1 Introduction

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 the calculus of constructions. 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.

2 Syntax

Here we describe the concrete ascii syntax for our language. We also describe the syntactic sugar users can enter. This includes unicode symbols.

Note that we have 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

We define the set of valid expressions of our language as the initial algebra, ie least prefixed point, of the endofunctor

$$F : X \mapsto \{E \,|\, \{E_0, E_1, \ldots, E_n\} \subseteq X \text{ and } \frac{E_0 \quad E_1 \quad \ldots \quad E_n}{E} \text{ is a rule instance}\}$$

on some complete powerset lattice. Such a construction can be justified by the famous fixed point theorems of Tarski and Kleene.

Before we list the full set of rules, we first introduce some metavariables.

2.1.1 Metavariables

1. A, B, T, E range over expressions.
2. x, y, z range over variables.

2.1.2 Syntax rules

1. **Sorts**

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

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

   Our language has 2 sorts, just like the original CoC. In the spirit of Curry Howard, our language makes no distinction between types and propositions and so users can enter `Prop` (short for "proposition") in place of `Type`.

   This syntactix sugar has the unfortunate consequence of conflating the syntactic roles of propositions and types. Types which we think of as data can be used interchangeably with types which we view as propositions.

   This may be counter-intuitive for users because in first (or higher) order logic, there is a clear distinction between terms and well-formed formulae, which belong to different syntactic categories.

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

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

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

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

$$\frac{x_i \qquad T_i \qquad E}{\text{fun}\,(x_0 : T_0)\;(x_1 : T_1)\;\ldots\;(x_n : T_n) \text{ => } E}$$

We also treat fun $(x_0 : T_0)\;(x_1 : T_1)\;\ldots\;(x_n : T_n)$ => E as syntactic sugar for

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

As syntactic sugar, we allow users to use `lambda` and λ in place of `fun`.

5. **Pi and Sigma type**

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

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

As with functions, this is syntactic sugar for

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

If there is only one pair of input type and type annotation following the `Pi`, the brackets may be ommitted so users can enter `Pi x : A, B` instead of `Pi (x : A), B`.

The syntax rules for Sigma, ie Sigma $(x : A), B$ and the above syntactic sugar work the same as with Pi above.

We also allow users to enter, in place of `Pi`, `forall` or using unicode, $\forall$ and Π. Similarly, users can enter `exists`, Σ and $\exists$ instead of `Sigma`.

In the event that the output type does not depend on the input type, users may enter `A -> B` and `A * B` in place of `Pi (x : A), B` and `Sigma (x : A), B`.

6. **Sigma constructor**

$$\frac{E_1 \quad E_2}{(E_1,\, E_2)}$$

7. **Sigma eliminators**

$$\frac{E}{\text{fst } E}$$

$$\frac{E}{\text{snd } E}$$

8. **Type ascriptions**

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

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

9. **Local let bindings**

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

10. **Sum type**

$$\frac{A \quad B}{A + B}$$

11. **Sum type constructor**

$$\frac{E}{\text{inl}\, E}$$

$$\frac{E}{\text{inr}\, E}$$

These are meant for introducing the left and right components of a disjoint sum, ie $\text{inl}\, E$ constructs an expression of type $A + B$ given an expression E of type A. Similarly, $\text{inr}\, E$ constructs an $A + B$ given E of type B.

12. **Sum type eliminator**
Given expressions E and variables x and y, users can perform case analysis on a sum type via the `match` construct below

```
match E with
| inl x -> E1
| inr y -> E2
end
```

The ordering of both clauses can be swapped so users can also enter

```
match E with
| inr y -> E2
| inl x -> E1
end
```

Note here that x is bound in the expression that is E_1, while y is bound in E_2.

This `match` construct is fashioned after the similarly named pattern matching construct in ML like languages. However, unlike those, we do not implement actual pattern matching since pattern matching for dependent types is not an easy problem. Our `match` construct serves the sole purpose of allowing one to perform case analysis on sum.

For convenience, we write $\text{match}(E, x \to E_1, y \to E_2)$ to denote this construct.

2.1.3 Syntactic sugar: wildcard variables

In the event where the user does not intend to use the variable being bound, like say in a lambda expression, the underscore, `_`, can be used in place of a variable name. For instance, one can enter `fun _ => Type` in place of `fun x => Type`. As in other languages, `_` is not a valid identifier that can be used anywhere else in an expression. It can only be used in place of a variable in a binder.

