1 Preliminaries

1.1 Overview

BIG TODO: Write something about Pure Type Systems and the lambda cube and how this language is essentially an extension of the Calculus of Constructions, with stuff like top level definitions and lightweight Typed Source esque type inference.

Our language is a dependently typed lambda calculus, based on what is known in the literature as λP or $\lambda\Pi$. The main references which we base our language on are this paper on $\lambda\Pi$ and this implementation of a dependent type theory.

In System F, types may depend on other type variables, thus enabling parametrically polymorphic functions. In our language, we generalize this to allow types to depend on terms as well. In order to achieve this, we promote all types to expressions, so we no longer have separate syntactic categories for them.

Yes, this means that *everything* in our language is an expression, even those things which we call types! With this in mind, a type is then an expression, say T, which satisfies a very specific typing judgment, ie $\Gamma \vdash T \Leftarrow \text{Type}$. We'll revisit this again later.

The main aim of our project is to implement a simple lambda calculus with the dependent function type. These are also known as Pi types in the literature. These generalize the simple function type $A \to B$ by allowing the output type B to now *depend on the value* of the input expression. The type of functions in our language is now written as $\Pi_{x:A}B(x)$ where we write $B(x)$ for the return type to emphasize that x may appear free in B.

The set theoretic analogue to this dependent Pi type is the generalized cartesian product. Given a set A and a family of sets $\langle B_x \mid x \in A\rangle$ indexed by the elements $x \in A$, we can form the generalized cartesian product, denoted

$$\Pi_{x\in A}B_x = \left\{ f : A \to \bigcup_{x\in A} B_x \mid \forall x \in A,\, f(x) \in B_x \right\}$$

Functions that inhabit this set are known as *choice functions* in set theory. Such choice functions associate to each $x \in A$, an element $f(x)$ in B_x. As a fun fact, the Axiom of Choice asserts that this set is nonempty if every B_x is inhabited.

Another core feature of our language will be type inference. This will be implemented alongside typechecking, using a fancy *bidirectional typechecking* algorithm. Don't worry, we'll formalize all this later. For now, this just means that typechecking and type inference are mutually recursive processes.

Our language also doesn't have full blown recursion and is *strongly normalizing* in that every sequence of beta reductions will always terminate in a unique head normal form. This allows us to safely normalize all terms to full head normal form.

TODO: explain why our language is strongly normalizing and why this allows us to normalize everything all the way.

1.2 Metavariables

1. A, B, T, E range over expressions.
2. x, y, z range over variables.
3. ν, τ range over expressions that are in *full* head normal form.
4. s ranges over all sorts, ie Type and Kind.

We often add primes and subscripts, like E' or ν_2 while referring to these.

2 Syntax

Here we describe the concrete ascii syntax for our language. We have a separate syntax for expressions and statements, the latter of which function like top-level commands. These will be used by the user to interact with our language.

2.1 Syntax for expressions

1. **Sorts**

$$\frac{}{\text{Type}}$$

$$\frac{}{\text{Kind}}$$

2. **Variables**

$$\frac{}{x}$$

3. **Optional parentheses**

$$\frac{E}{(\,E\,)}$$

4. **Function abstraction**

$$\frac{x \qquad E}{\text{fun } x \text{ => } E}$$

$$\frac{x_0 \qquad x_1 \qquad \ldots \qquad x_n \qquad E}{\text{fun } x_0\ x_1\ \ldots\ x_n \text{ => } E}$$

Note that all functions in our language are unary and so fun $x_0\,x_1\ \ldots\ x_n$ => E is syntactic sugar for

$$\text{fun } x_0 \text{ => (fun } x_1 \text{ => } \ldots \text{(fun} \, x_n \text{ => } E))$$

Simimarly, one can also provide optional type annotations for input variables. This is to help the type checker infer the type of a function.

$$\frac{x \quad T \quad E}{\text{fun}\,(x : T)\ \texttt{=>}\ E}$$

$$\frac{x_i \quad T_i \quad E}{\text{fun}\,(x_0 : T_0)\ (x_1 : T_1)\ \dots\ (x_n : T_n)\ \texttt{=>}\ E}$$

