Higher-Order Polymorphism

Vincent St-Amour

March 8, 2012

1 Why should I care?

Let's assume we're in $\lambda_{\rightarrow}$with products.

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\ e \mid (e, e) \mid \text{fst } e \mid \text{snd } e \qquad v ::= \lambda x{:}\tau.\, e \mid (v, v)$$

$$\tau ::= \tau \to \tau \mid \tau \times \tau \qquad \Gamma ::= \emptyset \mid \Gamma, x : \tau$$

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

$$\text{T-Var}\ \frac{}{\Gamma \vdash x : \tau}\ (\tau = \Gamma(x)) \qquad \text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{T-Fst}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \qquad \text{T-Snd}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2}$$

How do we encode triples that contain elements of the same type?

$$\text{Triple } \alpha = \alpha \times \alpha \times \alpha$$

Easy, right? That's what we've been doing all semester. But let's stop for a second and think: what *is* this thing? What kind of object is this is $\lambda_{\rightarrow}$? The answer is: it's not.

All this time, we've been appealing to our meta-language to compensate for the insufficient abstraction facilities of our language. Triple is really a type macro, not an object in our language.

Next question: what does it *mean*? Well, it kind of looks like a function definition of some sort, so we could assume that it performs some kind of substitution. But we can't really be sure, since we don't really have semantics for our meta-language.

More questions remain: what's the scope of this "definition"? This side of the board? The rest of the lecture? Who gets to define these shorthands? The designer of the type system? Should programmers using the system be in on the fun?

In the rest of the lecture, we'll give these constructions meaning by including them in the type system.

2 Type Operators and Kinding

We've decided that we wanted to add things like Triple to our type system. Now the question is: where do we put them? They work with types, so it seems sensible to put them at the type level. Now, how do should we represent them? They clearly are some kinds of functions, so let's represent them as functions. Since we have functions, we also need variables and applications.

$$\tau ::= \ \ldots \ \mid \ \alpha \ \mid \ \lambda\alpha.\,\tau \ \mid \ \tau\ \tau$$

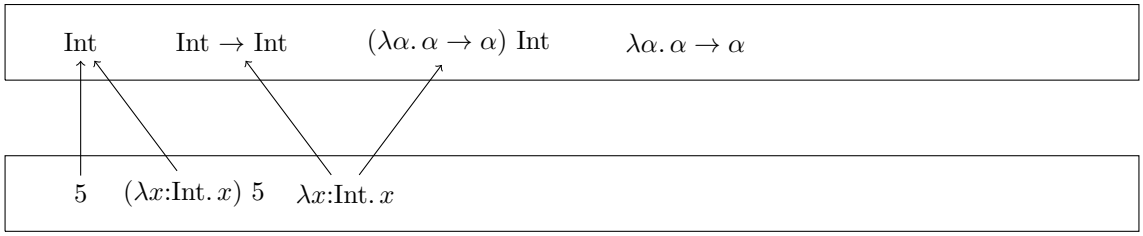
Groovy. Let's see what Triple looks like.

$$\text{Triple} \ = \lambda\alpha.\,\alpha \times \alpha \times \alpha$$

Awesome. Can we find a term that inhabits it?

Oh.

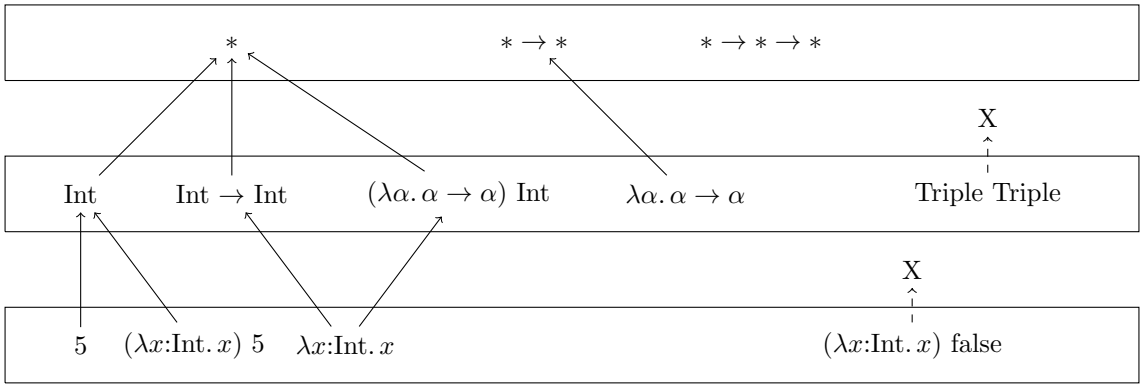
It seems that there really are multiple *kinds* of things at the type level, and not all of them classify terms.



