Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 218 (2008) 313–336
www.elsevier.com/locate/entcs

Dynamic Translucency with Abstraction Kinds and Higher-Order Coercions

Andreas Rossberg[1]

Max Planck Institute for Software Systems
Saarbrücken, Germany

Abstract

When a module language is combined with forms of non-parametric type analysis, abstract types require an opaque dynamic representation in order to maintain abstraction safety. As an idealisation of such a module language, we present a foundational calculus that combines higher-order type generation, modelling type abstraction, with singleton kinds, modelling translucency. In this calculus, type analysis can dynamically exploit translucency, without breaking abstraction. Abstract types are classified by a novel notion of abstraction kinds. These are analogous to singletons, but instead of inducing equivalence they induce an isomorphism that is witnessed by explicit type coercions on the term level. To encompass higher-order forms of translucent abstraction, we give an account for higher-order coercions in a rich type system with higher-order polymorphism and dependent kinds. The latter necessitate the introduction of an analogous notion of kind coercions on the type level. Finally, we give an abstraction-safe encoding of ML-style module sealing in terms of higher-kinded type generation and higher-order coercion.

Keywords: Abstract types, modules, generativity, coercions, parametricity, singleton kinds

1 Introduction

Modules play an important role in the construction of large-scale software and appear in various forms in many programming languages. Particularly sophisticated module systems have been developed and investigated in the context of the ML family of languages [19,15,13,17,25,6].

Two features form the core of a module type system in this line of work:

(i) *Type abstraction* provides for encapsulation by hiding the definition of a type exported by a module.

(ii) *Type sharing* enables module signatures to specify several types as equivalent, without necessarily exposing their representation.

[1] Email: rossberg@mpi-sws.mpg.de

doi:10.1016/j.entcs.2008.10.019

```
signature COMPLEX =
sig
  type cplx
  type base = real
  val mk : base → base → cplx
  val abs : cplx → base
  val add : cplx → cplx → cplx
end
```

```
structure C :> COMPLEX =
struct
  type cplx = real × real
  type base = real
  fun mk x y = (x, y)
  fun abs z = sqrt (sqr z.1 + sqr z.2)
  fun add z1 z2 = (z1.1 + z2.1, z1.2 + z2.2)
end
```

Fig. 1. Complex numbers in ML

These two aspects are combined in the notion of *translucency*, where module signatures can specify type components using an arbitrary mixture of opaque (abstract) and transparent (concrete) type specifications [13]. Sharing between two abstract types can be expressed by keeping one abstract, while exposing the other as manifestly equal to the first.

In their seminal paper, Mitchell & Plotkin explain type abstraction as existential quantification [20]. Subsequent work on ML-style modules has mostly followed this view, albeit with more sophisticated type theories involving dependent types or kinds. Type sharing and translucency can be modelled elegantly with *singleton kinds* [2,27]: classifying a type variable α with the singleton kind $\mathrm{S}(\tau)$ effectively says that $\alpha \equiv \tau$. Thereby, existentially quantifying over a type at singleton kind makes it transparent.

As a running example, consider the ML signature COMPLEX given in Figure 1, which specifies the interface for a rudimentary module of complex numbers. It declares an abstract type cplx for complex values, but also a concrete type base manifestly equal to real, denoting the vector base of the complex representation. Combining existential quantification with singleton kinds, this signature can be encoded with the following type:

$$\begin{aligned} \mathit{COMPLEX} \quad = \quad & \exists\alpha : \Omega \times \mathrm{S}(\mathit{real})\,. \\ & (\alpha.2 \to \alpha.2 \to \alpha.1) \times (\alpha.1 \to \alpha.2) \times (\alpha.1 \to \alpha.1 \to \alpha.1) \end{aligned}$$

This interpretation, also known as *phase splitting* [15], separates the signature into two parts: the *static* part, consisting of a (type-level) Cartesian product of all type components, and the *dynamic* part, a (term-level) product of all value components. The first component of the static part, representing type complex, has ground kind Ω, providing no information about its witness. In contrast, the second component, corresponding to the specification for base, has singleton kind $\mathrm{S}(\mathit{real})$, thus exposing the underlying definition. The tuple type describing the dynamic part can refer to both types in a uniform manner, like in the ML signature.

In Figure 1 we ascribe the signature COMPLEX to an implementation C, which hides the module's representation of type cplx, while revealing the identity of base. Following Mitchell & Plotkin, such an ascription corresponds to constructing an appropriate value $\langle \tau, e \rangle$ of existential type $\mathit{COMPLEX}$, as reflected in Figure 2.

This interpretation of type abstraction is solely static — in particular, it crucially

$$
\begin{array}{lcl}
C & = & \langle\langle \mathit{real} \times \mathit{real}, \mathit{real}\rangle, \\
 & & \ \langle \lambda x : \mathit{real}.\ \lambda y : \mathit{real}.\ \langle x, y\rangle, \\
 & & \ \ \lambda z : \mathit{real} \times \mathit{real}.\ \mathit{sqrt}\,(\mathit{sqr}\ z.1 + \mathit{sqr}\ z.2), \\
 & & \ \ \lambda z_1 : \mathit{real} \times \mathit{real}.\ \lambda z_2 : \mathit{real} \times \mathit{real}.\ \langle z_1.1 + z_2.1, z_1.2 + z_2.2\rangle\rangle \\
 & & \ \rangle\ \mathsf{as}\ \mathit{COMPLEX}
\end{array}
$$

Fig. 2. Type abstraction via existential quantification

relies on parametricity in the underlying language [21]. Without parametricity, existential abstraction is no longer safe, since clients might potentially uncover the representation of an abstract type at runtime. This is particularly apparent in languages with reflective features, such as dynamic type analysis (typecase) [1,14]. Using such a feature, we could write a function to "cast" the abstract type to its representation:

$$
\begin{array}{l}
\mathsf{let}\langle \alpha_C, x_C\rangle = C\ \mathsf{in}\ \ldots \\
\quad \mathit{cast}_C = \lambda z{:}\alpha_C.1.\ \mathsf{typecase}\, z : \alpha_C.1\ \mathsf{of}\ z' : \mathit{real} \times \mathit{real} \Rightarrow z'\ \mathsf{else}\ \langle 0, 0\rangle
\end{array}
$$

The dynamic semantics of existential elimination will simply substitute α_C with the respective witness, such that the typecase always succeeds (takes the first branch).

Unfortunately, forms of type analysis are indispensable to safely deal with *open* systems. For example, Alice ML [23] features modules as first-class *packages* suitable for dynamic import and export. When a package is *unpacked*, its signature is dynamically matched against a static annotation, to ensure type safety. A purely static interpretation of type abstraction would enable abusing this feature to write a function like cast_C.

The question we address in this paper hence is: *What is an adequate* dynamic *interpretation of type abstraction with translucency?*

Dynamic Type Generation with Translucency

A refined model for type abstraction that provides a natural *dynamic* interpretation for type abstraction and does not suffer from parametricity violations is *dynamic generativity* [22,28,8]. In this approach, type abstraction is interpreted as the generation of a fresh type name. In a formal calculus, this can be modelled by introducing a term construct $\mathsf{new}\, t \approx \tau\ \mathsf{in}\ e$ that generates a new type name and binds it to t. Within the body e the type variable t is known to be *isomorphic* to its underlying *representation* τ. However, t is not *equivalent* to τ — dynamic type analysis will properly distinguish the two. Any client code outside the original scope of the new construct has no way of discovering the isomorphism, it will just see an opaque type name.

In this paper, we extend earlier work on type generation by presenting the foundational calculus $\lambda^{\omega}_{\mathrm{SA}}$ that combines it with (higher-order) singleton kinds. This combination provides a dynamic interpretation of ML-style module types with translucency. In particular, it is sufficiently rich to form the semantic basis for dynamic extensions to module type systems, such as those found in Alice ML [23].

Abstraction Kinds

To type check the body e of an expression $\mathsf{new}\, t \approx \tau\ \mathsf{in}\ e$, the static semantics needs a way to record the isomorphism $t \approx \tau$ in the typing environment. Previous work has done so by extending typing environments in an ad-hoc manner.

A more regular and expressive approach, which we propose in this paper, is to extend the kind system: in a fashion similar to singleton kinds $\mathrm{S}(\tau)$ that classify types as *equivalent* to τ, we introduce *abstraction kinds* $\mathrm{A}(\tau)$ that classify types as *isomorphic* to τ. More precisely, in the typing rule for new, the type variable t is entered into the typing environment with kind $\mathrm{A}(\tau)$.

Analogous to singleton kinds [27], higher-order forms of abstraction kinds are encodable. This observation leads to a more regular treatment of higher-kinded abstract types than in previous work.

Coercions and Sealing

Because a generated type t and its representation τ are merely isomorphic, values in the $\lambda^{\omega}_{\mathrm{SA}}$-calculus have to be explicitly *coerced* between these types. In the simplest case, coercions apply to individual values of the abstract type. We write $\mathcal{C}^{+}_{t\approx\tau}$ for the coercion function $\tau \to t$, and $\mathcal{C}^{-}_{t\approx\tau}$ for its inverse.

However, coercing individual values is not a realistic model for a module system. Modules are typically implemented solely in terms of their representation types and then abstracted *a posteriori*, in one atomic sealing operation applied to the entire implementation. [2] A more realistic approach leads to the concept of *higher-order type coercions* [22,28] that apply to whole module implementations.

Similar notions of type coercion appear in a variety of other contexts, such as the compilation of subtyping [3], unboxing transformations [16], hybrid type systems [10], or contracts [9]. In most of these works, the host type system is comparably inexpressive. We show how type coercions generalise to a higher-order polymorphic type system with singleton and dependent kinds. The latter induce significant complications, and necessitate the definition of higher-order *kind coercions* on the level of types. Fortunately, kind coercions are encodable.

Ultimately, we are able to define a dynamic version of an ML-style module sealing operator as syntactic sugar in our system, by means of a simple combination of higher-order type generation and higher-order type coercion.

2 Basic System

In the remainder of this paper, we are going to devise the $\lambda^{\omega}_{\mathrm{SA}}$-calculus as a foundational model for a higher-order language with modules and dynamic typing. It extends the higher-order polymorphic λ-calculus with the following features:

- *Existential types* and *pair kinds* to express modules.
- *Singleton kinds*, *subkinding* and *subtyping* to express translucency.

[2] Some languages with simpler module systems, such as Haskell, actually require explicit individual coercions in the disguise of constructor applications and pattern matches.

$$
\begin{array}{lll}
\text{(base kinds)} & K & ::= \Omega \mid \mathrm{A}(\tau) \\
\text{(kinds)} & \kappa & ::= K \mid \mathrm{S}_K(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa \\
\text{(types)} & \tau & ::= \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \\
 & & \quad \mid \lambda\alpha{:}\kappa.\tau \mid \tau\tau \mid \langle \tau, \tau \rangle \mid \tau.1 \mid \tau.2 \\
\text{(terms)} & e & ::= x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e, e \rangle \mid \mathsf{let}\langle x, x \rangle = e \;\mathsf{in}\; e \\
 & & \quad \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle \tau, e \rangle \mid \mathsf{let}\langle \alpha, x \rangle = e \;\mathsf{in}_\tau\; e \\
 & & \quad \mid \mathsf{new}\,\alpha \approx \tau \;\mathsf{in}_\tau\; e \mid \mathcal{C}^+_{\tau\approx\tau} \mid \mathcal{C}^-_{\tau\approx\tau} \mid \mathsf{typecase}\,e{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}\; e
\end{array}
$$

Fig. 3. Core $\lambda^{\omega}_{\mathrm{SA}}$

- *Dynamic type analysis* to express runtime typing.
- *Type generativity* to give a dynamic interpretation to type abstraction.
- *Higher-order coercions* to express sealing.

