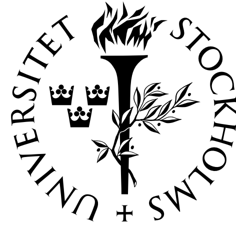

SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Mechanizing bidirectional type checking

av

Martin Svanberg

2023 - K8

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Mechanizing bidirectional type checking

Martin Svanberg

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Evan Cavallo

2023

Abstract

We present an extension of the simply typed lambda calculus including coproducts, and prove that the properties of progress and preservation hold for this language. We then implement bidirectional type checking for this language and prove its correctness. Finally, we show how the type checking algorithm can be extended to support polymorphism. All proofs are mechanized in Agda, a proof assistant based on dependent type theory.

Contents

1 Background **3**
1.1 Constructivism . . . 3
1.2 Dependent types . . . 4
1.3 Agda . . . 5

2 Language formalization **8**
2.1 Untyped lambda calculus . . . 8
2.2 Types . . . 11
2.3 Contexts . . . 13
2.4 Typing rules . . . 14
2.5 Substitution . . . 17
2.6 Reduction rules . . . 19
2.7 Progress and evaluation . . . 22

3 Bidirectional typechecking **25**
3.1 Terms . . . 26
3.2 Typing rules . . . 27
3.3 Lemmas . . . 29
3.4 Synthesizing and inheriting . . . 33
3.5 Soundness . . . 37
3.6 Annotatability . . . 38

4 Polymorphism **41**
4.1 Intrinsic presentation . . . 44
4.2 Type checking preliminaries . . . 45
4.3 Terms and typing judgments . . . 46
4.4 Lemmas . . . 48
4.5 Synthesizing and inheriting . . . 50

5 Conclusion **53**

Chapter 1

Background

In this thesis, we will take the simply typed lambda calculus as a starting point and extend it with a few more constructs, including coproducts and polymorphism. For this language we will then implement an algorithm known as bidirectional type checking, used by many real-world programming language implementations, and prove the soundness, completeness, and decidability of the algorithm using the proof assistant Agda. We assume that the reader has some familiarity with functional programming and mathematical logic.

To start off, we will introduce some of the history and theory behind proof assistants. There is far more to say about these matters than can be effectively communicated in this thesis, but we will strive to provide a minimal overview.

1.1 Constructivism

In the early 20th century, some mathematicians sought to reformulate mathematics on principles of constructivism. While there are several different schools of thought about what constructive mathematics means, the common understanding in modern times is based on the intuitionistic BHK-interpretation pioneered by Brouwer, Heyting, and Kolmogorov. This represents a form of mathematics where a "statement is *true* if we have proof of it, and *false* if we can show that the assumption that there is a proof for the statement leads to a contradiction." [TD88, p. 4]

This has consequences for how we interpret logical operations. For example, "proof of $A \vee B$ is given by presenting either proof of A or B (plus the stipulation that we want to regard the proof presented as evidence for $A \vee B$)." [TD88, p. 9]. It does not suffice to prove $A \vee B$ without having proof for one of A or B. Instead we must *construct* the proof of the statement from proofs of its constituent parts. This stands in contrast to classical logic, where we admit proofs that do not tell us how we construct solutions to the statements we intend to prove. As an example, consider the following theorem:

Theorem. *There exist irrational numbers a, b such that a^b is rational.*

Proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational, in which case we take $a = b = \sqrt{2}$, or it is irrational in which case we take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. □

While this constitutes a proof under classical logic, we are unable to say which choice of a and b satisfies the property. [BPI22, section 1] This is not a valid line of reasoning in constructive mathematics, which can be seen as a stricter form of mathematics where we require explicit constructions of our objects of study.

