$\lambda C+$ specification

Watt Seng Joe Abdul Haliq S/O Abdul Latiff

April 20, 2021

Contents

1 Introduction **3**
1.1 $\lambda C+$ and CoC . . . 3
1.1.1 Similarities . . . 3
1.1.2 Extensions to CoC . . . 4

2 Metavariables **4**

3 Syntax **5**
3.1 Syntax for expressions . . . 5
3.1.1 Syntax rules . . . 5
3.1.2 Wildcard variables . . . 8
3.1.3 A word about notation . . . 8
3.2 Syntax for statements and programs . . . 9
3.3 Lean inspired syntactic sugar . . . 10

4 Binders and capture avoiding substitutions **10**
4.1 Free variables . . . 11
4.2 Substitution . . . 11

5 Semantics **12**
5.1 Expressions . . . 12
5.1.1 Overview of reduction and normalization . . . 12
5.1.2 Neutral expressions, head normal form, beta equivalence . 13
5.1.3 Contexts . . . 14
5.1.4 Big step normalization . . . 15
5.1.5 Well formed types . . . 17
5.1.6 Static semantics and bidirectional typechecking . . . 17
5.2 Statements and programs . . . 21
5.2.1 Configuration, ie state of program . . . 21
5.2.2 Big step semantics for programs . . . 22
5.2.3 Putting the transition system together . . . 23
5.2.4 Output expression of evaluating a program . . . 23

6 Limitations and future work **24**
6.1 Lack of strong normalization and consistency 24
6.2 Negation, falsity and truth 25
6.3 Lack of real type inference 25
6.4 Conflation of Prop and Type 26

1 Introduction

$\lambda C+$ is a proof assistant grounded in the Curry-Howard correspondence. The main aim of $\lambda C+$ is to implement a typed lambda calculus containing builtin types to encode the following logical connectives as per this correspondence [1]

1. Universal quantifier $\forall$
2. Implication $\rightarrow$
3. Conjunction $\wedge$
4. Disjunction $\vee$
5. Existential quantifier $\exists$

Another important goal of $\lambda C+$ is to provide some constructs like syntactic sugar that let us write longer proofs more conveniently.

At the core of our language is the Calculus of Constructions (CoC) as originally formulated by Coquand and Huet in [4]. A good introduction to typed lambda calculi and CoC is the book in [10].

Syntactically, $\lambda C+$ was heavily inspired by Standard ML and the Lean theorem prover. Semantically, we define both a dynamic and static semantics. For this, we use a big-step operational semantics inspired by Plotkin's seminal work [12], as well as a bidirectional typechecking algorithm as found in [8].

In the next section, we discuss how our language extends CoC. Thereafter, we will formally define the syntax and semantics of $\lambda C+$.

1.1 $\lambda C+$ and CoC

1.1.1 Similarities

Following CoC, $\lambda C+$ does not have separate syntactic categories for types and expressions. Thus, everything, including types themselves, are expressions. However, not all expressions are types. Those expressions which we call types are expressions, say T, that satisfy a very specific typing judgment, namely $\Gamma \vdash T \Leftarrow s$, where $s \in \{\texttt{Type}, \texttt{Kind}\}$. We'll revisit this again later.

Note that this means that unlike the polymorphic lambda calculus, we do not have separate constructs for declaring types and binding them.

`Type` and `Kind` are the 2 sorts of $\lambda C+$, inherited from CoC. These are often called $*$ and $\Box$ in the literature. `Type` can be thought of as the "type of all types" and so expressions with a type of `Type` are types. `Kind` is then the type of the expression `Type` and `Kind` itself doesn't have a type. CoC includes it primarily to prevent Girard's paradox (see [2]).

Finally, as with CoC, $\lambda C+$ doesn't have a separate sort for propositions, so `Prop` is identified with `Type`.

[1] Note that negation ($\neg$), falsity ($\bot$) and truth ($\top$) are not implemented in the language.

1.1.2 Extensions to CoC

$\lambda C+$ extends CoC in 2 distinct ways. The first is that we provide more types to model the various connectives more naturally. CoC in its original formulation does not contain inductive types and only has the Pi type (aka the dependent function type).

While this can encode $\forall$ and $\rightarrow$ naturally, $\wedge$, $\vee$ and $\exists$ have to be encoded in terms of the Pi. Chapter 7 of [10] explains how this can be done. However, we wanted to implement these directly via their Curry-Howard interpretation.