We also treat $\text{fun}\,(x_0 : T_0)\ (x_1 : T_1)\ \dots\ (x_n : T_n)\ \texttt{=>}\ E$ as syntactic sugar for

$$\text{fun}\,(x_0 : T_0)\ \texttt{=>}\ (\text{fun}\,(x_1 : T_1)\ \texttt{=>}\ \dots(\text{fun}(x_n : T_n)\ \texttt{=>}\ E))$$

5. **Pi type**

$$\frac{x \quad A \quad B}{\text{Pi}\,(x : A),\ B}$$

$$\frac{x_i \quad A_i \quad B}{\text{Pi}\,(x_0 : A_0)\,(x_1 : A_1)\ \dots\ (x_n : A_n),\ B}$$

As with functions, this is syntactic sugar for

$$\text{Pi}\,(x_0 : A_0),\ (\text{Pi}\,(x_1 : A_1),\ \dots\ (\text{Pi}\,(x_n : A_n),\ B))$$

6. **Type ascriptions**

$$\frac{E \quad T}{(E : T)}$$

This functions similarly to other functional languages in that it's used mainly to provide a type annotation. For instance, it can be used to help the type checker if it's unable to infer the type of an expression.

7. **Local let bindings**

$$\frac{x \quad E \quad E'}{\text{let } x := E \text{ in } E'}$$

2.2 Syntax for statements

Statements are top level commands through which users interact with our language. Programs in our language will be *nonempty* sequences of these statements.

1. **Def**

$$\frac{x \quad E}{\text{def } x := E}$$

This creates a top level, global definition, binding the variable x to the expression given by E.

2. **Axiom**

$$\frac{x \quad T}{\text{axiom } x : T}$$

This defines the variable x to have a type of T with an unknown binding. The interpreter will treat x like an unknown, indeterminate constant.

3. **Check**

$$\frac{E}{\text{check } E}$$

This instructs the interpreter to compute the type of the expression E and output it.

4. **Eval**

$$\frac{E}{\text{eval } E}$$

This instructs the interpreter to fully normalize the expression E to full head normal form.

Note that we often write λx, E instead of fun x => E as in the concrete syntax. Similarly, we also write $\lambda(x : T)$, E in place of fun $(x : T)$ => x. Finally, we use $\Pi_{x:A} B(x)$ to abbreviate Pi $(x : A)$, B.

3 Capture avoiding substitutions

Here we define the notion of capture avoiding substitutions. For this, we first formalize the notion of free variables. Practically, we will implement substitutions and stuff using de bruijn indices so this formalization is for purposes of metatheory only.

3.1 Free variables

$$\begin{aligned}
\mathrm{FV}(x) &:= \{x\} \\
\mathrm{FV}(E_1\, E_2) &:= \mathrm{FV}(E_1) \cup \mathrm{FV}(E_2) \\
\mathrm{FV}(\lambda\, x,\, E(x)) &:= \mathrm{FV}(E) \setminus \{x\} \\
\mathrm{FV}(\lambda\, (x : T),\, E(x)) &:= \mathrm{FV}(T) \cup (\mathrm{FV}(E) \setminus \{x\}) \\
\mathrm{FV}(\text{let } x := E \text{ in } E') &:= \mathrm{FV}((\lambda x,\, E')\ E) \\
\mathrm{FV}(\Pi_{x:A} B(x)) &:= \mathrm{FV}(A) \cup (\mathrm{FV}(B) \setminus \{x\})
\end{aligned}$$

Note that in Pi expressions, the input type A is actually an expression itself and so may contain free variables. It's important to note that the $\Pi_{x:A}$, like a lambda is a binder binding x in B. However, this does not bind x in the input type, A. If x does appear in the input type A, then it is free there.