2.1.4 A word about notation

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

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.

We also allow users to enter `constant x : T` instead of `axiom x : T`, as syntactic sugar. Note that since our system conflates the notion of types and propositions, introducing a constant `x` of type `T` is akin to introducing an assumption (ie axiom).

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.

In the next section, we will properly define the notion of normalization using a big step semantics. For now it suffices to say that normalizing an expression is the process of fully evaluating it until no more simplifications can be performed.

3 Capture avoiding substitutions

Here we define the notion of capture avoiding substitutions. For this, we formalize the notion of free variables and substitution.

3.1 Free variables

We define the set of free variables of an expression E, ie $\mathrm{FV}(E)$ recursively.

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

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, it is free there.

$$\mathrm{FV}(\Sigma_{x:A} B(x) := \mathrm{FV}(\Pi_{x:A} B(x))$$

Sigma expressions follow the same rules as with Pi expressions, with x being bound in the output type B but not in the input type A.

$$\begin{aligned}
\mathrm{FV}(A + B) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\
\mathrm{FV}(\mathrm{match}(E, x \to E_1, y \to E_2)) &:= E \cup (E_1 \setminus \{x\}) \cup (E_2 \setminus \{y\}))
\end{aligned}$$

Note that in match expressions, x is bound in E_1 and y is bound in E_2.

For the expressions fst E, snd E, inl E and inr E, the set of free variables is precisely $\mathrm{FV}(E)$, since those keywords are treated like constants.

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''.

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

We also define $(\lambda\, (x : T),\, E)[x \mapsto E']$ similar to the case without 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.

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

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.

For Sigma expressions, we define $(\Sigma_{x:A}B(x))[x \mapsto B']$ in a similar fashion as with the case of Pi.

The substitution rules for `fst`, `snd`, `inl` and `inr` are trivial and thus ommitted.

Finally, we avoid specifying substitution formally for match expressions since it's tedious. The key thing to note is that in the expression $\mathrm{match}(E, x \to E_1, y \to E_2)$, x is bound in E_1 and y is bound in E_2.

4 Big step operational semantics

In this section, we define a big-step semantics for our language. We follow the approach of Plotkin, as described in his famous technical report on structural operational semantics.

4.1 Expressions

4.1.1 Head normal form, neutral expressions, beta equivalence

Head normal form and neutral expressions

In the simply typed lambda calculus in which there is only the simple function type and lambda abstraction, normalization is the process in which expressions are simplified through repeated applications of the beta reduction rule into a form in which no further beta reductions can occur anywhere in the expression, even underneath the outermost lambda. The resulting expression is said to be in *head normal form*. In this section, we want to define similar notions for our richer language.

First, note that we have more than just function types in our language. In particular, we have 3 main types, namely `Pi`, `Sigma` and `Sum`. Also, "types" themselves are expressions too.

Next note that we will be performing normalization and substitution with respect to a context, which contains definitions which the user declares globally, via `def` and `axiom` statements.

Since `axiom` statements introduce variables at the global level with a type but no binding, expressions can now contain free variables. Hence we need a definition of head normal form and normalization that takes care of expressions with free variables. For this, we must introduce the concept of a *neutral expression*.

Neutral expressions are those which cannot be reduced via one of the elimination rules (ie the ones for `Pi`, `Sigma` and `Sum` types) because the expression to be eliminated is a free variable rather than an expression of the appropriate type.

With this, we generalize the notion of head normal form to be one in which the expression cannot be further simplified by applying any of the 3 elimination rules corresponding to the aforementioned types.

Normalization is then the process in which we repeatedly simplify an expression using these elimination rules, with the aim of reducing it to a head normal form. In the rules to follow, we shall see that this head normal form of an expression, should it exist, is unique, just as in the simply typed lambda calculus.

Definition 4.1 (Neutral expressions and head normal form)**.** We now define, using mutual recursion, the subset of neutral and normalized expressions via the following judgments:

1. $\mathrm{hnf}(E)$ This asserts that E is an expression that is in head normal form.
2. $\mathrm{neutral}(E)$ This asserts that E is a neutral expression.