We begin our presentation with a simplified version of the calculus that only provides basic coercions and generativity for ground types. We will add higher-order coercions and higher-order generativity in subsequent sections.

Figure 3 gives the syntax of the basic system, which we call Core $\lambda^{\omega}_{\mathrm{SA}}$. It consists of the (impredicative) higher-order λ-calculus including all canonical product constructions, extended with singleton kinds, simple type analysis, and the minimal functionality for dynamic type abstraction.

The expression form $\mathsf{new}\,\alpha \approx \tau \;\mathsf{in}_{\tau'}\; e$ generates a fresh abstract type, with representation τ. The variable α acts as an internal name for the abstract type within the body e (of α-free type τ'). At the same time, it represents a capability for performing coercions over the new type: an *upward* coercion $\mathcal{C}^+_{\alpha\approx\tau}(e)$ serves as an introduction form for values of type α, while the dual *downward* coercion $\mathcal{C}^-_{\alpha\approx\tau}(e)$ is the respective elimination form. Where clear from context, we occasionally drop the "$\approx\tau$" part from coercions to avoid notational clutter.

The last term construct expresses a simple form of non-parametric type analysis: evaluating $\mathsf{typecase}\,e_1{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_2 \;\mathsf{else}\; e_3$ compares the dynamic instantiations of type τ_1 (of e_1) with τ_2. If they match, the e_2 branch is taken, binding x to the value of e_1, otherwise e_3 is evaluated. Matching is up to subtyping, so that typecase is reminiscent of a "downcast". Much more expressive forms of type analysis can be found in literature, but this variant is sufficient to encode the type dynamic [1] as the trivial existential $\exists\alpha{:}\Omega.\alpha$ with introduction and elimination forms

$$
\begin{array}{rcl}
\mathsf{dyn}\,e : \tau & := & \langle \tau, e \rangle \\
\mathsf{check}\,e : \tau \;\mathsf{else}\; e' & := & \mathsf{let}\langle \alpha, x \rangle = e \;\mathsf{in}_\tau\; \mathsf{typecase}\,x{:}\alpha \;\mathsf{of}\; x'{:}\tau.x' \;\mathsf{else}\; e'
\end{array}
$$

Here, check works up to subtyping, and thus can explore translucency dynamically.

The type language of $\lambda^{\omega}_{\mathrm{SA}}$ is completely standard. Note that type names are simply represented by type variables. On the kind level, we distinguish between *base kinds* K, i.e. ground kind and abstraction kinds, and general kinds including singletons and standard dependent products and sums. Base singletons can only

$$
\begin{array}{l}
C' = \mathsf{new}\ \mathit{cplx} \approx \mathit{real} \times \mathit{real}\ \mathsf{in} \\
\quad \langle \langle \mathit{cplx}, \mathit{real} \rangle, \\
\quad\ \langle \lambda x : \mathit{real}.\ \lambda y : \mathit{real}.\ \mathcal{C}^+_{\mathit{cplx}} \langle x, y \rangle, \\
\quad\ \ \lambda z : \mathit{cplx}.\ \mathit{sqrt}\,(\mathit{sqr}\,(\mathcal{C}^-_{\mathit{cplx}} z).1 + \mathit{sqr}\,(\mathcal{C}^-_{\mathit{cplx}} z).2), \\
\quad\ \ \lambda z_1 : \mathit{cplx}.\ \lambda z_2 : \mathit{cplx}.\ \mathcal{C}^+_{\mathit{cplx}} \langle (\mathcal{C}^-_{\mathit{cplx}} z_1).1 + (\mathcal{C}^-_{\mathit{cplx}} z_2).1, (\mathcal{C}^-_{\mathit{cplx}} z_1).2 + (\mathcal{C}^-_{\mathit{cplx}} z_2).2 \rangle \rangle \\
\quad \rangle
\end{array}
$$

Fig. 4. Type abstraction via type generativity

be formed over base kinds. Higher-order singletons are obtained following Stone & Harper's definition [27], with the added case $\mathrm{S}(\tau : \mathrm{A}(\tau')) := \mathrm{S}_{\mathrm{A}(\tau')}(\tau)$ for singletons at abstraction kind. [3]

Figure 4 shows how our module of complex numbers can be expressed using Core $\lambda^{\omega}_{\mathrm{SA}}$: it still is an existential package, but this time the abstract type name is represented by a fresh type name *cplx* that is used as a witness for $\alpha.1$. To match the "signature" type *COMPLEX*, all operations have to use coercions for mitigating between *cplx* and its internal representation $\mathit{real} \times \mathit{real}$.

2.1 Static Semantics

The static semantics of $\lambda^{\omega}_{\mathrm{SA}}$ consists of eight standard judgements:

(environment formation)	$\Gamma \vdash \Box$	(kind equivalence)	$\Gamma \vdash \kappa \equiv \kappa' : \Box$
(kind formation)	$\Gamma \vdash \kappa : \Box$	(type equivalence)	$\Gamma \vdash \tau \equiv \tau' : \kappa$
(type formation)	$\Gamma \vdash \tau : \kappa$	(kind inclusion)	$\Gamma \vdash \kappa \leq \kappa' : \Box$
(term formation)	$\Gamma \vdash e : \tau$	(type inclusion)	$\Gamma \vdash \tau \leq \tau' : \kappa$

Most of the rules defining these judgements are entirely standard. In particular, rules regarding singletons and extensionality are taken almost verbatim from Stone & Harper [27]. The full type system can be found in the Appendix, here we focus on the parts that are particular to our calculus.

Figure 5 shows the central rules. Abstraction kinds are formed like singleton kinds, except that we in fact generalise singleton formation to arbitrary base kinds. The same generalisation applies to the singleton introduction rule TEXT-SING. Note that there is no introduction or elimination rule for abstraction kinds. Types of abstraction kind are only introduced via new (i.e., there are no closed types of abstraction kind), and like singletons, abstraction kinds are eliminated simply by kind subsumption (rule KSABS-LEFT).

For terms, we find straightforward rules for type generation and type analysis. Coercions require the annotated type to have abstraction kind, and are assigned cor-

[3] Note that it is crucial to distinguish $\mathrm{S}_{\mathrm{A}(\tau')}(\tau)$ from $\mathrm{S}_{\Omega}(\tau)$. Conflating the two would necessitate a subkinding rule that allows deriving $\mathrm{S}(\tau) \leq \mathrm{A}(\tau')$ if $\tau : \mathrm{A}(\tau')$. However, such a rule would enable recovering abstraction kind dynamically for a plain abstract type name (of static kind Ω) — and thus break abstraction safety. E.g. consider $\lambda m{:}(\exists \alpha{:}\Omega.\alpha \to \mathit{int}).\ \mathsf{let}\langle \alpha, f \rangle = m\ \mathsf{in}\ \mathsf{typecase}\langle \alpha, f \rangle : (\exists \beta{:}\mathrm{S}(\alpha).\beta \to \mathit{int})\ \mathsf{of}\ m' : (\exists \beta{:}\mathrm{A}(\mathit{int}).\beta \to \mathit{int}).\ \mathsf{let}\langle \beta, f' \rangle = m'\ \mathsf{in}\ f'\ (\mathcal{C}^+_{\beta}\ 0) \ldots$ applied to $(\mathsf{new}\ \alpha \approx \mathit{int}\ \mathsf{in}_{\exists \alpha:\Omega.\alpha \to \mathit{int}}\ \langle \alpha, \lambda x{:}\alpha.\ 1/(\mathcal{C}^-_{\alpha}\ x) \rangle)$.

Kind Formation $(\Gamma \vdash \kappa : \Box)$

$$\textsc{Kabs}\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) : \Box} \qquad \textsc{Ksing}\ \frac{\Gamma \vdash \tau : K}{\Gamma \vdash \mathrm{S}_K(\tau) : \Box}$$

Type Formation $(\Gamma \vdash \tau : \kappa)$

$$\textsc{Text-sing}\ \frac{\Gamma \vdash \tau : K}{\Gamma \vdash \tau : \mathrm{S}_K(\tau)}$$

Term Formation $(\Gamma \vdash e : \tau)$

$$\textsc{Enew}\ \frac{\Gamma, \alpha{:}\mathrm{A}(\tau_1) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2} \qquad \textsc{Eclose}\ \frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash \langle \tau, e \rangle : \exists \alpha{:}\kappa.\tau_2}$$

$$\textsc{Eup}\ \frac{\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \mathcal{C}^{+}_{\tau_1 \approx \tau_2} : \tau_2 \to \tau_1} \qquad \textsc{Edown}\ \frac{\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \mathcal{C}^{-}_{\tau_1 \approx \tau_2} : \tau_1 \to \tau_2}$$

$$\textsc{Ecase}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{typecase}\, e_1{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}_\tau\ e_3 : \tau}$$

Kind Equivalence $(\Gamma \vdash \kappa \equiv \kappa' : \Box)$

$$\textsc{KQabs}\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \Box} \qquad \textsc{KQsing}\ \frac{\Gamma \vdash \tau \equiv \tau' : K \qquad \Gamma \vdash K \equiv K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \equiv \mathrm{S}_{K'}(\tau') : \Box}$$

Type Equivalence $(\Gamma \vdash \tau \equiv \tau' : \kappa)$

$$\textsc{TQext-sing}\ \frac{\Gamma \vdash \tau : \mathrm{S}_K(\tau'') \qquad \Gamma \vdash \tau' : \mathrm{S}_K(\tau'')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_K(\tau'')}$$

Kind Inclusion $(\Gamma \vdash \kappa \leq \kappa' : \Box)$

$$\textsc{KSabs}\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \mathrm{A}(\tau') : \Box} \qquad \textsc{KSsing}\ \frac{\Gamma \vdash \tau \equiv \tau' : K \qquad \Gamma \vdash K \leq K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \leq \mathrm{S}_{K'}(\tau') : \Box}$$

$$\textsc{KSabs-left}\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \Omega : \Box} \qquad \textsc{KSsing-left}\ \frac{\Gamma \vdash \tau : K \qquad \Gamma \vdash K \leq K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \leq K' : \Box}$$

Type Inclusion $(\Gamma \vdash \tau \leq \tau' : \kappa)$

$$\textsc{TSuniv}\ \frac{\Gamma \vdash \kappa' \leq \kappa : \Box \qquad \Gamma, \alpha{:}\kappa' \vdash \tau \leq \tau' : \Omega \qquad \Gamma \vdash \forall \alpha{:}\kappa.\tau : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \leq \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$\textsc{TSexist}\ \frac{\Gamma \vdash \kappa \leq \kappa' : \Box \qquad \Gamma, \alpha{:}\kappa \vdash \tau \leq \tau' : \Omega \qquad \Gamma \vdash \exists \alpha{:}\kappa'.\tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \leq \exists \alpha{:}\kappa'.\tau' : \Omega}$$

Fig. 5. Selected typing rules for Core $\lambda^{\omega}_{\mathrm{SA}}$

responding function type. As a technical detail, existential formation (rule ECLOSE) does not require a type annotation: thanks to subtyping (explained below), assigning singleton kind to the witness τ allows deriving a (fully transparent) existential type that is principal. The burden of annotation is shifted to new-expressions, which need to be given a τ_2 that avoids the local type variable α.

