15-417 HOT Compilation (Spring 2018)

Scribe: Nick Roberts
Professor: Karl Crary
Carnegie Mellon University
Last built: January 30, 2018

1 Definition of F_ω *(1/16/2018)*

Note: In this section we skip the naïve formulation given by Prof. Crary (without kinds) and immediately introduce kinds.

First, we define type constructors. We often refer to these simply as "constructors." By convention, τ denotes a nullary constructor, but c may be used in general without fear.

$$c, \tau \ ::= \ \alpha \mid c \to c \mid \forall(\alpha : \kappa).c \mid \lambda(\alpha : \kappa).c \mid c\, c$$

α denotes a type variable. We use these in quantification and type abstraction. It may be instantiated during application. κ denotes a kind, which we now define:

$$\kappa \ ::= \ \texttt{type} \mid \kappa \to \kappa$$

Henceforth we use `T` to denote `type`. Where would we be without terms to inhabit types?

$$e \ ::= \ x \mid \lambda(x : \tau).e \mid e\, e \mid \Lambda(\alpha : \kappa).e \mid e[\tau]$$

Our context, Γ, may contain judgments pertaining to types and terms.

$$\Gamma \ ::= \ \varepsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha : \kappa$$

Sometimes, for the latter judgment, you will see $\alpha :: \kappa$, but this is not too important. Proceeding from this context, we first define inductively the judgment that $\Gamma \vdash c : \kappa$

Judgment 1.1 (Type kind) $\Gamma \vdash c : \kappa$

$$\frac{\Gamma(\alpha) : \kappa}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash c_1 : \mathtt{T} \quad \Gamma \vdash c_2 : \mathtt{T}}{\Gamma \vdash c_1 \to c_2 : \mathtt{T}} \qquad \frac{\Gamma, \alpha : \kappa \vdash c : \mathtt{T}}{\Gamma \vdash \forall(\alpha : \kappa).c : \mathtt{T}}$$

$$\frac{\Gamma \vdash c_1 : \kappa \to \kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1\, c_2 : \kappa'} \qquad \frac{\Gamma, \alpha : \kappa \vdash c : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).c : \kappa \to \kappa'}$$

Using this judgment, we next define the judgment $\Gamma \vdash e : \tau$.

Judgment 1.2 (Term type) $\Gamma \vdash e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau' \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \lambda(x : \tau).e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}$$

$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau} \qquad \frac{\Gamma \vdash e : \forall(\alpha : \kappa).\tau' \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e[\tau] : [\tau/\alpha]\tau'}$$

But these rules aren't sufficient: if $f : \forall(\alpha : \mathtt{T} \to \mathtt{T}).\alpha\ \mathtt{int} \to \mathtt{unit}$, we would want $f[\lambda(\beta : \mathtt{T}).\beta]\ 12 : \mathtt{unit}$, but currently this is not the case. We need a way to show $(\lambda\beta.\beta)\mathtt{int} \equiv \mathtt{int}$. To do so, we define a new judgment:

$$\Gamma \vdash c \equiv c' : \kappa$$

and add to Judgment 1.2 the inference rule

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathtt{T}}{\Gamma \vdash e : \tau'}\ (\textit{equivalence})$$

Judgment 1.3 (Equivalence) $\Gamma \vdash c \equiv c' : \kappa$

Equivalence rules

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c \equiv c : \kappa} \qquad \frac{\Gamma \vdash c \equiv c' : \kappa}{\Gamma \vdash c' \equiv c : \kappa} \qquad \frac{\Gamma \vdash c_1 \equiv c_2 : \kappa \quad \Gamma \vdash c_2 \equiv c_3 : \kappa}{\Gamma \vdash c_1 \equiv c_3 : \kappa}$$

Compatibility rules

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \mathtt{T} \quad \Gamma \vdash c_2 \equiv c_2' : \mathtt{T}}{\Gamma \vdash (c_1 \to c_2) \equiv (c_1' \to c_2') : \mathtt{T}} \qquad \frac{\Gamma, \alpha : \kappa \vdash c \equiv c' : \mathtt{T}}{\Gamma \vdash \forall(\alpha : \kappa).c \equiv \forall(\alpha : \kappa).c' : \mathtt{T}}$$

$$\frac{\Gamma, \alpha : \kappa \vdash c \equiv c' : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).c \equiv \lambda(\alpha : \kappa.c') \equiv \kappa \to \kappa'} \qquad \frac{\Gamma \vdash c_1 \equiv c_1' : \kappa \to \kappa' \quad \Gamma \vdash c_2 \equiv c_2' : \kappa}{\Gamma \vdash c_1\ c_2 \vdash c_1'\ c_2' : \kappa'}$$

These rules defined a congruence relation. We haven't yet added the interesting rules. Note the enforcement that some constructors be types ($\mathtt{T}$), since they certainly may not be λ-abstractions. At this point in the semester, the κ constraints may be omitted, since they are uniquely determined by the **regularity conditions** (assuming $\vdash \Gamma$ ok):

- If $\Gamma \vdash c_1 \equiv c_2 : \kappa$, then $\Gamma \vdash c_1 : \kappa$ and $\Gamma \vdash c_2 : \kappa$.
- If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \mathtt{T}$.

