1 Introduction

Our language is a playground for the Curry-Howard correspondence. At its core is the impredicative type theory known as the Calculus of Constructions (CoC). Syntactically, it feels much like the Lean theorem prover, which like Coq, is based on a predicative variant of CoC with inductive types.

Following CoC, we stay true to the spirit of Curry-Howard and identify propositions as data types. Through this, all the logical connectives and quantifiers of intuitionistic logic are given a computational interpretation. The end result is a programming language which can be viewed as a theorem prover of sorts.

Unfortunately, we are unsure if our language is logically consistent like the original CoC since we do implement many more types to model the various logical connectives more naturally. Still, we hope that it inspires others to discover the joy of Curry-Howard.

In this document, we provide a big-step semantics for our language, upon which our interpreter was based. Our approach to semantics is heavily inspired by Plotkin's seminal work on structural operational semantics.

Though our language specification is formal and mathematically intensive, we believe in the importance of rigour to ensure precision and clarity. This is especially important for our language since dependent type theory is rather complex.

A benefit of this approach is that not only did our specification serve as a guide for our implementation, many of the rules we formalized here translated almost directly into the code which we implemented.

Despite our rigour, it is our aim to provide sufficient exposition and justification for our design decisions. We hope that the reader will come to see that the ideas underlying our formalization are intuitive.

In the next section, we offer some background on dependent type theory in relation to the Curry-Howard correspondence. Thereafter, we discuss where our language stands in relation to CoC.

1.1 Dependent type theory and Curry-Howard

1.1.1 Pi type and universal quantification

Abstractly, CoC is a pure type system that sits atop of Barendregt's lambda cube. More concretely, it generalizes the simply typed and polymorphic lambda calculi by replacing the simple function type with the dependent Pi type. 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 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

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

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.

More generally, dependently typed languages allow types to depend on expressions as well. Now you might ask, of what use are dependent types?

Remember that the Curry-Howard correspondence says that simple type theory can be used to encode constructive propositional logic, with sum types (ie coproducts) corresponding to disjunction, products to conjunction, unit to $\top$ and void to $\bot$.

But what about the quantifiers $\forall$ and $\exists$? This is where dependent type theory comes in.

To prove the universally quantified statement, $\forall x \in A,\, P(x)$ by hand, we would assume that x is an arbitrary element of A and then proceed to prove that $P(x)$ holds. Well, according to the BHK interpretation of constructive logic, constructing a proof as such is akin to defining a function which takes as input an element x of type A and returns a proof of $P(x)$. What is the type of such a function? It is precisely the Pi type, $\Pi_{x:A} P(x)$!

In the event that the output type of a Pi type doesn't depend on the value of the input, we recover the simple function type, which we write as $A \to B$ or $\Pi_{x:A} B$. So we see that the Pi type allows us to model both logical implication as well as universal quantification.

1.1.2 Sigma and existential types

As for the existential quantifier, the BHK interpretation says that a proof of $\exists x \in A,\, P(x)$ corresponds to a pair, $(x,\, p)$ where x is an element of type A witnessing the existential and pf is a proof that for this x, $P(x)$ holds.

The Sigma type, denoted $\Sigma_{x:A} B(x)$, generalizes the simple procuct type for pairs in the simply typed lambda calculus. The type of the second component is now allowed to depend on the value of the first.

Now there are 2 ways in which the elimination rules can be formulated, each giving a different flavour of Sigma.

If we define the first and second projections, ie `fst` and `snd`, that operate on a pair, say $(x,\, p) : \Sigma_{x:A} B(x)$, we get a generalization of the ordinary product type. We call this a "strong" Sigma type, since we are able to project out the explicit witness x and p, the proof that it satisfies the property that is B.

Set theoretically, these Sigma types can be used to model subsets. Consider the notion of a subset, $\{x \in A \,|\, P(x)\}$, from set theory. By interpreting sets as types, we can model elements of this subtype of A as pairs containing an element of type A and a second element certifying that the first element does belong to this subtype. What would such a certificate be? Well, a proof that it satisfies the predicate P! Such a pair is of the Sigma type $\Sigma_{x:A} P(x)$!