Equivalence for types and kinds bears no surprises, subtyping and subkinding are more interesting: they enable forgetting singleton kinds (KSSING-LEFT) and abstraction kinds (KSABS-LEFT). A type of abstraction kind may thus be used at kind Ω, which allows the type to act as an actual abstract type name, beyond its use as a coercion key. For singletons, subsumption can go to the respective index kind, or a superkind of it, thereby forgetting the type identity. Sub*typing* then simply lifts the subkinding relation to quantified types, and ultimately, the entire type language (by standard rules). This realises translucent signature matching. For example, $\exists\alpha{:}\mathrm{S}_\Omega(\mathit{int}).\alpha \times \alpha \leq \exists\alpha{:}\Omega.\alpha \times \mathit{int}$ is derivable.

2.2 Dynamic Semantics

Figure 6 defines a small-step operational semantics for Core $\lambda^{\omega}_{\mathrm{SA}}$. Values are defined as a subset of expressions as usual and consist of λ-abstractions, tuples, existentials and values coerced to abstract type. Reduction is on *configurations* C, which are expressions paired with a *heap*. Heaps capture the type names generated with new as an ordered list of type variables classified by abstraction kind. Evaluating a new-expression pushes the bound type variable on the heap, taking advantage of the usual variable convention for freshness. The only rule for coercions is cancellation; note that soundness ensures that $\Delta \vdash \tau_- \equiv \tau'_- : \Omega$ and $\Delta \vdash \tau_+ \equiv \tau'_+ : \mathrm{A}(\tau_-)$.

Type analysis uses the subtyping judgement from the static semantics to check whether τ_1 matches τ_2. Since an evaluation context E cannot bind variables, both types are guaranteed to be closed up to type variables from the heap Δ.

3 Higher-order Coercions

Core $\lambda^{\omega}_{\mathrm{SA}}$ has obvious limitations: coercions have to be used all over an implementation, which has to have a handle on the abstract type name *beforehand*. In contrast, modules typically allow type abstraction *after the fact*, outside the actual implementation. How can we faithfully recover that flexibility?

The key is generalising the notion of coercion: instead of performing coercions on individual values of the abstract type, we allow *higher-order coercions*, which apply to any type. Figure 7 shows the syntax and rules of such coercions: where a basic coercion $\mathcal{C}^{+}_{\tau_+ \approx \tau_-}$ just represents a function of type $\tau_- \to \tau_+$, a higher-order coercion $\mathcal{C}^{+\alpha.\tilde{\tau}}_{\tau_+ \approx \tau_-}$ is a function of arbitrary type $\tilde{\tau}[\tau_-/\alpha] \to \tilde{\tau}[\tau_+/\alpha]$ — similarly for the downward directions. Basic coercions arise as the special cases $\mathcal{C}^{\pm\alpha.\alpha}_{\tau_+ \approx \tau_-}$. Here, *concrete* types $\tilde{\tau}$ are a syntactic subset of types that we will explain in Section 3.2. Types instantiating polymorphic variables are also restricted to be concrete.

$$
\begin{array}{lll}
\text{(values)} & v & ::= \lambda x{:}\tau.e \mid \langle v, v\rangle \mid \lambda\alpha{:}\kappa.e \mid \langle \tau, v\rangle \mid \mathcal{C}^{+}_{\tau\approx\tau}(v) \\
\text{(contexts)} & E & ::= \_ \mid E\,e \mid v\,E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathsf{let}\langle x, x\rangle = E \;\mathsf{in}\; e \mid E\,\tau \mid \langle \tau, E\rangle \\
 & & \quad \mid \mathsf{let}\langle \alpha, x\rangle = E \;\mathsf{in}_\tau\; e \mid \mathsf{typecase}\, E{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}\; e \\
\text{(heaps)} & \Delta & ::= \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tau) \\
\text{(configurations)} & C & ::= \Delta; e
\end{array}
$$

$$
\begin{array}{lll}
(\textsc{Rapp}) & \Delta; E[(\lambda x{:}\tau.e)\,v] & \to \Delta; E[e[v/x]] \\
(\textsc{Rproj}) & \Delta; E[\mathsf{let}\langle x_1, x_2\rangle{=}\langle v_1, v_2\rangle \;\mathsf{in}\; e] & \to \Delta; E[e[v_1/x_1][v_2/x_2]] \\
(\textsc{Rinst}) & \Delta; E[(\lambda\alpha{:}\kappa.e)\,\tau] & \to \Delta; E[e[\tau/\alpha]] \\
(\textsc{Ropen}) & \Delta; E[\mathsf{let}\langle \alpha, x\rangle{=}\langle \tau, v\rangle \;\mathsf{in}_{\tau'}\; e] & \to \Delta; E[e[\tau/\alpha][v/x]] \\
(\textsc{Rnew}) & \Delta; E[\mathsf{new}\;\alpha{\approx}\tau \;\mathsf{in}_{\tau'}\; e] & \to \Delta, \alpha{:}\mathrm{A}(\tau); E[e] \\
(\textsc{Rcancel}) & \Delta; E[\mathcal{C}^{-}_{\tau_+\approx\tau_-}(\mathcal{C}^{+}_{\tau'_+\approx\tau'_-} v)] & \to \Delta; E[v] \\
(\textsc{Rcase1}) & \Delta; E[\mathsf{typecase}\, v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_1[v/x]] \;\text{if}\; \Delta \vdash \tau_1 \le \tau_2 : \Omega \\
(\textsc{Rcase2}) & \Delta; E[\mathsf{typecase}\, v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_2] \quad\; \text{if}\; \Delta \nvdash \tau_1 \le \tau_2 : \Omega
\end{array}
$$

Fig. 6. Reduction for Core $\lambda^{\omega}_{\mathrm{SA}}$

$$
\begin{array}{lll}
\text{(expressions)} & e & ::= \cdots \mid e\,\tilde{\tau} \mid \langle \tilde{\tau}, e\rangle \mid \mathcal{C}^{+\alpha.\tilde{\tau}}_{\tau\approx\tau} \mid \mathcal{C}^{-\alpha.\tilde{\tau}}_{\tau\approx\tau} \\
\text{(values)} & v & ::= \cdots \mid \langle \tilde{\tau}, v\rangle \mid \mathcal{C}^{+\alpha.\alpha}_{\tau\approx\tau}(v)
\end{array}
$$

$$
\textsc{Eup}, \frac{\Gamma \vdash \tau_+ : \mathrm{A}(\tau_-) \qquad \Gamma, \alpha{:}\Omega \vdash \tilde{\tau} : \Omega}{\Gamma \vdash \mathcal{C}^{+\alpha.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha] \to \tilde{\tau}[\tau_+/\alpha]}
\qquad
\textsc{Edown}, \frac{\Gamma \vdash \tau_+ : \mathrm{A}(\tau_-) \qquad \Gamma, \alpha{:}\Omega \vdash \tilde{\tau} : \Omega}{\Gamma \vdash \mathcal{C}^{-\alpha.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_+/\alpha] \to \tilde{\tau}[\tau_-/\alpha]}
$$

$$
\begin{array}{lll}
(\textsc{Rarrow}) & \mathcal{C}^{\pm\alpha.\tau_1\to\tau_2}_{\tau_+\approx\tau_-}(v) & \to \lambda x_1{:}\tau_1[\tau_\pm/\alpha].\, \mathcal{C}^{\pm\alpha.\tau_2}_{\tau_+\approx\tau_-}(v\,(\mathcal{C}^{\mp\alpha.\tau_1}_{\tau_+\approx\tau_-} x_1)) \\
(\textsc{Rtimes}) & \mathcal{C}^{\pm\alpha.\tau_1\times\tau_2}_{\tau_+\approx\tau_-}(v) & \to \mathsf{let}\langle x_1, x_2\rangle = v \;\mathsf{in}\; \langle \mathcal{C}^{\pm\alpha.\tau_1}_{\tau_+\approx\tau_-} x_1, \mathcal{C}^{\pm\alpha.\tau_2}_{\tau_+\approx\tau_-} x_2\rangle \\
(\textsc{Runiv}) & \mathcal{C}^{\pm\alpha.\forall\beta:\kappa.\tau}_{\tau_+\approx\tau_-}(v) & \to \lambda\beta{:}\kappa[\tau_\pm/\alpha].\, \mathcal{C}^{\pm\alpha.\tau'}_{\tau_+\approx\tau_-}(v\,(\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta)) \\
 & & \quad \text{where } \tau' = \tau[(\mathcal{T}^{\alpha.\kappa}_{\alpha/\tau_\pm}\beta)/\beta] \\
(\textsc{Rexist}) & \mathcal{C}^{\pm\alpha.\exists\beta:\kappa.\tau}_{\tau_+\approx\tau_-}(v) & \to \mathsf{let}\langle \beta, x\rangle = v \;\mathsf{in}\; \langle \mathcal{T}^{\alpha.\kappa}_{\tau_\pm/\tau_\mp}\beta, \mathcal{C}^{\pm\alpha.\tau'}_{\tau_+\approx\tau_-} x\rangle \\
 & & \quad \text{where } \tau' = \tau[(\mathcal{T}^{\alpha.\kappa}_{\alpha/\tau_\mp}\beta)/\beta] \\
(\textsc{Rdrop}) & \mathcal{C}^{\pm\alpha.\beta}_{\tau_+\approx\tau_-}(v) & \to v \quad \text{if } \beta \neq \alpha \\
(\textsc{Rcancel}) & \mathcal{C}^{-\alpha.\alpha}_{\tau_+\approx\tau_-}(\mathcal{C}^{+\alpha.\alpha}_{\tau'_+\approx\tau'_-} v) & \to v
\end{array}
$$

Fig. 7. Extensions for higher-order coercions (reduction rules omit surrounding context $\Delta; E[\_]$)

3.1 Reduction

Higher-order coercions $\mathcal{C}^{\pm\alpha.\tau}_{\tau_+\approx\tau_-}(v)$ are reduced to basic coercions via the reduction rules given in Figure 7. Reduction is type-directed, driven by the *residual type* τ of the coercion. We assume that τ is implicitly normalised [27]. Essentially, the rules proceed by functorially mapping the coercion over the residual type, i.e., η-expanding their argument and pushing the coercion inwards. For monomorphic

types this approach is fairly standard. For quantified types however, the presence of dependent kinds requires a bit more thought.

Without dependent kinds, rule RCOERCE-UNIV might simply look as follows [22]:

$$\mathcal{C}^{\pm\alpha.\forall\beta{:}\kappa.\tau}_{\tau_+\approx\tau_-}(v) \quad \to \quad \lambda\beta{:}\kappa.\,\mathcal{C}^{\pm\alpha.\tau}_{\tau_+\approx\tau_-}(v\,\beta)$$

In $\lambda^{\omega}_{\mathrm{SA}}$ this rule is incorrect however, because α may occur free in κ. One might think that it is enough to simply substitute τ_+ or τ_- for α (depending on direction), but unfortunately that is not sufficient. Consider the coercion $\mathcal{C}^{+\alpha.\forall\beta:\mathrm{S}_\Omega(\alpha).\beta}_{\tau_+\approx\tau_-}(f)$: its reduct $\lambda\beta{:}\mathrm{S}_\Omega(\tau_+).\,\mathcal{C}^{+\alpha.\beta}_{\tau_+\approx\tau_-}(f\,\beta)$ would be ill-typed, because the instantiation $f\,\beta$ is not valid — f has type $\forall\beta{:}\mathrm{S}_\Omega(\tau_-).\beta$ (by rule EUP'), which clashes with the incompatible kind $\mathrm{S}_\Omega(\tau_+)$ assigned to the λ-bound β.

In general, we need to instantiate the polymorphic value with a type of kind $\kappa[\tau_-/\alpha]$ in the reduct, which we have to construct from β of kind $\kappa[\tau_+/\alpha]$. In other words, we have to *coerce* β from the latter kind to the former. We write such a type-level coercion as $\mathcal{T}^{\alpha.\kappa}_{\tau_+/\tau_-}(\tau)$ ($\mathcal{T}^{\alpha.\kappa}_{\tau_-/\tau_+}(\beta)$ here, due to its contravariant position).

