \mid \Delta[\alpha{::}\kappa]
\end{array}
$$

Figure 7: NIL Syntax

The syntax for the NIL language is given in figure 7: the only change from the MIL is the addition of the unlabelled singleton, $S(c)$. For the sake of clarity, we write kinds in this extended calculus as $\underline{k}$ instead of κ, which we reserve for the core calculus.

There are two points of importance to the extended system that are already evident in the syntax. The first is that the addition of unlabelled singletons does not *replace* the core singleton at kind T: the original singleton form is still present in syntax. The second point is that typing contexts are restricted to contain kinds κ from the core calculus only: there are no unlabelled singletons allowed in the context. These two facts are the key to making the algorithm terminate.

3.2 Algorithmic judgments

New judgements

$\Delta \models \underline{k}\backslash\kappa$	Kind standardization
$\Delta \models c\backslash c'$	Constructor standardization

New versions of old judgements

$\Delta \models \underline{k}$	Well-formed kinds.
$\Delta \models c \Downarrow \kappa$	Kind analysis
$\Delta \models c \Uparrow \kappa$	Principal kind synthesis
$\Delta \models \tau$	Well formed type
$\Delta \models e \Downarrow \tau$	Type analysis
$\Delta \models e \Uparrow \tau$	Type synthesis

Unchanged

Δ ok	Well-formed context
$\Delta \models \kappa_1 \preceq \kappa_2$	Subkinding.
$\Delta \vdash c_1 \equiv c_2 :: \kappa$	Constructor equivalence.

Figure 8: Nil declarative judgements

The judgements used to define the NIL typechecking algorithm are listed in figure 8, and are described in full in appendix B.1. The major change is the addition of two new judgements: kind standardization and constructor standardization. We call a kind *standard* if it contains no occurrences of unlabelled singletons. A constructor is standard if it contains only standard kinds. Notice that every standard kind is a MIL kind. These new judgements implement the process of putting a kind or constructor into standard form.

The kind standardization algorithm traverses compositionally over the structure of kinds until it reaches a singleton type. In the case that the singleton is not standard it is necessary to reconstruct the principal standard kind by calling the kind synthesis algorithm on the constructor.

$$\frac{\Delta \models c \Uparrow \kappa}{\Delta \models S(c)\backslash\kappa} \quad \text{Singleton Any}$$

If the singleton is already standard, all that remains to be done is to standardize the constructor.

$$\frac{\Delta \models c\backslash c'}{\Delta \models S_T(c)\backslash S_T(c')} \quad \text{Singleton Type}$$

The labelled singleton is important here: it provides a way of marking singletons which do not require further type reconstruction efforts.

Notice that the kind synthesis algorithm is designed to synthesize standard kinds for non-standard constructors. This mixing of the two systems is important for a number of reasons, but here we see how it comes into play during kind standardization: if kind synthesis returned non-standard kinds we would not have made progress here. This intertwining of the two systems is essential to the algorithm.

The constructor standardization algorithm is straightforward: it simply traverses the constructor, standardizing any kinds that it finds.

$$\frac{\Delta \models \underline{k}\backslash\kappa \quad \Delta[\alpha{::}\kappa] \models c\backslash c'}{\Delta \models \lambda\alpha{::}\underline{k}.c\backslash\lambda\alpha{::}\kappa.c'}\ \text{Lambda}$$

It is also possible to generalize the system slightly by using an intermediate form wherein non-standard constructors are allowed inside of standard singletons so that constructor standardization is no longer necessary. This is a relatively straightforward extension, and for the sake of brevity we do not elaborate on it here.

The kind synthesis algorithm now proceeds much as before, but with additional calls to the kind standardization algorithm where necessary to preserve the property that all kinds in the context are standard.

$$\frac{\begin{array}{ll}\Delta \models \underline{k} & \Delta \models \underline{k}\backslash\kappa \\ \Delta[\alpha{::}\kappa] \models c \Uparrow \kappa' & \alpha \notin \mathrm{Dom}(\Delta)\end{array}}{\Delta \models \lambda\alpha{::}\underline{k}.c \Uparrow \Pi(\alpha :: \kappa).\kappa'}\ \text{Lambda}$$

In the variable rule we can see the importance of this property.

$$\frac{\models \alpha :: \kappa \doteq \kappa'}{\Delta[\alpha{::}\kappa] \models \alpha \Uparrow \kappa'}\ \text{Variable}$$

Because the contents of the context are already standard, it is not necessary to call back to the kind-standardization algorithm here. Much as with labelled singletons in the kind standardization algorithm, this gives the algorithm a place to stop.

The fact that the kind synthesis algorithm returns standard kinds is also important internally to the algorithm in cases where it must inspect kinds. In the projection rule, the fact that the kind returned is standard means that the only possible form for the kind of the constructor is that of a pair, and hence no further work need be done to determine if the projection is well formed.

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models \pi_1\, c \Uparrow \kappa_1}\ \text{First projection}$$

The rest of the judgements change from the MIL only in minor ways: either additional cases to handle the new construct, or additional calls to kind standardization where needed. Interestingly, the subkinding and constructor equivalence algorithms carry over intact to the new system: it naturally falls out that the only calls to these algorithms are made with standardized arguments.

3.3 Soundness and Completeness

It is important for the purposes of the compiler that the extended system be complete with respect to the core system: that is, that all programs which could be typechecked in the core system can still be typechecked in the extended system. This property holds, as stated in theorem 3. The proof of this theorem follows almost trivially, since the NIL is a syntactic superset of the MIL and since the well-formedness judgements of the NIL closely parallel those of the MIL. For clarity in the statement of the theorems, we write the NIL well-formedness judgements with a superscripted turnstyle, as such: $\overset{+}{\models}$.

Theorem 3 (Completeness)
The extended system is complete with respect to the core system.

1. *if* $\Delta\,\mathrm{ok}$ *and* $\Delta \models \kappa$, *then* $\Delta \overset{+}{\models} \kappa$.

2. *if* $\Delta\,\mathrm{ok}$ *and* $\Delta \models c \Uparrow \kappa$, *then* $\Delta \overset{+}{\models} c \Uparrow \kappa$.

Proof. First observe that every MIL kind is a syntactically valid standard NIL kind. Then observe that the kind standardization algorithm is the identity on standard kinds. The proof then follows easily by induction over the structure of typing derivations. ∎

While completeness is the most important property, it is desirable that the system be sound with respect to the core system as well: that is, that it does not allow us to typecheck more programs than before. Theorem 4 states this property. The proof of this theorem is less obvious, but not significantly more difficult.

Theorem 4 (Soundness)
The extended system is sound with respect to the core system.

1. *if* $\Delta\,\mathrm{ok}$ *and* $\Delta \overset{+}{\models} \underline{k}$ *then there exists a* κ *such that* $\Delta \models \underline{k}\backslash\kappa$ *and* $\Delta \models \kappa$

2. *if* $\Delta\,\mathrm{ok}$ *and* $\Delta \overset{+}{\models} c \Uparrow \kappa$ *then there exists a* c' *such that* $\Delta \models c\backslash c'$ *and* $\Delta \models c' \Uparrow \kappa$

Proof. By induction over the structure of typing derivations ∎

These two basic theorems show that from a theoretical standpoint the NIL is a sensible extension of the MIL. The next section will show that in addition to being sound and complete with respect to the core system, the extended system is also decidable. This is the last and in some ways the most important property that the extended system must hold.

3.4 Termination Proofs

The proof of decidability of the extended system proceeds much as with the core system, defining measure functions which map derivations to pairs of natural numbers ordered lexicographically and using these to argue that the system is well-founded.

Definition 3

$$sz_{\underline{k}}(\underline{k}) = \begin{cases} 1 & \text{if } \underline{k} = T \\ sz_c(c) + 1 & \text{if } \underline{k} = S_T(c) \\ sz_c(c) + 1 & \text{if } \underline{k} = S(c) \\ sz_{\underline{k}}(\underline{k}_1) + sz_{\underline{k}}(\underline{k}_2) & \text{if } \underline{k} = \Sigma(\alpha :: \underline{k}_1).\underline{k}_2 \\ sz_{\underline{k}}(\underline{k}_1) + sz_{\underline{k}}(\underline{k}_2) & \text{if } \underline{k} = \Pi(\alpha :: \underline{k}_1).\underline{k}_2 \end{cases}$$

$$sz_c(c) \;=\; \begin{cases} 1 & \text{if } c = \alpha, Int, \mathit{Boxedfloat} \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \mu(\alpha,\beta).(c_1,c_2) \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1 \times c_2, c_1 \to c_2, c_1 + c_2 \\ sz_c(c') + 1 & \text{if } c = c' \; \mathit{array} \\ sz_c(c') + sz_{\underline{k}}(\underline{k}) & \text{if } c = \lambda\alpha{::}\underline{k}.c' \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = c_1\, c_2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c =< c_1, c_2 > \\ sz_c(c') + 1 & \text{if } c = c'.1, c'.2 \\ sz_c(c_1) + sz_c(c_2) & \text{if } c = \textbf{let}\ \alpha = c_1\ \textbf{in}\ c_2\ \textbf{end} \end{cases}$$

$$SZ(J) \;=\; \begin{cases} (sz_{\underline{k}}(\underline{k}), 0) & \text{if the conclusion is } \Delta \models \underline{k}\backslash\kappa \\ (sz_c(c), 0) & \text{if the conclusion is } \Delta \models c\backslash c' \\ (sz_{\underline{k}}(\underline{k}), 0) & \text{if the conclusion is } \Delta \models \underline{k} \\ (sz_c(c), 1) & \text{if the conclusion is } \Delta \models c \Downarrow \kappa \\ (sz_c(c), 0) & \text{if the conclusion is } \Delta \models c \Uparrow \kappa \end{cases}$$

As before, we argue that the measure is a well-defined function. Note that the selfification result of lemma 4 still holds, since selfification is only performed on standard kinds.