Note that if the second component of a pair doesn't depend on the first, we recover the usual pair (ie product) type, written $A \times B$ or $\Sigma_{x:A} B$. With this, we see that the Sigma type can encode 3 different notions, namely conjunction, existential quantification, and the subtype/subset relation!

Now to model the existential quantifier from logic as we are familiar with, we need a weaker elimination rule. Recall that when we eliminate an existentially quantified formula, say $\exists x, \varphi(x)$, the witness we obtain is completely arbitrary save for the fact that it satisfies φ.

To model this faithfully, we want a way for users to "unpack" pairs representing proofs of existentials in an opaque manner. This cannot be achieved via the elimination rule from the strong Sigma type which we saw above, because that would give users the explicit witness stored in the pair!

Thus we need dependent pairs, like Sigma types, but with a new elimination rule to enforce this layer of abstraction that prevents users from obtaining the original witness. What new kind of programming language construct can we use to model such pairs?

The answer lies in a feature that modern functional languages like Ocaml offer – existential types.

To see this, first recall that in Ocaml, we can define module signatures like

```
module type TY = sig
  type t
  val lst : t list
end
```

Such a signature can be identified with the *existential type* $\exists t, \{\text{lst} : t\,\text{list}\}$. The existential quantification here enforces an abstraction barrier, forcing users to manipulate such values opaquely, without knowing anything about `t`.

We can augment the polymorphic lambda calculus with these existential types with introduction and elimination rules. Note that in the rules below, X may occur free in T' here since T' is existentially quantified over the type variable X.

1. **Introduction**

$$\frac{\Gamma \vdash E : T'[X \mapsto T]}{\Gamma \vdash \{T,\, E\} \text{ as } T' : \exists X,\, T'} \text{ (T-Pack)}$$

   Notice how this looks a lot like the rule of existential introduction, with T being a witness and $T'[X \mapsto T]$ being the proposition that this T satisfies the property T'.

2. **Elimination**

$$\frac{\Gamma \vdash E : \exists X,\, T \qquad \Gamma,\, X,\, x : T \vdash E' : T'}{\Gamma \vdash \text{let } \{X,\, x\} = E \text{ in } E' : T'} \text{ (T-Unpack)}$$

Focusing on the types alone, we obtain

$$\frac{\Gamma \vdash \exists X,\, T_{12} \qquad \Gamma,\, T_{12} \vdash T_2}{\Gamma \vdash T_2}$$

which resembles the usual rule of existential elimination from logic:

$$\frac{\Gamma \vdash \exists x,\, \varphi(x) \qquad \Gamma,\, \varphi(x) \vdash \psi}{\Gamma \vdash \psi}$$

provided x does not occur free in Γ and ψ.

This rule says that we may derive the desired conclusion, ψ, under the assumption that x is completely arbitrary save for the fact that it satisfies φ. Note that we must also make sure to discharge the variable so that the conclusion, ψ, does not contain x free.

Modeling existential types as pairs denoted by $\{E_1,\, E_2\}$, we obtain the following introduction and elimination rules:

1. **Introduction**

$$\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : B[x \mapsto E_1]}{\Gamma \vdash \{E_1,\, E_2\} : \exists x : A,\, B(x)}$$

2. **Elimination**

$$\frac{\Gamma \vdash E : (\exists x : A,\, B(x)) \qquad \Gamma,\, x : A,\, y : B(x) \vdash E' : T}{\Gamma \vdash \text{let } \{x,\, y\} := E \text{ in } E' : T}$$

Thus we see that the Curry-Howard interpretation of the usual existential quantifier is a dependent variation of the existential type. Its cousin, the Sigma type, can then be seen as a very strong, constructive version of existential quantification in which all the proofs themselves carry explicit witnesses which can be projected out.

To distinguish this new variant of the Sigma from the previous, stronger one, we call these *existentials* and we write $\exists x : A,\, B(x)$ instead of $\Sigma_{x:A} B(x)$.