We'll call those types that classify terms *proper types* and we'll call the others *type operators*. Now, what about stuff like:

$$\text{Triple Triple}$$

Yeah, that makes no sense. We want to rule that out. Now remember, what did we use to rule out terms that made no sense? That's right, a type system! This suggests that we need a "type system" for types, too. To avoid confusion, we'll call that a *kind* system, since kinds are the "types" of types.



Let's design our kind system. What do you think it's going to look like? Well, our type level looks an awful lot like the term level for $\lambda_{\to}$, doesn't it? So, let's ask ourselves, What Would $\lambda_{\to}$ Do? The only thing missing is the annotations on the abstractions, so let's add them.

$$\tau ::= \ \ldots \ \mid \ \lambda\alpha{::}\kappa.\,\tau$$

Let's note that this annotation is a *kind* annotation.

Based on our intuition, let's define the kind system by mimicking $\lambda_{\to}$'s type system. Just like $\lambda_{\to}$needs a type environment (Γ), we will need a kind environment (Δ).

$$\Delta ::= \ \emptyset \ \mid \ \Delta, \alpha :: \kappa$$

We'll have two classes of kind expressions. $*$, for proper types, and $\kappa \Rightarrow \kappa$ for type operators.

$$\kappa ::= \ \ * \ \mid \ \kappa \Rightarrow \kappa$$

Now we have all the pieces to define the kinding judgement. This is what it means for a type to be well-kinded.

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

$$\text{K-TVar}\ \frac{}{\Delta \vdash \alpha :: \kappa}\ (\kappa = \Delta(\alpha)) \qquad \text{K-Abs}\ \frac{\Delta, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Delta \vdash \lambda \alpha{::}\kappa_1 . \tau :: \kappa_1 \Rightarrow \kappa_2} \qquad \text{K-App}\ \frac{\Delta \vdash \tau_1 :: \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau_2 :: \kappa_1}{\Delta \vdash \tau_1\ \tau_2 :: \kappa_2}$$

$$\text{K-Arrow}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \to \tau_2 :: *} \qquad \text{K-Prod}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \times \tau_2 :: *}$$

This looks just like the typing rules for $\lambda_{\to}$, as expected. The arrow and product rules look just like the rule for $\lambda_{\to}$'s $+$ and similar operations, which makes sense, given that $\to$ and $\times$ are just base operations on types.

Let's typecheck some terms now. Oh wait, do we need to update the typing judgement? We at least need to make sure that the types in our program are well-kinded. The abstraction rule is the only one that introduces types, so it's the only one we need to update.

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

$$\text{T-Abs}\ \frac{\cdot \vdash \tau_1 :: * \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 . e : \tau_1 \to \tau_2}$$

Now we can typecheck programs. Let's try this one:

$$\lambda x{:}(\text{Triple Int}).\ \text{fst}\ x$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha :: * \vdash \alpha :: *}\text{K-TVar}}{\alpha :: * \vdash \alpha \times \alpha \times \alpha :: *}\text{K-Prod}}{\cdot \vdash \lambda \alpha :: * . \alpha \times \alpha \times \alpha :: * \Rightarrow *}\text{K-Abs} \qquad \cdot \vdash \text{Int} :: *}{\cdot \vdash (\lambda \alpha :: * . \alpha \times \alpha \times \alpha)\ \text{Int} :: *}\text{K-App}}{\cdot \vdash (\text{Triple Int}) :: *}\text{Def-Triple} \qquad x : (\text{Triple Int}) \vdash \text{fst}\ x : ?}{\cdot \vdash \lambda x{:}(\text{Triple Int}).\ \text{fst}\ x : (\text{Triple Int}) \to ?}\text{T-Abs}$$