Assuming kind coercions given for a moment, we can try reducing to $\lambda\beta{:}\kappa[\tau_\pm/\alpha].\,\mathcal{C}^{\pm\alpha.\tau}_{\tau_+\approx\tau_-}(v\,(\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta))$. This makes the instantiation well-formed. Unfortunately, inserting a kind coercion for the argument now changes the resulting type of the instantiation: the type of $v\,(\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta)$ is $\tau[\tau_\mp/\alpha][\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta/\beta]$ — but the surrounding coercion expects $\tau[\tau_\mp/\alpha]$.

This final problem can be fixed if we find a suitable residual type τ' in the surrounding coercion that fulfills the following equations:

$$\tau'[\tau_\pm/\alpha] \equiv \tau[\tau_\pm/\alpha] \qquad (1)$$
$$\tau'[\tau_\mp/\alpha] \equiv \tau[\tau_\mp/\alpha][(\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta)/\beta] \qquad (2)$$

With such a type, the coercion yields type $\tau[\tau_\mp/\alpha][(\mathcal{T}^{\alpha.\kappa}_{\tau_\mp/\tau_\pm}\beta)/\beta] \to \tau[\tau_\pm/\alpha]$, making the entire reduct well-formed and type-preserving. With the admissible rules for kind coercions that we will give in the next section, it can be shown that the type τ' used in Figure 7 indeed has this property: it contains a kind coercion to the actual placeholder α of the surrounding coercion — we call that a *placeholder-targetting* coercion. A similar trick is employed in rule RCOERCE-EXIST.

3.2 Kind Coercions

Figure 8 gives the definition of kind coercions by induction over the *residual kind* κ. Unlike types, the shapes of kinds are invariant under substitution, such that kind coercions can be expanded statically and thus be treated as syntactic sugar. Otherwise, the definition closely resembles the reduction rules for type coercions. In particular, it employs placeholder-targetting coercions in the same manner.

$$
\begin{array}{lll}
\mathcal{T}^{\alpha.\Omega}_{\tau_+/\tau_-}(\tau) & := \tau & \\
\mathcal{T}^{\alpha.\mathrm{S}_\Omega(\tau')}_{\tau_+/\tau_-}(\tau) & := \tau'[\tau_+/\alpha] & \\
\mathcal{T}^{\alpha.\Pi\beta{:}\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}(\tau) & := \lambda\beta{:}\tilde\kappa_1[\tau_+/\alpha].\mathcal{T}^{\alpha.\tilde\kappa_2'}_{\tau_+/\tau_-}(\tau\,(\mathcal{T}^{\alpha.\tilde\kappa_1}_{\tau_-/\tau_+}\beta)) & \text{where } \tilde\kappa_2' = \tilde\kappa_2[(\mathcal{T}^{\alpha.\tilde\kappa_1}_{\alpha/\tau_+}\beta)/\beta] \\
\mathcal{T}^{\alpha.\Sigma\beta{:}\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}(\tau) & := \langle \mathcal{T}^{\alpha.\tilde\kappa_1}_{\tau_+/\tau_-}(\tau.1), \mathcal{T}^{\alpha.\tilde\kappa_2'}_{\tau_+/\tau_-}(\tau.2)\rangle & \text{where } \tilde\kappa_2' = \tilde\kappa_2[(\mathcal{T}^{\alpha.\tilde\kappa_1}_{\alpha/\tau_-}(\tau.1))/\beta]
\end{array}
$$

Fig. 8. Kind coercions

The following formation and equivalence rules can be shown admissible:

$$
\textsc{Tcoerce}^*\ \frac{\Gamma, \alpha{:}\Omega \vdash \tilde\kappa : \Box \qquad \Gamma \vdash \tau_- : \Omega \qquad \Gamma \vdash \tau_+ : \Omega}{\Gamma \vdash \mathcal{T}^{\alpha.\tilde\kappa}_{\tau_+/\tau_-} : \tilde\kappa[\tau_-/\alpha] \to \tilde\kappa[\tau_+/\alpha]}
$$

$$
\textsc{TQdrop}^*\ \frac{\Gamma \vdash \tau : \tilde\kappa[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\Omega \vdash \tilde\kappa : \Box \qquad \Gamma \vdash \tau_+ \equiv \tau_- : \Omega}{\Gamma \vdash \mathcal{T}^{\alpha.\tilde\kappa}_{\tau_+/\tau_-}(\tau) \equiv \tau : \tilde\kappa[\tau_-/\alpha]}
$$

The most visible difference with the formation rule Eup' for type coercions is that *source* type τ_- and *target* type τ_+ need not be related by an abstraction kind, because kind coercions do not have to reduce to a basic form. Consequently, kind coercions need no polarity, the inverse of $\mathcal{T}^{\alpha.\tilde\kappa}_{\tau_+/\tau_-}$ is simply $\mathcal{T}^{\alpha.\tilde\kappa}_{\tau_-/\tau_+}$. The equivalence rule is crucial to equation (1) above, and thus to the correctness of placeholder-targetting coercions. Also note that kind coercions (obviously) commute with substitution.

One important restriction on kind coercions is that they are not definable for residual abstraction kinds, because those lack the necessary extensionality principle. We call the syntactic subclasses of kinds and types containing no syntactic occurrences of abstraction kinds *concrete*, and use the meta variables $\tilde\kappa$ and $\tilde\tau$ to range over them. Because all coercions with non-concrete residual kind (or type) would eventually expand to a kind coercion over abstraction kind, we have to restrict the use of non-concrete residuals, as implemented by the grammar in Figure 7. [4]

4 Higher-order Generativity

So far, our language only allows the generation of ground types. Figure 9 gives the main extensions to Full $\lambda^{\omega}_{\mathrm{SA}}$, necessary to support higher-kinded type generativity.

The extensions to the syntax are straightforward: all occurrences of type names are now annotated with a (concrete) kind. Accordingly, the respective typing rules need to generalise the kinds assigned to these variables: instead of plain abstraction kind $\mathrm{A}(\tau)$, they now have to assign *higher-order abstraction kind* $\mathrm{A}(\tau : \tilde\kappa)$.

[4] It would be possible to encompass kind coercions at abstraction kind, but only for the price of making them primitive. However, since abstraction kinds are "internal" and never show up in an encoding of module signatures, the restriction is not substantial.

$$
\begin{array}{lll}
\text{(expressions)} & e & ::= \cdots \mid \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau\ \mathsf{in}_\tau\ e \mid \mathcal{C}^{+\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} \mid \mathcal{C}^{-\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} \\
\text{(values)} & v & ::= \cdots \mid \mathcal{C}^{+\alpha:\Omega.\alpha}_{\tau_+\approx\tau_-}(v) \\
\text{(path contexts)} & P & ::= \_ \mid P\,\tau \mid P.1 \mid P.2 \\
\text{(heaps)} & \Delta & ::= \cdot \mid \alpha{:}\mathrm{A}(\tau : \tilde{\kappa})
\end{array}
$$

$$
\textsc{New}',\ \frac{\Gamma \vdash \tau_1 : \tilde{\kappa} \qquad \Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2}
$$

$$
\textsc{Up}',\ \frac{\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})}{\Gamma \vdash \mathcal{C}^{+\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha] \to \tilde{\tau}[\tau_+/\alpha]}
$$

$$
\textsc{Down}',\ \frac{\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})}{\Gamma \vdash \mathcal{C}^{-\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_+/\alpha] \to \tilde{\tau}[\tau_-/\alpha]}
$$

$$
\begin{array}{rcl}
\Delta; E[\mathsf{new}\,\alpha{:}\tilde{\kappa}{\approx}\tau\ \mathsf{in}\ e] & \to & \Delta, \alpha{:}\mathrm{A}(\tau : \tilde{\kappa}); E[e] \\
\mathcal{C}^{\pm\alpha:\tilde{\kappa}.P[\beta]}_{\tau_+\approx\tau_-}(v) & \to & \mathcal{C}^{+\alpha':\tilde{\kappa}'.P[\alpha'/\beta:\tilde{\kappa}'][\tau_\pm/\alpha]}_{\beta\approx\tau'}\mathcal{C}^{\pm\alpha:\tilde{\kappa}.P[\tau'/\beta:\tilde{\kappa}']}_{\tau_+\approx\tau_-}\mathcal{C}^{-\alpha':\tilde{\kappa}'.P[\alpha'/\beta:\tilde{\kappa}'][\tau_\mp/\alpha]}_{\beta\approx\tau'}(v) \\
 & & \text{where } \beta \neq \alpha \text{ and } \Delta(\beta) = \mathrm{A}(\tilde{\tau}' : \tilde{\kappa}') \\
\mathcal{C}^{\pm\alpha:\tilde{\kappa}.P[\alpha]}_{\tau_+\approx\tau_-}(v) & \to & \mathcal{C}^{\pm\alpha:\Omega.\alpha}_{P[\tau_+/\alpha:\tilde{\kappa}][\tau_\pm/\alpha]\approx P[\tau_-/\alpha:\tilde{\kappa}][\tau_\pm/\alpha]}\mathcal{C}^{\pm\alpha:\tilde{\kappa}.P[\tau_\mp/\alpha:\tilde{\kappa}]}_{\tau_+\approx\tau_-}(v) \\
 & & \text{where } P \neq \_
\end{array}
$$

Fig. 9. Extensions for higher-order type generation (omitted $\Delta; E[\_]$ for coercion reduction rules)

$$
\begin{array}{rcl}
\mathrm{A}(\tau : \Omega) & := & \mathrm{A}(\tau) \\
\mathrm{A}(\tau : \mathrm{S}_\Omega(\tau')) & := & \mathrm{S}_\Omega(\tau') \\
\mathrm{A}(\tau : \Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Pi\alpha{:}\tilde{\kappa}_1.\mathrm{A}(\tau\,\alpha : \tilde{\kappa}_2) \\
\mathrm{A}(\tau : \Sigma\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Sigma\alpha{:}\mathrm{A}(\tau.1 : \tilde{\kappa}_1).\,\mathrm{A}(\mathcal{T}^{\alpha:\tilde{\kappa}_1.\tilde{\kappa}_2}_{\alpha/\tau.1}(\tau.2) : \tilde{\kappa}_2)
\end{array}
$$

Fig. 10. Higher-order abstraction kinds

4.1 Higher-order Abstraction Kinds

Lifting abstraction kinds to higher order does not require any extension to the type or kind language — Figure 10 shows how they are definable. The definition closely mirrors that of higher-order singleton kinds [27], with the only difference showing up in the Σ case: here, a single abstraction is split into two separate ones, one for each component. Since the second component may refer to the first, it has to cross the abstraction barrier of the first via a suitable kind coercion targetting α.