1.1.3 Summary of Curry-Howard correspondence

In essence, adding dependent types to the lambda calculus allows types to include expressions like variables. This give us the extra tools that we need to go beyond propositional logic and formalize constructive predicate logic!

The table below summarizes the correspondence between types and their Curry-Howard interpretation.

Type	Formal notation	Alt notation	Curry-Howard interpretation
Pi	$\Pi_{x:A} B(x)$		$\forall x \in A,\, B(x)$
	$\Pi_{x:A} B$	$A \to B$	$A \to B$
Sigma	$\Sigma_{x:A} B(x)$		$\{x \in A \mid B(x)\}$
	$\Sigma_{x:A} B$	$A \times B$	$A \wedge B$
Existential	$\exists x : A,\, B(x)$		$\exists x \in A,\, B(x)$
Sum	$A + B$		$A \vee B$
Unit	$\top$		$\top$ (ie logical truth)
Void	$\bot$		$\bot$ (ie logical falsity)

Curry-Howard also tells us that we can model the introduction and elimination rules for each logical connective and quantifier by the introduction and elimination rules of the corresponding type. This means that proving, say a universally quantified statement, corresponds to defining a function of the appropriate `Pi` type. Universal instantiation and modus ponens then correspond to function application.

These are the types which we implement in our language. In the next section, we discuss how our language resembles and differs from CoC.

1.2 Our language in comparison to CoC

Since our emphasis here is on simplicity, we stay true to the original formulation of CoC, by offering only 2 sorts, `Type` and `Kind`, as opposed to a countable hierarchy of type universes. We also do not treat inductive types.

As with CoC, `Type` in our language is the sort of propositions as well as datatypes. Informally, we can think of type as the type of all types. In other words, an element T of type `Type` can be interpreted as either being a proposition whose elements are proofs or a datatype like a set. Though this is in line with the Curry-Howard interpretation, this has the unfortunate consequence of conflating the syntactic roles of propositions and data types.

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.

As for the sort `Kind`, it's not all too useful for users. It's just here to prevent Girard's paradox, a type theoretic formulation of the set theoretic Burali-Forti argument. Through the construction of Girard's paradox, one can (in theory) provide a proof of false, ie an expression that inhabits the empty type. This compromises the logical consistency of the calculus.

It should also be noted that we focus on theorem proving via Curry-Howard over general programming. This means that our language does not have basic programming constructs like general recursion, conditional statements or even basic types like numbers and booleans.

Where our language differs from CoC is in the built-in types. In its original formulation as a pure type system, CoC only includes one built-in type, namely the Pi type. It does not have Sigmas or sums (ie coproducts). While there are ways to encode $\exists$, $\wedge$ and $\vee$ into CoC [insert ref to Type Theory and Formal Proof book], such encodings can be awkward to work with. Hence, we decided to also implement Sigmas, sum types and existentials as built-ins so that these can be modelled more naturally.

1.3 Proofs in our language

Before we dive into a formal description of the syntax and semantics of our language, let's first explore how we can write proofs in Curry-Howard style. To see how our language can be used, let's see some sample proofs. To prove a theorem by way of Curry-Howard, we first write down the proposition we want to prove as a type. Then we provide an expression of that corresponding type.

Below is a sample program which establishes the commutativity of disjunction.

```
theorem or_comm :
forall (P : Prop) (Q : Prop), (P \/ Q) -> Q \/ P :=
  fun P Q (p_or_q : P \/ Q) =>
    match p_or_q with
    | inl p => (inr p : Q \/ P)
    | inr q => (inl q : Q \/ P)
    end
```

Technically, the theorem named `or_comm` is a function whose body is given after the `:=` symbol. Its type is given on the second line, which amounts to the proposition that for any propositions P and Q, $P \vee Q$ implies $Q \vee P$.

Note that `forall` and `->` are both syntactic sugar for the `Pi` type, since the dependent function type allows us to encode both the universal quantifier and implication.

