richardwu.ca

CS 442 Final Exam Guide

Principle of Programming Languages

Prabhakar Ragde • Winter 2018 • University of Waterloo

Last Revision: April 11, 2018

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

In general, theorems stated below will need to be proved for each individual language and again for any changes to the language.
We use **structural induction** on either the terms or some other part of the theorem, for example induction on the **evaluation rules** for $\to$.

1 Evaluation rules

Definition 1.1 (Small-step evaluation)**.** We say $t \to t'$ or t reduces to t' if we can take 1 step (in terms of evaluation rules) to get from t to t'.

Theorem 1.1 (Determinsitic evaluation)**.** If $t \to t'$ and $t \to t''$, then $t' = t''$.

Theorem 1.2 (Normal form must be a value)**.** If t is in normal form, then t is a value.

Definition 1.2 (Multi-step evaluation)**.** We define the multi-step evaluation relation $\to^*$

$$t \to_F^* t \tag{F1}$$

$$\frac{t \to t'' \quad t'' \to_F^* t'}{t \to_F^* t'} \tag{F2}$$

$$t \to_M^* t \tag{M1}$$

$$\frac{t \to t'}{t \to_M^* t'} \tag{M2}$$

$$\frac{t \to_M^* t'' \quad t'' \to_M^* t'}{t \to_M^* t'} \tag{M3}$$

$$t \to_L^* t \tag{L1}$$

$$\frac{t \to_L^* t'' \quad t'' \to t'}{t \to_L^* t'} \tag{L2}$$

where we have three equivalent set of rules F, M, L for first, middle and last.

Theorem 1.3 (Unique normal form)**.** If $t \to^* t'$ and $t \to^* t''$, and t' and t'' are normal forms, then $t' = t''$.

Theorem 1.4 (Strong normalization)**.** For every term t there is a normal form t' such that $t \to^* t'$.

Definition 1.3 (Stuck terms)**.** A term that is in normal form but not a value is called **stuck**. For example in the untyped calculus with arithmetic, `succ true` is stuck.

Definition 1.4 (Big-step evaluation)**.** We say $t \Downarrow v$ which relates t to the value v to which it evaluates.

Definition 1.5 (Multi-step iff big-step)**.** For all terms t and all values v, $t \to^* v$ *iff* $t \Downarrow v$.

2 Untyped lambda calculus

Definition 2.1 (Abstractions)**.** The term $\lambda x.t$ is called an **abstraction** where x is called the **variable** and t the **body** (it is equivalent to a function with one parameter x).
Abstractions extend as far right i.e. $\lambda x.t\ s \equiv \lambda x.(t\ s)$.

Definition 2.2 (Applications)**.** The term $t\ s$ is called an **application** where t is the **rator** and s is the **rand**.
Applications are *left-associative* i.e. $t\ s\ r \equiv ((t\ s)\ r)$.

Definition 2.3 (Bound variables)**.** Within an abstraction, we say an occurrence of a variable x in t is **bounded** by the **binder** x in $\lambda x.t$.
Formally the set of bound variables $BV[t]$ for a term t is defined as

$$\begin{aligned} BV[x] &= \varnothing \\ BV[\lambda x.t] &= BV[t] \cup \{x\} \\ BV[t_1\ t_2] &= BV[t_1] \cup BV[t_2] \end{aligned}$$

Definition 2.4 (Free variables)**.** An occurrence of a variable x that is not bounded by any binders is called **free**.
Formally

$$\begin{aligned} FV[x] &= \{x\} \\ FV[\lambda x.t] &= FV[t] \setminus \{x\} \\ FV[t_1\ t_2] &= FV[t_1] \cup FV[t_2] \end{aligned}$$

Definition 2.5 (Occurrence vs variable)**.** An occurrence of a variable x is the actual instance. A variable x is just the name given to it (a variable *have many* occurrences).
For example, in $x\lambda x.x$, x is a variable that is both free and bounded but each *occurrence* of x is either free or bounded, and never both.