See how the typing judgement "calls out" to the kinding judgement? That part works out perfectly.[1] However, what happens on the right side?

$$x : (\text{Triple Int}) \vdash \text{fst}\ x : ?$$

[1]Except that Triple is still defined at the meta-level. We'll fix that soon. If it makes you happy, you can replace Triple with its definition wherever it appears.

fst needs something of some pair type, but gets something of type

$$(\lambda\alpha::*\,.\,\alpha \times \alpha \times \alpha)\ \text{Int}$$

so we're stuck. What do we do?

Clearly, we need a way to express that this type is *equivalent* to Int × Int × Int. Let's define a type equivalence relation!

$\boxed{\tau \equiv \tau'}$

$$\text{Q-Refl}\ \frac{}{\tau \equiv \tau} \qquad \text{Q-Symm}\ \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \text{Q-Trans}\ \frac{\tau \equiv \tau'' \qquad \tau'' \equiv \tau'}{\tau \equiv \tau'}$$

$$\text{Q-Arrow}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \text{Q-Prod}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'}$$

$$\text{Q-Abs}\ \frac{\tau \equiv \tau'}{\lambda\alpha::\kappa.\,\tau \equiv \lambda\alpha::\kappa.\,\tau'} \qquad \text{Q-App}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'} \qquad \text{Q-AppAbs}\ \frac{}{(\lambda\alpha::\kappa.\,\tau)\ \tau_2 \equiv \tau[\tau_2/\alpha]}$$

Now let's plug it in our typing relation to get rid of our earlier stalemate.

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

$$\text{T-Eq}\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \cdot \vdash \tau :: *}{\Gamma \vdash e : \tau}$$

Let's see if that actually fixes our problem.

$$\text{T-Fst}\ \frac{\text{T-Eq}\ \dfrac{\text{T-Var}\ \dfrac{}{x : (\text{Triple Int}) \vdash x : (\text{Triple Int})} \qquad \text{Q-AppAbs}\ \dfrac{}{(\lambda\alpha::*\,.\,\alpha \times \alpha \times \alpha)\ \text{Int} \equiv \text{Int} \times \text{Int} \times \text{Int}}}{x : (\text{Triple Int}) \vdash x : \text{Int} \times \text{Int} \times \text{Int}} \qquad \text{K-Prod}\ \dfrac{\cdot \vdash \text{Int} :: * \qquad \cdot \vdash \text{Int} :: *}{\cdot \vdash \text{Int} \times \text{Int} \times \text{Int} :: *}}{x : (\text{Triple Int}) \vdash \text{fst}\ x : \text{Int}}$$

It does, great.

So, what have we done? We have extended $\lambda_\to$with type operators, which are basically type-level functions. To do that, we needed to define a "type system" for types, and check well-kindedness of types as part of the typechecking process. We also needed to define a type equivalence relation, otherwise we'd have no way to actually *use* our new type operators. The resulting system is called λ_ω. A full definition (all in one place) is presented in the appendix.

Before we move on, some observations:

- We didn't add anything to $\lambda_\to$'s term level. All the action happened at the type level.
- We now have two λs, one for the term level, one for the type level. Once we add System F to the mix, it will get even more fun.
- The kinding judgement only needs Δ, not Γ. Similarly, the typing judgement only needs Γ, not Δ.
- When the typing relation calls out to the kinding relation, it always does do with an empty Δ. Why is that? Well, terms can't bind type variables, so they can't contribute anything useful to the kinding judgement. That will change soon.

3 Aside: Why Stop at Kinds?

We just added "types" to types. Couldn't we do the same for kinds?

Type systems answer: Kinds ought to be enough for anybody.

Math answer: Sure, let's call the next level up *sorts* and keep going! This is the *pure type systems* framework.

4 Higher-Order Polymorphism

We've achieved the first half of our motivation: we now know what Triple *means*. But the second half still remains: when we write

$$\text{Triple} = \lambda\alpha{::}\kappa.\, \alpha \times \alpha \times \alpha$$

what's the scope of this definition, and who gets to define these?