Thus $\lambda C+$ extends CoC by providing 3 additional types on top of the Pi type. These are

1. the (dependent) Sigma type $\Sigma_{x:A}B(x)$ [2]
   This is a dependent variation of the product type that allows the type of the 2nd component to depend on the value of the first.
   The simple pair type which models conjunction is then taken as an abbreviation for $\Sigma_{x:A}, B$ where x does not appear free in B.

2. the (non-dependent) sum type $A + B$
   These exist only at the type level, ie we can only form $A + B$: `Type` out of A : `Type` and B : `Type`. We do not allow either A : `Kind` or B : `Kind`.

3. the (dependent) existential type $\exists x : A,\ B(x)$
   This is a variation of the Sigma type with a "weaker" elimination rule to model the existential quantifier faithfully.

The second way in which $\lambda C+$ extends upon CoC is that we allow users to introduce global definitions via the `def x := E` *statement*. As a lambda calculus designed for theoretical study, CoC doesn't have statements, only expressions. However, writing programs without any means to *name* expressions, be it locally or globally is highly inconvenient.

Similarly, when we write mathematical proofs, it is common to introduce auxiliary definitions, lemmas and axioms [10]. For instance, we define a surjective function to be an $f : X \rightarrow Y$ such that $\forall y \in Y, \exists x \in X, f(x) = y$. We then prefer writing that f is surjective rather than repeatedly writing the long formula.

Thus we believe that it is necessary for any practical programming language and proof assistant to allow users to name things. For this, we augment CoC with the concept of a context, aka an environment, which will be used to store typing and binding information. New bindings can then be added to these contexts via the `def` statement.

2 Metavariables

We list all the metavariables that we will use here.

[2] We write $B(x)$ to indicate that the variable x may appear free in B. If we write B alone, as in $\Sigma_{x:A}, B$, then we mean that x cannot appear free in B.

1. s ranges over sorts, ie `Type` and `Kind`.
2. A, B, T, E range over expressions.
3. x, y, z range over variables.
4. ν, τ range over expressions in head normal form.
5. n ranges over neutral expressions.
6. Γ ranges over contexts.

Note that we often add subscripts and ' to them, like for instance E_1, ν'.

3 Syntax

$\lambda C+$ has 2 syntactic categories, namely expressions and statements. As with CoC, there are no separate syntactic categories for types and expressions. Here we describe the concrete ascii syntax for our language, alongside the syntactic sugar we provide.

At the top level, programs are nonempty sequences of statements, like `def`, `axiom`, `check` and `eval`.

3.1 Syntax for expressions

3.1.1 Syntax rules

1. **Sorts, variables and parentheses**

$$\frac{}{\texttt{Type}} \quad \frac{}{\texttt{Kind}} \quad \frac{}{x} \quad \frac{E}{(\,E\,)}$$

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

   Also, following other languages in the ML family, parentheses are only used for grouping.

2. **Function abstraction**

$$\frac{x \quad E}{\texttt{fun}\, x \texttt{ => } E}$$

   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.