Now, a proof of this proposition is then an expression of the corresponding type. It is a function which takes in 2 arbitrary propositions, ie types, say P and Q, and then a proof of $P \vee Q$. Thus we have on the third line, a function whose input parameters are named `P`, `Q` and `p_or_q`. We type annotate `p_or_q` with `P \/ Q` to indicate, for readability purposes, that it is supposed to be a proof of $P \vee Q$.

As for the output of the function, it should be a proof of $Q \vee P$. The most natural thing to do is to consider cases on $P \vee Q$ since that's the only useful assumption we have. By Curry-Howard, the rule of disjunction elimination, which lets us perform case analysis on a disjunct, is captured by the `match` construct.

Hence in the 4th line, we match on `p_or_q`. We have 2 cases to consider, namely the case when P is true and when Q is true. The branch given by `|inl p => ...` indicates that we want to consider the case when the left disjunct, P is true. A proof of P, which we name `p` is then bound in the expression after the arrow `=>`. In this case, we return `(inr p : Q \/ P)`. `inr` is the data constructor which introduces the right disjunct for the sum type which models disjunction. The type annotation is used to indicate to the interpreter that we intend to return a proof of $Q \vee P$.

For the other case, ie the branch with `|inr q => ...`, we have a proof of Q, named `q`, which is then bound in the expression after the `=>`. In this case, we return `(inl q : Q \/ P)`.

Noting that lines beginning with `--` are treated as comments and using syntactic sugar borrowed from the Lean theorem prover, we may rewrite this as

```
theorem or_comm :
forall (P : Prop) (Q : Prop), (P \/ Q) -> Q \/ P :=
-- Assume that P and Q are propositions and that p_or_q
-- is a proof that P \/ Q.
  assume P Q (p_or_q : P \/ Q),
    -- We proceed by considering cases on P \/ Q.
    match p_or_q with
    -- In the event where P is true, ie p is a proof of P,
    -- we may immediate conclude Q \/ P is true since
    -- (inr p) is a proof of it.
    | inl p => show Q \/ P, from inr p
    -- In the event where Q is true, ie q is a proof of Q,
    -- we use (inl q) to show Q \/ P.
    | inr q => show Q \/ P, from inl q
    end
```

which may read more naturally to those who are more mathematically inclined.

2 Syntax

In order to implement our rich type system, we promote types to the level of expressions in our language, 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, namely $\Gamma \vdash T \Leftarrow s$, where $s \in \{\text{Type}, \text{Kind}\}$. We'll revisit this again later.

For now, we describe the concrete ascii syntax for our language, alongside the syntactic sugar we provide. This includes unicode versions of logical symbols that we write on paper.

At the top level, programs will be nonempty sequences of statements. Statements function like top-level commands, as found in other theorem provers like Coq and Lean. These offer a way for users to interact with our system. Through statements, users can introduce global assumptions and definitions.

2.1 Syntax for expressions

The syntax for our language and syntactic sugar are heavily inspried by the Lean theorem prover.

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**

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

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

Since our language makes no distinction between types and propositions, we allow users to enter `Prop` (short for "proposition") in place of `Type`.

2. **Variables**

$$\frac{}{x}$$

3. **Optional parentheses**

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

4. **Function abstraction**

$$\frac{x \qquad E}{\text{fun } x \Rightarrow E}$$

Note that all functions in our language are unary. However, we allow users to enter `fun x_0 x_1 ... x_n => E`, which is desugared into

```
fun x_0 => (fun x_1 => ... (fun x_n => E))
```

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

$$\frac{x \qquad T \qquad E}{\text{fun}\,(x : T)\;\Rightarrow E}$$

Users can enter a mixture of typed and untyped input parameters, like for instance `fun x y (h : T) => E` which is desugared into

```
fun x => (fun y => (fun (h : T) => E))
```