Now for a more interesting rules to add to Judgment 1.3:

$$\frac{\Gamma \vdash c : \kappa \quad \Gamma, \alpha : \kappa \vdash c' : \kappa'}{\Gamma \vdash (\lambda(\alpha : \kappa).c')c \equiv [c/\alpha]c' : \kappa'}\ (\beta)$$

We can prove:

$$\frac{\dfrac{\dfrac{}{\beta : \mathtt{T} \vdash \beta : \mathtt{T}} \quad \dfrac{}{\mathtt{int} : \mathtt{T}}}{(\lambda\beta.\beta)\ \mathtt{int} \equiv \mathtt{int} : \mathtt{T}}}{(\lambda\beta.\beta)\ \mathtt{int} \to \mathtt{unit} \equiv \mathtt{int} \to \mathtt{unit} : \mathtt{T}}$$

What about η-expansion? something along the lines of:

$$\frac{\Gamma \vdash c : \kappa \to \kappa'}{\Gamma \vdash c \equiv \lambda(\alpha : \kappa).c\ \alpha : \kappa \to \kappa'}\ (\eta)$$

We want something more general, akin to running an experiment on two constructors of function kind. (Adding to Judgment 1.3.)

$$\frac{\Gamma, \alpha : \kappa \vdash c_1\ \alpha \equiv c_2\ \alpha : \kappa'}{\Gamma \vdash c_1 \equiv c_2 : \kappa \to \kappa'}\ (\textit{extensionality})$$

Exercise: prove η from this rule.

1.1 F_ω plus products

$$\begin{array}{rcl} \kappa & ::= & \cdots \mid \kappa \times \kappa \\ c & ::= & \cdots \mid \langle c, c \rangle \mid \pi_1\, c \mid \pi_2\, c \end{array}$$

Adding to Judgment 1.1:

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_2 : \kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2} \qquad \frac{\Gamma \vdash c : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i\, c : \kappa_i}$$

Adding to Judgment 1.3, the compatability rules:

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \kappa' \quad \Gamma \vdash c_2 \equiv c_2' : \kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle : \kappa_1 \times \kappa_2} \qquad \frac{\Gamma \vdash c_1 \equiv c_2 : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i\, c_1 \equiv \pi_i\, c_2 : \kappa_i}$$

Adding to Judgment 1.3, the "interesting" rules:

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_2 : \kappa_2}{\Gamma \vdash \pi_1 \langle c_1, c_2 \rangle \equiv c_1 : \kappa_i}\ (\beta_\pi) \qquad \frac{\Gamma \vdash \pi_i\, c_1 \equiv \pi_i\, c_2 : \kappa_i}{\Gamma \vdash c_1 \equiv c_2 : \kappa_1 \times \kappa_2}\ (\mathit{extensionality}_\pi)$$

1.2 Motivation

All of this is useful in the understanding of ML's module system. For example, we wish to view the type components of a module as a singular type componen*ent*, for which we must understand a pair of types. Furthermore, functors may be seen as creating type constructors, for example:

```
functor Foo (type t) = struct
  type u = t * t
end
```

which may be represented as $\lambda(t : \mathtt{T})\langle t \times t, \ldots \rangle$.

1.3 Remarks

- We must specify a separate level (kinds) rather than just describing types with types; this is described in the Burali-Forti paradox.
- Counting binders from the inside-out is called de Brujin indices, but counting from the outside-in is called de Brujin *levels*, which "no one uses."

2 Writing a typechecker for F_ω *(1/18/2018)*

In this lecture, we rephrase the declarative syntax in the language of moded judgments so that we can consider matters of input and output. As before, since a type may admit at most one kind in F_ω, much of the material introduced here will stay unnecessary until we have reached the singleton kind calculus.

2.1 Judgments

Here is a list of all judgments defined in today's lecture. We use a superscript $+$ to indicate the input and a superscript $-$ to indicate the output.