Lemma 6
SZ is a function.

Proof. It suffices to show that $sz_c()$, and $sz_{\underline{k}}()$ are well-defined. This follows by induction over the structure of $\underline{k}$ and c. ∎

Lemma 7
SZ is order preserving. That is,

$$J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$$

where $<$ is the lexicographic ordering on $N \times N$.

Proof. The proof proceeds by cases on the last rule used in J_2. See appendix B.2.1 for details. ∎

The main result then follows easily as before.

Theorem 5
The kind standardization, constructor standardization, kind synthesis, kind analysis, and kind well-formedness judgements are decidable.

Proof. By lemma 7, any infinite sequence of rule applications corresponds to an infinite descending chain of pairs of natural numbers ordered lexicographically, which contradicts the well-foundedness of $(N \times N, <)$. ∎

4 Conclusion

This paper presents a language very close to that actually used in the internals of the TILT compiler: a language whose design was driven not by the usual concerns of programer usability, but by the new concern of compiler usability. This difference in purpose leads to very different concerns than those normally encountered by language designers. We have discussed here some of the more important design decisions resulting from this in the original core calculus, and we have also described the extension of the calculus to allow unlabelled singletons for the purpose of providing compact representations of internal forms. This extension has been shown sound and complete, and decidable.

The work described here was a key part of making the TILT compiler run efficiently, and well. It is of particular interest because it presents a theoretical approach to solving a practical problem. This is indicative of the overall design philosophy of the TILT project: that a systematic and theoretically sound approach to practical problems provides significant engineering benefits. The use of a new language construct (unlabelled singletons) to achieve an engineering goal (better space efficiency) is an excellent example of how this can work.

References

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[HS96] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1996. (Supersedes [SH96]. Also published as Fox Memorandum CMU–CS–FOX–96–02R.).

[HS97] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997. (Supersedes [HS96] and [SH96]. Also published as Fox Memorandum CMU–CS–FOX–97–01.).

[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.

[SH96] Chris Stone and Robert Harper. A type-theoretic account of Standard ML 1996 (version 1). Technical Report CMU-CS-96-136, School of Computer Science, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, May 1996. (Also published as Fox Memorandum CMU-CS-FOX-96-02).

[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.

[TMC^{+}96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference* *on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

A MIL

A.1 Declarative judgements

Well Formed Context $\boxed{\Delta \text{ ok}}$

$$\frac{}{\bullet \text{ ok}}\ \text{Empty}$$

$$\frac{\Delta \text{ ok} \quad \Delta \vdash \kappa \quad \alpha \notin \text{Dom}(\Delta)}{\Delta[\alpha{::}\kappa] \text{ ok}}\ \text{Kind}$$

$$\frac{\Delta \text{ ok} \quad \Delta \vdash \tau \quad x \notin \text{Dom}(\Delta)}{\Delta[x : \tau] \text{ ok}}\ \text{Type}$$

Well Formed Kind $\boxed{\Delta \vdash \kappa}$

$$\frac{\Delta \text{ ok}}{\Delta \vdash T}\ \text{Type}$$

$$\frac{\Delta \vdash c :: T}{\Delta \vdash S_T(c)}\ \text{Singleton}$$

$$\frac{\Delta \vdash \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \vdash \kappa_2}{\Delta \vdash \Pi(\alpha :: \kappa_1).\kappa_2}\ \text{Pi}$$

$$\frac{\Delta \vdash \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \vdash \kappa_2}{\Delta \vdash \Sigma(\alpha :: \kappa_1).\kappa_2}\ \text{Sigma}$$

Sub-Kinding $\boxed{\Delta \vdash \kappa_1 \preceq \kappa_2}$

$$\frac{\Delta \text{ ok}}{\Delta \vdash T \preceq T}\ \text{Type}$$

$$\frac{\Delta \vdash S_T(c)}{\Delta \vdash S_T(c) \preceq T}\ \text{SingletonL}$$

$$\frac{\Delta \vdash c \equiv d :: T}{\Delta \vdash S_T(c) \preceq S_T(d)}\ \text{Singletons}$$

$$\frac{\Delta \vdash \kappa_1' \preceq \kappa_1 \quad \Delta[\alpha{::}\kappa_1'] \vdash \kappa_2 \preceq \kappa_2'}{\Delta \vdash \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'}\ \text{Pi}$$

$$\frac{\Delta \vdash \kappa_1 \preceq \kappa_1' \quad \Delta[\alpha{::}\kappa_1] \vdash \kappa_2 \preceq \kappa_2'}{\Delta \vdash \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'}\ \text{Sigma}$$

Well formed constructor $\boxed{\Delta \vdash c :: \kappa}$

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

$$\frac{\Delta\ \text{ok}}{\Delta \vdash \mathit{BoxFloat} :: T}\ \text{BoxFloat}$$

$$\frac{\Delta\ \text{ok}}{\Delta \vdash \mathit{Int} :: T}\ \text{Int}$$

$$\frac{\Delta[\alpha{::}T][\beta{::}T] \vdash c_1 :: T \quad \Delta[\alpha{::}T][\beta{::}T] \vdash c_2 :: T}{\Delta \vdash \mu(\alpha, \beta).(c_1, c_2) :: T}\ \text{Mu}$$

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 \times c_2 :: T}\ \text{Pair}$$

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 \rightarrow c_2 :: T}\ \text{Arrow}$$

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 + c_2 :: T}\ \text{Sum}$$

$$\frac{\Delta \vdash c_1 :: T \quad \Delta \vdash c_2 :: T}{\Delta \vdash c_1 +^i c_2 :: T}\ \text{KnownSum}$$

$$\frac{\Delta \vdash c :: T}{\Delta \vdash c\ \mathit{array} :: T}\ \text{Array}$$

$$\frac{\Delta \vdash \kappa \quad \Delta[\alpha{::}\kappa] \vdash c :: \kappa'}{\Delta \vdash \lambda(\alpha :: \kappa).c :: \Pi(\alpha :: \kappa).\kappa'}\ \text{Lambda}$$

$$\frac{\Delta \vdash c_1 :: \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c_2 :: \kappa_1}{\Delta \vdash c_1\, c_2 :: \{c_2/\alpha\}\kappa_2}\ \text{App}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle :: \kappa_1 \times \kappa_2}\ \text{Record}$$

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \vdash c.1 :: \kappa_1}\ \text{Proj1}$$

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \vdash c.2 :: \{c.1/\alpha\}\kappa_2}\ \text{Proj2}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \vdash c_2 :: \kappa_2}{\Delta \vdash \mathbf{let}\ \alpha = c_1\ \mathbf{in}\ c_2\ \mathbf{end} :: \{c_1/\alpha\}\kappa_2}\ \text{Let}$$

$$\frac{\Delta \vdash c :: T}{\Delta \vdash c :: S_T(c)}\ \text{Selfify}$$

$$\frac{\Delta \vdash c :: \kappa \quad \Delta \vdash \kappa \preceq \kappa'}{\Delta \vdash c :: \kappa'}\ \text{Subkind}$$

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c.1 :: \kappa_1'}{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1').\kappa_2}\ \text{Sigma Ext1}$$

$$\frac{\Delta \vdash c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \vdash c.2 :: \kappa_2'}{\Delta \vdash c :: \kappa_1 \times \kappa_2'}\ \text{Sigma Ext2}$$

$$\frac{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta[\alpha{::}\kappa_1] \vdash c\,\alpha :: \kappa_2'}{\Delta \vdash c :: \Pi(\alpha :: \kappa_1).\kappa_2'}\ \text{Pi Ext}$$

Well-formed Type

$\boxed{\Delta \vdash \tau}$

$$\frac{\Delta \vdash c :: T}{\Delta \vdash T(c)}\ \text{Constructor}$$

$$\frac{\Delta \vdash \kappa \quad \Delta[\alpha{::}\kappa] \vdash \tau_1 \quad \Delta[\alpha{::}\kappa] \vdash \tau_2}{\Delta \vdash (\alpha :: \kappa, \tau_1) \to \tau_2}\ \text{Arrow}$$

$$\frac{\Delta\ \text{ok}}{\Delta \vdash \mathit{Float}}\ \text{Float}$$

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2}\ \text{Float}$$

$$\frac{\Delta \vdash c :: \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \vdash \tau}{\Delta \vdash \mathbf{let}\ \alpha = c\ \mathbf{in}\ \tau\ \mathbf{end}}\ \text{Let}$$