The answer lies with System F. With System F, terms can bind types, so we can do things like:

$$(\Lambda\text{TripleInt}.\ \ldots)\ \text{Int} \times \text{Int} \times \text{Int}$$

and use TripleInt as a type in our program. Intuitively, if we were to combine $\lambda_\omega$and System F, we would be able to express things like:

$$(\Lambda\text{Triple}.\,(\Lambda\text{TripleInt}.\ \ldots)\ (\text{Triple Int}))\ \lambda\alpha{::}\kappa.\, \alpha \times \alpha \times \alpha$$

[2] In this example, the scope of Triple is clearly defined, and it's obvious that anyone can define their own type operators, which is what we set off to achieve. Let's see what the combination of $\lambda_\omega$and System F would look like.

We will extend our definition of $\lambda_\omega$to include System F. Most of the changes will be similar to those that turn $\lambda_{\to}$into System F. First, we extend the term syntax to include type abstraction and application:

$$e ::= \ \ldots \ \mid\ \Lambda\alpha{::}\kappa.\, e \ \mid\ e\,[\tau] \qquad\qquad v ::= \ \ldots \ \mid\ \Lambda\alpha{::}\kappa.\, e$$

Since we want to be able to abstract over both proper types and type operators (as seen in our example), we need a kind annotation on the Λs. We could have added these annotations in System F too, but since it has only one kind ($*$), they wouldn't be especially useful.

Then, we add universal types (again with kind annotations) to the grammar of types.

$$\tau ::= \ \ldots \ \mid\ \forall\alpha{::}\kappa.\tau$$

Kinds and environments are unchanged from λ_ω. Δ still maps type variables to kinds, unlike System F's, which just contains type variables (or maps them all to $*$, if you prefer).

We need to extend the kinding and type equivalence relations for universals.

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

$$\textsc{K-All}\ \frac{\Delta, \alpha :: \kappa \vdash \tau :: *}{\Delta \vdash \forall\alpha{::}\kappa.\tau :: *}$$

[2] However, since Triple and TripleInt are bound by a Λ, they would be abstract types. To be able to use these types in a non-abstract way, we'd need to package them as existentials. Type definition is tricky.

$\boxed{\tau \equiv \tau'}$

$$\text{Q-All}\ \frac{\tau \equiv \tau'}{\forall \alpha{::}\kappa.\tau \equiv \forall \alpha{::}\kappa.\tau'}$$

Finally, we need to update our typing rules to carry Δs around, like in System F, and add new rules for type abstraction and application.

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

$$\text{T-Var}\ \frac{}{\Delta; \Gamma \vdash x : \tau}\ (\tau = \Gamma(x)) \qquad \text{T-Abs}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Eq}\ \frac{\Delta; \Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \Delta \vdash \tau :: *}{\Delta; \Gamma \vdash e : \tau} \qquad \text{T-TAbs}\ \frac{\Delta, \alpha :: \kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha{::}\kappa.\, \tau : \forall \alpha{::}\kappa.\tau} \qquad \text{T-TApp}\ \frac{\Delta; \Gamma \vdash e : \forall \alpha{::}\kappa.\tau \qquad \Delta \vdash \tau' :: \kappa}{\Delta; \Gamma \vdash e\,[\tau'] : \tau[\tau'/\alpha]}$$

Some observations:

- In the abstraction and type equality rules, we now need Δ from the term level to do kind-checking, why is that? Since terms can now bind types, information about the kinds of type variables can now be found in terms, which means that terms can now provide useful information to the kinding relation.
- The type well-formedness condition in System F's type application rule now uses the (more interesting) kinding relation[3]. And of course, the type argument needs to be of the right kind.

The system we have at this point is called the higher-order polymorphic λ-calculus, F_ω (with products).

5 Aside: F What?

The ω in F_ω is no accident. F_ω can be defined as the limit of a hierarchy of type systems, which we will call F_1, F_2, etc.

The base case, F_1, is $\lambda_\to$. The only kind is $*$, and no quantification or type abstraction is allowed. F_2 is the second-order λ-calculus, System F. Still only one kind, but quantification is allowed.