It turns out that constructive mathematics is an especially well-suited foundation for computer-aided proofs of mathematics. Classical mathematics does not map cleanly to programming. Quoting Martin-Löf, "the mathematical notions have gradually received an interpretation, the interpretation which we refer to as classical, which makes them unusable for programming." [Mar82, p. 169]. Unlike classical proofs, "every constructive proof embodies an algorithm that, in principle, can be extracted and recast as a computer program; moreover, the constructive proof is itself a verification that the algorithm is correct." [BPI22, section 3.4]

Using constructive mathematics as a foundation, type theory develops the tools we need to enable automatic theorem proving.

1.2 Dependent types

Per Martin-Löf, professor emeritus at Stockholm University, developed intuitionistic type theory in a 1972 paper. [Mar72] In intuitionistic type theory, every mathematical object has a certain *type*. A type "is defined by describing what we have to do in order to construct an object of that type." [BPI22] The concrete technical differences between set theory and type theory are subtle, and it is unfortunately outside of the scope of this thesis to compare them. Per Martin-Löf's intuitionistic type theory, which we will interchangeably call dependent type theory, forms the theoretical foundation underpinning many modern proof assistants.

The key to making dependent type theory work as a foundation for mechanized proofs of constructive mathematics is the Curry-Howard correspondence, which states that "a proposition is the type of its proofs." [DP23, p. 2.2] This allows us to think of logical propositions as types and elements of types as evidence for the truth of their types, and hence their corresponding propositions. In simple words, a program which type checks is a proof of its type.

1.3 Agda

Agda is a dependently typed programming language and proof assistant originally developed at Chalmers University. Ulf Norell described it in detail in his PhD thesis "Dependently Typed Programming in Agda". [Nor09] Agda is a modern tool for mechanizing mathematics[1] in the style of a functional programming language like Haskell. We can neither cover Agda nor Haskell in depth here, but will rather give a brief overview of how we can prove theorems in Agda.

A trivial example

In Agda, we make use of the Curry-Howard correspondence to claim that types correspond to propositions. Take the following function as an example.

```
identity : A → A
identity x = x
```

The identity function has the type signature `A → A`. In typical programming parlance, we might say that `identity` takes an object of type `A` and produces another object of type `A`. But another way to look at the signature is to see the arrow as a logical implication, and that evidence of the proposition `A` implies evidence of the proposition `A`. This is tautological and the reader would likely accept this claim as true without requiring proof thereof, but our definition above contains not only a theorem but also a proof.

The argument states that, assuming that we have evidence `x` of `A`, and using this evidence we show that we have evidence of `A`. In usual logic notation we might express this as below to construct a lambda expression, which we will discuss in Chapter 2.

$$\frac{x : A \vdash x : A \quad A \text{ type}}{\cdot \vdash \lambda x.\, x : A \to A}$$

The proof is constructive since it not only proves the theorem $A \to A$, but also gives us a method for constructing evidence of the theorem.

Induction

Logical induction is an important method of writing proofs in Agda. We start by proving a base case which terminates our proof, and then typically prove a number of additional cases by showing how they reduce to proofs of other cases. Since Agda is a total programming language, proofs that don't terminate are not valid.

[1] Mechanization of mathematics means writing proofs in such a way that they can be checked by a computer

An example of this might be the following definition of the Peano numbers using a `zero` constructor and a `suc` constructor. Here, we have a data type that can represent natural numbers in the form `zero` for zero, `suc zero` for one, `suc (suc zero)` for two and so on.

```
data Nat : Set
  zero : Nat
  suc N : Nat → Nat
```

Confusingly, Agda uses the word `Set` to denote types. Defining an addition function now happens in the form of an inductive proof on two cases.

```
plus : Nat → Nat → Nat
plus zero b = b
plus (suc a) b = suc (plus a b)
```

Here we make use of *pattern matching*, a common technique in functional programming languages, to recursively destruct[2] the left-hand side of the addition until it is `zero`, at which point we are finished. Agda enables the use of pattern matching to construct proofs.