Well-typed term

$\boxed{\Delta \vdash e\ :\ \tau}$

$$\frac{\Delta\ \text{ok}}{\Delta \vdash x\ :\ \Delta(x)}\ \text{Variable}$$

$$\frac{\Delta \vdash e_1\ :\ \tau_1 \quad \Delta[x : \tau_1] \vdash e_2\ :\ \tau_2}{\Delta \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}\ :\ \tau_2}\ \text{LetE}$$

$$\frac{\Delta \vdash c :: \kappa \quad \Delta[\alpha{::}\kappa] \vdash e\ :\ \tau}{\Delta \vdash \mathbf{let}\ \alpha = c\ \mathbf{in}\ e\ \mathbf{end}\ :\ \mathbf{let}\ \alpha = c\ \mathbf{in}\ \tau\ \mathbf{end}}\ \text{LetC}$$

$$\frac{\begin{array}{l}\Delta \vdash \kappa \quad \Delta[\alpha{::}\kappa] \vdash \tau_1 \qquad\qquad\qquad\qquad \Delta[\alpha{::}\kappa] \vdash \tau_2 \\ \Delta[f : (\alpha :: \kappa, \tau_1) \to \tau_2][\alpha{::}\kappa][x : \tau_1] \vdash e\ :\ \tau_2\end{array}}{\Delta \vdash \mathbf{rec}\ f = \lambda(\alpha{::}\kappa, x : \tau_1) : \tau_2.e\ :\ (\alpha :: \kappa, \tau_1) \to \tau_2}\ \text{Rec}$$

$$\frac{\begin{array}{l}\Delta \vdash e_1\ :\ (\alpha :: \kappa, \tau_1) \to \tau_2 \quad \Delta \vdash c :: \kappa \\ \Delta \vdash e_2\ :\ \{c/\alpha\}\tau_1\end{array}}{\Delta \vdash e_1[c]e_2\ :\ \{c/\alpha\}\tau_2}\ \text{App}$$

$$\frac{\Delta \vdash e_1\ :\ \tau_1 \quad \Delta \vdash e_2\ :\ \tau_2}{\Delta \vdash \langle e_1, e_2 \rangle\ :\ \tau_1 \times \tau_2}\ \text{Pair}$$

$$\frac{\Delta \vdash e \;:\; \tau_1 \times \tau_2}{\Delta \vdash e.1 \;:\; \tau_1}\;\text{Proj1}$$

$$\frac{\Delta \vdash e \;:\; \tau_1 \times \tau_2}{\Delta \vdash e.2 \;:\; \tau_2}\;\text{Proj2}$$

$$\frac{\Delta\, \mathrm{ok}}{\Delta \vdash r \;:\; \mathit{Float}}\;\text{Float}$$

$$\frac{\Delta\, \mathrm{ok}}{\Delta \vdash n \;:\; T(\mathit{Int})}\;\text{Int}$$

$$\frac{\Delta \vdash e \;:\; \mathit{Float}}{\Delta \vdash \mathbf{boxfloat}(e) \;:\; T(\mathit{BoxFloat})}\;\text{Box}$$

$$\frac{\Delta \vdash e \;:\; \mathit{BoxFloat}}{\Delta \vdash \mathbf{unboxfloat}(e) \;:\; \mathit{Float}}\;\text{Unbox}$$

$$\frac{\begin{array}{l}\Delta \vdash e \;:\; T(c_1 + c_2)\\ \Delta[x : T(c_1 +^1 c_2)] \vdash e_1 \;:\; \tau \quad \Delta[x : T(c_1 +^2 c_2)] \vdash e_2 \;:\; \tau\end{array}}{\Delta \vdash \mathbf{case}_\tau\, e\, \mathbf{of}\, \{\mathbf{inl}(x) \Rightarrow e_1, \mathbf{inr}(x) \Rightarrow e_2\} \;:\; \tau}\;\text{Sumswitch}$$

$$\frac{\begin{array}{l}\Delta \vdash c_1 :: T \qquad \Delta \vdash c_2 :: T\\ \Delta \vdash e \;:\; T(c_1)\end{array}}{\Delta \vdash \mathbf{inl}_{c_1,c_2} e \;:\; T(c_1 + c_2)}\;\text{inl}$$

$$\frac{\begin{array}{l}\Delta \vdash c_1 :: T \qquad \Delta \vdash c_2 :: T\\ \Delta \vdash e \;:\; T(c_2)\end{array}}{\Delta \vdash \mathbf{inr}_{c_1,c_2} e \;:\; T(c_1 + c_2)}\;\text{inr}$$

$$\frac{\begin{array}{l}\Delta \vdash c :: T \qquad\qquad \Delta \vdash c \equiv \mu(\alpha,\beta).(c_1,c_2).i :: T\\ \Delta \vdash e \;:\; T(\{c.1, c.2/\alpha,\beta\}c_i)\end{array}}{\Delta \vdash \mathbf{roll}_c(e) \;:\; T(c)}\;\text{roll}$$

$$\frac{\Delta \vdash e \;:\; \tau \quad \Delta \vdash \tau \equiv T(\mu(\alpha,\beta).(c_1,c_2).i)}{\Delta \vdash \mathbf{unroll}(e) \;:\; T(\{\mu(\alpha,\beta).(c_1,c_2).1, \mu(\alpha,\beta).(c_1,c_2).2/\alpha,\beta\}c_i)}\;\text{unroll}$$

$$\frac{\Delta \vdash e \;:\; \tau \quad \Delta \vdash \tau \equiv T(c_1 +^i c_2)}{\Delta \vdash \mathbf{proj}_i(e) \;:\; T(c_i)}\;\text{proj}$$

$$\frac{\begin{array}{ll}\Delta \vdash e_1 \;:\; Int & \Delta \vdash c :: T \\ \Delta \vdash e_2 \;:\; T(c) & \end{array}}{\Delta \vdash \mathbf{array}_c(e_1, e_2) \;:\; T(c\, array)}\ \text{array}$$

$$\frac{\Delta \vdash e_1 \;:\; T(c\, array) \quad \Delta \vdash e_2 \;:\; T(Int)}{\Delta \vdash \mathbf{sub}[e_1](e_2, \,) : T(c)}\ \text{sub}$$

$$\frac{\Delta \vdash e_1 \;:\; T(BoxFloat\; array) \quad \Delta \vdash e_2 \;:\; T(Int)}{\Delta \vdash \mathbf{fsub}(e_1, e_2) \;:\; Float}\ \text{fsub}$$

A.2 Algorithmic judgements

Well Formed Kind

$$\boxed{\Delta \models \kappa}$$

$$\frac{}{\Delta \models T}\ \text{Type}$$

$$\frac{\Delta \models c \Downarrow T}{\Delta \models S_T(c)}\ \text{Singleton}$$

$$\frac{\Delta \models \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \models \kappa_2}{\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2}\ \text{Pi}$$

$$\frac{\Delta \models \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \models \kappa_2}{\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2}\ \text{Sigma}$$

Sub-Kinding

$$\boxed{\Delta \models \kappa_1 \preceq \kappa_2}$$

Assume that Δ, κ_1 and κ_2 are well-formed. Check that κ_1 is a subkind of κ_2.

$$\frac{}{\Delta \models T \preceq T}\ \text{Type}$$

$$\frac{}{\Delta \models S_T(c) \preceq T}\ \text{Singleton}$$

$$\frac{\Delta \models c \equiv d :: T}{\Delta \models S_T(c) \preceq S_T(d)}\ \text{Singletons}$$

$$\frac{\Delta \models \kappa_1' \preceq \kappa_1 \quad \Delta[\alpha :: \kappa_1'] \models \kappa_2 \preceq \kappa_2' \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'}$$

$$\frac{\Delta \models \kappa_1 \preceq \kappa_1' \quad \Delta[\alpha :: \kappa_1] \models \kappa_2 \preceq \kappa_2' \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'}$$

Selfification $\boxed{\models c :: \kappa \doteq \kappa'}$

Assume c and κ are well-formed with respect to some context. Return the most precise kind of c. Intuitively, this is the definition of a singleton at the higher kind.

$$\frac{}{\models c :: T \doteq S_T(c)}\ \text{Type}$$

$$\frac{}{\models c :: S_T(d) \doteq S_T(c)}\ \text{Singleton}$$

$$\frac{\models c\,\alpha :: \kappa_2 \doteq \kappa_2'}{\models c :: \Pi(\alpha :: \kappa_1).\kappa_2 \doteq \Pi(\alpha :: \kappa_1).\kappa_2'}\ \text{Pi}$$

$$\frac{\models c.1 :: \kappa_1 \doteq \kappa_1' \quad \models c.2 :: \{c.1/\alpha\}\kappa_2 \doteq \kappa_2'}{\models c :: \Sigma(\alpha :: \kappa_1).\kappa_2 \doteq \Sigma(\alpha :: \kappa_1').\kappa_2'}\ \text{Sigma}$$

Kind Analysis $\boxed{\Delta \models c \Downarrow \kappa}$

Assume Δ and κ are well formed. Check that c is well formed and can be given kind κ.

$$\frac{\Delta \models c \Uparrow \kappa' \quad \Delta \models \kappa' \preceq \kappa}{\Delta \models c \Downarrow \kappa}\ \text{Analysis}$$

Kind Synthesis $\boxed{\Delta \models c \Uparrow \kappa}$

Assumes that Δ is well-formed. Check that c is well-kinded, and construct κ s.t. $\Delta \models \kappa$ and c has kind κ.