$$\frac{x \quad T \quad E}{\texttt{fun}\,(x : T) \Rightarrow E}$$

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))
```

3. **Pi, Sigma and Existential type former**

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

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 Sigmas and existentials, ie `Sigma (x : A), B` and `exists (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`.

4. **Sigma data constructor and eliminator**

$$\frac{E_1 \quad E_2}{(E_1,\, E_2)} \qquad \frac{E}{\texttt{fst}\ E} \qquad \frac{E}{\texttt{snd}\ E}$$

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

5. **Existential data constructor and eliminator**

$$\frac{E_1 \quad E_2}{\{E_1,\, E_2\}} \qquad \frac{x \quad y \quad E \quad E'}{\text{let } \{x,\, y\} := E \text{ in } E'}$$

Note that for the eliminator, ie the rule on the right, x and y are bound in E' but not E.

6. **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.

7. **Local let bindings**

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

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'`

8. **Sum type former**

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

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

9. **Sum data constructor**

$$\frac{E}{\texttt{inl } E} \qquad \frac{E}{\texttt{inr } E}$$

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

10. **Sum 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 `match`$(E, x \to E_1, y \to E_2)$ to denote this construct.

3.1.2 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.

3.1.3 A word about notation

Note that we often write $\lambda x,\ E$ 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)`.

3.2 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**

$$\frac{x \quad E}{\texttt{def} \;\; x := E}$$

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**

$$\frac{x \quad T}{\texttt{axiom} \;\; x : T}$$

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 $\lambda C+$ conflates the notion of types and propositions, as introducing an arbitrary variable `x` of type `T` is akin to introducing an assumption (ie axiom).

3. **Check**

$$\frac{E}{\texttt{check} \; E}$$

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

4. **Eval**

$$\frac{E}{\texttt{eval} \; E}$$

This instructs the interpreter to normalize (ie fully reduce) the expression E.

3.3 Lean inspired 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` abbreviates

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

   This lets us 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 ..."

4 Binders and 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.

Though it is common for authors (like [8]) to omit this in their semantics, we prefer to be explicit about this since great care must be taken when implementing these. Readers should free to skip most of this section and return here only when they need to clarify doubts.

4.1 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.

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

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.

4.2 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` 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 $\texttt{match}(E, x \to E_1, y \to E_2)$, x is bound in E_1 and y is bound in E_2.

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

5 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.

5.1 Expressions

5.1.1 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 binders. The resulting expression is said to be in *head normal form*. In this section, we want to define similar notions for our richer language.

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 4 main types, namely Pi, Sigma, sum and existentials. Thus we need to generalize the

notion of normalization to account for these new types and their elimination rules.

Informally, an expression is in head normal form if no elimination rule can be applied anywhere within the expression, even underneath binders. For instance, the identity function $\lambda x,\ x$ is in head normal form but $(\lambda x,\ (\lambda x,\ x)\ x)$ is not since it can be reduced to $\lambda x,\ x$. Note that when we go underneath the outermost lambda to reduce the body, ie $(\lambda x,\ x)\ x$, the argument ie x in that function application is actually free. Thus our notion of normalization must now also account for free variables.

For this, we need to introduce the concept of the *neutral expression*. These are expressions which cannot be reduced by an eliminator because the expression to be eliminated is a free variable. As an example, suppose that f is a free variable of some function type and we have the expression

$$(f\ E_1\ \ldots\ E_n)$$

We cannot beta reduce and substitute in the body of f because the variable in head position, ie f, is free. We don't know anything about what the body of f looks like. The best we can do is normalize, ie fully reduce, each of the E_i to say E_i' and then leave it as $(f\ E_1'\ \ldots\ E_n')$.

Similar scenarios occur with expressions of the other types in our language. For instance, if p is a free variable of the Sigma type, `fst` p cannot project out the first component of p, so we normalize p to p' and then leave it as `fst` p'.

5.1.2 Neutral expressions, head normal form, beta equivalence

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

1. hnf(E)