5. **Pi and Sigma type**

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

As with functions, we provide

```
Pi (x0 : A0) (x1 : A1) ... (xn : An), B
```

as syntactic sugar for

```
Pi (x0 : A0), (Pi (x1 : A1), ... (Pi (xn : An), 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 Σ instead of `Sigma`.

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

6. **Sigma constructor**

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

This constructs a pair of some Sigma type.

7. **Sigma eliminators**

$$\frac{E}{\mathrm{fst}\ E}$$

$$\frac{E}{\mathrm{snd}\ E}$$

`fst` is used to obtain the first component of a pair and `snd` is used to obtain the second component.

8. **Existential**

$$\frac{E}{\mathrm{exists}\ E}$$

We allow users to enter $\exists$ in unicode instead of `exists`.

9. **Existential constructor**

$$\frac{E_1 \qquad E_2}{\{E_1,\, E_2\}}$$

   This constructs an existential, ie the weaker cousin of the Sigma.

10. **Existential eliminator**

$$\frac{x \qquad y \qquad E \qquad E'}{\text{let } \{x,\, y\} := E \text{ in } E'}$$

   As discussed in the intro, this models the rule of existential elimination. x and y are binders for E' but not E. Note that this is not the same as pair destructuring since this is used for unpacking the existential rather than the Sigma.

11. **Type ascriptions**

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

   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.

12. **Local let bindings**

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

   Local let bindings behave the same way as in ML-like languages. This binds the expression `E` to `x` in the body that is `E'`.

   In the event that the user does not provide a type annotation for the binding E, it will be inferred by the interpreter. Users can also provide an optional annotation via

```
let x : T := E in E'
```

   which is desugared into `let x := (E : T) in E'`

13. **Sum type**

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

   As syntactic sugar, users can write `A \/ B` instead since sum types model disjunction.

14. **Sum type constructor**

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

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

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

15. **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. Note that this corresponds to the rule of disjunction elimination.