In general, we define a hierarchy of kinds:

$$K_1 = \emptyset$$

$$K_{i+1} = \{*\} \cup \{\kappa_1 \Rightarrow \kappa_2 | K_1 \in K_i \wedge \kappa_2 \in K_{i+1}\}$$

$$K_\omega = \bigcup_{i \geq 1} K_i$$

Each F_{i+1} can quantify over types with kinds in K_{i+1} and abstract over types with kinds in K_i.

F_3 is the first system where kinding and type equivalence become interesting. At that point, we can quantify over type operators. It is also the last one to not be considered esoteric.[4]

F_ω is the limit of that hierarchy, where we can abstract and quantify over anything (provided everything is layered appropriately).

[3] In a sense, System F also has a kinding relation, but it's equivalent to "Is this term closed?". Interestingly, this corresponds to the untyped λ-calculus's "typing" relation.

[4] By Pierce, at least.

6 The λ Cube

If we look carefully at the systems we defined so far, we can see patterns emerge.

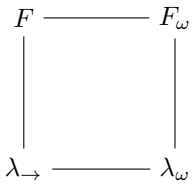
In $\lambda_{\rightarrow}$, terms (functions) can abstract over terms. This is reflected in the typing rules by the presence of Γ, which holds information provided by terms (functions) that is necessary to type-check other terms (their bodies).

In λ_{ω}, types (type operators) can abstract over types. This is reflected in the *kinding* rules by the presence of Δ, which holds information provided by types (type operators) that is necessary to kind-check other types (their bodies). This is in addition to terms abstracting over terms.

In System F, terms (type abstractions) can abstract over types. This is reflected in the *typing* rules by the presence of Δ, which holds information provided by terms (type abstractions) that is necessary to kind-check some types (their bodies). This is in addition to terms abstracting over terms.

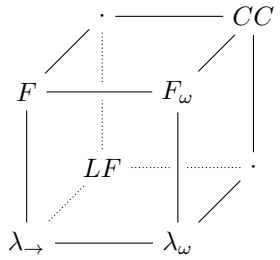
In F_{ω}, all three forms of abstraction are possible. As such, the typing relation features both Γ and Δ, and the kinding relation features Δ.

At this point, we can draw a nice square diagram to represent the relationship between these systems, let's call it the λ square:



The systems on top feature terms abstracting over types and the ones on the right feature types abstracting over types.

Now, if we look for a pattern in these patterns, we come to the following conclusion: What about types abtracting over terms? To represent that, we can add a third axis to our diagram. This axis corresponds to dependent types.[5] We end up with the λ cube:



[5] I won't say more on the subject. Paul will go into more detail tomorrow.

7 Soundness

The presence of computation at the type level in $\lambda_\omega$and $F_\omega$changes the proof of type soundness in intersting ways. We will briefly go over the structure of the proof to see what changes.

7.1 Before We Begin

As nice as our type equivalence relation is, having a directed, "computational" relation will be convenient when doing the proof. Let's define a notion of *parallel type reduction*,[6] which we will then prove to be equivalent to the type equivalence relation.

$\boxed{\tau \Longrightarrow \tau'}$

$$\text{QR-Refl}\ \frac{}{\tau \Longrightarrow \tau} \qquad \text{QR-Arrow}\ \frac{\tau_1 \Longrightarrow \tau_1' \qquad \tau_2 \Longrightarrow \tau_2'}{\tau_1 \to \tau_2 \Longrightarrow \tau_1' \to \tau_2'} \qquad \text{QR-All}\ \frac{\tau \Longrightarrow \tau'}{\forall \alpha{::}\kappa.\tau \Longrightarrow \forall \alpha{::}\kappa.\tau'} \qquad \text{QR-Abs}\ \frac{\tau \Longrightarrow \tau'}{\lambda \alpha{::}\kappa.\,\tau \Longrightarrow \lambda \alpha{::}\kappa.\,\tau'}$$

$$\text{QR-App}\ \frac{\tau_1 \Longrightarrow \tau_1' \qquad \tau_2 \Longrightarrow \tau_2'}{\tau_1\ \tau_2 \Longrightarrow \tau_1'\ \tau_2'} \qquad \text{QR-AppAbs}\ \frac{\tau \Longrightarrow \tau' \qquad \tau_2 \Longrightarrow \tau_2'}{(\lambda \alpha{::}\kappa.\,\tau)\ \tau_2 \Longrightarrow \tau'[\tau_2'/\alpha]}$$