   This asserts that E is an expression that is in head normal form.

2. neutral(E)
   This asserts that E is a neutral expression.

The rules are as follows:

$$\frac{\text{neutral}(E)}{\text{hnf}(E)} \qquad \frac{\text{hnf}(E)}{\text{hnf}(\lambda x,\ E)} \qquad \frac{\text{hnf}(E_1) \quad \text{hnf}(E_2)}{\text{hnf}(\Pi_{x:E_1} E_2)}$$

$$\frac{\text{hnf}(E_1) \quad \text{hnf}(E_2)}{\text{hnf}(\Sigma_{x:E_1} E_2)} \qquad \frac{\text{hnf}(E_1) \quad \text{hnf}(E_2)}{\text{hnf}(\exists x : E_1,\ E_2)} \qquad \frac{\text{hnf}(E_1) \quad \text{hnf}(E_2)}{\text{hnf}(E_1 + E_2)}$$

$$\frac{}{\text{neutral}(x)} \qquad \frac{\text{neutral}(E_1) \quad \text{hnf}(E_1)}{\text{neutral}(E_1\ E_2)}$$

$$\frac{\text{neutral}(E) \quad \text{hnf}(E_1) \quad \text{hnf}(E_2)}{\text{neutral}(\texttt{match}(E,\ x \to E_1,\ y \to E_2))} \qquad \frac{\text{neutral}(E)}{\text{neutral}(\texttt{fst}\ E)} \qquad \frac{\text{neutral}(E)}{\text{neutral}(\texttt{snd}\ E)}$$

$$\frac{\text{neutral}(E) \quad \text{hnf}(E')}{\text{neutral}(\text{let } \{x,\ y\} := E \text{ in } E')}$$

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

Finally, we also define the notion of and alpha and beta equivalence.

Definition 5.1 (Alpha and 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 $E_1 \equiv_{\alpha\beta} E_2$.

This concept of alpha and beta equivalence is important because in the bidirectional typechecking algorithm, we will need to compare types for beta equivalence.
For this, we will normalize them to head normal form and then compare them structurally, modulo alpha equivalence.

5.1.3 Contexts

This is a variation of the environments as seen in a big-step denotational semantics. These will be used to store variables along with their typing and binding

information when they are defined via the `def x := E` and `axiom x : T` statements.

We define contexts, denoted by the metavariable, Γ, to be lists of triples of the form (x, τ, ν) where τ and ν are allowed to be a special undefined value, which we denote by `und`.

Each triple can be read as

(variable name, type of variable, binding)

Note that this means we will be eagerly normalizing the type and binding of a variable before we store them in our contexts.

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

5.1.4 Big step normalization

The aim of normalization is to reduce an expression, through repeated applications of the eliminators for the 4 types, (ie Pi, Sigma, sum and existentials) to head normal form.

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.

1. **Type and variables**

$$\frac{}{\Gamma \vdash \texttt{Type} \Downarrow \texttt{Type}} \qquad \frac{(x, \tau, \nu) \in \Gamma}{\Gamma \vdash x \Downarrow \nu} \qquad \frac{(x, \tau, \text{und}) \in \Gamma}{\Gamma \vdash x \Downarrow x}$$

2. **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 tell the typechecker to ensure that E really has the prescribed type T.

3. **Pi type former and data constructor**

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

4. **Pi eliminator**

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

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

The first rule says that we normalize function applications via substitution with respect to the context Γ.

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 we are normalizing under binders.

5. **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 treating let $x := E$ in E' as the function application $(\lambda x,\, E')\ E$.

6. **Sigma type former and data constructor**

$$\frac{\Gamma \vdash A \Downarrow \tau \qquad \Gamma \vdash B \Downarrow \tau'(x)}{\Gamma \vdash \Sigma_{x:A} B(x) \Downarrow \Sigma_{x:\tau} \tau'(x)} \qquad \frac{\Gamma \vdash E_1 \Downarrow \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2}{\Gamma \vdash (E_1,\, E_2) \Downarrow (\nu_1,\, \nu_2)}$$

7. **Sigma eliminator**

$$\frac{\Gamma \vdash E \Downarrow (\nu_1,\, \nu_2)}{\Gamma \vdash \texttt{fst}\ E \Downarrow \nu_1} \qquad \frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \texttt{fst}\ E \Downarrow \texttt{fst}\ n}$$

$$\frac{\Gamma \vdash E \Downarrow (\nu_1,\, \nu_2)}{\Gamma \vdash \texttt{snd}\ E \Downarrow \nu_2} \qquad \frac{\Gamma \vdash E \Downarrow n}{\Gamma \vdash \texttt{snd}\ E \Downarrow \texttt{snd}\ n}$$

8. **Existential type former and data constructor**

$$\frac{\Gamma \vdash A \Downarrow \tau \qquad \Gamma \vdash B \Downarrow \tau'(x)}{\Gamma \vdash \exists x : A,\, B(x) \Downarrow \exists x : \tau,\, \tau'(x)} \qquad \frac{\Gamma \vdash E_1 \Downarrow \nu_1 \qquad \Gamma \vdash E_2 \Downarrow \nu_2}{\Gamma \vdash \{E_1,\, E_2\} \Downarrow \{\nu_1,\, \nu_2\}}$$

9. **Existential eliminator**

$$\frac{\Gamma \vdash E \Downarrow \{\nu_1,\, \nu_2\} \qquad \Gamma \vdash E'[x \mapsto \nu_1][y \mapsto \nu_2] \Downarrow \nu}{\Gamma \vdash \text{let } \{x,\, y\} := E \text{ in } E' \Downarrow \nu}$$

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

These rules are adapted from the elimination rules found in Chapter 24 of [11]. Note that the book defines a small-step semantics for evaluating expressions of the non-dependent variation.

10. **Sum type former and data constructor**

$$\frac{E_1 \Downarrow \nu_1 \quad E_2 \Downarrow \nu_2}{E_1 + E_2 \Downarrow \nu_1 + \nu_2} \qquad \frac{\Gamma \vdash E \Downarrow \nu}{\Gamma \vdash \texttt{inl}\ E \Downarrow \texttt{inl}\ \nu} \qquad \frac{\Gamma \vdash E \Downarrow \nu}{\Gamma \vdash \texttt{inr}\ E \Downarrow \texttt{inr}\ \nu}$$