The rules are as follows:

1. $$\frac{\mathrm{neutral}(E)}{\mathrm{hnf}(E)}$$

2. $$\frac{\mathrm{hnf}(E)}{\mathrm{hnf}(\lambda x,\, E)}$$

3. $$\frac{\mathrm{hnf}(E_1) \qquad \mathrm{hnf}(E_2)}{\mathrm{hnf}(\Pi_{x:E_1},\, E_2)}$$

4. $$\frac{\mathrm{hnf}(E_1) \qquad \mathrm{hnf}(E_2)}{\mathrm{hnf}(\Sigma_{x:E_1},\, E_2)}$$

5. $$\frac{\mathrm{hnf}(E_1) \qquad \mathrm{hnf}(E_2)}{\mathrm{hnf}(E_1 + E_2)}$$

6. $$\frac{}{\mathrm{neutral}(x)}$$

7.
$$\frac{\text{neutral}(E_1) \qquad \text{hnf}(E_1)}{\text{neutral}(E_1\ E_2)}$$

8.
$$\frac{\text{neutral}(E) \qquad \text{hnf}(E_1) \qquad \text{hnf}(E_2)}{\text{neutral}(\text{match}(E,\, x \to E_1,\, y \to E_2))}$$

9.
$$\frac{\text{neutral}(E)}{\text{neutral}(\text{fst}\ E)}$$

10.
$$\frac{\text{neutral}(E)}{\text{neutral}(\text{snd}\ E)}$$

Note that `let` expressions are not considered to be in head normal form because we treat expressions of the form `let x := E in E'` as syntactic sugar for the application $(\lambda x, E')\, E$ during the process of normalization.

In fact, we will see later that the typing rules for `let` expressions are also derived using this syntactic trick.

Definition 4.2 (Beta equivalence)**.** Finally, we also define the notion of beta equivalence. We say that two expressions are beta equivalent to each other, written $E_1 \equiv_\beta E_2$ if they have the same head normal form.

If both expressions are also alpha equivalent, ie they're equal up to renaming of bound variables, we write $E_1 \equiv_{\alpha\beta} E_2$.

4.1.2 Metavariables for neutral and normalized expressions

We now introduce 2 more types of metavariables corresponding to the new definitions given in the previous section.

1. ν, τ range over normalized expressions, ie those in head normal form.
2. n ranges over all neutral terms.

4.1.3 Contexts

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

$$(\text{variable name, type of variable, binding})$$

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 special undefined value, which we denote by `und`. Contexts have global scope and the top level statements `def` and `axiom`, return new contexts with updated bindings. The special `und` value is used for the bindings created by `axiom` statements and when we want to add a binding for type checking purposes. In such scenarios, we don't particularly care about the actual binding. We're only interested in the type of the variable.

We should mention that whenever we write, say $(x, \tau, \nu) \in \Gamma$, we allow the metavariable ν to be `und` as well. Also if there are multiple occurrences of x in Γ, as is the case when there is variable shadowing, we refer to the first occurrence of x.

4.1.4 Overview of judgement forms

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

1. $\mathcal{WF}(\Gamma)$
   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.

   It should be noted here that this isn't real type inference using constraint solving and unification. It's a form of lightweight type inference that can infer simple stuff like the return type of a function application but not the type parameter in the polymorphic identity function $\lambda\,(T : \text{Type})(x : T),\, x$.

3. $\Gamma \vdash E \Downarrow \nu$
   This $\cdot \vdash \cdot \Downarrow \cdot$ 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.

4.1.5 Big step normalization

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

Recall that the aim of normalization is to reduce an expression to head normal form, which includes neutral expressions and free variables as a subset.

The rules are as follows:

1. **Type**

$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \text{Type} \Downarrow \text{Type}}$$

2. **Variables**

$$\frac{\mathcal{WF}(\Gamma) \quad (x,\,\tau,\,\nu) \in \Gamma}{\Gamma \vdash x \Downarrow \nu}$$

$$\frac{\mathcal{WF}(\Gamma) \quad (x,\,\tau,\,\text{und}) \in \Gamma}{\Gamma \vdash x \Downarrow x}$$

3. **Type ascriptions**

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

This says type ascriptions do not play a role in computation. They're just there to help the type checker figure things out and for users to communicate their intentions.