Lemma: $\tau \equiv \tau'$ iff $\tau \Longleftrightarrow^* \tau'$
Proof:

- Case $\longleftarrow$: Obvious.
- Case $\longrightarrow$: The main mismatch between $\equiv$ and $\Longleftrightarrow^*$ is that $\equiv$ can use symmetry and transitivity whenever, while $\Longleftrightarrow^*$ can only use them at the outermost level (i.e. not when deriving $\Longrightarrow$).
  The key insight is that any derivation for $\tau \equiv \tau'$ can be transformed into a chain of derivations

  $$\tau \equiv \tau_1, \tau_1 \equiv \tau_2, \ldots, \tau_n \equiv \tau'$$

  none of which use transitivity, and only use transitivity at the end, or not at all.

In addition, we need to make sure that $\Longrightarrow$ is confluent, to ensure that two equivalent terms are guaranteed to eventually reduce to the same thing. The proof is not especially interesting; our type language is basically $\lambda_\to$, so the proof of confluence is very similar.

7.2 Preservation

As usual, to prove preservation, we need an inversion lemma.

Lemma (Inversion):

1. If $\Delta; \Gamma \vdash \lambda x{:}\tau_1'.\, e : \tau_1 \to \tau_2$ then $\tau_1 \equiv \tau_1'$ and $\Delta; \Gamma, x : \tau_1' \vdash e : \tau_2$. Also, $\Delta \vdash \tau_1' :: *$.
2. If $\Delta; \Gamma \vdash \Lambda \alpha{::}\kappa_1.\, e :: \forall \alpha{::}\kappa_2.\tau$ then $\kappa_1 = \kappa_2$ and $\Delta, \alpha :: \kappa_1; \Gamma \vdash e : \tau$.

To prove that, we will need an intermediate lemma:

Lemma (Perservation of shapes under reduction):

1. If $\tau_1 \to \tau_2 \Longrightarrow^* \tau'$ then $\tau' = \tau_1' \to \tau_2'$ with $\tau_1 \Longrightarrow^* \tau_1'$ and $\tau_2 \Longrightarrow^* \tau_2'$.
2. If $\forall \alpha{::}\kappa.\tau_2 \Longrightarrow^* \tau'$ then $\tau' = \forall \alpha{::}\kappa.\tau_2'$ with $\tau_2 \Longrightarrow^* \tau_2'$.

[6] This proof technique is used elsewhere as well. See, for example, Belo et al. 2011.

Proof is by straightforward induction.

Once we have the inversion lemma, the proof of preservation is mostly straightforward. The interesting case is the type application case, which needs a type substitution lemma (just like the application case needs a term substitution lemma).

Lemma (Type substitution):

1. If $\Delta, \alpha : \kappa' \vdash \tau :: \kappa$ and $\Delta \vdash \beta :: \kappa'$ then $\Delta[\beta/\alpha] \vdash \tau[\beta/\alpha] :: \kappa$.
2. If $\tau \equiv \tau'$ then $\tau[\beta/\alpha] \equiv \tau'[\beta/\alpha]$.
3. If $\Delta, \alpha :: \kappa; \Gamma \vdash e : \tau$ and $\Delta \vdash \tau' :: \kappa$ then $\Delta[\tau'/\alpha]; \Gamma[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.

7.3 Progress

As usual, to prove progress, we need a canonical forms lemma.

Lemma (Canonical forms):

1. If v is a closed value with type $\tau_1 \to \tau_2$, then v is an abstraction.
2. If v is a closed value with type $\forall \alpha :: \kappa . \tau$, then v is a type abstraction.