11. **Sum eliminator**

$$\frac{\Gamma \vdash E \Downarrow \texttt{inl}\ \nu \quad \Gamma \vdash E_1[x \mapsto \nu] \Downarrow \nu_1}{\Gamma \vdash \texttt{match}(E,\, x \to E_1,\, y \to E_2) \Downarrow \nu_1} \qquad \frac{\Gamma \vdash E \Downarrow \texttt{inr}\ \nu \quad \Gamma \vdash E_2[y \mapsto \nu] \Downarrow \nu_2}{\Gamma \vdash \texttt{match}(E,\, x \to E_1,\, y \to E_2) \Downarrow \nu_2}$$

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

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.

5.1.5 Well formed types

An expression T is said to be a well formed type with respect to the context Γ if it satisfies

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

This will be defined formally later via *type formation rules* in our static semantics. These rules tell us when an expression is a well formed type.

Informally, we can think of `Type` as the "type of all types" and so all well formed types are those expressions satisfying $\Gamma \vdash T \Leftarrow \texttt{Type}$, ie they can be checked to have type `Type`.
`Kind` is just here mainly to be the type of `Type` as well as the type of type constructors. (Type constructors are functions whose output type is `Type`). This was inherited from CoC.

5.1.6 Static semantics and bidirectional typechecking

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 semantics we present here follow [8], though we prefer using the notation $\cdot \vdash \cdot \Rightarrow \cdot$ and $\cdot \vdash \cdot \Leftarrow \cdot$ as opposed to $\cdot \vdash \cdot ::_{\uparrow} \cdot$ and $\cdot \vdash \cdot ::_{\downarrow} \cdot$.

We highly recommend the talk [1] for readers who need more intuition on this. 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.
This is why the $\lambda C+$ interpreter is able to infer the type for eliminators but struggles with data constructors. To help the interpreter infer the type of a data constructor, type annotations must be provided via ascriptions, eg we must write `((a, b) : A \/ B)` instead of just `(a, b)` alone.

In the rules below, we'll see that each of the 4 main types, ie Pi, Sigma, sum and existentials come with 3 rules, namely

* Type formation
* Introduction
* Elimination

Type formation rules tell us when an expression constructed using a type former like `Pi` or `+` is a well formed type. The introduction and elimination rules give us the typing rules for the data constructors and eliminators respectively.

1. **Var**

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

   Here, ν is allowed to be undefined but not τ. This says that we may infer the type of a variable if the type information is already in our context.

2. **Type**

$$\frac{}{\Gamma \vdash \texttt{Type} \Rightarrow \texttt{Kind}}$$

   This follows CoC.

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.

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

Note that the first rule doesn't allow users to assert that $(E : \texttt{Kind})$ since there is no s with $\Gamma \vdash \texttt{Kind} \Rightarrow s$. 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.

This is where we need to check types for alpha and beta equivalence.

5. **Pi type 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 $s_1,\, s_2 \in \{\texttt{Type}, \texttt{Kind}\}$. This was adapted from CoC.

6. **Pi introduction**

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

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

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. **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**

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

Note that we only treat local let bindings as function application, ie $((\lambda x,\, E')\ E)$ for the purposes of normalization, not for typechecking. For typechecking, we extend the context and typecheck the body E'.

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

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**

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

Notice how the type of the $\texttt{snd}\ E$ depends on the *value* of $\texttt{fst}\ E$.

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''.

15. **Sum formation**

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

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, 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 formers found in `Kind`.

16. **Sum introduction**

$$\frac{\Gamma \vdash E \Leftarrow \tau_1}{\Gamma \vdash \texttt{inl}\ E \Leftarrow \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash E \Leftarrow \tau_2}{\Gamma \vdash \texttt{inr}\ E \Leftarrow \tau_1 + \tau_2}$$

17. **Sum elimination**

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

5.2 Statements and programs

Recall that programs in our language are nonempty sequences of statements, with each statement being built out of expressions. We formalize the execution of programs by defining a *transition system*, ala [12].

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

5.2.1 Configuration, ie 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:** $(\emptyset, \langle S_0; \ldots; S_n \rangle, \texttt{Type})$

   This is the state in which we begin executing our programs. Initially, our global context is empty. `Type` is used as a dummy initial output value here.