3.2 Substitution

Here we define $E[x \mapsto E'] := E''$ to mean substituting all free occurrences of x by E' in the expression E yields another expression E''.

$$\begin{aligned}
x[x \mapsto E] &:= E \\
y[x \mapsto E] &:= y \quad (\text{if } y \neq x) \\
(E_1\ \ E_2)[x \mapsto E] &:= (E_1[x \mapsto E]\ \ E_2[x \mapsto E]) \\
(\lambda x,\ E)[x \mapsto E'] &:= \lambda x,\ E \\
(\lambda y,\ E)[x \mapsto E'] &:= \lambda y,\ E[x \mapsto E'] \quad (\text{if } y \notin \mathrm{FV}(E')) \\
(\lambda y,\ E)[x \mapsto E'] &:= \lambda z, E[y \mapsto z][x \mapsto E'] \\
&\qquad\quad (\text{if } z \notin FV(E) \cup FV(E') \cup \{x\}) \\
(\text{let } y := E \text{ in } E'')[x \mapsto E'] &:= ((\lambda y,\ E)\ E'')[x \mapsto E']
\end{aligned}$$

We also define $(\lambda\,(x : T),\ E)[x \mapsto E']$ similar to the case withouit the optional type annotation above. The only difference here is we also need to substitute x for E' in the type annotation, T. Note that we do not treat x as being bound in T here.

$$\begin{aligned}
(\Pi_{x:A} B(x))[x \mapsto B'] &:= \Pi_{x:A[x \mapsto B']} B(x) \\
(\Pi_{y:A} B(y))[x \mapsto B'] &:= \Pi_{y:A[x \mapsto B']} B[x \mapsto B'] \quad (\text{if } y \notin \mathrm{FV}(B')) \\
(\Pi_{y:A} B(y))[x \mapsto B'] &:= \Pi_{z:A[x \mapsto B']} B[y \mapsto z][x \mapsto B'] \\
&\qquad (\text{if } z \notin FV(B) \cup FV(B') \cup \{x\})
\end{aligned}$$

For Pi expressions, A is an expression and thus when substituting x by B', we must also perform the substitution in the input type A. However, as x is not bound there, we need not worry about capturing it when substituting there.

4 Big step operational semantics

Here we choose to use a big-step semantics for our language as it fits nicer with the functional style that we're using to write our recursive interpreter with.

4.1 Expressions

4.1.1 Contexts and judgment forms

Here we take the context, Γ, to be a list of triples of the form

(variable name, type of variable, binding)

We will use $\emptyset$ to denote the empty context, and :: to refer to the list cons operation.

The binding can be an expression or a sentinel value `undefined`. Contexts have global scope and can be modified at the top level by `def` and `axiom` statements. This `undefined` value is used for the bindings created by `axiom` statements.

We first give an overview of the main judgment forms which we will define in a mutually recursive fashion.

1. $\mathrm{wf}(\Gamma)$
   This asserts that a context Γ is well formed.

2. $\Gamma \vdash E \Leftarrow T$ and $\Gamma \vdash E \Rightarrow T$
   These judgments will be used to formalize our bidirectional typechecking algorithm. They form the typing rules for our language. For now, it suffices to say that $\Gamma \vdash E \Leftarrow T$ formalizes the meaning that given an expression E and some type T, we may verify that E has type T under the context Γ.

   On the other hand, $\Gamma \vdash E \Rightarrow T$ formalizes the notion of *type inference.* It says that from a context Γ, we may infer the type of E to be T.

3. $\Gamma \vdash E \Downarrow \nu$
   This $\Gamma \vdash \cdot \Downarrow \cdot$ relation is used in our definition of a big-step operational semantics to normalize expressions to their full head normal form. It says that with respect to a context Γ containing global bindings, we may normalize E to an expression, ν that is in head normal form.

4.1.2 Normalizing expressions

Before defining our type system given by the 2 judgments, $\Gamma \vdash \cdot \Leftarrow \cdot$ and $\Gamma \vdash \cdot \Rightarrow \cdot$, we must first define $\Gamma \vdash \cdot \Downarrow \cdot$, the big step semantics for evaluating expressions. This is because types are now expression and we will actually need to perform normalization while typechecking.