$\Gamma^+ \vdash e^+ \Rightarrow \tau^-$	type synthesis/inference
$\Gamma^+ \vdash e^+ \Leftarrow \tau^+$	type checking/analysis
$\Gamma^+ \vdash c^+ \Rightarrow \kappa^-$	kind synthesis
$\Gamma^+ \vdash c^+ \Leftarrow \kappa^+$	kind checking
$\Gamma^+ \vdash c^+ \Leftrightarrow c'^+$	algorithmic equivalence
$\Gamma^+ \vdash q_1^+ \leftrightarrow q_2^+ : \kappa^-$	algorithmic path equivalence
$c^+ \Downarrow q^-$	weak-head normalization
$c^+ \rightsquigarrow c^-$	weak-head reduction

2.2 Inference Rules

Judgment 2.1 (Type synthesis)

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}$$

$$\frac{\Gamma \vdash \tau \Leftarrow \mathtt{T} \quad \Gamma, x : \tau \vdash e \Rightarrow \tau'}{\Gamma \vdash \lambda(x : \tau).e \Rightarrow \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau \quad \tau \Downarrow \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau_2}$$

Unlike 15-317, we can synthesize the type of a lambda term since we have the type of the parameter specified in the term. In the last rule, we don't simply synthesize the type of e_1 and confirm that it is an arrow type. This is because the type could have redexes. We instead normalize to $\tau_1 \to \tau_2$.[1]

Judgment 2.2 (Type checking)

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \Gamma \vdash \tau \Leftrightarrow \tau' : \mathtt{T}}{\Gamma \vdash e \Leftarrow \tau} \qquad \frac{\Gamma, \alpha : \kappa \vdash e \Rightarrow \tau}{\Gamma \vdash \Lambda(\alpha : \kappa).e \Rightarrow \forall(\alpha : \kappa).\tau}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \quad \tau \Downarrow \forall(\alpha : \kappa).\tau' \quad \Gamma \vdash c \Leftarrow \kappa}{\Gamma \vdash e[c] \Rightarrow [c/\alpha]\tau'}$$

We do not need to check the validity of kinds—*yet* (ominously). The same footnote holds for the universal type.

[1] I'm still not clear on why we can rely on WHNF to correctly "normalize" when it seemed to be a theme of lecture that we cannot rely on this.

Algorithmic equivalence. An option that works for F_ω is to repeatedly contract redeces until reaching a normalized term. However, this doesn't generalize to the singleton kind calculus, so we develop more machinery here.

Judgment 2.3 (Kind synthesis)

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \Rightarrow \kappa}$$

$$\frac{\Gamma, \alpha : \kappa \vdash c \Rightarrow \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).c \Rightarrow \kappa \to \kappa'} \qquad \frac{\Gamma \vdash c_1 \Rightarrow \kappa \to \kappa' \quad \Gamma \vdash c_2 \Leftarrow \kappa}{\Gamma \vdash c_1\, c_2 \Rightarrow \kappa'}$$

$$\frac{\Gamma \vdash c_1 \Rightarrow \kappa_1 \quad \Gamma \vdash c_2 \Rightarrow \kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle \Rightarrow \kappa_1 \times \kappa_2} \qquad \frac{\Gamma \vdash c \Rightarrow \kappa' \quad \Gamma \vdash c \Rightarrow \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i\, c \Rightarrow \kappa_i}$$

$$\frac{\Gamma \vdash c_1 \Leftarrow \mathtt{T} \quad \Gamma \vdash c_2 \Leftarrow \mathtt{T}}{\Gamma \vdash c_1 \to c_2 \Rightarrow \mathtt{T}} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau \Leftarrow \mathtt{T}}{\Gamma \vdash \forall(\alpha : \kappa).\tau \Rightarrow \mathtt{T}}$$

$$\frac{\Gamma \vdash c \Rightarrow \kappa' \quad \kappa = \kappa'}{\Gamma \vdash c \Leftarrow \kappa}\ \textit{(check)}$$

We write *(check)* in such a way that we can more explicitly see what we'll generalize when we move to other calculi, particularly the notion of equality.

Now when we write the equivalence rules, think about extensionality. We recurse on the *kind* using the appropriate projection or function application. What we *don't* do is normalize both types and check equivalence, because this doesn't generalize to the singleton kind calculus.

Judgment 2.4 (Algorithmic equivalence)

$$\frac{\Gamma, \alpha : \kappa_1 \vdash c\,\alpha \Leftrightarrow c'\,\alpha : \kappa_2}{\Gamma \vdash c \Leftrightarrow c' : \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash \pi_1\, c \Leftrightarrow \pi_1\, c' : \kappa_1 \quad \Gamma \vdash \pi_2\, c \Leftrightarrow \pi_2\, c' : \kappa_2}{\Gamma \vdash c \Leftrightarrow c' : \kappa_1 \times \kappa_2}$$

$$\frac{c \Downarrow q \quad c' \Downarrow q' \quad \Gamma \vdash q \leftrightarrow q' : \mathtt{T}}{\Gamma \vdash c \Leftrightarrow c' : \mathtt{T}}$$

We can only normalize once we reach a type kind.