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.

5.2.2 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**

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

   The output of a `check` statement is τ, the type of E in head normal form.

2. **Eval**

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

   `eval` instructs the interpreter to normalize the given expression and output the normalized form. Note that we first verify that the expression is well typed before normalizing it.

3. **Axiom**

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

   `axiom` statements allow users to introduce variables with a type but no binding. First the type T is checked to be well formed. Then it is normalized to τ. Afterwards, the context Γ is extended with a new binding. Note that `und` here indicates that x has no binding, like an indeterminate variable.

   This means that when we enter `axiom x : T` into the interpreter, the output is the head normal form of T.

4. **Def**

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

`def` statements are used to introduce global bindings. The expression E is first typechecked, and then normalized to ν. Thereafter, the context Γ is extended with a new binding.

The output of a `def` statement is the normalized expression ν, and the its type, τ.

> This is why the interpreter outputs cryptic expressions of the form `(E : T)` after evaluating `def` statements.

5. **Sequences of statements**

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

To evaluate a sequence of statements, we evaluate the head and then recursively evaluate the tail with the possibly modified context.

5.2.3 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

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

Notice how our formalization mimics the definition of an automaton.

5.2.4 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

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

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.

6 Limitations and future work

6.1 Lack of strong normalization and consistency

The original formulation of CoC possesses some nice properties like type preservation (aka subject-reduction in the literature) and strong normalization [3, 7]. This means that any sequence of reductions (like beta reduction) using the various eliminators, when applied to *any* well typed expression, always terminates. Note that this is a very strong version of the progress property that says that well typed expressions never "get stuck".

[3] explains that this strong normalization property yields yet another nice property: the type system of CoC is logically consistent.

What does it mean for a type system to be logically consistent? Remember that the Curry-Howard correspondence allows us to identify propositions (ie logical formulae) as types and expressions as proofs. In this vein, we can view a type system as a *logical calculus* presented in natural deduction form.

In logic, a calculus is said to be inconsistent if there is a proof of false from it. By the Curry-Howard correspondence, an inconsistency arises in a type system if there is an expression (ie a proof) of the empty type (ie falsity). [3]

Many of these expressions are constructed via Girard's paradox [2]. These expressions are non-normalizing in that they have infinite reduction sequences. As an example, the looping combinator from the untyped lambda calculus

$$(\lambda x,\, (x\, x))\ (\lambda x,\, (x\, x))$$

is non-normalizing since we can keep beta reducing it forever. Thus being able to derive Girard's paradox comprises both strong normalization as well as consistency.

While CoC possesses the nice properties of strong normalization and consistency, these need not hold for $\lambda C+$ since we extend CoC with 3 additional types, ie Sigma, sum and existentials.

For instance, section 2.2.4 of [9] explains that our Sigma formation rule

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

allows for Girard's paradox to be derived since we allow for s_1 to be `Kind` and s_2 to be `Type`. Note that the paper uses *Prop* instead of `Type` as we do here.

This means that there are expressions in $\lambda C+$ that do not terminate, even though $\lambda C+$ forgoes recursion! Users are (in theory) able to construct these to provide a proof of false.

While this is undesirable as our language is aimed at formalizing logic and theorem proving, we are not particularly bothered since these expressions are

[3] Note that this concept of consistency is not the same as type safety. Many functional programming languages, like Haskell, are type safe but not consistent. To see this, note that the expression `undefined` in Haskell has type `a` where `a` is an arbitrary type variable. Since `a` is arbitrary, we could take this to be the empty type `Void`.

astronomically large and intricate [2]. Thus it is highly unlikely that users will stumble upon these issues unwittingly.

In order to provide a semantics for the Sigma type that doesn't affect the logical consistency of the type system, a common trick as found in [9] is to further stratify the universe of sorts from simply `Type : Kind` to a countable hierarchy of type universes. Proof assistants like Coq and Lean which are based on a variation of CoC with inductive types use this approach as well. However, adopting such a solution will drastically complicate our semantics and implementation.