More concretely, higher-order abstraction kinds must be defined such that

higher-order generalisations of the rules from Figure 5 are admissible:

$$\textsc{KAbs}^* \ \frac{\Gamma \vdash \tau : \tilde\kappa}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) : \Box} \qquad \textsc{KQAbs}^* \ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde\kappa \quad \Gamma \vdash \tilde\kappa \equiv \tilde\kappa' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \equiv \mathrm{A}(\tau' : \tilde\kappa') : \Box}$$

$$\textsc{KSAbs-left}^* \ \frac{\Gamma \vdash \tau : \tilde\kappa}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \leq \tilde\kappa : \Box} \qquad \textsc{KSAbs}^* \ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde\kappa \quad \Gamma \vdash \tilde\kappa \equiv \tilde\kappa' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \leq \mathrm{A}(\tau' : \tilde\kappa') : \Box}$$

In particular, an abstraction kind must be a subkind of its index (KSabs-left*). Now consider $\kappa = \mathrm{A}(\tau : \Sigma\alpha{:}\Omega.\, \mathrm{S}_\Omega(\alpha) \to \Omega)$ with $\tau = \langle \mathit{int}, \lambda\beta{:}\mathrm{S}_\Omega(\mathit{int}).\beta \rangle$. According to the definition, it expands to $\Sigma\alpha{:}\mathrm{A}(\mathit{int}).\, \Pi\beta{:}\mathrm{S}_\Omega(\alpha).\, \mathrm{A}(\beta)$ (modulo $\beta\eta$-reduction of constituent types). Clearly, this is a subkind of the original index, whereas other obvious candidates for the expansion of $\mathrm{A}(\tau : \Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2)$ (such as $\Sigma\alpha{:}\mathrm{A}(\tau.1 : \tilde\kappa_1).\mathrm{A}(\tau.2 : \tilde\kappa_2)$ or $\Sigma\alpha{:}\mathrm{A}(\tau.1 : \tilde\kappa_1).\mathrm{A}(\tau.2 : \tilde\kappa_2[\tau.1/\alpha])$) would fail to produce one.

4.2 Reduction of Higher-kinded Type Coercions

One interesting consequence of the way we define higher-order abstraction kinds is that we can keep the notion of value unchanged as well: abstract types can already be denoted by arbitrary type expressions τ_+, so that basic coercions subsume higher-kinded abstractions. To see why, consider an abstract type $\mathit{stack} : \mathrm{A}(\mathit{list} : \Omega \to \Omega)$. This kind decomposes into $\Pi\alpha{:}\Omega.\mathrm{A}(\mathit{list}\ \alpha)$. Consequently, a stack of integers can be formed as the value $s = \mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{stack\ int} \approx \mathit{list\ int}}[5, 2, 7]$ right away. Here, $\tau_+ = \mathit{stack\ int}$ is a type *path*, not just a name, but yet has abstraction kind $\mathrm{A}(\mathit{list\ int})$.

What we can *not* express with basic coercions is *further* abstracting the element type of s with respect to $\mathit{date} \approx \mathit{int}$, as in $\mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{stack\ date} \approx \mathit{stack\ int}}(s)$ — this application is ill-formed, because *stack date* has kind $\mathrm{A}(\mathit{list\ date})$, not $\mathrm{A}(\mathit{stack\ int})$. We need a higher-order coercion $\mathcal{C}^{+\alpha:\Omega.\mathit{stack}\,\alpha}_{\mathit{date} \approx \mathit{int}}(s)$ instead, where α appears in nested position.

The two additional reduction rules for coercions given in Figure 10 deal with cases like this, where the residual type is an abstract path $P[\alpha]$. For now, you should read occurrences of path replacement $P[\tau/\alpha : \kappa]$ in the rules as $P[\tau]$.

The first rule treats the case where the head of the path is another abstract type β. It looks up β's representation τ' on the heap, inserts an auxiliary coercion to τ', performs the actual coercion on the representation, and finally redoes the coercion to β. For example, $\mathcal{C}^{+\alpha:\Omega.\mathit{stack}\,\alpha}_{\mathit{date} \approx \mathit{int}}(s)$ will be split into three simpler coercions, $\mathcal{C}^{+\alpha':\Omega\to\Omega.\,\alpha'\ \mathit{date}}_{\mathit{stack} \approx \mathit{list}}(\mathcal{C}^{+\alpha:\Omega.\,\mathit{list}\,\alpha}_{\mathit{date} \approx \mathit{int}}(\mathcal{C}^{-\alpha':\Omega\to\Omega.\,\alpha'\ \mathit{int}}_{\mathit{stack} \approx \mathit{list}}(s)))$, and eventually become the properly staged $\mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{stack\ date} \approx \mathit{list\ date}}[\mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{date} \approx \mathit{int}}(5), \mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{date} \approx \mathit{int}}(2), \mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{date} \approx \mathit{int}}(7)]$.

The second rule is more subtle. It applies whenever we coerce at a (non-trivial) path $P[\alpha]$ headed by the placeholder variable itself, i.e. when we do a coercion that is higher-order and higher-kinded at the same time. Intuitively, the rule implements two simultaneous simplifications: (1) it *splits* the coercion into two, first coercing the type arguments (occurrences of α in P) and then the head of the residual type, and (2) it *grounds* the head coercion by lifting path information from the residual type to the abstract type itself, yielding a residual of ground kind Ω. For example, $\mathcal{C}^{+\alpha:(\Omega\to\Omega).\alpha\,(\alpha\ \mathit{int})}_{\mathit{stack} \approx \mathit{list}}([[7]])$ will produce $\mathcal{C}^{+\alpha:\Omega.\alpha}_{\mathit{stack}\ (\mathit{stack\ int}) \approx \mathit{list}\ (\mathit{stack\ int})}(\mathcal{C}^{+\alpha:(\Omega\to\Omega).\mathit{list}\ (\alpha\ \mathit{int})}_{\mathit{stack} \approx \mathit{list}}([[7]]))$, where the inner coercion

$$
\begin{array}{ll}
P[\tau_+/\tau_- : \tilde\kappa] & := P[^{\alpha:\tilde\kappa.\tilde\kappa}_{\tau_+/\tau_-}\alpha] \\
\_[^{\alpha:\tilde\kappa.\Omega}_{\tau_+/\tau_-}\tau] & := \tau[\tau_+/\alpha] \\
\_[^{\alpha:\tilde\kappa.\mathrm{S}_\Omega(\tau')}_{\tau_+/\tau_-}\tau] & := \tau[\tau_+/\alpha] \\
P[\_\,\tau'][^{\alpha:\tilde\kappa.\Pi\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := P[^{\alpha:\tilde\kappa.\tilde\kappa_2[\tau''/\alpha_1]}_{\tau_+/\tau_-}\tau\,\tau''] \text{ where } \tau'' = \mathcal{T}^{\alpha:\tilde\kappa.\tilde\kappa_1}_{\alpha/\tau_-}(\tau') \\
P[\_.1][^{\alpha:\tilde\kappa.\Sigma\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := P[^{\alpha:\tilde\kappa.\tilde\kappa_1}_{\tau_+/\tau_-}\tau.1] \\
P[\_.2][^{\alpha:\tilde\kappa.\Sigma\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := P[^{\alpha:\tilde\kappa.\tilde\kappa_2[\tau.1/\alpha_1]}_{\tau_+/\tau_-}\tau.2]
\end{array}
$$

Fig. 11. Coercive path replacement

treats the inner list, and the outer grounds the modified residual type by lifting $P[\_] = \_\,(\mathit{stack}\ \mathit{int})$ to the abstract type. With this rule, *all* coercions can ultimately be brought into ground form, which is in contrast to previous work [22,28].

Once more, singletons create a complication: it is not generally correct to split the coercion by simply replacing only the head of the path, even though we know that $\tau_\pm : \tilde\kappa$. What consists a valid split depends on the specifics of kind $\tilde\kappa$. To see why, consider $\kappa_S = \Sigma\alpha_1{:}\Omega.\Pi\alpha_2{:}\mathrm{S}_\Omega(\alpha_1).\Omega$ and $\tau_+ : \mathrm{A}(\tau_- : \kappa_S)$. Under this kind assignment, $P[\alpha] = \alpha.2\,(\alpha.1)$ is a valid residual type. However, forming $P[\tau_-] = \tau_-.2\,(\alpha.1)$ is not: $\alpha.1$ does not match the argument kind $\mathrm{S}_\Omega(\tau_-.1)$ of $\tau_-.2$!

To address this problem, the rules actually use *coercive path replacement* $P[\tau/\alpha : \kappa]$, which is defined in Figure 11. It coerces occurrences of α in P where necessary, such that the resulting type remains well-formed. In the case of $P[\alpha] = \alpha.2\,(\alpha.1)$ as before, $P[\tau_\pm/\alpha : \kappa_S] = \tau_\pm.2\,(\tau_\pm.1)$. In fact, there is no way to separate the two occurrences of α. Consequently, a coercion $\mathcal{C}^{+\alpha:\kappa_S.\alpha.2\,(\alpha.1)}_{\tau_+\approx\tau_-}(v)$ would not actually "split" the residual type, but ground it in one step with $\mathcal{C}^{+\alpha:\Omega.\alpha}_{\tau_+.2\,(\tau_+.1)\approx\tau_-.2\,(\tau_-.1)}(\mathcal{C}^{+\alpha:\kappa_S.\tau_-.2\,(\tau_-.1)}_{\tau_+\approx\tau_-}(v))$ (leaving a redundant inner coercion that eventually vanishes). This is consistent with the fact that $\tau_+.2\,(\tau_+.1) : \mathrm{A}(\tau_-.2\,(\tau_-.1))$.

Contrast this with a singleton-free variation of the example, using kind $\kappa_\Omega = \Sigma\alpha_1{:}\Omega.\Pi\alpha_2{:}\Omega.\Omega$. This results in $P[\tau_\pm/\alpha : \kappa_\Omega] = \tau_\pm.2\,(\alpha.1)$, and thus the coercion $\mathcal{C}^{+\alpha:\kappa_\Omega.\alpha.2\,(\alpha.1)}_{\tau_+\approx\tau_-}(v)$ is split into $\mathcal{C}^{+\alpha:\Omega.\alpha}_{\tau_+.2\,(\tau_+.1)\approx\tau_-.2\,(\tau_+.1)}(\mathcal{C}^{+\alpha:\kappa_\Omega.\tau_-.2\,(\alpha.1)}_{\tau_+\approx\tau_-}(v))$, where head and argument are indeed handled separately. Here, $\tau_+.2\,(\tau_+.1) : \mathrm{A}(\tau_-.2\,(\tau_+.1))$. In general, if $\tau_+ : \mathrm{A}(\tau_- : \tilde\kappa)$, we have $P[\tau_+/\alpha : \tilde\kappa] : \mathrm{A}(P[\tau_-/\alpha : \tilde\kappa])$, given a suitable P.

5 Results

We now have all necessary ingredients to define a dynamically adequate notion of *a posteriori* sealing as follows:

$$e :> \exists\alpha{:}\tilde\kappa.\tilde\tau \quad := \quad \mathsf{let}\langle\alpha, x\rangle = e\ \mathsf{in}\ \mathsf{new}\,\beta{:}\tilde\kappa \approx \alpha\ \mathsf{in}_{\exists\alpha:\tilde\kappa.\tilde\tau}\ \langle\beta, \mathcal{C}^{+\alpha:\tilde\kappa.\tilde\tau}_{\beta\approx\alpha}x\rangle$$

with the admissible typing rule

$$\textsc{Eseal}^* \quad \frac{\Gamma \vdash e : \exists \alpha{:}\tilde{\kappa}.\tilde{\tau}}{\Gamma \vdash (e :> \exists \alpha{:}\tilde{\kappa}.\tilde{\tau}) : \exists \alpha{:}\tilde{\kappa}.\tilde{\tau}}$$

Note how higher kinds allow uniformly generating just one type name for the whole static part of the module, and then coercing the dynamic part appropriately with a single higher-order coercion. Moreover, the definition of higher-order abstraction kinds ensures that singleton components remain transparent. Applying this to our complex number example lets us define the safe C' from Figure 4 simply as follows:

$$C' \quad = \quad C :> \mathit{COMPLEX}$$

where C is the original transparent definition from Figure 2.

The $\lambda^{\omega}_{\mathrm{SA}}$-calculus enjoys the usual soundness properties:

Lemma 5.1 (Preservation)
If $\Delta \vdash e : \tau$ *and* $(\Delta; e) \to (\Delta'; e')$, *then* $\Delta' \vdash e' : \tau$.