The proof for the first part goes as follows. There are only two kinds of values in plain F_ω: abstractions and type abstractions. Let's suppose for contradiction that we have a type abstraction with type $\tau_1 \to \tau_2$. Since type abstractions have types of the shape $\forall \alpha :: \kappa . \tau$ (by the typing rules), we need to show:

$$\tau_1 \to \tau_2 \equiv \forall \alpha :: \kappa . \tau$$

Which is equivalent to:

$$\tau_1 \to \tau_2 \Longleftrightarrow^* \forall \alpha :: \kappa . \tau$$

By confluence, this means that there exists a τ' to which these both reduce to. However, by preservation of shapes under reduction, we have that τ' must be both of the shape $\tau_1 \to \tau_2$ and of the shape $\forall \alpha :: \kappa . \tau$, which is a contradiction. The second part is proved in a similar fashion.

Once we have the canonical forms lemma, the proof for progress is straightforward.

8 Decidability

The last question we can ask is: is typechecking F_ω decidable? The answer is yes. Here's the rough intuition.

First of all, the kinding relation is decidable. It's easy to believe; our type language is $\lambda_\to$, which we know is strongly normalizing.[7]

Second, if we remove the equality rule from the typing relation, it clearly is decidable, since all the other rules are syntax-directed. If we replace the equality relation with the reduction relation we defined previously, and only apply it to well-kinded types, everything is still decidable, since well-kinded types are strongly normalizing.

[7] That's a great example of how strongly normalizing languages are useful.

9 F_ω in the Wild

We've looked at use cases for F_ω. Now let's look at some uses of it in actual programming languages.

9.1 Type Constructors in ML

ML type constructors map pretty much exactly to type operators. Our Triple example would be written in ML as follows:

```
type 'a triple = 'a * 'a * 'a
```

9.2 The ML Module System

The ML module system can be encoded in F_ω. Jon will talk about the encoding later in the semester.

9.3 Type Classes in Haskell

Type classes abstract over types. For example,[8] the `Show` type class:

```
class Show a
  method show ::  a -> string
```

can be expressed using a type operator:

$$\text{Show} = \lambda\alpha{::}*\,.\,\{\text{show} : (\alpha \to \text{String})\}$$

The record is "evidence" that α can behave as something of class `Show`. Functions requiring arguments of class `Show`, which have a type like:

```
forall a, b .  Show a, Show b => a -> b -> string
```

can be converted to evidence-passing style:

```
forall a, b .  Show a -> Show b -> a -> b -> string
```

where the records are passed explicitly, and the function found inside used to turn `a`s and `b`s to `string`s.

Type classes can also abstract over arrow kinds. For instance, the `Monad` type class:

```
class Monad (M:*->*)
  return ::  All a:*.a -> M a
  bind ::  All a,b:*.M a -> (a -> M b) -> M b
```

abstracts over type operators. Haskellers call these constructor classes.

[8] Examples from this section are from Greg Morrisett.

A Full Definition of λ_ω (Sans Operational Semantics)

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\ e \qquad\qquad v ::= \lambda x{:}\tau.\, e$$

$$\tau ::= \alpha \mid \lambda \alpha{::}\kappa.\, \tau \mid \tau\ \tau \mid \tau \to \tau \qquad\qquad \kappa ::= * \mid \kappa \Rightarrow \kappa$$

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \qquad\qquad \Delta ::= \emptyset \mid \Delta, \alpha :: \kappa$$

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

$$\text{K-TVar}\ \frac{}{\Delta \vdash \alpha :: \kappa}\ (\kappa = \Delta(\alpha)) \qquad \text{K-Abs}\ \frac{\Delta, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Delta \vdash \lambda \alpha{::}\kappa_1.\, \tau :: \kappa_1 \Rightarrow \kappa_2} \qquad \text{K-App}\ \frac{\Delta \vdash \tau_1 :: \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau_2 :: \kappa_1}{\Delta \vdash \tau_1\ \tau_2 :: \kappa_2}$$

$$\text{K-Arrow}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \to \tau_2 :: *}$$