Sigma types aside, we are also unsure if our variation of existential types can lead to any issues with strong normalization and consistency.

As we are not experts on the subject, our emphasis was on simplicity rather than these metatheoretic properties. We wanted a simple semantics that would allow us to implement a language that explores the Curry-Howard correspondence. As such, we were willing to make this trade off.

That said, we would like to explore how to address these issues in future once we acquire more expertise in type theory.

6.2 Negation, falsity and truth

Currently $\lambda C+$ doesn't include types for the following connectives:

* Negation, ie $\neg A$
* Falsity, ie $\bot$
* Truth, ie $\top$

According to the Curry-Howard correspondence, $\bot$ and $\top$ can be modeled by empty and singleton types respectively. Negation, ie $\neg A$, can then be treated as an abbreviation for $A \to \bot$. These should not prove difficult to add to our language but we decided to leave them out as we were running out of time for the project. We would however like to come back to add these in the future.

6.3 Lack of real type inference

In our static semantics, we used a bidirectional typechecking algorithm as in [8]. While this provides a limited form of type inference, it is highly limited compared to the unification based methods as used in algorithms like Hindley-Milner. As such $\lambda C+$ can be rather annoying to use since a lot of type annotations must be provided explicitly.

While this is undesirable and makes our language rather verbose, there is unfortunately, no easy solution. Type inference for dependently typed languages is unfortunately undecidable in general [6] since higher-order unification is needed to infer certain things, like the element of a Pi type [5]. Proof assistants such as Lean often use various heuristics [5] to handle this.

We chose to forgo this because they can get rather complicated. Fortunately, we found that bidirectional typechecking works well for dependently typed languages and it's not too difficult to formalize and implement.

Still, we recognize this is an avenue for future work. $\lambda C+$ could be made less verbose by implementing some heuristics for type inference. This would greatly improve the user friendliness of the language.

6.4 Conflation of Prop and Type

$\lambda C+$ stays true to the Curry-Howard correspondence by happily identifying propositions with types. Unfortunately, this may be unintuitive for users who are familiar with first (or higher order) logic.

Consider the type in the concrete syntax, `exists x : A, A`, where `A` is some arbitrary type. While this is a perfectly well formed type, it doesn't much sense from a logical point of view. The issue is that the type `A` is used in 2 different ways here. The first use of `A` suggests that it's being used like a *set*, with `x` being an element of it. But the second `A` suggests that it's being used as a *proposition*.

In logic, there is a clear distinction between terms and well formed formulae, which belong to different syntactic categories. Thus formulae like $\exists x \in A$, A are not well formed.

Though the Curry-Howard correspondence is able to give meaning to the type `exists x : A, A`, this may be counter-intuitive for some users.

Although we would like to address this, there is no easy fix for this because this was inherited from CoC. One way to fix this is to further stratify the universe of sorts. In other words, we would need to split `Prop` out into its own sort, giving us 3 sorts: `Prop`, `Type` and `Kind`.

The typing rules would then need to be adjusted to account for this. However, this would complicate our semantics and implementation and it may not preserve the consistency of the type system when viewed as a logical calculus. Thus we did not implement this.

References

[1] Christiansen, D. Bidirectional type checking. `https://www.youtube.com/watch?v=utyBNDj7s2w`, June 2019.

[2] Coquand, T. An analysis of girard's paradox. In *LICS* (1986), IEEE Computer Society, pp. 227–236.

[3] Coquand, T., and Gallier, J. H. A proof of strong normalization for the theory of constructions using a kripke-like interpretation. Tech. rep., 1990.

[4] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76*, 2 (1988), 95–120.

[5] de Moura, L. M., Avigad, J., Kong, S., and Roux, C. Elaboration in dependent type theory. *CoRR abs/1505.04324* (2015).

[6] Dowek, G. The undecidability of typability in the lambda-pi-calculus. In *Typed Lambda Calculi and Applications* (Berlin, Heidelberg, 1993), M. Bezem and J. F. Groote, Eds., Springer Berlin Heidelberg, pp. 139–145.

[7] Geuvers, H. A short and flexible proof of strong normalization for the calculus of constructions. In *Types for Proofs and Programs* (Berlin, Heidelberg, 1995), P. Dybjer, B. Nordström, and J. Smith, Eds., Springer Berlin Heidelberg, pp. 14–38.

[8] Löh, A., Mcbride, C., and Swierstra, W. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticae 102*, 2 (2010), 177–207.

[9] Luo, Z. An extended calculus of constructions.

[10] Nederpelt, R., and Geuvers, H. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[11] Pierce, B. C. *Types and Programming Languages*, 1 ed. MIT Press, Feb. 2002.

[12] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981.