A normal form is such that all redeces have been contracted. However, we only place in *weak head* normal form, where an arrow constructor or a universal constructor is at the outermost level. (That is, we don't recursively normalize.)

Judgment 2.5 (Weak-head normalization)

$$\frac{c \rightsquigarrow c' \quad c' \Downarrow c''}{c \Downarrow c''} \qquad \frac{c \not\rightsquigarrow}{c \Downarrow c}$$

Don't be too concerned about $c \not\rightsquigarrow$. In practice, we could implement this in ML by raising an exception if no reduction step is made.

Judgment 2.6 (Weak-head reduction)

$$\frac{}{(\lambda(\alpha : \kappa).c)\, c' \rightsquigarrow [c'/\alpha]c} \qquad \frac{}{\pi_i\, \langle c_1, c_2 \rangle \rightsquigarrow c_i}$$

$$\frac{c_1 \rightsquigarrow c_1'}{c_1\, c_2 \rightsquigarrow c_1'\, c_2} \qquad \frac{c \rightsquigarrow c'}{\pi_i\, c \rightsquigarrow \pi_i\, c'}$$

We only reduce under an application and a projection. This is so that when we encounter types such as $((\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha))\ c_3$, we can reduce them.

The definitions for path and whnf are curiously familiar.

$$\begin{aligned} \text{path } p &::= \alpha \mid p\ c \mid \pi_1\ p \mid \pi_2\ p \\ \text{whnf } q &::= p \mid c \to c \mid \forall(\alpha : \kappa).c \end{aligned}$$

Path is clearly a neutral term. Whnf is almost a neutral term, except the constituents (body of universal type and right/left of arrow may contain redeces). We have already gotten to kind `T`, which guarantees that this grammar is exhaustive.

Algorithmic structural equivalence should look familiar from constructive logic. The kind is synthesized as an output. Sometimes it is put back in as an input to the algorithmic equivalence judgment.

Judgment 2.7 (Algorithmic structural equivalence)

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \leftrightarrow \alpha : \kappa} \qquad \frac{\Gamma \vdash p \leftrightarrow p' : \kappa_1 \to \kappa_2 \quad \Gamma \vdash c \Leftrightarrow c' : \kappa_1}{\Gamma \vdash: p\ c \leftrightarrow p'\ c' : \kappa_2}$$

$$\frac{\Gamma \vdash p \leftrightarrow p' : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i\ p \leftrightarrow \pi_i\ p' : \kappa_i} \qquad \frac{\Gamma \vdash c_1 \Leftrightarrow c_1' : \mathtt{T} \quad \Gamma \vdash c_2 \Leftrightarrow c_2' : \mathtt{T}}{\Gamma \vdash (c_1 \to c_2) \leftrightarrow (c_1' \to c_2') : \mathtt{T}}$$

$$\frac{\Gamma, \alpha : \kappa \vdash c \Leftrightarrow c' : \mathtt{T}}{\Gamma \vdash \forall(\alpha : \kappa).c \leftrightarrow \forall(\alpha : \kappa).c' : \mathtt{T}}$$

2.3 Introduction to de Bruijn indices

With explicit variables, you must be careful to avoid capture. So let's instead count the number of binders. Substitution coincides with shifting of the indices.

$\uparrow_i^j$	Add j to all variables, except those bound within i binders
$\uparrow_i^j (\lambda.e) = \lambda.\ \uparrow_{i+1}^j e$	
$\uparrow_i^j k = k$	if $k < i$
$\uparrow_i^j k = k + j$	otherwise

3 Explicit substitution *(1/23/2018)*

In this lecture, we introduce a representation of substitutions as mathematical objects rather than resorting to meta-mathematical reasoning. We adopt the presentation of de Bruijn indices that will be used in the first project.

3.1 Intuition and Examples

Throughout, we use an *ordered context*. As a substitution is being performed, variables are bound to this context. There may still be free variables whose index exceeds the number of items in the context; we will have to decrement the indices of these.

Given the term (in de Bruijn indices)

$$(\lambda.\lambda.2 + 0 + 3)[M/0] \quad ,$$

it would be quite nice if this reduced to

$$(\lambda.\lambda.M + 0 + 2) \quad .$$

Notice again how we talk about substitution as being part of the *term* and of being *reduced*—this is the first sign that we're modeling substitution explicitly.