1. **Type**

$$\frac{\mathrm{wf}(\Gamma)}{\Gamma \vdash \mathrm{Type} \Downarrow \mathrm{Type}}$$

2. **Variables**

$$\frac{\mathrm{wf}(\Gamma) \qquad (x,\, \tau,\, \nu) \in \Gamma}{\Gamma \vdash x \Downarrow \nu}$$

$$\frac{\mathrm{wf}(\Gamma) \qquad (x,\, \tau,\, \mathrm{undefined}) \in \Gamma}{\Gamma \vdash x \Downarrow x}$$

3. **Type ascriptions**

$$\frac{\Gamma \vdash E \Downarrow \nu}{\Gamma \vdash (E : T) \Downarrow \nu}$$

4. **Pi elimination, ie function application**

$$\frac{\Gamma \vdash E_1 \Downarrow \lambda x,\, \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2 \qquad \Gamma \vdash \nu_1[x \mapsto \nu_2] \Downarrow \nu}{\Gamma \vdash E_1\ E_2 \Downarrow \nu}$$

5. **Normalizing under lambdas**

$$\frac{\Gamma \vdash E \Downarrow \nu}{\Gamma \vdash \lambda x,\, E,\, \Downarrow \lambda x,\, \nu}$$

$$\frac{\Gamma \vdash \lambda x,\, E \Downarrow \nu}{\Gamma \vdash \lambda\, (x : T),\, E,\, \Downarrow \nu}$$

This second rule says that optional type ascriptions do not play a role in normalization, ie we just ignore them.

6. **Pi**

$$\frac{\Gamma \vdash A \Downarrow \tau \qquad \Gamma \vdash B(x) \Downarrow \tau'(x)}{\Gamma \vdash \Pi_{x:A} B(x) \Downarrow \Pi_{x:\tau} \tau'(x)}$$

7. **Local let binding**

$$\frac{\Gamma \vdash E \Downarrow \nu \qquad \Gamma \vdash E'[x \mapsto \nu] \Downarrow \nu'}{\Gamma \vdash \text{let } x := E \text{ in } E' \Downarrow \nu'}$$

4.1.3 Well formed context and typing judgments

Definition 4.1 (Well formed type)**.** An expression T is said to be a well-formed type with respect to the context Γ if it satisfies

$$\Gamma \vdash T \Leftarrow \text{Type}$$

Informally, we can think of Type as the "type of all types" and so all all well formed types are expressions which we can check to have the type Type.

Definition 4.2 (Well formed context)**.** Note that our definition below implies that expressions and types that we store in our context are fully normalized to head normal form.

1. **Base case**

$$\text{wf}(\emptyset)$$

2. **Inductive cases**

$$\frac{\text{wf}(\Gamma) \qquad \Gamma \vdash \tau \Leftarrow \text{Type}}{\text{wf}((x,\, \tau,\, \nu) :: \Gamma)}$$

$$\frac{\text{wf}(\Gamma) \qquad \Gamma \vdash \tau \Leftarrow \text{Type}}{\text{wf}((x,\, \tau,\, \text{undefined}) :: \Gamma)}$$

Definition 4.3 (Bidirectional typechecking)**.** Here we define the 2 *mutually recursive* relations

1. $\Gamma \vdash \cdot \Rightarrow \cdot$ which corresponds to *inference*
2. $\Gamma \vdash \cdot \Leftarrow \cdot$ which corresponds to *checking*

The idea is that there are some expressions for which it is easier to *infer*, ie compute the type directly, while for others, it is easier to have the user supply a type annotation and then *check* that it is correct.

As a rule of thumb, it is often easier to check the type for term introduction rules while for elimination rules, it is usually easier to infer the type.

1. **Type ascriptions**

$$\frac{\Gamma \vdash T \Rightarrow s \qquad \Gamma \vdash T \Downarrow \tau \qquad \Gamma \vdash E \Leftarrow \tau}{\Gamma \vdash (E : T) \Rightarrow \tau}$$

   Optional type ascriptions allow the interpreter to infer the type of an expression. This is useful for lambda abstractions in particular because it's kinda hard to infer the type of a function like $\lambda x,\ x$ without any further contextual information.

2. **Var**

$$\frac{\mathrm{wf}(\Gamma) \qquad (x,\ \tau,\ v) \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}$$

   This says that we may infer the type of a variable if the type information is already in our context.

3. **Type**

$$\frac{\mathrm{wf}(\Gamma)}{\Gamma \vdash \mathrm{Type} \Rightarrow \mathrm{Kind}}$$

4. **Check**

$$\frac{\Gamma \vdash E \Rightarrow \tau' \qquad \tau \equiv_{\alpha\beta} \tau'}{\Gamma \vdash E \Leftarrow \tau}$$

   Given a context Γ, if we can compute the type of an expression E to be τ, then if we are also given τ, we can verify that E indeed has type τ.