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

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

$$\frac{\Gamma \vdash E_1 \Downarrow n \quad \Gamma \vdash E_2 \Downarrow \nu}{\Gamma \vdash E_1\ E_2 \Downarrow n\ \nu}$$

The last rule handle the cases of neutral expressions.

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 type constructor**

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

7. **Local let binding**

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

8. **Normalizing under pair constructor**

$$\frac{\Gamma \vdash E_1 \Downarrow \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2}{\Gamma \vdash \langle E_1,\, E_2 \rangle \Downarrow \langle \nu_1,\, \nu_2 \rangle}$$

9. **Sigma type constructor**

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

10. **Sigma elimination**

$$\frac{\Gamma \vdash E \Downarrow (\nu_1,\, \nu_2)}{\Gamma \vdash \operatorname{fst} E \Downarrow \nu_1}$$

$$\frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \operatorname{fst} E \Downarrow \operatorname{fst} n}$$

$$\frac{\Gamma \vdash E \Downarrow (\nu_1,\, \nu_2)}{\Gamma \vdash \operatorname{snd} E \Downarrow \nu_2}$$

$$\frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \operatorname{snd} E \Downarrow \operatorname{snd} n}$$

11. **Sum type constructor**

$$\frac{E_1 \Downarrow \nu_1 \qquad E_2 \Downarrow \nu_2}{E_1 + E_2 \Downarrow \nu_1 + \nu_2}$$

12. **Normalizing under sum data constructors**

$$\frac{\Gamma \vdash E \Downarrow \nu}{\Gamma \vdash \text{inl } E \Downarrow \text{inl } \nu}$$

$$\frac{\Gamma \vdash E \Downarrow \operatorname{inl} \nu}{\Gamma \vdash \text{inr } E \Downarrow \nu}$$

13. **Normalizing sum eliminator**

$$\frac{\Gamma \vdash E \Downarrow \operatorname{inl} \nu \qquad \Gamma \vdash E_1[x \mapsto \nu] \Downarrow \nu_1}{\Gamma \vdash \operatorname{match}(E,\, x \to E_1,\, y \to E_2) \Downarrow \nu_1}$$

$$\frac{\Gamma \vdash E \Downarrow \operatorname{inr} \nu \qquad \Gamma \vdash E_2[y \mapsto \nu] \Downarrow \nu_2}{\Gamma \vdash \operatorname{match}(E,\, x \to E_1,\, y \to E_2) \Downarrow E_2[y \mapsto \nu_2]}$$

$$\frac{\Gamma \vdash E \Downarrow n \qquad \Gamma \vdash E_1 \Downarrow \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2}{\Gamma \vdash \operatorname{match}(E,\, x \to E_1,\, y \to E_2) \Downarrow \operatorname{match}(n,\, x \to \nu_1,\, y \to \nu_2)}$$

Note that the lack of a normalization rule for `Kind` is deliberate. The reason is that we will ensure that we only normalize expressions after we typecheck them, and we will see in the next section that `Kind` has no type. Thus we will never need to normalize it.

4.1.6 Well formed context and typing judgments

Definition 4.3 (Well formed type and type constructor)**.** An expression T is said to be a well-formed type with respect to the context Γ if it satisfies

$$\Gamma \vdash T \Leftarrow s$$

Informally, we can think of Type as the "type of all small types" and so all small well formed types are those expressions satisfying $\Gamma \vdash T \Leftarrow \text{Type}$, ie they can be checked to have type Type.

Those T satisfying $\Gamma \vdash T \Leftarrow \text{Kind}$ instead represent the type of type constructors. In other words, the type of a type constructor is a Kind, just like in Haskell.

Definition 4.4 (Well formed context)**.** Note that our definition below implies that expressions and types that we store in our context are normalized to head normal form.

1. **Base case**

$$\mathcal{WF}(\emptyset)$$

2. **Inductive cases**

$$\frac{\mathcal{WF}(\Gamma) \qquad \Gamma \vdash \tau \Rightarrow s}{\mathcal{WF}((x,\, \tau,\, \nu) :: \Gamma)}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad \Gamma \vdash \tau \Rightarrow s}{\mathcal{WF}((x,\, \tau,\, \text{und}) :: \Gamma)}$$

Definition 4.5 (Bidirectional typechecking)**.** Here we define the 2 *mutually recursive* relations

1. $\cdot \vdash \cdot \Rightarrow \cdot$ which corresponds to *inference*
2. $\cdot \vdash \cdot \Leftarrow \cdot$ 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 introduction rules while for elimination rules, it is usually easier to infer the type.