$\boxed{\tau \equiv \tau'}$

$$\text{Q-Refl}\ \frac{}{\tau \equiv \tau} \qquad \text{Q-Symm}\ \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \text{Q-Trans}\ \frac{\tau \equiv \tau'' \qquad \tau'' \equiv \tau'}{\tau \equiv \tau'} \qquad \text{Q-Arrow}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \text{Q-Abs}\ \frac{\tau \equiv \tau'}{\lambda \alpha{::}\kappa.\, \tau \equiv \lambda \alpha{::}\kappa.\, \tau'}$$

$$\text{Q-App}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'} \qquad \text{Q-AppAbs}\ \frac{}{(\lambda \alpha{::}\kappa.\, \tau)\ \tau_2 \equiv \tau[\tau_2/\alpha]}$$

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

$$\text{T-Var}\ \frac{}{\Gamma \vdash x : \tau}\ (\tau = \Gamma(x)) \qquad \text{T-Abs}\ \frac{\cdot \vdash \tau_1 :: * \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Eq}\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \cdot \vdash \tau :: *}{\Gamma \vdash e : \tau}$$

B Full Definition of F_ω (Sans Operational Semantics)

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\ e \mid \Lambda\alpha{::}\kappa.\, e \mid e\,[\tau] \qquad v ::= \lambda x{:}\tau.\, e \mid \Lambda\alpha{::}\kappa.\, e$$

$$\tau ::= \alpha \mid \lambda\alpha{::}\kappa.\, \tau \mid \tau\ \tau \mid \tau \to \tau \mid \forall\alpha{::}\kappa.\tau \qquad \kappa ::= * \mid \kappa \Rightarrow \kappa$$

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \qquad \Delta ::= \emptyset \mid \Delta, \alpha :: \kappa$$

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

$$\text{K-TVar}\ \frac{}{\Delta \vdash \alpha :: \kappa}\ (\kappa = \Delta(\alpha)) \qquad \text{K-Abs}\ \frac{\Delta, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Delta \vdash \lambda\alpha{::}\kappa_1.\, \tau :: \kappa_1 \Rightarrow \kappa_2} \qquad \text{K-App}\ \frac{\Delta \vdash \tau_1 :: \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau_2 :: \kappa_1}{\Delta \vdash \tau_1\ \tau_2 :: \kappa_2}$$

$$\text{K-Arrow}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \to \tau_2 :: *} \qquad \text{K-All}\ \frac{\Delta, \alpha :: \kappa \vdash \tau :: *}{\Delta \vdash \forall\alpha{::}\kappa.\tau :: *}$$

$\boxed{\tau \equiv \tau'}$

$$\text{Q-Refl}\ \frac{}{\tau \equiv \tau} \qquad \text{Q-Symm}\ \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \text{Q-Trans}\ \frac{\tau \equiv \tau'' \qquad \tau'' \equiv \tau'}{\tau \equiv \tau'} \qquad \text{Q-Arrow}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \text{Q-Abs}\ \frac{\tau \equiv \tau'}{\lambda\alpha{::}\kappa.\, \tau \equiv \lambda\alpha{::}\kappa.\, \tau'}$$

$$\text{Q-App}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'} \qquad \text{Q-AppAbs}\ \frac{}{(\lambda\alpha{::}\kappa.\, \tau)\ \tau_2 \equiv \tau[\tau_2/\alpha]} \qquad \text{Q-All}\ \frac{\tau \equiv \tau'}{\forall\alpha{::}\kappa.\tau \equiv \forall\alpha{::}\kappa.\tau'}$$

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

$$\text{T-Var}\ \frac{}{\Delta; \Gamma \vdash x : \tau}\ (\tau = \Gamma(x)) \qquad \text{T-Abs}\ \frac{\Delta \vdash \tau_1 :: * \qquad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Eq}\ \frac{\Delta; \Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \Delta \vdash \tau :: *}{\Delta; \Gamma \vdash e : \tau} \qquad \text{T-TAbs}\ \frac{\Delta, \alpha :: \kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha{::}\kappa.\, \tau : \forall\alpha{::}\kappa.\tau} \qquad \text{T-TApp}\ \frac{\Delta; \Gamma \vdash e : \forall\alpha{::}\kappa.\tau \qquad \Delta \vdash \tau' :: \kappa}{\Delta; \Gamma \vdash e\,[\tau'] : \tau[\tau'/\alpha]}$$