5. **Pi formation**

$$\frac{\Gamma \vdash A \Rightarrow s_1 \qquad \Gamma \vdash A \Downarrow \tau \qquad (x,\ \tau,\ \mathrm{undefined}) :: \Gamma \vdash B(x) \Rightarrow s_2}{\Gamma \vdash \Pi_{x:A} B(x) \Rightarrow s_2}$$

   Note that this is a rule schema with the metavariables $s_1,\ s_2 \in \{\mathrm{Type},\ \mathrm{Kind}\}$

6. **Pi introduction**

$$\frac{(x,\, \tau,\, \text{undefined}) :: \Gamma \vdash E \Leftarrow \tau'(x)}{\Gamma \vdash \lambda x,\, E \Leftarrow \Pi_{x:\tau}\tau'(x)}$$

Note that our language (currently at least) doesn't support optional type ascriptions for input parameters, and so we don't have an inference rule for inferring the type of a lambda abstraction.

TODO: Say something about alpha renaming.

$$\frac{\Gamma \vdash T \Rightarrow s \qquad \Gamma \vdash T \Downarrow \tau \qquad (x,\, \tau,\, \text{undefined}) :: \Gamma \vdash E \Rightarrow \tau'(x)}{\Gamma \vdash \lambda\,(x : T),\, E \Rightarrow \Pi_{x:\tau}\tau'(x)}$$

The second rule says that if the user type annotates the input argument of the function, then we can try to infer the type of the output and consequently, the type of the function as a whole.

7. **Function application, ie Pi elimination**

$$\frac{\Gamma \vdash E_1 \Rightarrow \Pi_{x:\tau}\tau'(x) \qquad \Gamma \vdash E_2 \Leftarrow \tau \qquad \Gamma \vdash \tau'[x \mapsto E_2] \Downarrow \tau''}{E_1\ E_2 \Rightarrow \tau''}$$

8. **Local let binding**

$$\frac{\Gamma \vdash E \Rightarrow \tau \qquad (x,\, \tau,\, \text{undefined}) :: \Gamma \vdash E' \Rightarrow \tau'(x) \qquad \Gamma \vdash \tau'[x \mapsto E] \Downarrow \tau''}{\Gamma \vdash \text{let } x := E \text{ in } E' \Rightarrow \tau''}$$

The nice thing about these rules is that we can translate it almost directly into a typechecking and inference algorithm! More precisely, we may translate $\Gamma \vdash E \Rightarrow T$ into a function called check(Γ, E, T) which checks if the expression E really has the type T given a context Γ.

Similarly, $\Gamma \vdash E \Leftarrow T$ gives us a the function infer(Γ, E) which outputs the inferred type of E given the context Γ.

In the literature, such rules are called *syntax directed*, as the algorithm closely follows the formalization of the corresponding judgments.

TODO: This next part on admissible rules could be expanded on and better elaborated.

It's worth noting that since we normalize everything to full head normal form, the following *conversion rule* is a theorem:

$$\frac{\Gamma \vdash E \Leftarrow \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash E \Leftarrow \tau'}\ \text{Conv}$$

where we use $\tau \equiv_\beta \tau'$ to mean that we may obtain one from the other through a sequence of beta reductions. In this case, we say that τ and τ' are beta equivalent to each other.

Also, since we will be using de bruijn indices internally, lambda terms will always be deemed equal up to alpha renaming, so we actually have

$$\frac{\Gamma \vdash E \Leftarrow \tau \qquad \tau \equiv_{\alpha\beta} \tau'}{\Gamma \vdash E \Leftarrow \tau'}\ \text{Conv}$$

Note the extra α in the subscript for $\equiv$ next to the β.

Another kind of equivalence is *eta equivalence*. To explain this, consider the terms $\lambda x,\, B\, x$ and B. Both of them have exactly the same behavior. This is one form of eta equivalence. In lambda calculus terms, the latter is obtained from the former by an eta reduction.

Now, we may view the lambda as the data constructor for the Pi type and function application as the eliminator. Then the expression $\lambda x,\, B\, x$ is obtained from B by first eliminating B to obtain Bx and then using the lambda constructor to obtain $\lambda x,\, B\, x$.