```
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
```

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 sums.

Note here that `x` and `y` are binders, with `x` being bound in `E1` and `y` being bound in `E2`. `inl` and `inr` are used to indicate which case we are considering.

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

2.1.3 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 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`, and $\exists x : A$, $B(x)$ to abbreviate `exists x : A, B(x)`.

2.2 Syntax for statements and programs

Statements are top level commands through which users interact with our language. Programs in our language will be *nonempty* sequences of these statements. Lines that begin with `--` are treated as comments.

We have 4 key statements in our language, fashioned after Coq and Lean.

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.

Like with local `let` bindings, we also allow type annotations on def. Similarly, we treat `def x : T := E` as syntactic sugar for `def x := (E : T)`.

Since Curry-Howard allows us to interpret `(E : T)` as an assertion that `E` is a proof of the proposition that is `T`, we also allow users to write `lemma x : T := E` and `theorem x : T := E` instead of `def x : T := 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.

Later, 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.

2.3 Lean-esque syntactic sugar

Following Lean, we provide some lightweight syntactic sugar that allows users to write structured proofs in a manner more akin to how one would do so on paper.

1. `assume x0 ... xn, E` is syntactic sugar for `fun x0 ... xn => E`

   Similarly, type annotations can be given, so

   `assume (x0 : T0) ... (xn : Tn), E` desugars into

   `fun (x0 : T0) ... (xn : Tn) => E`

   Users can write `assume ...` in place of `fun ...` to introduce an arbitrary variable or hypothesis into the context to prove a universally quantified statement or implication.

2. `have x : T, from E, E'` is sugar for `let x : T := E in E'`

   This models how we write "We have ... because of ..." in pen and paper proofs. The variable `x` is used to bind the expression `E` that is the proof of the proposition `T` in the remainder of the proof that is `E'`.

3. `have T, from E, E'` is sugar for `let this : T := E in E'`

   In the event that the expression is not named, an implicit `"this"` name is given to the expression.

4. `show T, from E` is sugar for `(E : T)`

   This models how we would conclude a pen and paper proof by writing something along the lines of "Finally we have shown that ... because of ..."

3 Capture avoiding substitutions

In this section, we define the concept of free variables and capture avoiding substitutions. This will be important when we later define normalization and reduction in our big-step semantics.

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` and existential 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.

Finally, for existential elimination, we have

$$\mathrm{FV}(\text{let } \{x,\, y\} := E \text{ in } E') := \mathrm{FV}(E) \cup (\mathrm{FV}(E') \setminus \{x,\, y\})$$

since x and y are bound in E' but not E.

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.

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

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

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

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

For existential elimination, ie $\text{let } \{x,\, y\} := E \text{ in } E'$, note that x and y are bound in E' but not E.

4 Semantics

We define the semantics for expressions first and statements later. For expressions, we formulate a static type system along with a big-step operational semantics which is used for evaluation. For statements, we define evaluation in terms of a transition system, ie a generalized automaton. To evaluate programs, ie sequences of statements, we do not have a separate typechecking and evaluation phase. We typecheck and evaluate each statement entered line by line, as if they were entered in a REPL enviroment. This makes our implementation well suited for use in interactive environments.

The style of operational semantics presented here is a hybrid between the small-step dynamic semantics and big-step denotational semantics. This means that we will be carrying around an environment and performing syntactic substitutions with respect to it.

4.1 Expressions

4.1.1 Overview of reduction and normalization

In the theory of lambda calculi, the notion of computation is captured by the process of beta reduction. Normalization is then the process in which expressions are reduced 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 unlike the simply typed lambda calculus and even CoC, we have more than just function types in our language. In particular, we have 3 main types, namely `Pi`, `Sigma` and `Sum`. Thus we need to generalize the notion of normalization to account for these new types and their elimination rules. Intuitively, we want to repeatedly reduce an expression using all of these rules until we can reduce no further.

Next note that we will be reducing expressions and substituting with respect to an environment, 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 also need a definition of head normal form and reduction that takes care of expressions with free variables.

In the next section, we define the head normal form of an expression by way of a new concept – the *neutral expression*. After that, we introduce the *context*, which is our take on the environment from denotational semantics. We will then be in a position to discuss normalization.

4.1.2 Neutral expressions, head normal form, beta equivalence

Informally, 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 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.

Formally we define, using mutual recursion, the subset of neutral and normalized expressions via the following judgments:

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

The rules are as follows:

1. $$\frac{\text{neutral}(E)}{\text{hnf}(E)}$$
2. $$\frac{\text{hnf}(E)}{\text{hnf}(\lambda x,\, E)}$$

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.

$$\frac{\mathrm{neutral}(E_1) \qquad \mathrm{hnf}(E_1)}{\mathrm{neutral}(E_1\ E_2)}$$

8.

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

9.

$$\frac{\mathrm{neutral}(E)}{\mathrm{neutral}(\mathrm{fst}\ E)}$$

10.

$$\frac{\mathrm{neutral}(E)}{\mathrm{neutral}(\mathrm{snd}\ E)}$$

Note that `let ... in ...` 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.

Finally, we also define the notion of beta equivalence.

Definition 4.1 (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 $E_1 \equiv_{\alpha\beta} E_2$.

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 A brief overview of contexts