$$\frac{\models \alpha :: \kappa \doteq \kappa'}{\Delta[\alpha :: \kappa] \models \alpha \Uparrow \kappa'}\ \text{Variable}$$

$$\frac{}{\Delta \models \mathit{BoxFloat} \Uparrow S_T(\mathit{BoxFloat})}\ \text{BoxFloat}$$

$$\frac{}{\Delta \models \mathit{Int} \Uparrow S_T(\mathit{Int})}\ \text{Int}$$

$$\frac{\Delta[\alpha{::}T][\beta{::}T] \models T \Downarrow \quad \Delta[\alpha{::}T][\beta{::}T] \models T \Downarrow \quad \alpha, \beta \notin \mathrm{Dom}(\Delta)}{\Delta \models \mu(\alpha,\beta).(c_1,c_2) \Uparrow S_T(\mu(\alpha,\beta).(c_1,c_2).1) \times S_T(\mu(\alpha,\beta).(c_1,c_2).2)}\ \text{Mu}$$

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 \times c_2 \Uparrow S_T(c_1 \times c_2)}\ \text{pair}$$

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 \to c_2 \Uparrow S_T(c_1 \to c_2)}\ \text{Arrow}$$

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 + c_2 \Uparrow S_T(c_1 + c_2)}\ \text{Sum}$$

$$\frac{\Delta \models c_1 \Downarrow T \quad \Delta \models c_2 \Downarrow T}{\Delta \models c_1 +^i c_2 \Uparrow S_T(c_1 +^i c_2)}\ \text{KnownSum}$$

$$\frac{\Delta \models c \Downarrow T}{\Delta \models c\ \mathit{array} \Uparrow S_T(c\ \mathit{array})}\ \text{Array}$$

$$\frac{\Delta \models \kappa \quad \Delta[\alpha :: \kappa] \models c \Uparrow \kappa' \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \lambda\alpha{::}\kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa'}\ \text{Lambda}$$

$$\frac{\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2 \quad \Delta \models c_2 \Downarrow \kappa_1}{\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2}\ \text{App}$$

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta \models c_2 \Uparrow \kappa_2}{\Delta \models \langle c_1, c_2 \rangle \Uparrow \kappa_1 \times \kappa_2}\ \text{Record}$$

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.1 \Uparrow \kappa_1}\ \text{Proj1}$$

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.2 \Uparrow \{c.1/\alpha\}\kappa_2}\ \text{Proj2}$$

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2 \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \mathbf{let}\ \alpha = c_1\ \mathbf{in}\ c_2\ \mathbf{end} \Uparrow \{c_1/\alpha\}\kappa_2}\ \text{Let}$$

Well-formed Type

$\boxed{\Delta \models \tau}$

Assume Δ is well-formed. Check that τ is well-formed.

$$\frac{\Delta \models c \Downarrow T}{\Delta \models T(c)}\ \text{Constructor}$$

$$\frac{\begin{array}{ll}\Delta \models \kappa & \alpha \notin \mathrm{Dom}(\Delta) \\ \Delta[\alpha{::}\kappa] \models \tau_1 & \Delta[\alpha{::}\kappa] \models \tau_2\end{array}}{\Delta \models (\alpha :: \kappa, \tau_1) \to \tau_2}\ \text{ArrowType}$$

$$\frac{}{\Delta \models \mathit{Float}}\ \text{Float}$$

$$\frac{\Delta \models \tau_1 \quad \Delta \models \tau_2}{\Delta \models \tau_1 \times \tau_2}\ \text{PairType}$$

$$\frac{\Delta \models c \Uparrow \kappa_1 \quad \Delta[\alpha{::}\kappa_1] \models \tau \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \mathbf{let}\,\alpha = c\,\mathbf{in}\,\tau\,\mathbf{end}}\ \text{Let}$$

Type Analysis $\boxed{\Delta \models e \Downarrow \tau}$

Assume Δ and τ are well-formed. Check that e is well-typed, and has type τ.

$$\frac{\Delta \models e \Uparrow \tau' \quad \Delta \models \tau' \equiv \tau}{\Delta \models e \Downarrow \tau}\ \text{Analysis}$$

Type Synthesis $\boxed{\Delta \models e \Uparrow \tau}$

Assume Δ is well-formed. Check that e is well-formed and construct its type τ, where $\Delta \models \tau$

$$\frac{}{\Delta[x : \tau] \models x \Uparrow \tau}\ \text{Variable}$$

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta[x : \tau_1] \models e_2 \Uparrow \tau_2 \quad x \notin \mathrm{Dom}(\Delta)}{\Delta \models \mathbf{let}\,x = e_1\,\mathbf{in}\,e_2\,\mathbf{end} \Uparrow \tau_2}\ \text{lete}$$

$$\frac{\Delta \models c \Uparrow \kappa \quad \Delta[\alpha{::}\kappa] \models e \Uparrow \tau \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \models \mathbf{let}\,\alpha = c\,\mathbf{in}\,e\,\mathbf{end} \Uparrow \mathbf{let}\,\alpha = c\,\mathbf{in}\,\tau\,\mathbf{end}}\ \text{letc}$$

$$\frac{\begin{array}{l}\Delta \models \kappa \quad \Delta[\alpha{::}\kappa] \models \tau_1 \quad \Delta[\alpha{::}\kappa] \models \tau_2 \\ \Delta[f : (\alpha :: \kappa, \tau_1) \to \tau_2][\alpha{::}\kappa][x : \tau_1] \models e \Downarrow \tau_2 \quad f, x, \alpha \notin \mathrm{Dom}(\Delta))\end{array}}{\Delta \models \mathbf{rec}\,f = \lambda(\alpha{::}\kappa, x : \tau_1) : \tau_2.e \Uparrow (\alpha :: \kappa, \tau_1) \to \tau_2}\ \text{rec}$$

$$\frac{\Delta \models e_1 \Uparrow (\alpha :: \kappa, \tau_1) \to \tau_2 \quad \Delta \models c_{i+1} \Downarrow \{\overline{c}^{i} / \overline{\alpha}^{i}\}\kappa_{i+1} \quad \Delta \models e_2 \Downarrow \{\overline{c}^{n} / \overline{\alpha}^{n}\}\tau_1}{\Delta \models e_1[c]e_2 \Uparrow \mathbf{let}\,\alpha = c\,\mathbf{in}\,\tau_2\,\mathbf{end}}\ \text{app}$$

$$\frac{\Delta \models e_1 \Uparrow \tau \quad \Delta \models \tau \mapsto T(c_1 \to c_2) \quad \Delta \models e_2 \Downarrow T(c_1)}{\Delta \models e_1[]e_2 \Uparrow T(c_2)}\ \text{MonoApp}$$

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta \models e_2 \Uparrow \tau_2}{\Delta \models \langle e_1, e_2 \rangle \Uparrow \tau_1 \times \tau_2}\ \text{pair}$$

$$\frac{\Delta \models e \Uparrow \tau \quad \Delta \models \Delta \mapsto \tau\tau_1 \times \tau_2}{\Delta \models e.1 \Uparrow \tau_1}\ \text{type\_proj1}$$

$$\frac{\Delta \models e \Uparrow \tau \quad \Delta \models \tau \mapsto \tau_1 \times \tau_2}{\Delta \models e.2 \Uparrow \tau_2}\ \text{type\_proj2}$$

$$\frac{}{\Delta \models r \Uparrow \mathit{Float}}\ \text{Float}$$

$$\frac{}{\Delta \models n \Uparrow T(\mathit{Int})}\ \text{int}$$

$$\frac{\Delta \models e \Downarrow \mathit{Float}}{\Delta \models \mathbf{boxfloat}(e) \Uparrow T(\mathit{BoxFloat})}\ \text{box}$$

$$\frac{\Delta \models e \Downarrow T(\mathit{BoxFloat})}{\Delta \models \mathbf{unboxfloat}(e) \Uparrow \mathit{Float}}\ \text{unbox}$$

$$\frac{\begin{array}{lll} \Delta \models e \Uparrow \tau_e & \Delta \models \tau_e \mapsto T(c_1 + c_2) & \Delta \models \tau \\ \Delta[x : c_1 +^1 c_2] \models e_1 \Downarrow \tau & \Delta[x : c_1 +^2 c_2] \models e_2 \Downarrow \tau & \end{array}}{\Delta \models \mathbf{case}_\tau\, e\, \mathbf{of}\, \{\mathbf{inl}(x) \Rightarrow e_1, \mathbf{inr}(x) \Rightarrow e_2\} \Uparrow \tau_1}\ \text{sumswitch}$$

$$\frac{\begin{array}{ll} \Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models e \Downarrow T(c_1) & \end{array}}{\Delta \models \mathbf{inl}_{c_1,c_2} e \Uparrow T(c_1 + c_2)}\ \text{inl}$$

$$\frac{\begin{array}{ll} \Delta \models c \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models e \Downarrow T(c_2) & \end{array}}{\Delta \models \mathbf{inr}_{c_1,c_2} e \Uparrow T(c_1 + c_2)}\ \text{inr}$$

$$\frac{\begin{array}{ll} \Delta \models c \Downarrow T & \Delta \models c \mapsto \mu(\alpha,\beta).(c_1,c_2).iT \\ \Delta \models e \Downarrow T(\{c.1, c.2/\alpha,\beta\}c_i) & \end{array}}{\Delta \models \mathbf{roll}_c(e) \Uparrow T(c)}\ \text{roll}$$

$$\frac{\Delta \models e \Uparrow \tau \quad \Delta \models \tau \mapsto T(\mu(\alpha,\beta).(c_1,c_2).i)}{\Delta \models \mathbf{unroll}(e) \Uparrow T(\{\mu(\alpha,\beta).(c_1,c_2).1, \mu(\alpha,\beta).(c_1,c_2).2/\alpha,\beta\}c_i)}\ \text{unroll}$$

$$\frac{\Delta \models e \Uparrow \tau \quad \Delta \models \tau \mapsto T(c_1 +^i c_2)}{\Delta \models \mathbf{proj}_i(e) \Uparrow T(c_i)} \ \text{proj}$$

$$\frac{\begin{array}{l}\Delta \models e_1 \Downarrow T(Int) \quad \Delta \models c \Downarrow T \\ \Delta \models e_2 \Downarrow T(c)\end{array}}{\Delta \models \mathbf{array}_c(e_1, e_2) \Uparrow T(c\, array)} \ \text{array}$$

$$\frac{\Delta \models e_1 \Uparrow]\tau \quad \Delta \models \tau \mapsto T(c'\, array) \quad \Delta \models e_2 \Downarrow Int}{\Delta \models \mathbf{sub}[e_1](e_2, \Uparrow)T(c')} \ \text{sub}$$

$$\frac{\Delta \models e_1 \Downarrow T(BoxFloat\, array) \quad \Delta \models e_2 \Downarrow T(Int)}{\Delta \models \mathbf{fsub}(e_1, e_2) \Uparrow Float} \ \text{fsub}$$