The other context explored today is *shifting* (or lifting), which involves yet more terms to the ordered context. Let's give some examples of familiar terms and their de Bruijn equivalents.

$$w \vdash \lambda y.\lambda z.z + w \qquad \Longrightarrow \qquad \lambda.\lambda.0 + 2 \tag{1}$$
$$w, x \vdash \lambda y.\lambda z.z + w \qquad \Longrightarrow \qquad \lambda.\lambda.0 + 3 \tag{2}$$

In (2), we must shift the index of w by 1 to refer to the correct position in the ordered context.

3.2 Modelling substitutions explicitly

We model our approach after `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-54.pdf`, taking some liberties, like indexing from 0.

First, our grammar of terms:

$$M \ ::= \ i \mid \lambda.M \mid M\,M \mid M[\sigma]$$

i denotes de Bruijn indices, and postfix brackets indicate substitution. Now, our grammar of substitutions σ.

$$\sigma \ ::= \ M \cdot \sigma \mid \uparrow^n$$

$\cdot$ functions as cons[2]. We usefully abbreviate $\uparrow^0$ as id, and $\uparrow^1$ as $\uparrow$.

Substitution is given meaning by equations:

$$\begin{aligned}
0[M \cdot \sigma] &= M \\
(i+1)[M \cdot \sigma] &= i[\sigma] \\
i[\uparrow^n] &= i + n \\
(M_1\ M_2)[\sigma] &= M_1[\sigma]\ M_2[\sigma] \\
(\lambda.M)[\sigma] &= \lambda.M[0 \cdot (\sigma \circ \uparrow)]
\end{aligned}$$

[2]In lecture, we used a period ".", but this notation is misleadingly suggestive of binding.

Unlike the reference, we don't model $\circ$ as primitive but rather define it as a binary operator over substitutions. Our goal is for $M[\sigma \circ \sigma'] = M[\sigma][\sigma']$. Here's how you compute it, assuming $\cdot$ to bind tighter than $\circ$.

$$\begin{aligned}
\uparrow^m \circ \uparrow^n &= \uparrow^{m+n} \\
\mathrm{id} \circ M \cdot \sigma &= M \cdot \sigma \\
\uparrow^{m+1} \circ M \cdot \sigma &= \uparrow^n \cdot \sigma \\
M \cdot \sigma \circ \sigma' &= M[\sigma'] \cdot (\sigma \circ \sigma')
\end{aligned}$$

Composition here can be pronounced "before." The last rule is the only interesting one, indicating that later substitutions may depend on prior ones, since we must perform $M[\sigma']$.

For example, we would like the following to hold:

$$\begin{aligned}
(0+1)[M\cdot \uparrow \circ \uparrow^4] &= (0+1)[M\cdot \uparrow][\uparrow^4] \\
&= (M+1)[\uparrow^4] \\
&= M[\uparrow^4] + 5
\end{aligned}$$

And by our equations, it in fact does.

$$\begin{aligned}
(0+1)[M\cdot \uparrow \circ \uparrow^4] &= (0+1)[M[\uparrow^4] \cdot (\uparrow \circ \uparrow^4)] \\
&= (0+1)[M[\uparrow^4]\cdot \uparrow^5] \\
&= M[\uparrow^4] + 5
\end{aligned}$$

3.3 Rule conversions

With these notions of indices and binding, we convert some rules from from Judgment 2.6 and Judgment 2.4. Importantly, the context becomes ordered with this conversion.

Judgment 3.1 (Rule conversions with de Bruijn indices)

$$\frac{}{(\lambda(\alpha : \kappa).c)\ c' \rightsquigarrow [c'/\alpha]c} \quad \Longrightarrow_{\text{de Bruijn}} \quad \frac{}{(\lambda\kappa.c)\ c' \rightsquigarrow c[c' \cdot \mathrm{id}]}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash c\ \alpha \Leftrightarrow c'\ \alpha : \kappa_2}{\Gamma \vdash c \Leftrightarrow c' : \kappa_1 \to \kappa_2} \quad \Longrightarrow_{\text{de Bruijn}} \quad \frac{\Gamma, \kappa_1 \vdash c[\uparrow]\ 0 \Leftrightarrow c'[\uparrow]\ 0 : \kappa_2}{\Gamma \vdash c \Leftrightarrow c' : \kappa_1 \to \kappa_2}$$

3.4 Substitutions we'll use

Rather than using substitutions in their full generality, we are really concerned with substitutions of the form $[0 \cdot 1 \cdots n-1 \cdot M_1[\uparrow^n] \cdots M_k[\uparrow^n] \cdot \uparrow^{n+\ell}]$. This is still pretty general:

1. $[M \cdot \mathrm{id}]$ has $n = 0, k = 1, \ell = 0$.

2. $[\uparrow^\ell]$ has $n = 0, k = 0$.