Dependent types in Agda

Agda also supports types that are indexed or parametrized by elements of types, dependent types, which are a tool that we will make heavy use of in our proofs. The canonical example of a dependent type is the vector type, whose type describes the number of elements in the vector. It can be defined as below:

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Note that the vector type is parametrized by the type `A`, and that the signature of the data type looks like a function from `Nat` to `Set`. This example also demonstrates a number of other features, including the definition of the infix operator `_::_` as well as implicit arguments given in curly braces. Agda will try to infer implicit arguments, leading to shorter code, but we can also provide implicit arguments explicitly using a similar curly brace syntax.

[2] Destructing is also known as destructuring in other languages

This was a glimpse of how Agda looks and behaves. The interested reader will want to consult a proper introduction to the language in order to truly understand it, for example the official Agda documentation [Agd23] or *Programming Language Foundations in Agda* [WKS22], which this thesis leans heavily on. We will now start looking into how to describe a programming language in Agda.

Chapter 2

Language formalization

In this chapter we will describe our language under study, starting from a description using ordinary notation from mathematical logic and ending with a mechanized formalization in Agda.

2.1 Untyped lambda calculus

We start by describing the untyped lambda calculus. It will have little relevance to the rest of the thesis, as we will be concerned with a language that has more terms, and eventually also typing rules. But it provides a starting point for us to build upon.

The lambda calculus is one of the simplest Turing-complete languages, and certainly one of the oldest. It is remarkably simple but can compute anything a modern programming language can compute. The lambda calculus has the following syntax, given in Backus-Naur form:

$$
\begin{array}{lll}
t ::= & & \\
\mid & x & \text{variable} \\
\mid & \lambda\, x \Rightarrow t & \text{abstraction} \\
\mid & t \cdot t & \text{application}
\end{array}
$$

We define a *variable* as a placeholder for other terms, and a *lambda abstraction* as a way to represent functions. The lambda abstraction *binds* a variable (x above) and gives it a name[1] that can only be used in the body of the abstraction. Conversely, we also define *function application* which removes the abstraction by substituting a term for the bound variable. Taken together, they form a primitive but powerful language.

[1] In our initial presentation variables have ordinary alphabetic names, but we will use a different naming method called de Bruijn indices in the mechanization.

Following convention, function application is left-associative. As an example, we can express numbers in this language using Church numerals. Church numerals encode the number n as the operator that composes a function with itself n times.

$$
\begin{aligned}
&0 \equiv \lambda\, s \Rightarrow \lambda\, z \Rightarrow z \\
&1 \equiv \lambda\, s \Rightarrow \lambda\, z \Rightarrow s \cdot z \\
&2 \equiv \lambda\, s \Rightarrow \lambda\, z \Rightarrow s \cdot (s \cdot z) \\
&\dots
\end{aligned}
$$

Here, z indicates a function that represents zero and s a function that represents the successor function. Given a natural number, the successor function gives the next natural number. In this fashion, we can also define a plus function:

$$a + b \equiv \lambda\, a \Rightarrow \lambda\, b \Rightarrow \lambda\, s \Rightarrow \lambda\, z \Rightarrow a \cdot s \cdot (b \cdot s \cdot z)$$

In order to make use of this definition, we need to be able to reduce expressions. To do this, we define a number of reduction rules. The first one is called β-reduction, which replaces the *formal parameter* of the function with the *actual parameter*. [WKS22, Lambda] In the expression $(\lambda\, x \Rightarrow x) \cdot y$, x is the formal parameter and y is the actual parameter. We then need two rules that let us perform reductions in the left-hand side and right-hand side of an application. These are called $\xi\text{-}\cdot_1$ and $\xi\text{-}\cdot_2$, respectively.

Reduction rules can be composed. For example, the rule $\xi\text{-}\cdot_1\ \beta\text{-}\lambda$ will β-reduce in the left-hand side of an application.