1. **Var**

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

This says that we may infer the type of a variable if the type information is already in our context. Note that in this rule, we also allow ν to be `und`.

2. **Type**

$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \text{Type} \Rightarrow \text{Kind}}$$

3. **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.

$$\frac{\Gamma \vdash E \Leftarrow \text{Kind}}{\Gamma \vdash (E : \text{Kind}) \Rightarrow \text{Kind}}$$

Note that the first rule doesn't allow users to assert that $(E : \text{Kind})$ since there is no s with $\Gamma \vdash \text{Kind} \Rightarrow s$. This second rule allows users to assert that `Type` and type constructors have type `Kind`.

4. **Check**

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

This says that to check if E has type τ with respect to a context Γ, we may first infer the type of E. Suppose it is τ'. Then if we also find that τ and τ' are α and β equivalent to each other, we may conclude that E indeed has type τ.

Remark. Note that this rule, together with the one on type ascriptions, is the reason why we need to perform computations on types, ie normalize them, while typechecking expressions.

To see this, suppose the user requests that we typecheck an expression of the form $(x : T)$ where T is some complicated expression entered by the user. Intuitively, we want to grab the type of x from the context and then check that it is equal to the annotated type of T.

Unlike the simply typed and polymorphic lambda calculi, our type system is much richer and so checking types for equality is not so simple. How do we check if 2 "types" are equal when they are really just expressions? One idea is to fully simplify them to their respective unique head normal forms, via normalization, and then check if those are structurally equal.

However, it must be noted that throughout these rules, we only normalize expressions after we typecheck them. We always check that a type is well formed, ie that $\Gamma \vdash T \Rightarrow s$, before we try to normalize T.

5. **Pi formation**

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

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

6. **Pi introduction**

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

Note that in the event that the input variable of the lambda abstraction and Pi are different, then we must perform an α renaming so that the variable being bound in $(\lambda x,\, E)$ and $\Pi_{y:\tau} \tau'(y)$ are the same.

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

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**

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

9. **Sigma formation**

$$\frac{\Gamma \vdash A \Rightarrow s_1 \qquad \Gamma \vdash A \Downarrow \tau \qquad (x,\, \tau,\, \text{und}) :: \Gamma \vdash B(x) \Rightarrow s_2}{\Gamma \vdash \Sigma_{x:A} B(x) \Rightarrow s_2}$$

As with the rule for Pi formation, $s_1,\, s_2 \in \{\text{Type, Kind}\}$.

10. **Sigma introduction**

$$\frac{\Gamma \vdash E_1 \Leftarrow \tau_1 \qquad \Gamma \vdash \tau_2[x \mapsto E_1] \Downarrow \tau_2' \qquad \Gamma \vdash E_2 \Leftarrow \tau_2'}{\Gamma \vdash \langle E_1,\, E_2 \rangle \Leftarrow \Sigma_{x:\tau_1} \tau_2(x)}$$

11. **Sigma elimination**

$$\frac{\Gamma \vdash E \Rightarrow \Sigma_{x:\tau_1} \tau_2(x)}{\Gamma \vdash \mathrm{fst}\, E \Rightarrow \tau_1}$$

$$\frac{\Gamma \vdash E \Rightarrow \Sigma_{x:\tau_1} \tau_2(x) \qquad \Gamma \vdash \tau_2[x \mapsto \mathrm{fst}\, E] \Downarrow \tau_2'}{\Gamma \vdash \mathrm{snd}\, E \Rightarrow \tau_2'}$$

12. **Sum formation**

$$\frac{\Gamma \vdash A \Leftarrow \mathrm{Type} \qquad \Gamma \vdash B \Leftarrow \mathrm{Type}}{\Gamma \vdash A + B \Rightarrow \mathrm{Type}}$$

Note that this rule says that users can only make construct a sum, aka coproduct, out of types that live in the universe `Type`, not `Kind`.