Definition 2.6 (Substitution)**.** We denote **substitution** as $[x \mapsto t_2]t_1$ which is used to evaluate applications to abstractions i.e. $(\lambda x.t_1)\ t_2$.
To avoid substituting bound variables further down or *variable capture*, we have the following formal definition

$$\begin{aligned} &[x \mapsto s]x = s && \\ &[x \mapsto s]y = y && \\ &[x \mapsto s]t_1\ t_2 = [x \mapsto s]t_1\ [x \mapsto s]t_2 && \\ &[x \mapsto s]\lambda x.t = t && \\ &[x \mapsto s]\lambda y.t = \lambda y.[x \mapsto s]t && y \notin FV[s] \\ &[x \mapsto s]\lambda y.t = \lambda z.[x \mapsto s][y \mapsto z]t && y \in FV[s],\ z \text{ fresh} \end{aligned}$$

Definition 2.7 (Alpha equivalence)**.** We denote α-equivalence as the relation $=_\alpha$ defined as

$$\frac{y \notin FV[t]}{\lambda x.t =_\alpha \lambda y.[x \mapsto y]t}$$
$$\frac{t_1 =_\alpha t_1'}{t_1\ t_2 =_\alpha t_1'\ t_2}$$
$$\frac{t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1\ t_2'}$$
$$\frac{t =_\alpha t'}{\lambda x.t =_\alpha \lambda x.t'}$$

Definition 2.8 (Beta reduction)**.** We call the evaluation rule $(\lambda x.t_1)\ t_2 \to_\beta [x \mapsto t_2]\ t_1$ β-reduction.
Such an expression i.e. an application where the rator is an abstraction is called a **redex**.

Theorem 2.1 (Church-Rosser theorem (confluence))**.** If E_1, E_2, E_3 are terms such that $E_1 \to_\beta^* E_2$ and $E_1 \to_\beta^* E_3$, then there exists E_4 such that $E_2 \to_\beta^* E_4$ and $E_3 \to_\beta^* E_4$.

Definition 2.9 (Beta reduction strategies)**.** Since a term can contain many redexes, there exist different evaluation strategies that evaluates the redexes in a prescribed order:

Full beta reduction Reduce redexes in any arbitrary order. We *lose determinancy* with full β-reduction.

The inference rules are

$$(\lambda x.t_1)\ t_2 \to_\beta [x \mapsto t_2]\ t_1$$
$$\frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'}$$
$$\frac{t_1 \to_\beta t_1'}{t_1\ t_2 \to_\beta t_1'\ t_2}$$
$$\frac{t_2 \to_\beta t_2'}{t_1\ t_2 \to_\beta t_1\ t_2'}$$

Normal order (NOR) Reduce the **leftmost, outermost** redexes first i.e. do not evaluate rands until all applications have been fully evaluated. NOR is *deterministic*.

Inference rules are non-trivial to formalize.

Call-by-name Similar to NOR but leave rand un-evaluated. *Do not* reduce redexes inside body of abstractions.

The inference rules are

$$(\lambda x.t_1)\ t_2 \to_\beta [x \mapsto t_2]\ t_1$$
$$\frac{t_1 \to_\beta t_1'}{t_1\ t_2 \to_\beta t_1'\ t_2}$$

A lazy variation called **call-by-need** caches the values of evaluated subexpressions (used by Haskell).

Call-by-value Reduce the **leftmost, innermost** redexes first. *Do not* reduce redex inside body of abstractions.

It is **strict** or eagar: arguments are evaluated regardless of whether they are actually used.

The inference rules are

$$(\lambda x.t_1)\ v_2 \to_\beta [x \mapsto v_2]\ t_1$$

$$\frac{t_1 \to_\beta t_1'}{t_1\ t_2 \to_\beta t_1'\ t_2}$$

$$\frac{t_2 \to_\beta t_2'}{v_1\ t_2 \to_\beta v_1\ t_2'}$$

2.1 Programming in untyped lambda calculus

We can express many programming semantics and/or traditional primitives using lambda calculus.

Definition 2.10 (Identity function)**.**

$$\textbf{id} = \lambda x.x$$

Definition 2.11 (Booleans and if statements)**.** We define booleans with the idea of if statements in mind.

$$\textbf{true} = \lambda x.\lambda y.x$$
$$\textbf{false} = \lambda x.\lambda y.y$$
$$\textbf{if B then T else F} = B\ T\ F$$

where B is either **true** or **false**.
We can define **and**, **or**, and **not** easily with **if**.