The $\xi\text{-}\cdot_2$ rule has the additional requirement that the left-hand side needs to be a *value*, a notion that we will define more clearly later, but which we for now can think of as an expression that is fully reduced. This implies an essential ordering of the rules: we must reduce the left term in an application in order to get a value that we can use to reduce the second term. Note that this is not the only possible evaluation order, but merely the one we have chosen.

$$\frac{}{(\lambda\, x \Rightarrow N) \cdot V \rightarrow N[x := V]}\ \beta\text{-}\lambda$$

$$\frac{L \rightarrow L'}{L \cdot M \rightarrow L' \cdot M}\ \xi\text{-}\cdot_1$$

$$\frac{M \rightarrow M' \quad V \text{ value}}{V \cdot M \rightarrow V \cdot M'}\ \xi\text{-}\cdot_2$$

Figure 2.1: Untyped lambda calculus reduction rules

Using notation from logic, we define these rules in Figure 2.1. These rules suffice to reduce expressions in the lambda calculus to values.

A reduction of $1 + 1$ using our reduction rules then looks as follows:

$$\begin{aligned}
1 &\equiv \lambda\, s \Rightarrow \lambda\, z \Rightarrow s \cdot z \\
1 + 1 &\equiv (\lambda\, a \Rightarrow \lambda\, b \Rightarrow \lambda\, s \Rightarrow \lambda\, z \Rightarrow a \cdot s \cdot (b \cdot s \cdot z)) \cdot 1 \cdot 1 \cdot s \cdot z \\
&\xrightarrow{\xi\text{-}\cdot_1\, \xi\text{-}\cdot_1\, \xi\text{-}\cdot_1\, \beta\text{-}\lambda} (\lambda\, b \Rightarrow \lambda\, s \Rightarrow \lambda\, z \Rightarrow 1 \cdot s \cdot (b \cdot s \cdot z)) \cdot 1 \cdot s \cdot z \\
&\xrightarrow{\xi\text{-}\cdot_1\, \xi\text{-}\cdot_1\, \beta\text{-}\lambda} (\lambda\, s \Rightarrow \lambda\, z \Rightarrow 1 \cdot s \cdot (1 \cdot s \cdot z)) \cdot s \cdot z \\
&\xrightarrow{\xi\text{-}\cdot_1\, \beta\text{-}\lambda} (\lambda\, z \Rightarrow 1 \cdot s \cdot (1 \cdot s \cdot z)) \cdot z \\
&\xrightarrow{\beta\text{-}\lambda} 1 \cdot s \cdot (1 \cdot s \cdot z) \\
&\xrightarrow{\xi\text{-}\cdot_1\, \beta\text{-}\lambda} ((\lambda\, z \Rightarrow s) \cdot z) \cdot (1 \cdot s \cdot z) \\
&\xrightarrow{\xi\text{-}\cdot_1\, \beta\text{-}\lambda} s \cdot (1 \cdot s \cdot z) \\
&\xrightarrow{\xi\text{-}\cdot_2\, \xi\text{-}\cdot_1\, \beta\text{-}\lambda} s \cdot ((\lambda\, z \Rightarrow s \cdot z) \cdot z) \\
&\xrightarrow{\xi\text{-}\cdot_2\, \beta\text{-}\lambda} s \cdot (s \cdot z)
\end{aligned}$$

Since two is the successor of the successor of zero, we have reached the correct answer. This was a fairly cumbersome way of proving that $1 + 1 = 2$, but it illustrates how we can define reduction rules in order to evaluate our expressions.

We will now take a few liberties and add a few more terms to our untyped lambda calculus. Figure 2.2 defines them in Backus-Naur form.

$$
\begin{array}{lll}
L, M, N ::= & & \\
\quad \mid & x & \text{variable} \\
\quad \mid & \lambda\, x \Rightarrow N & \text{abstraction} \\
\quad \mid & L \cdot M & \text{application} \\
\quad \mid & \text{zero} & \text{zero} \\
\quad \mid & \text{suc}\, M & \text{successor} \\
\quad \mid & \text{case}\mathbb{N}\, L\, [\text{zero} \Rightarrow M \mid \text{suc}\, M \Rightarrow N] & \text{case}\mathbb{N} \\
\quad \mid & \mu\, x \Rightarrow N & \text{fixpoint} \\
\quad \mid & \text{inj}_1\, N & \text{left injection} \\
\quad \mid & \text{inj}_2\, N & \text{right injection} \\
\quad \mid & \text{case}\uplus\, L\, [\text{inj}_1\, M \Rightarrow N \mid \text{inj}_2\, M \Rightarrow N] & \text{case}\uplus \\
\quad \mid & \langle M , N \rangle & \text{product} \\
\quad \mid & \text{proj}_1\, N & \text{left projection} \\
\quad \mid & \text{proj}_2\, N & \text{right projection}
\end{array}
$$