Natural Kind Extraction

$\boxed{\Delta \models p \leadsto \kappa}$

Assumes that Δ and p are well-formed. Returns the unselfified kind of p.

$$\frac{}{\Delta[\alpha{::}\kappa] \models \alpha \leadsto \kappa} \ \text{Variable}$$

$$\frac{\Delta \models p \leadsto \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models p.1 \leadsto \kappa_1} \ \text{Proj1}$$

$$\frac{\Delta \models p \leadsto \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models p.2 \leadsto \{p.1/\alpha\}\kappa_2} \ \text{Proj2}$$

$$\frac{\Delta \models p \leadsto \Pi(\alpha :: \kappa_1).\kappa_2}{\Delta \models p\, c \leadsto \{c/\alpha\}\kappa_2} \ \text{App}$$

Weak Head Beta Short Form

$\boxed{\Delta \models c \hookrightarrow c'}$

$$\frac{\Delta \models c_1 \hookrightarrow \lambda\alpha :: \kappa.c_1 \quad \Delta \models \{c_2/\alpha\}c_1 \hookrightarrow c}{\Delta \models c_1\, c_2 \hookrightarrow c} \ \text{App}$$

$$\frac{\Delta \models c \hookrightarrow \langle c_1, c_2 \rangle \quad \Delta \models c_1 \hookrightarrow c_1'}{\Delta \models c.1 \hookrightarrow c_1'} \ \text{Proj1}$$

$$\frac{\Delta \models c \hookrightarrow \langle c_1, c_2 \rangle \quad \Delta \models c_2 \hookrightarrow c_2'}{\Delta \models c.2 \hookrightarrow c_2'} \ \text{Proj2}$$

$$\frac{\Delta \models \{c_1/\alpha\}c_2 \hookrightarrow c}{\Delta \models \textbf{let}\, \alpha = c_1 \,\textbf{in}\, c_2 \,\textbf{end} \hookrightarrow c} \text{ Let}$$

$$\frac{}{\Delta \models c \hookrightarrow c} \text{ Otherwise}$$

Constructor Weak Head Normal Form

$\boxed{\Delta \models c \mapsto c'}$

Assumes that Δ and c are well-formed. Returns the head normal form of c.

$$\frac{\Delta \models c \hookrightarrow p \quad \Delta \models p \rightsquigarrow S_T(c') \quad \Delta \models c' \mapsto c''}{\Delta \models c \mapsto c''} \text{ Pathequation}$$

$$\frac{\Delta \models c \hookrightarrow p \quad \Delta \models p \rightsquigarrow \kappa \quad \kappa \neq S_T(c')}{\Delta \models c \mapsto p} \text{ Pathnoequation}$$

$$\frac{\Delta \models c \hookrightarrow c' \quad c' \text{ not a path}}{\Delta \models c \mapsto c'} \text{ NonPath}$$

Type Weak Head Normal Form

$\boxed{\Delta \models \tau \mapsto \tau'}$

$$\frac{\Delta \models \{c/\alpha\}\tau \mapsto \tau'}{\Delta \models \textbf{let}\, \alpha = c \,\textbf{in}\, \tau \,\textbf{end} \mapsto \tau'} \text{ Let}$$

$$\frac{\Delta \models c \mapsto c_1 \times c_2}{\Delta \models T(c) \mapsto T(c_1) \times T(c_2)} \text{ Conpair}$$

$$\frac{\Delta \models c \mapsto c'}{\Delta \models T(c) \mapsto T(c')} \text{ Inclusion}$$

$$\frac{}{\Delta \models \tau \mapsto \tau} \text{ Otherwise}$$

A.3 Termination Proofs

A.3.1 Proof of Lemma 2

To show: SZ is order preserving. That is, $J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$

Proof. We proceed by cases on the conclusion of J_2.

1. $\Delta \models T \preceq T$. Vacuously true: J_2 is minimal. and hence has nothing smaller than it.

2. $\Delta \models S_T(c) \preceq T$. Vacuously true: J_2 is again minimal.

3. $\Delta \models S_T(c) \preceq S_T(d)$. The rule for this judgement has no sub-kinding derivations, and hence has nothing smaller than it. The only subgoal is an equivalence derivation, which has been shown to be decidable separately [SH99].

4. $\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2 \preceq \Pi(\alpha :: \kappa_1').\kappa_2'$. Suppose $J_1 \prec J_2$. From the subkinding rule for the Π kind, we see that there are two possibilities for the conclusion of J_1:

   (a) $\Delta \models \kappa_1' \preceq \kappa_1$

   $$\begin{array}{rcl} SZ(J_1) & = & sz(\kappa_1') + sz(\kappa_1) \\ & < & sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ & = & SZ(J_2) \end{array}$$

   (b) $\Delta[\alpha{::}\kappa_1'] \models \kappa_2 \preceq \kappa_2'$

   $$\begin{array}{rcl} SZ(J_1) & = & sz(\kappa_2') + sz(\kappa_2) \\ & < & sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ & = & SZ(J_2) \end{array}$$

5. $\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2 \preceq \Sigma(\alpha :: \kappa_1').\kappa_2'$. Suppose $J_1 \prec J_2$. From the subkinding rule for the Σ kind, we see that there are two possibilities for the conclusion of J_1:

   (a) $\Delta \models \kappa_1 \preceq \kappa_1'$.

   $$\begin{array}{rcl} SZ(J_1) & = & sz(\kappa_1') + sz(\kappa_1) \\ & < & sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ & = & SZ(J_2) \end{array}$$

   (b) $\Delta[\alpha{::}\kappa_1] \models \kappa_2 \preceq \kappa_2'$.

   $$\begin{array}{rcl} SZ(J_1) & = & sz(\kappa_2') + sz(\kappa_2) \\ & < & sz(\kappa_1') + sz(\kappa_1) + sz(\kappa_2') + sz(\kappa_2) \\ & = & SZ(J_2) \end{array}$$

■

A.3.2 Proof of Lemma 5

To show: SZ is order preserving. That is, $J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$ where $<$ is the lexicographic ordering on $N \times N$.

Proof. The proof proceeds by cases over the conclusion of J_2, demonstrating that each immediate subderivation is strictly smaller according to the given metric. We ignore subderivations that correspond to judgements which are independently known to be decidable, such as subkinding and constructor equivalence. Technically, this may be viewed as using the constant measure that always returns zero for these judgements.

1. Well Formed Kind $\Delta \models \kappa$ We proceed by subcases on the form of κ.

   (a) T No premises.

(b) $S_T(c)$

$$\begin{aligned} SZ(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= SZ(\Delta \models S_T(c)) \end{aligned}$$

(c) $\Pi(\kappa_1 :: \kappa_2)$.

i.

$$\begin{aligned} SZ(\Delta \models \kappa_1) &= (sz_\kappa(\kappa_1), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

ii.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa_1] \models \kappa_2) &= (sz_\kappa(\kappa_2), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

(d) $\Sigma(\kappa_1 :: \kappa_2)$.

i.

$$\begin{aligned} SZ(\Delta \models \kappa_1) &= (sz_\kappa(\kappa_1), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

ii.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa_1] \models \kappa_2) &= (sz_\kappa(\kappa_2), 0) \\ &< (sz_\kappa(\kappa_1) + sz_\kappa(\kappa_2), 0) \\ &= SZ(\Delta \models \Sigma(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

2. Kind Analysis $\Delta \models c \Downarrow \kappa$.

$$\begin{aligned} SZ(\Delta \models c \Uparrow \kappa') &= (sz_c(c), 0) \\ &< (sz_c(c), 1) \\ &= SZ(\Delta \models c \Downarrow \kappa) \end{aligned}$$

3. Kind Synthesis $\Delta \models c \Uparrow \kappa$

Variable By lemma 4

BoxFloat No premises

Int No premises

Mu

(a)

$$\begin{aligned} SZ(\Delta[\alpha :: T, \beta :: T] \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(\alpha, \beta).(c_1, c_2)), 0) \\ &= SZ(\Delta \models \mu(\alpha, \beta).(c_1, c_2) \Uparrow \kappa) \end{aligned}$$

where $\kappa = S_T(\mu(\alpha,\beta).(c_1,c_2).1) \times S_T(\mu(\alpha,\beta).(c_1,c_2).2)$.