Definition 2.12 (Lists)**.** Lists are defined with list functions in mind.

$$\textbf{cons} = \lambda f.\lambda r.\lambda m.m\ f\ r$$
$$\textbf{first} = \lambda p.p\ \textbf{true}$$
$$\textbf{rest} = \lambda p.p\ \textbf{false}$$
$$\textbf{empty} = \lambda m.\textbf{true}$$
$$\textbf{empty?} = \lambda p.p\ \lambda f.\lambda r.\textbf{false}$$

Definition 2.13 (Natural numbers)**.** We could treat a natural number n as a list of length n (difficult to define arithmetic functions) or use **Church numerals**.

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z \\
c_1 &= \lambda s.\lambda z.s\ z \\
c_2 &= \lambda s.\lambda z.s\ (s\ z) \\
&\dots \\
\textbf{succ} &= \lambda n.\lambda s.\lambda z.s\ (n\ s\ z) \\
\textbf{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\
\textbf{mult} &= \lambda m.\lambda n.m\ (\textbf{plus}\ n)\ c_0 \\
&= \lambda m.\lambda n.\lambda s.\lambda z.m\ (n\ s)\ z \\
\textbf{expt} &= \lambda m.\lambda n.\lambda s.\lambda z.n(\textbf{times}\ m)\ c_1 \\
&= \lambda m.\lambda n.n\ m
\end{aligned}
$$

where s and z can be thought of *successor* and *zero*, respectively.
pred (and thus **sub**) are a little less trivial

$$
\begin{aligned}
\textbf{zz} &= \text{pair}\ c_0\ c_0 \\
\textbf{ss} &= \lambda p.\text{pair}\ (\textbf{snd}\ p)\ (\textbf{plus}\ c_1\ (\textbf{snd}\ p)) \\
\textbf{pred} &= \lambda n.\lambda s.\lambda z.\textbf{fst}\ (n\ ss\ zz)
\end{aligned}
$$

Definition 2.14 (Recursion with Y or fix combinator)**.** In order to perform recursion in lambda calculus, we can use the **Y combinator**

$$\mathbf{Y} = \lambda f.(\lambda r.f\ (r\ r))\ (\lambda r.f\ (r\ r))$$

This works under NOR but not call-by-value. We need to use η-expansion (eta expansion)

$$\mathbf{Y} = \lambda f.(\lambda r.(\lambda y.f\ (r\ r)\ y))\ (\lambda r.(\lambda y.f\ (r\ r)\ y))$$

where something like the **fact** factorial function is a **fixed point** of the Y combinator i.e. **Y fact** = **fact (Y fact)**.

3 Typed lambda calculus

We can perform **static analysis** on the derivative tree of an expression (rather than *dynamic analysis*) to reason about the type of the value the expression evaluates to.
Note our **typechecking** will not rule out stuck values (it is *conservative*).

Definition 3.1 (Typing relation)**.** A term `t` has type `T`, which we denote `t:T`.

Lemma 3.1 (Inversion lemma)**.** If `t:R`, the types of the subterms can be derived from `R` and its corresponding derivative rules.
For example, if `true:R`, then `R = Bool`. Or if `if b then t else f:R`, then `b:Bool`, `t:R`, and `f:R`.

Theorem 3.1 (Unique type)**.** If `t` is typable then its type is unique, and there is only one derivation (i.e. unique proof tree).

Theorem 3.2 (Progress)**.** If `t:T`, then either `t` is a value or there is some `t'` such that $t \to t'$.

Theorem 3.3 (Preservation)**.** If `t:T` and $t \to t'$, then `t':T`.

Definition 3.2 (Type terms)**.** We generally denote uninterpreted base types (something that can be represented as any type) as capital alphabets e.g. `A`.
We denote types for abstractions with the type constructor $T \to T$. Note $\to$ is *right-associative* i.e. $A \to B \to C \equiv A \to (B \to C)$.

Definition 3.3 (Simply-typed typing rules)**.** For the simply typed lambda calculus, we have the following rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1 . t) : (T_1 \to T_2)} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : (T_1 \to T_2) \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \quad \text{(T-APP)}$$

where Γ is the **typing context** which keeps track of types of bound variables (i.e. from abstractions) as we recurse down into the derivation tree.