Figure 2.2: Extended untyped lambda calculus terms

The first three terms comprise the standard lambda calculus. The μ term gives us a way to express recursion without the Y combinator,[2] the zero and suc terms let us express natural numbers without Church numerals, and the case term gives us a way to pattern match on natural numbers. We will not define their reduction rules yet.

We also add two more constructs to our language: products and coproducts. A product is the type of a term that contains two other terms. In typical programming languages these might be known as structs or tuples. Pairs are constructed using the $\langle M , N \rangle$ syntax, and are destructed using the projective terms. Coproducts are terms that contain either one term or another but not both. This is akin to the `Either` type in Haskell. Coproducts are constructed using the injective constructors, and destructed using the case$\uplus$ term.

2.2 Types

The goal of this thesis is to present an algorithm for typechecking. In order to do that, we need to have types. Types allow us to prove that our expressions are well-formed before we reduce them. We will now begin to talk about types and start slowly introducing how we might go about representing them in Agda. The presentation in this section borrows

[2] The Y combinator is a clever construction which allows for the definition of recursive functions in the lambda calculus. The μ term is a mere convenience in the untyped lambda calculus, but is essential when trying to add typing rules since Y is not well-typed.

heavily from [WKS22]. For concision, we may omit explicit citations to it. We will point out where we deviate from the structure laid out by the book.

We will now define our language in *intrinsic* form. This means that types and terms are defined at the same time, such that it is not possible to refer to a term without reference to its corresponding type. This ensures that expressions in our language are always well-typed, and we piggyback off the Agda type checker to ensure correctness. An alternative approach is to define the language in *extrinsic* form, where untyped terms are defined independently of their types. According to [MZ13], "in the intrinsic view, all expressions carry a type, and there is no need (or even sense) to consider the meaning of "untyped" expressions; while in the extrinsic view, every expression carries an independent meaning, and typing judgments serve to assert some property of that meaning." We will have to give an extrinsic view later on when we implement our own type checker.

This thesis is presented in *literate* style, meaning that code is interleaved with the text and could be executed were the surrounding text to be removed.[3] This means that the reader can be assured that all code has passed Agda's type checker.

First, we define some preliminaries including importing modules from the standard library and some mixfix operators. Agda allows the usage of *mixfix operators*, where arguments can appear before, after, or in the middle of operators. This enables us to write code that looks closer to the notation we're used to from mathematics, but can occasionally be quite confusing to read. We also make heavy use of Unicode symbols, which again brings us closer to mathematical convention but may look jarring to readers used to more conventional programming languages.

```
module thesis.src.Intrinsic where

open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _<_; _≤?_; z≤n; s≤s)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Decidable using (True; toWitness)

infix 2 _⟶_

infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infixr 7 _⇒_
```

[3]Strictly speaking the code is not fully executable, since some uninteresting parts have been omitted in order to save space.

```
infix 5 ƛ_
infix 5 μ_
infixl 7 _·_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_
```

Next, it is time to define the types we will use in this language. We have a total of four types: natural numbers, functions, products, and coproducts. Collectively they can describe every possible term in our language. They are defined using an inductive type definition, allowing them to be arbitrarily nested. For example, we could write `N × N → N ⊎ N` to describe the type of functions which take a pair of natural numbers and return an injection of a natural number.

```
data Type : Set where
  `ℕ : Type
  _⇒_ : Type → Type → Type
  _`×_ : Type → Type → Type
  _`⊎_ : Type → Type → Type
```

2.3 Contexts

Contexts will be of key importance when working out our typing rules. Contexts should be a familiar concept from logic, but here we explain briefly how we use them for typing rules. A context is an ordered list of assumptions, in our case typing judgments of the form $x : A$ stating that the term x has type A. We stack these judgments in a context, which we will usually call Γ, in lists such as $\emptyset, x : A, y : B, \ldots$. The comma operator is a left-associative operator and extends contexts with new judgments, always starting with the empty context $\emptyset$.