3. For σ of the desired form, the substitution $[0 \cdot (\sigma \circ \uparrow)]$ retains the form:

$$0 \cdot ((0 \cdot 1 \cdots n-1 \cdot M_1[\uparrow^n] \cdots M_k[\uparrow^n] \cdot \uparrow^{n+\ell}) \circ \uparrow)$$

$$\Downarrow$$

$$0 \cdot 1 \cdots n \cdot M_1[\uparrow^{n+1}] \cdots M_k[\uparrow^{n+1}] \cdot \uparrow^{n+\ell+1}$$

3.5 The type theory of explicit substitution

Assuming the type theory of the underlying language, we want to show that:

$$\textit{If} \quad \Gamma \vdash \sigma : \Gamma' \quad \textit{and} \quad \Gamma' \vdash M : A, \quad \textit{then} \quad \Gamma \vdash M[\sigma] : A[\sigma].$$

There are a few things to note. One is that the type of a substitution σ can be described be an ordered list of types; in other words, the type of σ is a context Γ'. Next, the use of $A[\sigma]$ indicates that the type A may refer to variables in the context Γ. This will require some thought about dependent types. Finally, this framework applies equally well to M indicating a term and A a type as it does to M indicating a type and A a kind.

Below, we assume Γ to be an ordered context of the form ε or Γ, A.

Judgment 3.2 (Substitution typing—general rules)

$$\frac{\Gamma \vdash \sigma : \Gamma' \quad \Gamma \vdash M : A[\sigma]}{\Gamma \vdash M \cdot \sigma : \Gamma', A}$$

$$\frac{\Gamma = \Gamma', A_1, \ldots, A_k}{\Gamma \vdash \uparrow^k : \Gamma'} \qquad \frac{}{\varepsilon \vdash \mathrm{id} : \varepsilon}$$

We can use these rules to perform a simple derivation after converting to de Bruijn indices. We wish to derive the RHS of:

$$y : \texttt{int} \vdash [\texttt{int}/\alpha, y/x] : \alpha : \texttt{T}, x : \alpha \qquad \Longrightarrow_{\text{deBruijn}} \qquad \texttt{int} \vdash \texttt{int} \cdot 0 \cdot \uparrow : \varepsilon, \texttt{T}, 0$$

We show this, explicitly including ε as part of the types of substitutions (so that it is slightly easier to distinguish judgments on the types of substitutions and judgments on the kinds of types).

$$\frac{\dfrac{\dfrac{}{\texttt{int} \vdash \uparrow : \varepsilon} \quad \dfrac{}{\texttt{int} \vdash \texttt{int} : \texttt{T}[\uparrow]}}{\texttt{int} \vdash \texttt{int} \cdot \uparrow : \varepsilon, \texttt{T}} \qquad \dfrac{\dfrac{}{\texttt{int} \vdash 0 : \texttt{int}}}{\texttt{int} \vdash 0 : 0[\texttt{int}. \uparrow]}}{\texttt{int} \vdash 0 \cdot \texttt{int} \cdot \uparrow : \varepsilon, \texttt{T}, 0}$$

We perform transformations on the rules of Judgment 1.1, now, in this type-theoretic setting.

Judgment 3.3 (Type kind with substitutions)

$$\frac{\Gamma(\alpha) : \kappa}{\Gamma \vdash \alpha : \kappa} \quad \Longrightarrow \quad \frac{\Gamma(i) : \kappa}{\Gamma \vdash i : \kappa[\uparrow^{i+1}]}$$

$$\frac{\Gamma, \alpha : \kappa \vdash c : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).c : \kappa \to \kappa'} \quad \Longrightarrow \quad \frac{\Gamma, \kappa \vdash c : \kappa'[\uparrow]}{\Gamma \vdash \lambda \kappa.c : \kappa \to \kappa'}$$

Unchanged rules:

$$\frac{\Gamma \vdash c_1 : \kappa \to \kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1\, c_2 : \kappa'} \qquad \frac{\Gamma, \alpha : \kappa \vdash c : \texttt{T}}{\Gamma \vdash \forall(\alpha : \kappa).c : \texttt{T}}$$

The $\forall$ rule may be surprising, since there are no shifts involved, but this is only because there's no need to shift `T` up as in the lambda rule; `T` has no variables.

Finally, we give the same sort of transformations for our term language, transforming rules from Judgment 1.2.