Definition 3.4 (Proof trees)**.** We show an expression has a given type by constructing a proof tree using the typing rules

$$\dfrac{\dfrac{\dfrac{\texttt{x:Bool} \in \texttt{x:Bool}}{\texttt{x:Bool} \vdash \texttt{x : Bool}}\ \text{T-VAR}}{\vdash \lambda\texttt{x:Bool.x : Bool}\to\texttt{Bool}}\ \text{T-ABS} \quad \dfrac{}{\vdash \texttt{true : Bool}}\ \text{T-TRUE}}{\vdash (\lambda\texttt{x:Bool.x})\ \texttt{true : Bool}}\ \text{T-APP}$$

Theorem 3.4 (Curry-Howard correspondence)**.** We can relate propositional logic to typed lambda calculus

Logic	Typed LC
propositions	types
$\to_i$	Abs
$\to_e$	App
S	Var
proof of α	term of type α (i.e. proof tree)
proof-checking	type-checking

Lemma 3.2 (Substitution lemma)**.** To prove the preservation theorem for simply-typed lambda calculus with abstractions and application, we claim:
If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

Theorem 3.5 (Strong normalization)**.** Every reduction sequence of every well-typed term of the simply-typed lambda calculus is of finite length.
Note: this implies we cannot do general recursion e.g. with the Y combinator.

3.1 Extensions to simply-typed

Definition 3.5 (Unit type)**.** We denote `():Unit` which is used for void functions e.g. `printf`.

Definition 3.6 (Sequencing)**.** We can sequence expressions `t1;t2` using $(\lambda x : \text{Unit}.t_2)\ t_1$ where x does not occur free in t_2.

Definition 3.7 (Ascription)**.** Optinal type annotations on expressions e.g. `t as T`.

Definition 3.8 (Let bindings)**.** Before types, `let` was syntatic sugar for an application of an abstraction. With types, we have the typing rule for `let x = E in B`:

$$\frac{\Gamma \vdash E : T_1 \quad \Gamma, x : T_1 \vdash B : T_2}{\Gamma \vdash \text{let x = E in B} : T_2} \qquad \text{(T-LET)}$$

`letrec` can be typed similarly.

Definition 3.9 (Product types)**. Pairs**, **tuples**, and **records** have **product types** e.g. the type of a pair whose components have type T_1 and T_2 is $T_1 \times T_2$.

Definition 3.10 (Sum types)**.** Variant types have **sum types**: this is common for algebraic data types.
Do we type `Int+Bool` as `Int`? Do we type `3` as `Int` or `Int+String`?
For binary variant types, we use `inl` and `inr` to inject types left and right, respectively.
So `inl A` has type `A + B` for any type `B`, and similarly `inr`.
To facilitate typechecking of these injection terms, we need to annotate them i.e. `inl A as A+B` (otherwise we can't figure out the other type). So we have the typing rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2}$$
$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 \text{ as } T_1 + T_2 : T_1 + T_2}$$

We also use the `case` construct to de-construct binary variant terms where we have the evaluation rules

$$\texttt{case inl v as T of inl x => s | inl y => t} \to [x \mapsto v]s$$
$$\texttt{case inr v as T of inl x => s | inl y => t} \to [y \mapsto v]t$$

For example we might have `case addr as Addr of inl x => x.firstlast | inr y => y.name` (this is similar to `match` statements in OCaml and `case` statements in Haskell).
Thus we have the typing rule for `case`

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma, x : T_1 \vdash s : T \quad \Gamma, y : T_2 \vdash t : T}{\Gamma \vdash \text{case } t_0 \text{ of inl } x => s | \text{ inr } y => t : T}$$

In OCaml, we use **labelled union** with unique constructor names. The compiler replaces all unique constructors with their corresponding type annotations.

Definition 3.11 (General recursion)**.** Recall we previous used the **Y combinator** for recursion. The raw lambda expression was untypeable. Instead, we can define a `fix` primitive where `fix f = f (fix f)`.
We have the evaluation rule

$$\text{fix } \lambda x : T.t \to [x \mapsto \text{fix } \lambda x : T.t]t$$

and the inference rule

$$\frac{\Gamma \vdash f : T \to T}{\Gamma \vdash \text{fix } f : T}$$

Thus `letrec` can be desugared as

$$\texttt{letrec x:T = t in s} \equiv \texttt{let x:T = fix (}\lambda\texttt{x.t) in s}$$