Lemma 5.2 (Progress) *If* $\Delta \vdash e : \tau$, *then either* $e = v$, *or* $(\Delta; e) \to (\Delta'; e')$.

More interestingly, type checking is decidable for Full $\lambda^{\omega}_{\mathrm{SA}}$:

Lemma 5.3 (Decidability) *For each* $\lambda^{\omega}_{\mathrm{SA}}$ *judgement there exists an algorithm that is sound, complete and terminating.*

All algorithms and proofs can be found in [24]. The main difficulty is posed by type equivalence with singleton kinds. Fortunately, we kept our type language close enough to Stone & Harper [27], so that only minor extensions to their algorithms and proofs were required, mostly for dealing with the extended notion of base kinds.

6 Related and Future Work

Dynamic type generation for abstract types can be found in a number of previous works, including our own [26,22,18,28,8]. Likewise, notions of higher-order coercion are a recurring scheme, as already mentioned in Section 1. However, we are not aware of any previous work that investigates either in the context of a type system such as ours, which includes both translucency and type analysis or features dependent kinds.

In [22] we first presented a calculus similar to $\lambda^{\omega}_{\mathrm{SA}}$, but without translucency. Lacking the regularity of abstraction kinds as well, higher-kinded coercions could not be reduced to a simple canonical form, requiring more ad-hoc rules.

Vytiniotis et al. [28] is the only other work that combines a (much richer) type analysis construct with higher-kinded type generativity and higher-order coercions, but without modelling translucency. Not having abstraction kinds, their system syntactically distinguishes between basic and higher-order coercions: their equivalent of $\mathcal{C}^{+\alpha:(\Omega\to\Omega).\alpha\,\tau}_{\beta\approx\tau\_}(v)$ would 'forget' the residual type annotation and reduce to

$\mathcal{C}^{+}_{\beta\approx\tau\_}(v)$, resulting in a lack of a principal type property. For example, $\mathcal{C}^{+}_{\beta\approx\lambda\alpha.int}(3)$ could be assigned infinitely many incompatible types. It is not obvious that type checking is decidable, considering higher-order cases like $\mathcal{C}^{+}_{\beta\approx\lambda\alpha:\Omega\to\Omega.int}(3)$ that may ask for higher-order unification to infer a higher-kinded argument to β.

Crary [3] presents a coercion calculus for eliminating subtyping and bounded quantification. His language is equipped with intersection types, but does not feature higher-order types or dependent kinds. That allows him to express higher-order coercions by a separate coercion language, and have a simple proof of erasability. Interestingly, Crary's development requires defining a meta function *map* that applies a pair of positive and negative coercions to all occurrences of a type variable in a type. This roughly corresponds to a simple notion of kind coercion. In another article [4], Crary gives an elimination transformation for singleton kinds. His definition of singleton expansion conincides with the degenerate case of kind coercion where the placeholder α does not actually occur in the residual kind.

Grossman et al.'s *abstraction brackets* [12] are very similar to higher-order coercions over ground types, but in a simply-typed system. Unlike coercions, nested brackets are *merged* on reduction, dropping all intermediate type annotations. It is not clear if that would leave sufficient information for decidable type checking in a richer type system like ours, particularly with higher kinds and subtyping.

Dreyer employs explicit generativity to model type abstraction of recursive modules [8]. For that purpose, he separates generation of type names ($\mathsf{new}\ \ \alpha{\uparrow}\kappa\ \mathsf{in}\ e$) from definition ($\mathsf{set}\ \ \alpha{:}{\approx}\tau\ \mathsf{in}\ e$). An effect system ensures linearity of the definitions. He does not consider type analysis, and thus avoids the need for coercions by allowing abstract type names to become transparent at run time.

Dreyer's thesis [7] describes a problem with the interplay between singleton kinds and higher-order recursive types, where well-formedness of a recursive type does not imply well-formedness of its unrolling. This is due to the same problem that occurs in our setting in the reduction of higher-kinded coercions and led us to introduce coercive path replacement. Dreyer's solution is to syntactically restrict the kinds on μ-bound type variables, which is not a viable option in our case.

Crary et al. [5] show how type analysis can be translated into term-level dispatch by introducing *term representations* of types. This translation can be employed to selectively prevent the analysis of certain types, simply by not introducing a representation for them. However, the choice cannot be made on a level that is fine-grained enough for dynamic translucency. For example, if β is the name of an abstract type, then inspection of $int \times \beta$ can only be prevented altogether, whereas our system still allows inspecting the product, but not β itself.

Instead of type names, generative type *tags* [11] can also ensure encapsulation of abstract values. However, they do not enable regular $\mathsf{typecase}$ to translucently match abstract types, and they do not easily support higher kinds.

The most interesting question to address in future work is formally proving a form of representation independence for our system. In [24] we prove a rather weak syntactic "Opacity" property, similar to Grossman et al.'s syntactic "Value Abstraction" [12]. We would like to obtain stronger results by applying semantic

techniques like logical relations to this non-parametric setting.

Acknowledgement

I thank Derek Dreyer for helpful comments and suggestions.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *TOPLAS*, 13(2), 1991.

[2] David Aspinall. Subtyping with singleton types. In *Computer Science Logic*, LNCS 933, 1995.

[3] Karl Crary. Typed compilation of inclusive subtyping. In *ICFP*, 2000.

[4] Karl Crary. Sound and complete elimination of singleton kinds. *TOCL*, 8(2), 2007.

[5] Karl Crary, Stephanie Weirich, and Greg Morisett. Intensional polymorphism in type-erasure semantics. *JFP*, 12(6), 2002.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL*, 2003.

[7] Derek Dreyer. *Understanding and Evolving the ML Module System*. Phd thesis, Carnegie Mellon University, 2005.

[8] Derek Dreyer. Recursive type generativity. *JFP*, 17(4&5), 2007.

[9] Robert Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[10] Cormac Flanagan. Hybrid type checking. In *POPL*, 2006.

[11] Neal Glew. Type dispatch for named hierarchical types. In *ICFP*, 1999.

[12] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *TOPLAS*, 22(6), 2000.

[13] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL*, 1995.

[15] Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL*, 1990.

[16] Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL*, 1992.

[17] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL*, 1994.

[18] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *ICFP*, 2003.

[19] David MacQueen. Using dependent types to express modular structure. In *POPL*, 1986.

[20] John Mitchell and Gordon Plotkin. Abstract types have existential type. *TOPLAS*, 10(3), 1988.

[21] J. Reynolds. Types, abstraction and parametric polymorphism. In *Information Procesing*, 1983.

[22] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP*, 2003.

[23] Andreas Rossberg. The missing link – dynamic components for ML. In *ICFP*, 2006.

[24] Andreas Rossberg. *Typed Open Programming*. Dissertation, Saarland University, 2007. available from http://www.mpi-sws.mpg.de/~rossberg/papers/thesis.pdf.

[25] Claudio Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[26] Peter Sewell. Modules, abstract types, and distributed versioning. In *POPL*, 2001.

[27] Chris Stone and Robert Harper. Extensional equivalence and singleton types. *TOCL*, 7(4), 2006.

[28] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *TLDI*, 2005.

A Core $\lambda^{\omega}_{\mathrm{SA}}$ Summary

A.1 Syntax

$$
\begin{array}{lrcl}
\text{base kinds} & K & ::= & \Omega \mid \mathrm{A}(\tau) \\
\text{kinds} & \kappa & ::= & K \mid \mathrm{S}_K(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa \\
\text{types} & \tau & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \\
 & & & \mid \lambda\alpha{:}\kappa.\tau \mid \tau\,\tau \mid \langle \tau, \tau \rangle \mid \tau.1 \mid \tau.2 \\
\text{terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e, e \rangle \mid \mathsf{let}\langle x, x \rangle = e \;\mathsf{in}\; e \\
 & & & \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle \tau, e \rangle \mid \mathsf{let}\langle \alpha, x \rangle = e \;\mathsf{in}_\tau\; e \\
 & & & \mid \mathsf{new}\,\alpha \approx \tau\;\mathsf{in}_\tau\; e \mid \mathcal{C}^{+}_{\tau\approx\tau} \mid \mathcal{C}^{-}_{\tau\approx\tau} \mid \mathsf{typecase}\,e{:}\tau\;\mathsf{of}\;x{:}\tau.e\;\mathsf{else}_\tau\;e \\
\text{environments} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\kappa \\
\text{heaps} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tau) \\
\text{configurations} & C & ::= & \Delta; e
\end{array}
$$

Higher-order Singletons $\quad \mathrm{S}(\tau : \kappa)$

$$
\begin{array}{rcl}
\mathrm{S}(\tau : \Omega) & := & \mathrm{S}_\Omega(\tau) \\
\mathrm{S}(\tau : \mathrm{A}(\tau')) & := & \mathrm{S}_{\mathrm{A}(\tau')}(\tau) \\
\mathrm{S}(\tau : \mathrm{S}_K(\tau')) & := & \mathrm{S}_K(\tau') \\
\mathrm{S}(\tau : \Pi\alpha{:}\kappa_1.\kappa_2) & := & \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2) \\
\mathrm{S}(\tau : \Sigma\alpha{:}\kappa_1.\kappa_2) & := & \mathrm{S}(\tau.1 : \kappa_1) \times \mathrm{S}(\tau.2 : \kappa_2[\tau.1/\alpha])
\end{array}
$$

A.2 Static Semantics

Environment Formation $\quad \Gamma \vdash \Box$

$$
\cdot \vdash \Box \qquad \frac{\Gamma \vdash \kappa : \Box}{\Gamma, \alpha{:}\kappa \vdash \Box}\,(\alpha \notin \mathrm{Dom}(\Gamma)) \qquad \frac{\Gamma \vdash \tau : \Omega}{\Gamma, x{:}\tau \vdash \Box}\,(x \notin \mathrm{Dom}(\Gamma))
$$

Kind Formation $\quad \Gamma \vdash \kappa : \Box$

$$
\frac{\Gamma \vdash \Box}{\Gamma \vdash \Omega : \Box} \qquad \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) : \Box} \qquad \frac{\Gamma \vdash \tau : K}{\Gamma \vdash \mathrm{S}_K(\tau) : \Box} \qquad \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box} \qquad \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box}
$$

Kind Equivalence $\quad \Gamma \vdash \kappa \equiv \kappa' : \Box$

$$
\frac{\Gamma \vdash \Box}{\Gamma \vdash \Omega \equiv \Omega : \Box} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \Box} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : K \qquad \Gamma \vdash K \equiv K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \equiv \mathrm{S}_{K'}(\tau') : \Box}
$$

$$
\frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \equiv \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box} \qquad \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \equiv \Sigma\alpha{:}\kappa_1'.\kappa_2' : \Box}
$$

Kind Inclusion $\quad \Gamma \vdash \kappa \le \kappa' : \Box$

$$
\frac{\Gamma \vdash \Box}{\Gamma \vdash \Omega \le \Omega : \Box}
$$

$$
\frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \le \mathrm{A}(\tau') : \Box} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : K \qquad \Gamma \vdash K \le K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \le \mathrm{S}_{K'}(\tau') : \Box}
$$

$$
\frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \le \Omega : \Box} \qquad \frac{\Gamma \vdash \tau : K \qquad \Gamma \vdash K \le K' : \Box}{\Gamma \vdash \mathrm{S}_K(\tau) \le K' : \Box}
$$

$$
\frac{\Gamma \vdash \kappa_1' \le \kappa_1 : \Box \qquad \Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \le \kappa_2' : \Box \qquad \Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \le \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box}
$$

$$
\frac{\Gamma \vdash \kappa_1 \le \kappa_1' : \Box \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \le \kappa_2' : \Box \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1'.\kappa_2' : \Box}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \le \Sigma\alpha{:}\kappa_1'.\kappa_2' : \Box}
$$