Judgment 3.4 (Term type with substitution)

$$\frac{\Gamma(x) : \tau}{\Gamma \vdash x : \tau} \quad \Longrightarrow \quad \frac{\Gamma(i) : \tau}{\Gamma \vdash i : \tau[\uparrow^{i+1}]}$$

$$\frac{\Gamma \vdash e : \forall(\alpha : \kappa).\tau' \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e[\tau] : [\tau/\alpha]\tau'} \quad \Longrightarrow \quad \frac{\Gamma \vdash e : \forall\kappa.\tau \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c \cdot \mathrm{id}] : \tau[c \cdot \mathrm{id}]}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau' \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \lambda(x : \tau).e : \tau \to \tau'} \quad \Longrightarrow \quad \frac{\Gamma, \tau \vdash e : \tau'[\uparrow]}{\Gamma \vdash \lambda\tau.e : \tau \to \tau'}$$

Unchanged rules:

$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

Note that the lambda rule above was extrapolated by myself and not specifically covered in lecture. Mistakes are mine. It is known.

4 Singleton Kind Calculus *(1/25/2018)*

4.1 Remarks from previous lecture

- Although in the previous section I treated $M[\sigma]$ as a term, Prof. Crary wanted to present it as an operation on terms.
- $M\cdot \uparrow^k \equiv M \cdot k \cdot k+1 \cdot \cdots$

4.2 Toward the Singleton Kind Calculus

Starting with F_ω + products, let's add one more kind scheme.

$$\kappa \;::=\; \mathtt{T} \;|\; \kappa \to \kappa \;|\; \kappa \times \kappa \;|\; \mathtt{S}(c)$$

From a set-theoretic point of view, $\mathtt{S}(c)$ is the singleton set $\{c\}$, but we of course look down on set theory, so don't go too far with this.

For sure, $\mathtt{int} : \mathtt{S}(\mathtt{int})$. But we also want that $(\lambda\alpha.\alpha)\ \mathtt{int} : \mathtt{S}(\mathtt{int})$. (The kind of the parameter α was unspecified in class, but it seems that we want to allow $\alpha : \mathtt{T}$ in addition to the obvious $\alpha : \mathtt{S}(\mathtt{int})$.)

Let's review the components of a basic ML signature:

```
type t
type u
val a : t
```

The type component of this signature is $\alpha : \mathtt{T} \times \mathtt{T}$, and the value component is $a : \pi_2\ \alpha$. In an explicit, made-up ML syntax, this would look like:

```
typeconstructor t : type
typeconstructor u : type
val a : t
```

This is nice and breezy, but what happens if we need to introduce sharing constraints between types? The classic example of the non-geopolitical Diamond Import Problem is where a lexer and parser signature both reference a symbol table module. A compiler, needing to make use of both a lexer and a parser, must specify that its lexer and parser don't deviate in symbol table. (A remark was made that Haskell solves this using "sharing by construction (parameterization)," in contrast to the ML orthodoxy's "sharing by specification (fibration)." There are category-theoretical implications to this all, believe it or not. I don't.)

A complication: how to kind $\lambda(\alpha : \mathtt{T}).\mathtt{int}$? Surely it's $\mathtt{T} \to \mathtt{T}$, but also it's $\mathtt{T} \to \mathtt{S}(\mathtt{int})$. Fine, but what about $\lambda(\alpha : \mathtt{T}) : \alpha$? Again, $\mathtt{T} \to \mathtt{T}$ should groove, but we have no way to assign the kind $\mathtt{T} \to \mathtt{S}(?)$. This formulation doesn't appear to be entirely satisfactory, so let's nix it and throw dependent kinds into the mix.

4.3 Dependent kinds

We instead adopt this grammar of kinds.

$$\kappa \;::=\; \mathtt{T} \;|\; \Pi(\alpha : \kappa).\kappa \;|\; \Sigma(\alpha : \kappa).\kappa \;|\; \mathtt{S}(c)$$

Somehow, we want to set this up so:

$$\begin{aligned} \lambda(\alpha : \mathtt{T}).\alpha &: \Pi(\alpha : \mathtt{T}).T \\ &: \Pi(\alpha : \mathtt{T}).\mathtt{S}(\alpha) \end{aligned}$$

where the second option is clearly the best. So good that no other type has that kind.

Arrows (exponentiation) are iterated products; pairs (products) are iterated sums. We can still retain the old notation where there is no dependence. Where we would say $\langle \texttt{int}, \texttt{int} \rangle : \texttt{T} \times \texttt{T}$ before, we now say $\langle \texttt{int}, \texttt{int} \rangle : \Sigma(\alpha : \texttt{T}).\texttt{S}(\alpha)$, and eruditely.

For a signature with kind $\Sigma(\texttt{t} : \texttt{S}(\texttt{int})).\texttt{S}(\texttt{t} \times \texttt{t})$, look no further than

```
type t = int
type u = t * t
```

But we impugn kinds with dependence. It's possible for a kind to be bad, so we resort to the usual interrogation tactics.