Definition 3.12 (Recursive types)**.** Suppose we want to define the recursive type `type NatList = Nil | Cons of Int * NatList`.
In a **nominative** type system, type equality is based on names and type checking works as expected. Nominative systems however limit expressivity and flexibility (e.g. dealing with polymorphic data types `List[T]` where functions may act on general `List` types).
We define recursive type similar to recursion in untyped lambda calculus: we pull out the type as `X` as define the **recursive operator** μ such that `NatList =` μ`X.<nil:Unit,cons:{Nat,X}>`.
Unfolding happens when we take

$$\mu X.T \to [X \mapsto \mu X.T]T$$

e.g. μ`X.<nil:Unit,cons:{Nat,X}>` unfolds to μ`X.<nil:Unit,cons:{Nat,`μ`X.<nil:Unit,cons:{Nat,X}>}>`.
In a **equi-recursive** type system, unfolded/folded versions are treated all the same (however one needs a way to manage infinite trees).
In an **iso-recursive** type system, unfolded/folded versions are isomorphic, so types will need to be folded/unfolded during evaluation. That is: *construction* of a value will need to be *folded* into its recursive type; *pattern matching* will need to *unfold* the value.

Definition 3.13 (References)**.** A reference of a value `t` can be constructed via `ref t`.
Derefencing a variable is denoted `!t` while assigning a value to a reference is denoted `t := t`.
Typing rules are straightforward

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_1}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1}{\Gamma \vdash !t_1 : T_1}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \texttt{Unit}}$$

For evaluation, we need the concept of a **store**, denoted μ, addressed by **locations** (i.e. memory and addresses). A reference is thus a location and a store is a partial function from location to value.
We append $|\mu$ to all our previous evaluation rules, e.g.

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1'\ t_2 \mid \mu'}$$

The store μ is thus updated with uses of references

$$\frac{t \mid \mu \to t' \mid \mu'}{!t \mid \mu \to !t' \mid \mu'}$$

$$\frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu}$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1 := t_2 \mid \mu \to t_1' := t_2 \mid \mu'}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1 := t_2 \mid \mu \to v_1 := t_2' \mid \mu'}$$

$$l := v \mid \mu \to \texttt{unit} \mid [l \mapsto v]\mu$$

$$\frac{t \mid \mu \to t' \mid \mu'}{\texttt{ref } t \mid \mu \to \texttt{ref } t' \mid \mu'}$$

$$\frac{l \notin dom(\mu)}{\texttt{ref } v_1 \mid \mu \to l \mid (\mu, l \mapsto v_1)}$$

4 Type inference

Definition 4.1 (Type substitution)**.** A **type substitution** σ maps uninterpreted type variables to types.
For example, $\sigma = \{A \mapsto Nat, B \mapsto (A \to Bool)\}$. In this case, $dom(\sigma) = \{A, B\}$ and $range(\sigma) = \{Nat, (A \to Bool)\}$.
Substitution on a type can be defined as

$$\sigma(X) = T \text{ if } (X \mapsto T) \in \sigma$$
$$\sigma(X) = X \text{ if } X \notin dom(\sigma)$$
$$\sigma(T_1 \to T_2) = \sigma(T_1) \to \sigma(T_2)$$

and base cases like $\sigma(Nat) = Nat$.
Substitution on type environments follows similarly.

Definition 4.2 (Substitution composition)**.** We can compose substitutions σ and γ denoted $\sigma \circ \gamma$

$$\sigma \circ \gamma = \{X \mapsto \sigma(T) \mid (X \mapsto T) \in \gamma\} \cup \{X \mapsto T \mid X \notin dom(\gamma), (X \mapsto T) \in \sigma\}$$

In inference rules, we denote the four-place relation $\Gamma \vdash t : T \mid \sigma$ as $\sigma\Gamma \vdash \sigma t : T$.

Theorem 4.1 (Typing with substitutions)**.** If $\Gamma \vdash t : T$, then $\sigma\Gamma \vdash \sigma t : \sigma T$.

Definition 4.3 (Unification)**.** When type inferencing, we sometimes need to solve type equations. This is called unification.
Unification returns a type substitution σ. It is given as

$$unify(T, T) = []$$
$$unify(X, T) = unify(T, X) = [X \mapsto T] \text{ if } X \notin FV[T]$$
$$unify(S_1 \to S_2, T_1 \to T_2) = \sigma_2 \circ \sigma_1$$
$$\text{where}$$
$$\sigma_1 = unify(S_1, T_1)$$
$$\sigma_2 = unify(\sigma_1 S_2, \sigma_1 T_2)$$

otherwise unification fails.