We define contexts, denoted by the metavariable, Γ, to be lists 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.

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.

Most of the time, when we work with contexts, we want them to be *well formed* in the sense that we want the contexts that we deal with in our typing and normalization judgments to satisfy some additional invariants.

Primarily, we want the "types" that we store in the context to be "valid types" rather than just being expressions in our language. We will discuss this in more detail later, alongside the static semantics.

4.1.4 Overview of judgement forms

Since we will be defining some judgment forms (including normalization and typing related ones) in a mutually recursive fashion, we first give an overview of them.

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 $\lambda\,(T : \text{Type})(x : T),\, x$. Unfortunately, this means that our language can be rather verbose as a lot of explicit type parameters must be provided.

3. $\Gamma \vdash E \Downarrow \nu$
   The $\cdot \vdash \cdot \Downarrow \cdot$ relation is our big-step normalization process. 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

As mentioned earlier, the aim of normalization is to reduce an expression, through repeated applications of the 3 elimination rules (ie `Pi`, `Sigma` and `Sum`) to head normal form, which includes neutral expressions and free variables as a subset.

One may notice that this intuitive formulation resembles a small-step semantics. Indeed, we may obtain a contraction rule from each elimination rule and then formalize a one-step transition relation, say $\Gamma \vdash E \longmapsto E'$. From that, we can then derive its reflexive, transitive closure, $\Gamma \vdash E \longmapsto^* E'$.

However we take a different approach. We define a big-step semantics directly because this translates nicer into a recursive interpreter written in a functional style.

In the rules below, note how our elimination rules account for expressions in head normal form, ie ν and τ, as well as neutral ones, ie n. Also notice how all the base cases are annotated with $\mathcal{WF}(\Gamma)$ judgments in the hypotheses. In other words, we always assume that our contexts satisfy some extra invariants when normalizing expressions using them. These details will be covered later.

1. **Type**

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

2. **Variables**

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

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

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. Users can also use these as a way to document their intentions.

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

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

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

The first rule says that we normalize function applications via syntactic substitution, with respect to an environment. Notice how this is a hybrid between the styles of dynamic semantics and denotational semantics.

The second rule utilizes the previously introduced concept of a neutral expression to handle the case when the elimination rule cannot be used because the expression at the head of a function application is a free variable. Remember that such variables can exist because users are allowed to introduce indeterminate constants as the top level via `axiom` statements.

5. **Normalizing under lambdas**

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

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

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

6. **Pi type constructor**

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

7. **Local let binding**

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

This rule was derived by desugaring let $x := E$ in E' into the function application $(\lambda x,\, E')\ E$.

8. **Normalizing under pair constructor**

$$\frac{\Gamma \vdash E_1 \Downarrow \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2}{\Gamma \vdash (E_1,\, E_2) \Downarrow (\nu_1,\, \nu_2)}$$

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**

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

$$\frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \mathrm{fst}\, E \Downarrow \mathrm{fst}\, n}$$

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

$$\frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \mathrm{snd}\, E \Downarrow \mathrm{snd}\, n}$$

11. **Existential type constructor**

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

12. **Normalizing under existential pair constructor**

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

13. **Existential eliminator**

$$\frac{\Gamma \vdash E \Downarrow \nu \qquad \Gamma \vdash \lambda x,\, \lambda y,\, E' \Downarrow \lambda x,\, \lambda y, \nu'}{\Gamma \vdash \text{let } \{x,\, y\} := E \text{ in } E' \Downarrow \text{let } \{x,\, y\} := \nu \text{ in } \nu'}$$

14. **Sum type constructor**

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

15. **Normalizing under sum data constructors**

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

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

16. **Normalizing sum eliminator**

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

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

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

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 types and contexts

Recall that when defining the notion of a context earlier, we said that we normally only work with well formed ones. In particular, this means that the types we store in the context are "valid". Here we properly define these concepts.

First we introduce a new metavariable s to denote either of the 2 sorts, `Type` and `Kind`.

Definition 4.2 (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 types" and so all well formed types are those expressions satisfying $\Gamma \vdash T \Leftarrow \mathrm{Type}$, ie they can be checked to have type Type.

With this, we define well formed contexts as contexts satisfying 2 key invariants:

1. Well formed contexts only contain well formed types.
2. The expressions representing the type and binding of a variable are both in head normal form.

Observe that in order to maintain these 2 invariants while typechecking an expression, we will need to perform normalization on types themselves. To see this, consider what happens when we want to typecheck a lambda abstraction. Intuitively, we procceed as in the simply typed lambda calculus. We want to add the variable and its type to the context and continue typechecking the body. But remember, we must ensure that the context remains well formed when typechecking the body. For this, we must check that the type is well formed and normalize it to head normal form before we can add it to the context.