Judgment 4.1 (Innocence of kinds (kind formation)) $\Gamma \vdash \kappa : \text{kind}$

$$\frac{}{\Gamma \vdash \texttt{T} : \text{kind}} \qquad \frac{\Gamma \vdash \tau : \texttt{T}}{\Gamma \vdash \texttt{S}(\tau) : \text{kind}}$$

$$\frac{\Gamma \vdash \kappa_1 : \text{kind} \quad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \text{kind}}{\Gamma \vdash \Pi(\alpha : \kappa_1).\kappa_2 : \text{kind}} \qquad \frac{\Gamma \vdash \kappa_1 : \text{kind} \quad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \text{kind}}{\Gamma \vdash \Sigma(\alpha : \kappa_1).\kappa_2 : \text{kind}}$$

4.4 Equivalence in kind

Before, in the judgment $\Gamma \vdash c_1 \equiv c_2 : \kappa$, neither the context Γ nor the kind κ was needed. But, just as Mercury casts capricious shadows on the surface of Mars, so too introduces the fallen kind new pathways to deceit. Dependent, yes—dependent on sin.

$$\lambda(\alpha : \texttt{T}).\alpha \overset{?}{\equiv} \lambda(\alpha : \texttt{T}).\texttt{int}$$

At $\texttt{T} \to \texttt{T}$, no. At $\texttt{S}(\texttt{int}) \to \texttt{T}$, yeah. Subkinding helps: $\texttt{S}(\texttt{int}) \to \texttt{T} \geq \texttt{T} \to \texttt{T}$. The kind κ is necessary to check equivalence. The context is clearly necessary for dependent kinds, and I don't feel like reproducing the example from lecture.

Here, our declarative judgments:

$\Gamma \vdash \kappa : \text{kind}$	kind formation (done!)
$\Gamma \vdash \kappa_1 \equiv \kappa_2 : \text{kind}$	kind equivalence
$\Gamma \vdash \kappa_1 \leq \kappa_2$	subkinding
$\Gamma \vdash c : \kappa$	type formation
$\Gamma \vdash c_1 \equiv c_2 : \kappa$	type equivalence
$\Gamma \vdash e : c$	term formation
$\vdash \Gamma$ ok	context ok-ness

Above the line is where the action is. Below, unchanged. Let's briefly state regularity conditions. If $\vdash \Gamma$ ok, then:

(i) if $\Gamma \vdash \kappa \equiv \kappa' : \text{kind}$, then $\Gamma \vdash \kappa : \text{kind}$ and $\Gamma \vdash \kappa : \text{kind}$.

(ii) if $\Gamma \vdash \kappa \leq \kappa'$, then $\Gamma \vdash \kappa : \text{kind}$ and $\Gamma \vdash \kappa : \text{kind}$.

(iii) if $\Gamma \vdash c : \kappa$, then $\Gamma \vdash \kappa : \text{kind}$.

(iv) if $\Gamma \vdash c \equiv c' : \kappa$, then $\Gamma \vdash c : \kappa$ and $\Gamma \vdash c' : \kappa$.

(v) if $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \texttt{T}$.

We start with defining kind checking for types.

Judgment 4.2 (Type formation (incomplete)) $\Gamma \vdash c : \kappa$

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

$$\frac{\Gamma \vdash \kappa : \text{kind} \quad \Gamma, \alpha : \kappa \vdash c : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).c : \Pi(\alpha : \kappa).\kappa'} \qquad \frac{\Gamma \vdash c_1 : \Pi(\alpha : \kappa).\kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1\, c_2 : [c_2/\alpha]\kappa'}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \text{kind} \quad \Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_2 : [c_1/\alpha]\kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle : \Sigma(\alpha : \kappa_1).\kappa_2}$$

In the elimination rule for dependent products, κ and κ' live under a different number of binders. Watch out when implementing de Bruijn indices.

The regularity condition in the premise of the dependent sum rule is required. For example, if $\gamma : \mathtt{S}(\mathtt{int}) \to \mathtt{T}$, the substitution of a type constructor for γ in $\langle \mathtt{int}, \gamma\ \mathtt{int} \rangle$ will appear to license the kind $\Sigma(\alpha : \mathtt{T}).\mathtt{S}(\gamma\ \alpha)$, but this in fact should not be the case.

That the dependent sum is dual to the dependent product is to be expected, and is "standard for dependent types." This is not entirely satisfactory, so hopefully more exposure to this notion will make this clear. (Presumably, π_1 and π_2 are still eliminatory for dependent sums, but this went unmentioned.)

4.5 Remarks

- Why can't the declaration `type t = int` be interpreted as introducing an alias for the type `int`? Prof. Crary's explanation was that the signature with this binding is a subkind of the signature specifying only `type t`, and that the alias interpretation is not satisfactory in this regard.

- For more on the Diamond Import Problem, see Pierce's *Advanced Topics in Types and Programming Languages*.