(b) Similar

Pair

(a)

$$\begin{aligned} SZ(\Delta \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1 \times c_2), 0) \\ &= SZ(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1 \times c_2)) \end{aligned}$$

(b) Similarly for the second premise.

Arrow As with the Pair case.

Sum As with the Pair case.

Array

$$\begin{aligned} SZ(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= (sz_c(c\ array), 0) \\ &= SZ(\Delta \models c\ array \Uparrow S_T(c\ array)) \end{aligned}$$

Lambda

(a)

$$\begin{aligned} SZ(\Delta \models \kappa) &= (sz_\kappa(\kappa), 0) \\ &< (sz_\kappa(\kappa) + sz_c(c), 0) \\ &= (sz_c(\lambda\alpha{::}\kappa.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}\kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

(b)

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa] \models c \Uparrow \kappa') &= (sz_c(c), 0) \\ &< (sz_\kappa(\kappa) + sz_c(c), 0) \\ &= (sz_c(\lambda\alpha{::}\kappa.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}\kappa.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

App

(a)

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1\, c_2), 0) \\ &= SZ(\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2) \end{aligned}$$

(b)

$$\begin{aligned} SZ(\Delta \models c_2 \Downarrow \kappa_1) &= (sz_c(c_2), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1\, c_2), 0) \\ &= SZ(\Delta \models c_1\, c_2 \Uparrow \{c_2/\alpha\}\kappa_2) \end{aligned}$$

Record

(a)

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \kappa_1) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (, sz_c(< c_1, c_2 >), 0) \\ &= SZ(\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2) \end{aligned}$$

(b)

$$\begin{aligned} SZ(\Delta \models c_2 \Uparrow \kappa_2) &= (sz_c(c_2), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(< c_1, c_2 >), 0) \\ &= SZ(\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2) \end{aligned}$$

Proj1

$$\begin{aligned} SZ(\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2) &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= (sz_c(c.1), 0) \\ &= SZ(\Delta \models c.1 \Uparrow \kappa_1) \end{aligned}$$

Proj2 As with Proj1

Let

(a)

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \kappa_1) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\textbf{let}\, \alpha = c_1 \,\textbf{in}\, c_2 \,\textbf{end}), 0) \\ &= SZ(\Delta \models \textbf{let}\, \alpha = c_1 \,\textbf{in}\, c_2 \,\textbf{end} \Uparrow \{c_1/\alpha\}\kappa_2) \end{aligned}$$

(b)

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2) &= (sz_c(c_2), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\textbf{let}\, \alpha = c_1 \,\textbf{in}\, c_2 \,\textbf{end}), 0) \\ &= SZ(\Delta \models \textbf{let}\, \alpha = c_1 \,\textbf{in}\, c_2 \,\textbf{end} \Uparrow \{c_1/\alpha\}\kappa_2) \end{aligned}$$

■

B NIL (Extended MIL)

B.1 Algorithmic judgments

Kind Standardization

$\boxed{\Delta \models \underline{k} \backslash \kappa}$

$$\frac{}{\Delta \models T \backslash T} \text{ Type}$$

$$\frac{\Delta \models c\backslash c'}{\Delta \models S_T(c)\backslash S_T(c')} \;\text{Singleton Type}$$

$$\frac{\Delta \models c \Uparrow \kappa}{\Delta \models S(c)\backslash \kappa} \;\text{Singleton Any}$$

$$\frac{\Delta \models \underline{k}_1\backslash\kappa_1 \quad \Delta[\alpha{::}\kappa_1] \models \underline{k}_2\backslash\kappa_2}{\Delta \models \Pi(\alpha :: \underline{k}_1).\underline{k}_2\backslash\Pi(\alpha :: \kappa_1).\kappa_2} \;\text{Pi}$$

$$\frac{\Delta \models \underline{k}_1\backslash\kappa_1 \quad \Delta[\alpha{::}\kappa_1] \models \underline{k}_2\backslash\kappa_2}{\Delta \models \Sigma(\alpha :: \underline{k}_1).\underline{k}_2\backslash\Sigma(\alpha :: \kappa_1).\kappa_2} \;\text{Sigma}$$

Constructor standardization $\boxed{\Delta \models c\backslash c'}$

All cases proceed compositionally over the structure of the constructors except for the following cases:

$$\frac{\Delta \models \underline{k}\backslash\kappa \quad \Delta[\alpha{::}\kappa] \models c\backslash c'}{\Delta \models \lambda\alpha{::}\underline{k}.c\backslash\lambda\alpha{::}\kappa.c'} \;\text{Lambda}$$

$$\frac{\begin{array}{ll}\Delta \models c_1\backslash c_1' & \Delta \models c_1 \Uparrow \kappa \\ \Delta[\alpha{::}\kappa] \models c_2\backslash c_2' & \end{array}}{\Delta \models \mathbf{let}\; \alpha = c_1 \;\mathbf{in}\; c_2 \;\mathbf{end}\backslash\mathbf{let}\; \alpha = c_1' \;\mathbf{in}\; c_2' \;\mathbf{end}} \;\text{Let}$$

Type standardization $\boxed{\Delta \models t\backslash\tau}$

$$\frac{\Delta \models c\backslash c'}{\Delta \models T(c)\backslash T(c')} \;\text{Constructor}$$

$$\frac{\begin{array}{ll}\Delta \models \underline{k}\backslash\kappa & \Delta[\alpha{::}\kappa] \models t_1\backslash\tau_1 \\ \Delta[\alpha{::}\kappa][x : \tau_1] \models t_2\backslash\tau_2 & \end{array}}{\Delta \models (\alpha :: \underline{k}, x : t_1) \to t_2\backslash(\alpha :: \kappa, \tau_1) \to \tau_2} \;\text{Arrow}$$

$$\frac{}{\Delta \models \mathit{Float}\backslash\mathit{Float}} \;\text{Float}$$

$$\frac{\Delta \models t_1\backslash\tau_1 \quad \Delta \models t_2\backslash\tau_2}{\Delta \models t_1 \times t_2\backslash\tau_1 \times \tau_2} \;\text{Pair}$$

Well Formed Kind

$\boxed{\Delta \models \underline{k}}$

The Type and Singleton Type rules are as before.

$$\frac{\Delta \models c \Downarrow T}{\Delta \models S(c)}\ \text{Singleton Any}$$

$$\frac{\begin{array}{l}\Delta \models \underline{k}_1 \qquad \Delta \models \underline{k}_1 \backslash \kappa_1 \\ \Delta[\alpha{::}\kappa_1] \models \underline{k}_2\end{array}}{\Delta \models \Pi(\alpha :: \underline{k}_1).\underline{k}_2}\ \text{Pi}$$

$$\frac{\begin{array}{l}\Delta \models \underline{k}_1 \qquad \Delta \models \underline{k}_1 \backslash \kappa_1 \\ \Delta[\alpha{::}\kappa_1] \models \underline{k}_2\end{array}}{\Delta \models \Sigma(\alpha :: \underline{k}_1).\underline{k}_2}\ \text{Sigma}$$

Sub-Kinding

$\boxed{\Delta \models \kappa_1 \preceq \kappa_2}$

We do not need to redefine subkinding for the extended NIL - all queries will be restricted to core syntax.

Kind Analysis

$\boxed{\Delta \models c \Downarrow \kappa}$

Note that we restrict this judgement to core kinds. Assume Δ and κ are well formed. Check that c is well formed and can be given kind $\underline{k}$.

$$\frac{\Delta \models c \Uparrow \kappa' \quad \Delta \models \kappa' \preceq \kappa}{\Delta \models c \Downarrow \kappa}\ \text{Analysis}$$

Kind Synthesis

$\boxed{\Delta \models c \Uparrow k}$

Assumes that Δ is well-formed. Check that c is well-kinded, and construct κ s.t. $\Delta \models \kappa$ and c has kind κ.

$$\frac{\models \alpha :: \kappa \doteq \kappa'}{\Delta[\alpha :: \kappa] \models \alpha \Uparrow \kappa'}\ \text{Variable}$$

$$\frac{}{\Delta \models \mathit{BoxFloat} \Uparrow S_T(\mathit{BoxFloat})}\ \text{BoxFloat}$$

$$\frac{}{\Delta \models \mathit{Int} \Uparrow S_T(\mathit{Int})}\ \text{Int}$$

$$\frac{\begin{array}{l}\Delta[\alpha{::}T][\beta{::}T] \models c_1 \Downarrow T \quad \Delta[\alpha{::}T][\beta{::}T] \models c_2 \Downarrow T \\ \Delta[\alpha{::}T][\beta{::}T] \models c_1 \backslash c_1' \quad \Delta[\alpha{::}T][\beta{::}T] \models c_2 \backslash c_2' \quad a, b \notin \mathrm{Dom}(\Delta)\end{array}}{\Delta \models \mu(\alpha,\beta).(c_1, c_2) \Uparrow S_T(\mu(\alpha,\beta).(c_1', c_2').1) \times S_T(\mu(\alpha,\beta).(c_1', c_2').2)}\ \mu$$

$$\frac{\begin{array}{ll}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')}\ \text{Pair}$$

$$\frac{\begin{array}{ll}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 \rightarrow c_2 \Uparrow S_T(c_1' \rightarrow c_2')}\ \text{Arrow}$$

$$\frac{\begin{array}{ll}\Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2'\end{array}}{\Delta \models c_1 + c_2 \Uparrow S_T(c_1' + c_2')}\ \text{Sum}$$

$$\frac{\Delta \models c \Downarrow T \quad \Delta \models c \backslash c'}{\Delta \models c\ \mathit{array} \Uparrow S_T(c'\ \mathit{array})}\ \text{Array}$$

$$\frac{\begin{array}{ll}\Delta \models \underline{k} & \Delta \models k \backslash \kappa \\ \Delta[\alpha :: \kappa] \models c \Uparrow \kappa' & \alpha \notin \mathrm{Dom}(\Delta)\end{array}}{\Delta \models \lambda\alpha{::}\underline{k}.c \Uparrow \Pi(\alpha :: \kappa).\kappa'}\ \text{Lambda}$$

$$\frac{\begin{array}{ll}\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2 & \Delta \models c_2 \Downarrow \kappa_1 \\ \Delta \models c_2 \backslash c_2' & \end{array}}{\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2}\ \text{App}$$

$$\frac{\Delta \models c_1 \Uparrow \kappa_1 \quad \Delta \models c_2 \Uparrow \kappa_2}{\Delta \models < c_1, c_2 > \Uparrow \kappa_1 \times \kappa_2}\ \text{Record}$$

