THE SINGLETON KIND CALCULUS

Abstract. In this section we develop the singleton kind calculus. A singleton kind $S(c)$ is the kind of all constructors that are equivalent to c. The addition of these new kinds will be useful to explain module signatures later on.

1. Syntax

The singleton kind calculus is built on top of souped-up F^ω.

$$\begin{array}{lll} k & ::= & \texttt{type} \mid k \to k \mid k * k \mid S(c) \mid \Pi\alpha : k.\ k \mid \Sigma\alpha : k.\ k \\ c & ::= & \alpha \mid c \to c \mid \forall\alpha : k.c \mid \lambda\alpha : k.c \mid c\ c \mid \langle c, c\rangle \mid \pi_1\ c \mid \pi_2\ c \\ e & ::= & x \mid \lambda x : c.e \mid e\ e \mid \Lambda\alpha : k.e \mid e[c] \\ \Gamma & ::= & \cdot \mid \Gamma, x : c \mid \Gamma, \alpha : k \end{array}$$

2. Motivation

Consider the following ML signature.

```
sig
    type t
    type 'a u
    type ('a, 'b) v
    type w = int
end
```

The first three types can be assigned kinds in F^ω in a straight forward way.

$$\begin{array}{l} \texttt{t} : \texttt{type} \\ \texttt{u} : \texttt{type} \to \texttt{type} \\ \texttt{v} : \texttt{type} \times \texttt{type} \to \texttt{type} \end{array}$$

But how do we kind `w`? Remember, `int` is not a kind, so it doesn't make sense to say $\texttt{w} : \texttt{int}$. But it's not quite right to say $\texttt{w} : \texttt{type}$ either, because `w` cannot stand for arbitrary types. We therefore write $\texttt{w} : S(\texttt{int})$: $S(\texttt{int})$ is the kind containing exactly `int` and all those types equivalent to `int`, such as $(\lambda\alpha : \texttt{type}.\ \alpha)\ \texttt{int}$. The other new kind constructs, $\Pi\alpha : k.\ k$ and $\Sigma\alpha : k.\ k$ (which are called dependent function spaces and dependent sums respectively), exist to solve the the analogous problem for kinding assignments to polymorphic types in signatures:

```
sig
    type 'a t = 'a list
end
```

This will become more clear once the rules are enumerated.

3. Definitions

In this section the following judgements will be defined.

Judgement	Description
$\Gamma \vdash k : \texttt{kind}$	k is a kind
$\Gamma \vdash k \equiv k' : \texttt{kind}$	kind equivalence
$\Gamma \vdash k \leq k'$	subkinding
$\Gamma \vdash c : k$	c has kind k
$\Gamma \vdash c \equiv c' : k$	constructor equivalence
$\Gamma \vdash e : \tau$	e has type τ

A complete list would also include the judgement $\Gamma \vdash \tau : \texttt{type}$ but these rules are exactly the same as in F^ω so we will omit them. Begining with the rules for well-formed kinds:

3A
$$\frac{}{\Gamma \vdash \tau : \texttt{kind}}$$

3B
$$\frac{\Gamma \vdash c : \tau}{\Gamma \vdash S(c) : \texttt{kind}}$$

3C
$$\frac{\Gamma \vdash k_1 : \texttt{kind} \quad \Gamma, k_1 : \texttt{kind} \vdash k_2 : \texttt{kind}}{\Gamma \vdash \Pi\alpha : k_1.\ k_2 : \texttt{kind}}$$

3D
$$\frac{\Gamma \vdash k_1 : \texttt{kind} \quad \Gamma, k_1 : \texttt{kind} \vdash k_2 : \texttt{kind}}{\Gamma \vdash \Sigma\alpha : k_1.\ k_2 : \texttt{kind}}$$

Definitional equality of kinds:

3E
$$\frac{\Gamma \vdash k : \texttt{kind}}{\Gamma \vdash k \equiv k : \texttt{kind}}$$

3F
$$\frac{\Gamma \vdash k_1 \equiv k_2 : \texttt{kind}}{\Gamma \vdash k_2 \equiv k_1 : \texttt{kind}}$$

3G
$$\frac{\Gamma \vdash k_1 \equiv k_2 : \texttt{kind} \quad \Gamma \vdash k_2 \equiv k_3 : \texttt{kind}}{\Gamma \vdash k_1 \equiv k_3 : \texttt{kind}}$$

3H
$$\frac{\Gamma \vdash c \equiv c' : \texttt{type}}{\Gamma \vdash S(c) \equiv S(c') : \texttt{kind}}$$

3I
$$\frac{\Gamma \vdash k_1 \equiv k_1' : \texttt{kind} \quad \Gamma, \alpha : k_1 \vdash k_2 \equiv k_2' : \texttt{kind}}{\Gamma \vdash \Pi\alpha : k_1.\ k_2 \equiv \Pi\alpha : k_1'.\ k_2 : \texttt{kind}}$$

3J
$$\frac{\Gamma \vdash k_1 \equiv k_1' : \texttt{kind} \quad \Gamma, \alpha : k_1 \vdash k_2 \equiv k_2' : \texttt{kind}}{\Gamma \vdash \Sigma\alpha : k_1.\ k_2 \equiv \Sigma\alpha : k_1'.\ k_2 : \texttt{kind}}$$

Kind membership — which constructors belong to a given kind:

3K
$$\frac{\alpha : k \in \Gamma}{\Gamma \vdash \alpha : k}$$

3L
$$\frac{\Gamma \vdash c_1 : \mathtt{type} \quad \Gamma \vdash c_2 : \mathtt{type}}{\Gamma \vdash c_1 \to c_2 : \mathtt{type}}$$

3M
$$\frac{\Gamma \vdash k : \mathtt{kind} \quad \Gamma, \alpha : k \vdash c : \mathtt{type}}{\Gamma \vdash \forall \alpha : k.\ c : \mathtt{type}}$$

3N
$$\frac{\Gamma \vdash k_1 : \mathtt{kind} \quad \Gamma, \alpha : k_1 \vdash c : k_2}{\Gamma \vdash \lambda \alpha : k_1.\ c : \Pi \alpha : k_1.\ k_2}$$

3O
$$\frac{\Gamma \vdash c_1 : \Pi \alpha : k.\ k' \quad \Gamma \vdash c_2 : k}{\Gamma \vdash c_1\ c_2 : [c_2/\alpha]k'}$$

3P
$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : [c_1/\alpha]k_2 \quad \Gamma, \alpha : k_1 \vdash k_2 : \mathtt{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle : \Sigma \alpha : k_1.\ k_2}$$

3Q
$$\frac{\Gamma \vdash c : \Sigma \alpha : k_1.\ k_2}{\Gamma \vdash \pi_1\ c : k_1}$$

3R
$$\frac{\Gamma \vdash c : \Sigma \alpha : k_1.\ k_2}{\Gamma \vdash \pi_2\ c : [\pi_1 c/\alpha]k_2}$$

3S
$$\frac{\Gamma \vdash c : \mathtt{type}}{\Gamma \vdash c : S(c)}$$

Notice that even though $\Sigma\alpha : k_1.\ k_2$ is called a dependent sum, it behaves like a product.