Recalling the Curry-Howard correspondence which identifies types and propositions, we see sum types as a way to model disjunction. Hence we do not see a need to allow users to construct sums out of large types like type constructors found in `Kind`.

Also we are unsure if the logical consistency of the system can be preserved if we allow for $A + B$ to be formed when one is of type `Type` while the other has type `Kind`.

13. **Sum introduction**

$$\frac{\Gamma \vdash E \Leftarrow \tau_1}{\Gamma \vdash \mathrm{inl}\, E \Leftarrow \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash E \Leftarrow \tau_2}{\Gamma \vdash \mathrm{inr}\, E \Leftarrow \tau_1 + \tau_2}$$

14. **Sum elimination**

$$\frac{\Gamma \vdash E \Rightarrow \tau_1 + \tau_2 \qquad (x,\, \tau_1,\, \mathrm{und}) :: \Gamma \vdash E_1 \Rightarrow \tau_1' \qquad (y,\, \tau_2,\, \mathrm{und}) :: \Gamma \vdash E_2 \Rightarrow \tau_2' \qquad \tau_1' \equiv_{\alpha\beta} \tau_2'}{\Gamma \vdash \mathrm{match}(E,\, x \to E_1,\, y \to E_2) \Rightarrow \tau_1'}$$

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 $\mathrm{check}(\Gamma,\, E,\, T)$ 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 $\mathrm{infer}(\Gamma,\, E)$ 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.

It's worth noting that η equivalence is not respected by our type system. By that we mean that the following rule does not hold:

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

In other words, 2 types that are η equivalent to one another will *not* be judged as equal types.

One way to fix this is to allow the type checker to eta expand terms, via eliminating and then applying the data constructor while computing the beta normal form. However, for simplicity, we chose to follow the original formulation of the Calculus of Constructions and ignore this.

4.2 Big step semantics for programs

Programs in our language are nonempty sequences of statements, with each statement being built from some expression. Following Plotkin's approach to structural operational semantics, we formalize the execution of program by defining a *transition system*.

Such transition systems behave like finite automata, although they aren't restricted to having finitely many states or transitions. In the next section, we define the notion of a configuration, which plays the same role as states in automata. Thereafter, we formalize the big-step transition relation for executing programs.

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

   Initially, our global context is empty. Also, `Type` is merely a dummy value and could have very well be replaced by `Kind`.

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

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

4.2.2 Big step semantics for programs

We define the big step transition relation, $\cdot \Downarrow \cdot$, via a new judgment form in our calculus.

It should be noted that in the rules below, 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**

$$\frac{\Gamma \vdash T \Rightarrow s \qquad \Gamma \vdash T \Downarrow \tau}{(\Gamma,\, \langle \text{axiom}\, x : T \rangle,\, E) \Downarrow ((x,\, \tau,\, \text{und}) :: \Gamma,\, \langle \rangle,\, x)}$$

4. **Def**

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

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 Putting the transition system together

Letting S denote the (infinite) set of all configurations, and defining

$$F := \{(\Gamma,\, \langle \rangle,\, E) \in S \mid E \text{ expression}\}$$

we obtain a transition system, given by the tuple

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

Notice how our formalization mimics the definition of an automaton.

4.2.4 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 until we reach a final configuration.

5 Metatheoretic discussion

The original Calculus of Constructions is known to be strongly normalizing in that the normalization process, when applied to any well typed term, always terminates. Furthermore, it has decidable typechecking and is logically consistent when its type system is viewed as a logical calculus, with the Pi type corresponding to universal quantification.

TODO

Say something about why we think our bidirectional typechecking works. https://arxiv.org/pdf/2102.06513.pdf

Also say something about how "impredicative" Sigma types breaks consistency due to Girard's paradox and that the only (?) way to fix that is to use a predicative hierarchy of type universe with either cumulativitiy or universe polymorphism.

Mention that this issue may not be a big one because the proof term in Girard's is astronomical and Idk if our system can even handle anything that big since it isn't very efficient (cos we substitute naively rather than use normalization by evaluation).

If users are really afraid, just avoid using higher order existential quantification and stick to first order. Alternatively, can encode higher order existential quantification in terms of universal in CoC (see Type Theory and Formal Proof book).

https://era.ed.ac.uk/bitstream/handle/1842/12487/Luo1990.Pdf

6 Technical implementation