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2}{\Delta \models c.1 \Uparrow \kappa_1}\ \text{Proj1}$$

$$\frac{\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2 \quad \Delta \models c \backslash c'}{\Delta \models c.2 \Uparrow \{c'.1/\alpha\}\kappa_2}\ \text{Proj2}$$

$$\frac{\begin{array}{ll}\Delta \models c_1 \Uparrow \kappa_1 \Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2 & \\ \Delta \models c_1 \backslash c_1' & \alpha \notin \mathrm{Dom}(\Delta)\end{array}}{\Delta \models \textbf{let}\ \alpha = c_1\ \textbf{in}\ c_2\ \textbf{end} \Uparrow \{c_1'/\alpha\}\kappa_2}\ \text{Let}$$

Well-formed Type $\boxed{\Delta \models \tau}$

Assume Δ is well-formed. Check that τ is well-formed.

$$\frac{\Delta \models c \Downarrow T}{\Delta \models T(c)}\ \text{Constructor}$$

$$\frac{\begin{array}{ll} \Delta \models \underline{k} & \Delta \models \underline{k}\backslash\kappa \\ \Delta[\alpha{::}\kappa] \models \tau_1 & \Delta[\alpha{::}\kappa] \models \tau_1\backslash\tau_1' \\ \Delta[\alpha{::}\kappa][x : \tau_1'] \models \tau_2 & \alpha \notin \mathrm{Dom}(\Delta) \end{array}}{\Delta \models (\alpha :: \underline{k}, x : \tau_1) \rightarrow \tau_2}\ \text{ArrowType}$$

$$\frac{}{\Delta \models \mathit{Float}}\ \text{Float}$$

$$\frac{\Delta \models \tau_1 \quad \Delta \models \tau_2}{\Delta \models \tau_1 \times \tau_2}\ \text{PairType}$$

Type Analysis

$\boxed{\Delta \models e \Downarrow \tau}$

Note that we restrict this to core types. Assume Δ and t are well-formed. Check that e is well-typed, and has type τ.

$$\frac{\Delta \models e \Uparrow \tau' \quad \Delta \models \tau' \equiv \tau}{\Delta \models e \Downarrow \tau}\ \text{Analysis}$$

Type Synthesis

$\boxed{\Delta \models e \Uparrow \tau}$

Assume Δ is well-formed. Check that e is well-formed and construct its type τ, such that $\Delta \models \tau$

$$\frac{}{\Delta[x : \tau] \models x \Uparrow \tau}\ \text{variable}$$

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta[x : \tau_1] \models e_2 \Uparrow \tau_2 \quad x \notin \mathrm{Dom}(\Delta)}{\Delta \models \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \,\mathbf{end} \Uparrow \tau_2}\ \text{lete}$$

$$\frac{\begin{array}{ll} \Delta \models c \Uparrow \kappa & \Delta \models c\backslash c' \\ \Delta[\alpha{::}\kappa] \models e \Uparrow \tau & \alpha \notin \mathrm{Dom}(\Delta) \end{array}}{\Delta \models \mathbf{let}\, \alpha = c \,\mathbf{in}\, e \,\mathbf{end} \Uparrow \{c'/\alpha\}\tau}\ \text{letc}$$

$$\frac{\begin{array}{ll} \Delta \models \underline{k} & \Delta \models \underline{k}\backslash\kappa \\ \Delta[\alpha{::}\kappa] \models \tau_1 & \Delta[\alpha{::}\kappa] \models \tau_1\backslash\tau_1' \\ \Delta[\alpha{::}\kappa] \models \tau_2 & \Delta[\alpha{::}\kappa] \models \tau_2\backslash\tau_2' \\ \Delta[\alpha{::}\kappa][x : \tau_1'][f : (\alpha :: \kappa, \tau_1') \rightarrow \tau_2'] \models e \Downarrow \tau_2' & \\ f, x, \alpha \notin \mathrm{Dom}(\Delta) & \end{array}}{\Delta \models \mathbf{rec}\, f = \lambda(\alpha{::}k, x : \tau_1) : \tau_2.e \Uparrow (\alpha :: \kappa, \tau_1') \rightarrow \tau_2'}\ \text{rec}$$

$$\frac{\begin{array}{ll} \Delta \models e_1 \Uparrow (\alpha :: \kappa, \tau_1) \rightarrow \tau_2 & \Delta \models c \Downarrow \kappa \\ \Delta \models c\backslash c' & \Delta \models e_2 \Downarrow \{c'/\alpha\}\tau_1 \end{array}}{\Delta \models e_1[c]e_2 \Uparrow \{c'/\alpha\}\tau_2}\ \text{app}$$

$$\frac{\Delta \models e_1 \Uparrow T(c_e) \quad \Delta \models c_e \mapsto c_1 \to c_2 \quad \Delta \models e_2 \Downarrow T(c_1)}{\Delta \models e_1[]e_2 \Uparrow T(c_2)} \ \text{Monomorphic app}$$

$$\frac{\Delta \models e_1 \Uparrow \tau_1 \quad \Delta \models e_2 \Uparrow \tau_2}{\Delta \models < e_1, e_2 > \Uparrow \tau_1 \times \tau_2} \ \text{pair}$$

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.1 \Uparrow \tau_1} \ \text{type\_proj1}$$

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \mapsto c_1 \times c_2}{\Delta \models e.1 \Uparrow T(c_1)} \ \text{con\_proj1}$$

$$\frac{\Delta \models e \Uparrow \tau_1 \times \tau_2}{\Delta \models e.2 \Uparrow \tau_2} \ \text{type\_proj2}$$

$$\frac{\Delta \models e \Uparrow T(c) \quad \Delta \models c \mapsto c_1 \times c_2}{\Delta \models e.2 \Uparrow T(c_2)} \ \text{con\_proj2}$$

$$\frac{}{\Delta \models r \Uparrow Float} \ \text{float}$$

$$\frac{}{\Delta \models n \Uparrow T(Int)} \ \text{int}$$

$$\frac{\Delta \models e \Downarrow Float}{\Delta \models \mathbf{boxfloat}(e) \Uparrow T(BoxFloat)} \ \text{box}$$

$$\frac{\Delta \models e \Downarrow T(BoxFloat)}{\Delta \models \mathbf{unboxfloat}(e) \Uparrow Float} \ \text{unbox}$$

$$\frac{\begin{array}{ll} \Delta \models e \Uparrow T(c) & \Delta \models c \mapsto c_1 + c_2 \\ \Delta[x : T(c_1 +^1 c_2)] \models e_1 \Uparrow \tau_1 & \Delta[x : T(c_1 +^1 c_2)] \models e_2 \Uparrow \tau_2 \\ \Delta \models \tau_1 \equiv \tau_2 & \end{array}}{\Delta \models \mathbf{case}_e\, x\, \mathbf{of}\, \{\mathbf{inl}(e_1) \Rightarrow e_2, \mathbf{inr}(e_1) \Rightarrow \Uparrow\}\tau_1} \ \text{sumswitch}$$

$$\frac{\begin{array}{ll} \Delta \models c_1 \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2' \\ \Delta \models e \Downarrow T(c_1') & \end{array}}{\Delta \models \mathbf{inl}_{c_1,c_2} e \Uparrow T(c_1' + c_2')} \ \text{inl}$$

$$\frac{\begin{array}{ll}\Delta \models c \Downarrow T & \Delta \models c_2 \Downarrow T \\ \Delta \models c_1 \backslash c_1' & \Delta \models c_2 \backslash c_2' \\ \Delta \models e \Downarrow T(c_2') & \end{array}}{\Delta \models \mathbf{inr}_{c_1,c_2} e \Uparrow T(c_1' + c_2')}\ \text{inr}$$

$$\frac{\begin{array}{ll}\Delta \models c \Downarrow T & \Delta \models c \backslash c' \\ \Delta \models e_1 \Downarrow T(Int) & \Delta \models e_2 \Downarrow T(c') \end{array}}{\Delta \models \mathbf{array}_c(e_1, e_2) \Uparrow T(c'\ array)}\ \text{array}$$

$$\frac{\Delta \models e_1 \Uparrow T(c) \quad \Delta \models c \mapsto c'\ array \quad \Delta \models e_2 \Downarrow Int}{\Delta \models \mathbf{sub}[e_1](e_2, \Uparrow) T(c')}\ \text{sub}$$

$$\frac{\Delta \models e_1 \Downarrow T(BoxFloat\ array) \quad \Delta \models e_2 \Downarrow T(Int)}{\Delta \models \mathbf{fsub}(e_1, e_2) \Uparrow Float}\ \text{fsub}$$

B.2 Termination Proofs

B.2.1 Proof of Lemma 7

To show: SZ is order preserving. That is, $J_1 \prec J_2 \Rightarrow SZ(J_1) < SZ(J_2)$ where $<$ is the lexicographic ordering on $N \times N$.