Definition 4.4 (Algorithm W)**.** We can now define type inference rules for our simply-typed lambda calculus

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \varnothing}$$

$$\frac{\Gamma, x : X \vdash t : T \mid \sigma}{\Gamma \vdash (\lambda x : X.t) : (\sigma X \to T) \mid \sigma}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid \sigma_1 \quad \sigma_1\Gamma \vdash t_2 : T_2 \mid \sigma_2 \quad \sigma_3 = unify(T_2 \to X, \sigma_2 T_1) \quad X \text{ fresh}}{\Gamma \vdash t_1\ t_2 : \sigma_3 X \mid \sigma_3 \circ \sigma_2 \circ \sigma_1}$$

Definition 4.5 (Principal unifier)**.** We say $\sigma \sqsubseteq \sigma'$ (σ is more general than σ') if $\sigma' = \gamma \circ \sigma$ for some γ.
A **principal unifier** is σ that unifies a constraint set C such that $\sigma \sqsubseteq \sigma'$ for any σ' unifying C.

Definition 4.6 (Let polymorphism)**.** In `let id = λx.x in if id true then id 1 else 0`, `id` is typed as $X \to X$ which gets specialized.
We thus need to type `id` as $\forall X.X \to X$ internally and specialize it on application. We can only quantify variables α with $\forall$ variables that are **not free** (or do not appear) in the type context: A variable is **free** in the type context if it is not quantified.
Typing rules are

$$\frac{\Gamma \vdash E : T_1 \quad \Gamma, x : \forall\alpha.T_1 \vdash B : T_2}{\Gamma \vdash \text{let x = E in B} : T_2}$$

$$\frac{x : \forall\alpha.T \in \Gamma}{\Gamma \vdash x : [\alpha_i \to X_i]T}$$

where each X_i is a fresh variable.
There is a loophole with references: thus the **value restriction** only permits syntatic values (constants, variables, and abstractions) to be quantified.

5 Type classes in Haskell

Definition 5.1 (Functors)**.** The Functor type class allow us to map a function over a "boxed" type e.g. `Maybe`

```
1  class Functor f where
2    fmap :: (a -> b) -> f a -> f b
```

Note this only works for unary functions.

Definition 5.2 (Applicatives)**.** The Applicative type class generalizes Functors to chain multiple applications of functions (note how it takes in a Functor/Applicative class and spits out a Functor/Applicative class unlike Functors).

```
1  class Functor f => Applicative f where
2    pure :: a -> f a
3    <*> :: f (a -> b) -> f a -> f b
```

For example the following are equivalent

```
1  pure (*) <*> (+1) <*> (*4) $ 2
2  fmap (*) (+1) <*> (*4) $ 2
3  (*) <$> (+1) <*> (*4) $ 2
```

where `fmap = <$>`. Note that the Applicative class being used here is the `((->) r)` instance.

5.1 Higher-order polymorphism (System F)

Definition 5.3 (Type abstractions and applications)**.** Similar to how a variable x can be abstracted out in $\lambda x.t$, we can also abstract out a type variable X in $\lambda X.t$ to support general polymorphic functions.
We thus get the typing rules

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X.T} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[S] : [X \mapsto S]\ T} \quad \text{(T-TAPP)}$$

We can thus construct proof trees for typechecking polymorphic expressions

$$\dfrac{\dfrac{\dfrac{}{X : X \in X, x : X}}{X, x : X \vdash x : X}\ \text{T-Var}}{X \vdash (\lambda x : X.x) : (X \to X)}\ \text{T-Abs}}{\vdash (\lambda X.\lambda x : X.x) : \forall X.X \to X}\ \text{T-TAbs}$$

$$\dfrac{\dfrac{\vdash 3 : \mathsf{Int}}{\vdash (\lambda X.\lambda x : X.x) : \forall X.(X \to X)}}{\vdash (\lambda X.\lambda x : X.x)[\mathsf{Int}] : \mathsf{Int} \to \mathsf{Int}}\ \text{T-TApp}}{\vdash (((\lambda X.\lambda x : X.x)[\mathsf{Int}])\ 3) : \mathsf{Int}}\ \text{T-App}$$