More generally, we say that 2 terms are eta equivalent to each other if one can be obtained from the other by eliminating the term and then reconstructing the original term using the constructor. For instance, if we had a binary coproduct, we could project out both components and then recombine them to obtain the original sum.

Unfortunately our type system is intensional rather than extensional in that the following rule does *not* hold:

$$\frac{\Gamma \vdash E \Leftarrow \tau \qquad \tau =_{\eta} \tau'}{\Gamma \vdash E \Leftarrow \tau'}$$

One way to fix this is to allow the type checker to eta expand terms, via eliminating and then applying the data constructor, after computing the beta normal form.

4.2 Programs

To specify the big step semantics for evaluating programs, we first define the notion of a *configuration*.

4.2.1 Configuration/state of program

A configuration has the form

$$(\Gamma,\, \langle S_0;\, \ldots;\, S_n\rangle,\, E)$$

Configurations are triples representing the instantaneous state during an execution of a program. Here, Γ denotes the current state of the global context and the sequence $\langle S_0;\, \ldots;\, S_n\rangle$ denotes the next statements to be executed. The expression E is used to indicate the output of the previously executed statement.

With this view, given a program, $\langle S_0;\, \ldots;\, S_n\rangle$, we also define:

1. **Initial configuration**
   $(\emptyset,\, \langle S_0;\, \ldots;\, S_n\rangle,\, \text{Type})$

2. **Final configurations**
   These are all configurations of the form $(\Gamma,\, \langle\rangle,\, E)$

In the next section, we define the big-step relation for programs $\cdot \Downarrow \cdot$ using this notion of a configuration.

4.2.2 Big step semantics for programs

We interleave type checking and evaluation, ie we type check and then evaluate each statement, one at a time, carrying the context along as we go. We do not statically type check the whole program first, then evaluate afterwards.

1. **Check**

$$\frac{\Gamma \vdash E \Rightarrow \tau}{(\Gamma,\, \langle \text{check } E\rangle,\, E') \Downarrow (\Gamma,\, \langle\rangle,\, \tau)}$$

2. **Eval**

$$\frac{\Gamma \vdash E \Rightarrow \tau \qquad \Gamma \vdash E \Downarrow \nu}{(\Gamma,\, \langle \text{eval } E\rangle,\, E') \Downarrow (\Gamma,\, \langle\rangle,\, \nu)}$$

3. **Axiom**

$$\frac{\Gamma \vdash T \Leftarrow \text{Type} \qquad \Gamma \vdash T \Downarrow \tau}{(\Gamma,\, \langle \text{axiom}\, x : T\rangle,\, E') \Downarrow ((x,\, \tau,\, \text{undefined}) :: \Gamma,\, \langle\rangle,\, \tau)}$$

4. **Def**

$$\frac{\Gamma \vdash E \Rightarrow \tau \qquad \Gamma \vdash E \Downarrow \nu}{(\Gamma,\, \langle \text{def}\, x := E\rangle,\, E') \Downarrow ((x,\, \tau,\, \nu) :: \Gamma,\, \langle\rangle,\, (\nu : \tau))}$$

5. **Sequences of statements**

$$\frac{(\Gamma,\, \langle S_0\rangle,\, E) \Downarrow (\Gamma',\, \langle\rangle,\, E') \qquad (\Gamma',\, \langle S_1;\, \ldots;\, S_n\rangle,\, E) \Downarrow (\Gamma'',\, \langle\rangle,\, E'')}{(\Gamma,\, \langle S_0;\, S_1;\, \ldots;\, S_n\rangle,\, E) \Downarrow (\Gamma'',\, \langle\rangle,\, E'')}$$

4.2.3 Output expression of evaluating a program

With these rules, given a user-entered program, say $\langle S_0; \ldots; S_n\rangle$, we define the output expression of a program to be the E such that

$$(\emptyset,\, \langle S_0;\, \ldots\; S_n\rangle,\, \text{Type}) \Downarrow (\Gamma,\, \langle\rangle,\, E)$$

In other words, the expression output to the user is the expression obtained by beginning with the initial configuration and then recursively evaluating it until we reach a final configuration.