Type Formation $\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{}{\Gamma \vdash \square}$$

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

$$\frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 : \Omega}$$

$$\frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 : \Omega} \qquad \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \exists\alpha{:}\kappa_1.\tau_2 : \Omega}$$

$$\frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2 \qquad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\,\tau_2 : \kappa_2[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha] \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \langle\tau_1, \tau_2\rangle : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau.1 : \kappa_1} \qquad \frac{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau.2 : \kappa_2[\tau.1/\alpha]}$$

$$\frac{\Gamma \vdash \tau : K}{\Gamma \vdash \tau : \mathrm{S}_K(\tau)}$$

$$\frac{\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2' \qquad \Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \kappa_2 \qquad \Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2' : \square}{\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau.1 : \kappa_1 \qquad \Gamma \vdash \tau.2 : \kappa_2[\tau.1/\alpha] \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \kappa \le \kappa' : \square}{\Gamma \vdash \tau : \kappa'}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$\frac{}{\Gamma \vdash \square}$$

$$\frac{}{\Gamma \vdash \alpha \equiv \alpha : \Gamma(\alpha)}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \qquad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \Omega} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \qquad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \Omega}$$

$$\frac{\Gamma \vdash \kappa \equiv \kappa' : \square \qquad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \forall\alpha{:}\kappa.\tau \equiv \forall\alpha{:}\kappa'.\tau' : \Omega} \qquad \frac{\Gamma \vdash \kappa \equiv \kappa' : \square \qquad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \exists\alpha{:}\kappa.\tau \equiv \exists\alpha{:}\kappa'.\tau' : \Omega}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \qquad \Gamma, \alpha{:}\kappa_1 \vdash \tau \equiv \tau' : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau \equiv \lambda\alpha{:}\kappa_1'.\tau' : \Pi\alpha{:}\kappa_1.\kappa_2} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Pi\alpha{:}\kappa_1.\kappa_2 \qquad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Gamma \vdash \tau_1\,\tau_2 \equiv \tau_1'\,\tau_2' : \kappa_2[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1 \qquad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2[\tau_1/\alpha] \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \langle\tau_1, \tau_2\rangle \equiv \langle\tau_1', \tau_2'\rangle : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau.1 \equiv \tau'.1 : \kappa_1} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau.2 \equiv \tau'.2 : \kappa_2[\tau.1/\alpha]}$$

$$\frac{\Gamma \vdash \tau : \mathrm{S}_K(\tau'') \qquad \Gamma \vdash \tau' : \mathrm{S}_K(\tau'')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_K(\tau'')}$$

$$\frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha \equiv \tau'\,\alpha : \kappa_2 \qquad \Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2' \qquad \Gamma \vdash \tau' : \Pi\alpha{:}\kappa_1.\kappa_2''}{\Gamma \vdash \tau \equiv \tau' : \Pi\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau.1 \equiv \tau'.1 : \kappa_1 \qquad \Gamma \vdash \tau.2 \equiv \tau'.2 : \kappa_2[\tau.1/\alpha] \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \tau \equiv \tau' : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau' \equiv \tau : \kappa} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \tau' \equiv \tau'' : \kappa}{\Gamma \vdash \tau \equiv \tau'' : \kappa} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \le \kappa' : \square}{\Gamma \vdash \tau \equiv \tau' : \kappa'}$$

Type Inclusion $\boxed{\Gamma \vdash \tau \leq \tau' : \kappa}$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau \leq \tau' : \kappa}$$

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' : \Omega} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_1' : \Omega \qquad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' : \Omega}$$

$$\frac{\Gamma \vdash \kappa' \leq \kappa : \Box \qquad \Gamma, \alpha{:}\kappa' \vdash \tau \leq \tau' : \Omega \qquad \Gamma \vdash \forall \alpha{:}\kappa.\tau : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \leq \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$\frac{\Gamma \vdash \kappa \leq \kappa' : \Box \qquad \Gamma, \alpha{:}\kappa \vdash \tau \leq \tau' : \Omega \qquad \Gamma \vdash \exists \alpha{:}\kappa'.\tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \leq \exists \alpha{:}\kappa'.\tau' : \Omega}$$

$$\frac{\Gamma \vdash \tau \leq \tau' : \kappa \qquad \Gamma \vdash \tau' \leq \tau'' : \kappa}{\Gamma \vdash \tau \leq \tau'' : \kappa}$$

Term Formation $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \Box}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\langle x_1, x_2 \rangle = e_1 \;\mathsf{in}\; e_2 : \tau}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\tau} \qquad \frac{\Gamma \vdash e : \forall \alpha{:}\kappa.\tau \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash \langle \tau, e \rangle : \exists \alpha{:}\kappa.\tau_2} \qquad \frac{\Gamma \vdash e_1 : \exists \alpha{:}\kappa.\tau_2 \qquad \Gamma, \alpha{:}\kappa, x{:}\tau_2 \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathsf{let}\langle \alpha, x \rangle = e_1 \;\mathsf{in}_\tau\; e_2 : \tau}$$

$$\frac{\Gamma, \alpha{:}\mathrm{A}(\tau_1) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2} \qquad \frac{\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \mathcal{C}^{+}_{\tau_1 \approx \tau_2} : \tau_2 \to \tau_1} \qquad \frac{\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \mathcal{C}^{-}_{\tau_1 \approx \tau_2} : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{typecase}\, e_1{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}_\tau\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \leq \tau' : \Omega}{\Gamma \vdash e : \tau'}$$

Configuration Formation $\boxed{\Gamma \vdash C : \tau}$

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \Delta; e : \tau}$$

A.3 Admissible Rules

Kind Formation $\boxed{\Gamma \vdash \kappa : \Box}$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) : \Box}$$

Type Formation $\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau : \mathrm{S}(\tau : \kappa)}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \Box}$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \equiv \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \equiv \mathrm{S}(\tau' : \kappa') : \Box}$$

Type Equivalence $\left|\ \Gamma \vdash \tau \equiv \tau' : \kappa\ \right|$

$$\frac{\Gamma \vdash \tau : \mathrm{S}(\tau'' : \kappa) \qquad \Gamma \vdash \tau' : \mathrm{S}(\tau'' : \kappa)}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}(\tau'' : \kappa)}$$

$$\frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2 \qquad \Gamma \vdash \tau_1 : \kappa_1}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 \equiv \tau_2[\tau_1/\alpha] : \kappa_2[\tau_1/\alpha]}$$

$$\frac{\Gamma \vdash \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2\,\alpha \equiv \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle.1 \equiv \tau_1 : \kappa_1} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle.2 \equiv \tau_2 : \kappa_2}$$

$$\frac{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \langle \tau.1, \tau.2 \rangle \equiv \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

Kind Inclusion $\left|\ \Gamma \vdash \kappa \leq \kappa' : \Box\ \right|$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \leq \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \mathrm{S}(\tau' : \kappa') : \Box} \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \kappa : \Box}$$

A.4 Dynamic Semantics

Values and Contexts

$$\begin{array}{lrcl} \text{values} & v & ::= & \lambda x{:}\tau.e \mid \langle v, v \rangle \mid \lambda\alpha{:}\kappa.e \mid \langle \tau, v \rangle \mid \mathcal{C}^{+}_{\tau \approx \tau}(v) \\ \text{contexts} & E & ::= & \_ \mid E\,e \mid v\,E \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathsf{let}\langle x, x \rangle = E \;\mathsf{in}\; e \\ & & & \quad \mid E\,\tau \mid \langle \tau, E \rangle \mid \mathsf{let}\langle \alpha, x \rangle = E \;\mathsf{in}\; e \\ & & & \quad \mid \mathsf{typecase}\; E{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}\; e \end{array}$$

Reduction rules

$$\begin{array}{lll} (\textsc{Rapp}) & \Delta; E[(\lambda x{:}\tau.e)\,v] & \to \Delta; E[e[v/x]] \\ (\textsc{Rproj}) & \Delta; E[\mathsf{let}\langle x_1, x_2 \rangle = \langle v_1, v_2 \rangle \;\mathsf{in}\; e] & \to \Delta; E[e[v_1/x_1][v_2/x_2]] \\ (\textsc{Rinst}) & \Delta; E[(\lambda\alpha{:}\kappa.e)\,\tau] & \to \Delta; E[e[\tau/\alpha]] \\ (\textsc{Ropen}) & \Delta; E[\mathsf{let}\langle \alpha, x \rangle = \langle \tau, v \rangle \;\mathsf{in}_{\tau'}\; e] & \to \Delta; E[e[\tau/\alpha][v/x]] \\ (\textsc{Rnew}) & \Delta; E[\mathsf{new}\;\alpha \approx \tau \;\mathsf{in}_{\tau'}\; e] & \to \Delta, \alpha{:}\mathrm{A}(\tau); E[e] \\ (\textsc{Rcancel}) & \Delta; E[\mathcal{C}^{-}_{\tau_+ \approx \tau_-}(\mathcal{C}^{+}_{\tau'_+ \approx \tau'_-}(v))] & \to \Delta; E[v] \\ (\textsc{Rcase1}) & \Delta; E[\mathsf{typecase}\; v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_1[v/x]] \quad \text{if } \Delta \vdash \tau_1 \leq \tau_2 : \Omega \\ (\textsc{Rcase2}) & \Delta; E[\mathsf{typecase}\; v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_2] \quad \text{if } \Delta \not\vdash \tau_1 \leq \tau_2 : \Omega \end{array}$$

B Full $\lambda^{\omega}_{\mathrm{SA}}$

B.1 Syntax

$$\begin{array}{lrcl} \text{expressions} & e & ::= & \cdots \mid e\,\tilde{\tau} \mid \langle \tilde{\tau}, e \rangle \mid \mathsf{new}\;\alpha{:}\tilde{\kappa} \approx \tilde{\tau} \;\mathsf{in}_\tau\; e \mid \mathcal{C}^{+\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tilde{\tau} \approx \tilde{\tau}} \mid \mathcal{C}^{-\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tilde{\tau} \approx \tilde{\tau}} \\ \text{concrete kinds} & \tilde{\kappa} & ::= & \Omega \mid \mathrm{S}_\Omega(\tau) \mid \Pi\alpha{:}\tilde{\kappa}.\tilde{\kappa} \mid \Sigma\alpha{:}\tilde{\kappa}.\tilde{\kappa} \\ \text{concrete types} & \tilde{\tau} & ::= & \alpha \mid \tilde{\tau} \to \tilde{\tau} \mid \tilde{\tau} \times \tilde{\tau} \mid \forall\alpha{:}\tilde{\kappa}.\tilde{\tau} \mid \exists\alpha{:}\tilde{\kappa}.\tilde{\tau} \mid \lambda\alpha{:}\tilde{\kappa}.\tilde{\tau} \mid \tilde{\tau}\,\tilde{\tau} \mid \langle \tilde{\tau}, \tilde{\tau} \rangle \mid \tilde{\tau}.i \\ \text{heaps} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tilde{\tau} : \tilde{\kappa}) \end{array}$$