However, it must be noted that we only normalize expressions after we typecheck them. At no point in our formulation in the next section, do we attempt to normalize an expression before checking that it has a well formed type.

Definition 4.3 (Well formed context)**.** Recalling that $\emptyset$ denotes the empty list and that :: denotes the list cons operation, we define precisely the judgment $\mathcal{WF}(\Gamma)$.

1. **Base case**

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

2. **Inductive cases**

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

Here we allow both τ and ν to be undefined.

4.1.7 Static semantics, ie typing rules

Here we define the 2 *mutually recursive* relations in our bidirectional typechecking algorithm.

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**

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

ν is allowed to be undefined here. This says that we may infer the type of a variable if the type information is already in our context.

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 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. This rule, together with the one on type ascriptions, is precisely the reason why we eagerly normalize types and maintain the 2 invariants required for our contexts to be well formed.

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. In fact, "types" are just expressions!

Ah but since we always normalize types before dealing with them, assuming that terms have a unique head normal form, we will normalize T and T' to τ and τ' respectively. After that, we need only compare them for structural equality, modulo alpha equivalence. Such an operation is trivial to implement using an implementation that represets variables by de bruijn indices.

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 $s_1,\, s_2 \in \{\text{Type},\, \text{Kind}\}$.

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**

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

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, $s_1,\, s_2 \in \{\text{Type},\, \text{Kind}\}$.

10. **Sigma introduction**

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

11. **Sigma elimination**

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

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

12. **Existential formation**

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

13. **Existential introduction**

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

14. **Existential elimination**

$$\frac{\Gamma \vdash E \Rightarrow \exists x : \tau,\, \tau'(x) \qquad (y,\, \tau'(x),\, \text{und}) :: (x,\, \tau, \text{und}) :: \Gamma \vdash E' \Rightarrow \tau'' \qquad x,\, y \notin \text{FV}(\tau'')}{\Gamma \vdash \text{let } \{x,\, y\} := E \text{ in } E' \Rightarrow \tau''}$$

Here we write $x,\, y \notin \text{FV}(\tau'')$ to emphasize that x and y should not occur free in the output type that is τ''.

By adding a fresh, unknown variable x, of the right type to the context, we essentially abstract away the first component of the underlying pair.

Notice how the rules for existentials in our language are similar to the typing rules for existential types. Also, take note of how this elimination rule models the rule of existential elimination in natural deduction as we discussed in the intro

$$\frac{\Gamma \vdash \exists x,\, \varphi(x) \qquad \Gamma,\, \varphi(x) \vdash \psi}{\Gamma \vdash \psi}$$

provided x does not occur free in Γ and ψ.

15. **Sum formation**

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

Note that this rule says that users can only 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`.

16. **Sum introduction**

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

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

Without the appropriate type ascription, we can't infer the type of `inl E` and `inr E`. These are meant to be used in conjunction with a type ascription, eg `(inl E : A + B)`.

17. **Sum elimination**

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

Note that the data constructors `(. , .)`, `{. , .}`, `inl` and `inr` do not come with type annotations and thus we are unable to infer the type of the constructed term. Hence they will need to be paired with type ascriptions to help the interpreter figure out what type the expression has.

For instance, to help the interpreter figure out what type `inl E` should be, we can ascribe `(inl E : A + B)` and the interpreter will then check if `E` has type `A`.

Similarly, we can't infer the type of an unannotated lambda like $\lambda x,\, x$. Users must either annotate the input type or ascribe a type to the whole function.

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

Similarly, $\Gamma \vdash E \Rightarrow T$ gives us 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.

4.2 Statements and programs

Programs in our language are nonempty sequences of statements, with each statement being built from some expression. We formalize the execution of programs by defining a *transition system*.

Before that, we first define the notion of a configuration, which plays the same role as states in automata.

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 sequence of statements to be executed next. 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 the initial and final configurations.

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

   This is the state in which we begin executing our programs. 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** $(\Gamma, \langle\rangle, E)$ After executing programs in our language, if all goes well, we'll end up in one of these states.

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.

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**

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

2. **Eval**

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

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**

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

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

TODO: flesh out this part

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.

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

Here we make an assumption that head normal forms are *unique*. This is not a far fetched assumption because it holds for the original Calculus of Constructions as well as various extensions of it that include `Sigma` types among others. [insert reference to Luo's phd thesis]

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, this significantly complicates discussions of the metatheoretic properties of the language and so for simplicity, we chose to follow the original formulation of the Calculus of Constructions and ignore this.

6 Technical implementation

Work in progress

Mention the pains of de bruijn indices.

7 More sample proofs