Proof. The proof proceeds by cases over the conclusion of J_2, demonstrating that each immediate subderivation is strictly smaller according to the given metric. We ignore subderivations that correspond to judgements which are independently known to be decidable, such as subkinding and constructor equivalence. Technically, this may be viewed as using the constant measure that always returns zero for these judgements.

- Kind standardization $\Delta \models k \backslash \kappa$

Type No premises

Singleton_Type

$$\begin{array}{rcl} SZ(\Delta \models c \backslash c') & = & (sz_c(c), 0) \\ & < & (sz_c(c) + 1, 0) \\ & = & SZ(\Delta \models S_T(c) \backslash S_T(c')) \end{array}$$

Singleton_Any

$$\begin{array}{rcl} SZ(\Delta \models c \Uparrow \kappa) & = & (sz_c(c), 0) \\ & < & (sz_c(c) + 1, 0) \\ & = & SZ(\Delta \models S(c) \backslash \kappa) \end{array}$$

Pi

1.

$$\begin{aligned} SZ(\Delta \models k_1 \backslash \kappa_1) &= (sz_\kappa(k_1), 0) \\ &< (sz_\kappa(\Pi(\alpha :: k_1).k_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: k_1).k_2 \backslash \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa_1] \models k_2 \backslash \kappa_2) &= (sz_\kappa(k_2), 0) \\ &< (sz_\kappa(\Pi(\alpha :: k_1).k_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: k_1).k_2 \backslash \Pi(\alpha :: \kappa_1).\kappa_2) \end{aligned}$$

Sigma As with the Pi case.

- Constructor standardization (All cases except those below are just decomposition of the constructor)

Lambda

1.

$$\begin{aligned} SZ(\Delta \models k \backslash \kappa) &= (sz_\kappa(k), 0) \\ &< (sz_c(\lambda\alpha{::}k.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}k.c \backslash \lambda\alpha{::}\kappa.c') \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa] \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(\lambda\alpha{::}k.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}k.c \backslash \lambda\alpha{::}\kappa.c') \end{aligned}$$

Let The size of the original derivation is

$$SZ(\Delta \models \textbf{let}\ \alpha = c_1\ \textbf{in}\ c_2\ \textbf{end} \backslash \textbf{let}\ \alpha = c_1'\ \textbf{in}\ c_2'\ \textbf{end}) = (sz_c(c_1) + sz_c(c_2), 0)$$

1.

$$\begin{aligned} SZ(\Delta \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \kappa) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \end{aligned}$$

3.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa] \models c_2 \backslash c_2') &= (sz_c(c_2), 0) \\ &< (sz_c(c_2) + sz_c(c_1), 0) \end{aligned}$$

- Well Formed Kind $\Delta \models k$

  Type, Singleton_Type As before

  Singleton_Any

$$\begin{aligned} SZ(\Delta \models c \Uparrow \kappa) &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= SZ(\Delta \models S(c)) \end{aligned}$$

  Pi

  1.

$$\begin{aligned} SZ(\Delta \models k_1) &= (sz_\kappa(k_1), 0) \\ &< (sz_\kappa(k_1) + sz_\kappa(k_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: k_1).k_2) \end{aligned}$$

  2.

$$\begin{aligned} SZ(\Delta[\alpha{::}\kappa_1] \models k_2) &= (sz_\kappa(k_2), 0) \\ &< (sz_\kappa(k_1) + sz_\kappa(k_2), 0) \\ &= SZ(\Delta \models \Pi(\alpha :: k_1).k_2) \end{aligned}$$

  Sigma As with the Pi case.

- Kind Analysis remains unchanged.

- Kind Synthesis $\Delta \models c \Uparrow \kappa$

  Variable By lemma 4. Note that kinds in the context are restricted to the core syntactic forms.

  BoxFloat No premises

  Int No premises

  μ Let $\kappa = S_T(\mu(\alpha, \beta).(c_1', c_2').1) \times S_T(\mu(\alpha, \beta).(c_1', c_2').2)$

  1.

$$\begin{aligned} SZ(\Delta[\alpha{::}T][\beta{::}T] \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(a = c_1, b = c_2)), 0) \\ &= SZ(\Delta \models \mu(\alpha, \beta).(c_1, c_2) \Uparrow \kappa) \end{aligned}$$

  2.

$$\begin{aligned} SZ(\Delta[\alpha{::}T][\beta{::}T] \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mu(\alpha, \beta).(c_1, c_2)), 0) \end{aligned}$$

  3. The cases for c_2 are exactly the same.

Pair

1.

$$\begin{aligned} SZ(\Delta \models c_1 \Downarrow T) &= (sz_c(c_1), 1) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1 \times c_2), 0) \\ &= SZ(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models c_1 \backslash c_1') &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= SZ(\Delta \models c_1 \times c_2 \Uparrow S_T(c_1' \times c_2')) \end{aligned}$$

3. Similarly for the c_2 premises.

Arrow As with the Pair case.

Sum As with the Pair case.

Array

1.

$$\begin{aligned} SZ(\Delta \models c \Downarrow T) &= (sz_c(c), 1) \\ &< (sz_c(c) + 1, 0) \\ &= (sz_c(c\ array), 0) \\ &= SZ(\Delta \models c\ array \Uparrow S_T(c\ array)) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(c) + 1, 0) \\ &= SZ(\Delta \models c\ array \Uparrow c'\ array) \end{aligned}$$

Lambda

1.

$$\begin{aligned} SZ(\Delta \models k) &= (sz_\kappa(k), 0) \\ &< (sz_\kappa(k) + sz_c(c), 0) \\ &= (sz_c(\lambda\alpha{::}k.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models k \backslash \kappa) &= (sz_\kappa(k), 0) \\ &< (sz_c(\lambda\alpha{::}k.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

3.

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa] \models c \Uparrow \kappa') &= (sz_c(c), 0) \\ &< (sz_c(\lambda\alpha{::}k.c), 0) \\ &= SZ(\Delta \models \lambda\alpha{::}k.c \Uparrow \Pi(\alpha :: \kappa).\kappa') \end{aligned}$$

App

1.

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \Pi(\alpha :: \kappa_1).\kappa_2) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(c_1\, c_2), 0) \\ &= SZ(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models c_2 \Downarrow \kappa_1) &= (sz_c(c_2), 1) \\ &< (sz_c(c_1\, c_2), 0) \\ &= SZ(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{aligned}$$

3.

$$\begin{aligned} SZ(\Delta \models c_2 \backslash c_2') &= (sz_c(c_2), 0) \\ &< (sz_c(c_1\, c_2), 0) \\ &= SZ(\Delta \models c_1\, c_2 \Uparrow \{c_2'/\alpha\}\kappa_2) \end{aligned}$$

Record As before

Proj1 As before

Proj2

1.

$$\begin{aligned} SZ(\Delta \models c \Uparrow \Sigma(\alpha :: \kappa_1).\kappa_2) &= (sz_c(c), 0) \\ A &< (sz_c(c) + 1, 0) \\ &= (sz_c(c.1), 0) \\ &= SZ(\Delta \models c.1 \Uparrow \{c'.1/\alpha\}\kappa_2) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta \models c \backslash c') &= (sz_c(c), 0) \\ &< (sz_c(c.1), 0) \\ &= SZ(\Delta \models c.1 \Uparrow \{c'.1/\alpha\}\kappa_2) \end{aligned}$$

Let

1.

$$\begin{aligned} SZ(\Delta \models c_1 \Uparrow \kappa_1) &= (sz_c(c_1), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mathbf{let}\, \alpha = c_1\, \mathbf{in}\, c_2\, \mathbf{end}), 0) \\ &= SZ(\Delta \models \mathbf{let}\, \alpha = c_1\, \mathbf{in}\, c_2\, \mathbf{end} \Uparrow \{c_1'/\alpha\}\kappa_2) \end{aligned}$$

2.

$$\begin{aligned} SZ(\Delta[\alpha :: \kappa_1] \models c_2 \Uparrow \kappa_2) &= (sz_c(c_2), 0) \\ &< (sz_c(c_1) + sz_c(c_2), 0) \\ &= (sz_c(\mathbf{let}\, \alpha = c_1\, \mathbf{in}\, c_2\, \mathbf{end}), 0) \\ &= SZ(\Delta \models \mathbf{let}\, \alpha = c_1\, \mathbf{in}\, c_2\, \mathbf{end} \Uparrow \{c_1'/\alpha\}\kappa_2) \end{aligned}$$

3.

$$\begin{array}{rcl}
SZ(\Delta \models c_1 \backslash c_1') & = & (sz_c(c_1), 0) \\
 & < & (sz_c(\mathbf{let}\ \alpha = c_1\ \mathbf{in}\ c_2\ \mathbf{end}), 0) \\
 & = & SZ(\Delta \models \mathbf{let}\ \alpha = c_1\ \mathbf{in}\ c_2\ \mathbf{end} \Uparrow \{c_1'/\alpha\}\kappa_2)
\end{array}$$

■