Kind Coercions $\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau/\tau}$

$$
\begin{array}{lcl}
\mathcal{T}^{\alpha:\kappa.\Omega}_{\tau_+/\tau_-}(\tau) & := & \tau \\
\mathcal{T}^{\alpha:\kappa.\mathrm{S}_\Omega(\tau')}_{\tau_+/\tau_-}(\tau) & := & \tau'[\tau_+/\alpha] \\
\mathcal{T}^{\alpha:\kappa.\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2}_{\tau_+/\tau_-}(\tau) & := & \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_2[\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_1}_{\alpha/\tau_+}\alpha_1/\alpha_1]}_{\tau_+/\tau_-}(\tau\,(\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_1}_{\tau_-/\tau_+}\alpha_1)) \\
\mathcal{T}^{\alpha:\kappa.\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2}_{\tau_+/\tau_-}(\tau) & := & \langle \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_1}_{\tau_+/\tau_-}(\tau.1), \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_2[\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}_1}_{\alpha/\tau_-}(\tau.1)/\alpha_1]}_{\tau_+/\tau_-}(\tau.2)\rangle
\end{array}
$$

Higher-Order Abstraction Kinds $\mathrm{A}(\tau:\tilde{\kappa})$

$$
\begin{array}{rcl}
\mathrm{A}(\tau:\Omega) & := & \mathrm{A}(\tau) \\
\mathrm{A}(\tau:\mathrm{S}_\Omega(\tau')) & := & \mathrm{S}_\Omega(\tau') \\
\mathrm{A}(\tau:\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Pi\alpha{:}\tilde{\kappa}_1.\mathrm{A}(\tau\,\alpha:\tilde{\kappa}_2) \\
\mathrm{A}(\tau:\Sigma\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Sigma\alpha{:}\mathrm{A}(\tau.1:\tilde{\kappa}_1).\mathrm{A}(\mathcal{T}^{\alpha:\tilde{\kappa}_1.\tilde{\kappa}_2}_{\alpha/\tau.1}(\tau.2):\tilde{\kappa}_2)
\end{array}
$$

Sealing $e :> \tilde{\tau}$

$$e :> \exists\alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 \qu:= \quad \mathsf{let}\langle\alpha, x\rangle = e\ \mathsf{in}_{\exists\alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \mathsf{new}\,\beta{:}\tilde{\kappa}_1 \approx \alpha\ \mathsf{in}_{\exists\alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \langle\beta, \mathcal{C}^{+\alpha:\tilde{\kappa}_1.\tilde{\tau}_2}_{\beta\approx\alpha}x\rangle$$

B.2 Static Semantics

Term formation $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash \tau_1 : \tilde{\kappa} \qquad \Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2}$$

$$\frac{\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})}{\Gamma \vdash \mathcal{C}^{+\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha] \to \tilde{\tau}[\tau_+/\alpha]}$$

$$\frac{\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})}{\Gamma \vdash \mathcal{C}^{-\alpha:\tilde{\kappa}.\tilde{\tau}}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha] \to \tilde{\tau}[\tau_+/\alpha]}$$

B.3 Admissible Rules

Kind Formation $\boxed{\Gamma \vdash \kappa : \Box}$

$$\frac{\Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) : \Box}$$

Type Formation $\left|\ \Gamma \vdash \tau : \kappa\ \right|$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_- : \kappa \qquad \Gamma \vdash \tau_+ : \kappa}{\Gamma \vdash \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau_+/\tau_-} : \tilde{\kappa}[\tau_-/\alpha] \to \tilde{\kappa}[\tau_+/\alpha]}$$

Term Formation $\left|\ \Gamma \vdash e : \tau\ \right|$

$$\frac{\Gamma \vdash e : \tilde{\tau}}{\Gamma \vdash e :> \tilde{\tau} : \tilde{\tau}}$$

Kind Equivalence $\left|\ \Gamma \vdash \kappa \equiv \kappa' : \Box\ \right|$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa} \qquad \Gamma \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \equiv \mathrm{A}(\tau' : \tilde{\kappa}') : \Box}$$

Type Equivalence $\left|\ \Gamma \vdash \tau \equiv \tau' : \kappa\ \right|$

$$\frac{\begin{array}{c}\Gamma \vdash \kappa \equiv \kappa' : \Box \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma, \alpha{:}\kappa' \vdash \tilde{\kappa}' : \Box \\ \Gamma \vdash \tau_+ \equiv \tau'_+ : \kappa \qquad\qquad \Gamma \vdash \tau_- \equiv \tau'_- : \kappa \\ \Gamma \vdash \tilde{\kappa}[\tau_+/\alpha] \equiv \tilde{\kappa}'[\tau'_+/\alpha] : \Box \qquad \Gamma \vdash \tilde{\kappa}[\tau_-/\alpha] \equiv \tilde{\kappa}'[\tau'_-/\alpha] : \Box\end{array}}{\Gamma \vdash \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau_+/\tau_-} \equiv \mathcal{T}^{\alpha:\kappa'.\tilde{\kappa}'}_{\tau'_+/\tau'_-} : \tilde{\kappa}[\tau_+/\alpha]}$$

$$\frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ \equiv \tau_- : \kappa}{\Gamma \vdash \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau_+/\tau_-}(\tau) \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}$$

$$\frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau_-/\tau_+}(\mathcal{T}^{\alpha:\kappa.\tilde{\kappa}}_{\tau_+/\tau_-}(\tau)) \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}$$

Kind Inclusion $\left|\ \Gamma \vdash \kappa \leq \kappa' : \Box\ \right|$

$$\frac{\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa} \qquad \Gamma \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \leq \mathrm{A}(\tau' : \tilde{\kappa}') : \Box} \qquad \frac{\Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \leq \tilde{\kappa} : \Box}$$

B.4 Dynamic Semantics

Values and Contexts

$$\begin{array}{lll} \text{values} & v & ::= \cdots \mid \langle \tilde{\tau}, v \rangle \mid \mathcal{C}^{+\alpha:\Omega.\alpha}_{\tilde{\tau} \approx \tilde{\tau}}(v) \\ \text{path contexts} & P & ::= \_ \mid P\,\tau \mid P.1 \mid P.2 \end{array}$$

Reduction rules

$$\Delta; E[\mathsf{new}\ \alpha{:}\tilde\kappa{\approx}\tau\ \mathsf{in}\ e] \ \to\ \Delta, \alpha{:}\mathrm{A}(\tau : \tilde\kappa); E[e]$$

$$\begin{array}{ll}
\mathcal{C}^{\pm\alpha:\Omega.\tau_1\to\tau_2}_{\tau_+\approx\tau_-}(v) & \to\ \lambda x_1{:}\tau_1[\tau_\pm/\alpha].\, \mathcal{C}^{\pm\alpha:\Omega.\tau_2}_{\tau_+\approx\tau_-}(v\,(\mathcal{C}^{\mp\alpha:\Omega.\tau_1}_{\tau_+\approx\tau_-} x_1)) \\
\mathcal{C}^{\pm\alpha:\Omega.\tau_1\times\tau_2}_{\tau_+\approx\tau_-}(v) & \to\ \mathsf{let}\langle x_1, x_2\rangle = v\ \mathsf{in}\ \langle \mathcal{C}^{\pm\alpha:\Omega.\tau_1}_{\tau_+\approx\tau_-} x_1, \mathcal{C}^{\pm\alpha:\Omega.\tau_2}_{\tau_+\approx\tau_-} x_2\rangle \\
\mathcal{C}^{\pm\alpha:\Omega.\forall\beta:\kappa.\tau}_{\tau_+\approx\tau_-}(v) & \to\ \lambda\beta{:}\kappa[\tau_\pm/\alpha].\, \mathcal{C}^{\pm\alpha:\Omega.\tau'}_{\tau_+\approx\tau_-}(v\,(\mathcal{T}^{\alpha:\Omega.\kappa}_{\tau_\mp/\tau_\pm}\beta)) \\
 & \quad \text{where } \tau' = \tau[(\mathcal{T}^{\alpha:\Omega.\kappa}_{\alpha/\tau_\pm}\beta)/\beta] \\
\mathcal{C}^{\pm\alpha:\Omega.\exists\beta:\kappa.\tau}_{\tau_+\approx\tau_-}(v) & \to\ \mathsf{let}\langle\beta, x\rangle = v\ \mathsf{in}\ \langle \mathcal{T}^{\alpha:\Omega.\kappa}_{\tau_\pm/\tau_\mp}\beta, \mathcal{C}^{\pm\alpha:\Omega.\tau'}_{\tau_+\approx\tau_-} x\rangle \\
 & \quad \text{where } \tau' = \tau[(\mathcal{T}^{\alpha:\Omega.\kappa}_{\alpha/\tau_\mp}\beta)/\beta] \\
\mathcal{C}^{\pm\alpha:\tilde\kappa.P[\beta]}_{\tau_+\approx\tau_-}(v) & \to\ \mathcal{C}^{+\alpha':\tilde\kappa'.P[\alpha'/\beta:\tilde\kappa'][\tau_\pm/\alpha]}_{\beta\approx\tau'}\mathcal{C}^{\pm\alpha:\tilde\kappa.P[\tau'/\beta:\tilde\kappa']}_{\tau_+\approx\tau_-}\mathcal{C}^{-\alpha':\tilde\kappa'.P[\alpha'/\beta:\tilde\kappa'][\tau_\mp/\alpha]}_{\beta\approx\tau'}(v) \\
 & \quad \text{where } \beta\neq\alpha \text{ and } \Delta(\beta) = \mathrm{A}(\tau' : \tilde\kappa') \\
\mathcal{C}^{\pm\alpha:\tilde\kappa.P[\alpha]}_{\tau_+\approx\tau_-}(v) & \to\ \mathcal{C}^{\pm\alpha:\Omega.\alpha}_{P[\tau_+/\alpha:\tilde\kappa][\tau_\pm/\alpha]\approx P[\tau_-/\alpha:\tilde\kappa][\tau_\pm/\alpha]}\mathcal{C}^{\pm\alpha:\tilde\kappa.P[\tau_\mp/\alpha:\tilde\kappa]}_{\tau_+\approx\tau_-}(v) \\
 & \quad \text{where } P \neq _ \\
\mathcal{C}^{-\alpha:\Omega.\alpha}_{\tau_+\approx\tau_-}(\mathcal{C}^{+\alpha:\Omega.\alpha}_{\tau'_+\approx\tau'_-}v) & \to\ v
\end{array}$$

Notes: 1. Omitted surrounding $\Delta; E[\_]$ in reduction rules for coercions.
2. All RHS variables fresh.

Path Replacement

$$\begin{array}{lll}
P[\tau_+/\tau_- : \tilde\kappa] & := & P^{[\alpha:\tilde\kappa.\tilde\kappa}_{\tau_+/\tau_-}\,\alpha] \\
^{[\alpha:\tilde\kappa.\Omega}_{\tau_+/\tau_-}\tau] & := & \tau[\tau+/\alpha] \\
^{[\alpha:\tilde\kappa.\mathrm{S}_\Omega(\tau')}_{\tau_+/\tau_-}\tau] & := & \tau[\tau+/\alpha] \\
P[_\,\tau'][^{[\alpha:\tilde\kappa.\Pi\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := & P^{[\alpha:\tilde\kappa.\tilde\kappa_2[\tau''/\alpha_1]}_{\tau_+/\tau_-}\tau\,\tau''] \text{ where } \tau'' = \mathcal{T}^{\alpha:\tilde\kappa.\tilde\kappa_1}_{\alpha/\tau_-}(\tau') \\
P[_.1][^{[\alpha:\tilde\kappa.\Sigma\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := & P^{[\alpha:\tilde\kappa.\tilde\kappa_1}_{\tau_+/\tau_-}\tau.1] \\
P[_.2][^{[\alpha:\tilde\kappa.\Sigma\alpha_1:\tilde\kappa_1.\tilde\kappa_2}_{\tau_+/\tau_-}\tau] & := & P^{[\alpha:\tilde\kappa.\tilde\kappa_2[\tau.1/\alpha_1]}_{\tau_+/\tau_-}\tau.2]
\end{array}$$