The context represents assumptions we make during our derivation process. In our case, the assumptions are term judgments. When we introduce a variable, it refers to a assumption in the context. Abstractions and case terms discharge assumptions, removing them from the context. A full program ready to be evaluated has an empty context, indicating that there are no *free variables*, variables that haven't been bound.

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

[WKS22] initially uses named terms in the context. We will avoid this, since it leads to a longer, less compact presentation, and forces us to deal with the problem of non-unique names. Instead we will use de Bruijn indices.

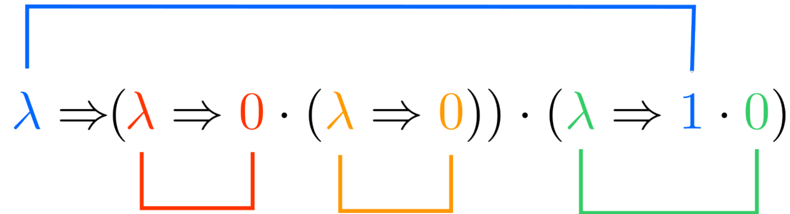

Figure 2.3: Example of de Bruijn indices mapped to lambda abstractions, inspired by [Cha09]

Figure 2.3 illustrates how we can use numbers to label variables in a context relative to the location of terms. When working in the innermost lambda abstraction, the number 0 refers to the most-recently introduced variable, or the top-most assumption on the context stack. The number 1 refers to the second-to-last assumption in the context, and so on. These numbers are called de Bruijn indices.

We will also need to define a lookup judgment $\Gamma \ni x$ describing the location of a term x in context Γ. We use two constructors `Z` and `S`, analogous to our earlier definition of the natural numbers. `Z` corresponds to the de Bruijn index 0, and successive uses of `S` give us subsequent indices.

```
data _∋_ : Context → Type → Set where
  Z : ∀ {Γ A}
    → Γ , A ∋ A
  S_ : ∀ {Γ A B}
    → Γ ∋ A
    → Γ , B ∋ A
```

Here we see our first use of a dependent type. The type for lookup judgments depends on a context and a type, and each constructor universally quantifies over contexts and types.

2.4 Typing rules

There is a standard approach for typing the lambda calculus known as the simply typed lambda calculus. Its typing rules are given below in Figure 2.4, along with typing rules for our language extensions.

We will now replicate these typing rules in Agda using a type representing judgments of the form $\Gamma \vdash A$. The general structure of this comes from [WKS22], but we've extended it for our new terms.

$$\frac{x : \Gamma \ni A}{\Gamma \vdash x : A}\text{ T-VAR} \qquad \frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x \Rightarrow N : A \to B}\text{ T-}\lambda$$

$$\frac{\Gamma \vdash L : A \Rightarrow B \quad \Gamma \vdash M : A}{\Gamma \vdash L \cdot M : B}\text{ T-}\cdot$$

$$\frac{}{\Gamma \vdash \text{zero} : \mathbb{N}}\text{ T-ZERO} \qquad \frac{\Gamma \vdash N : \mathbb{N}}{\Gamma \vdash \text{suc}\, N : \mathbb{N}}\text{ T-SUC}$$

$$\frac{\Gamma \vdash L : \mathbb{N} \quad \Gamma \vdash M : C \quad \Gamma, \mathbb{N} \vdash N : C}{\Gamma \vdash \text{case}\mathbb{N}\, L\, [\text{zero} \Rightarrow M \mid \text{suc}\, N] : C}\text{ T-CASE}\mathbb{N}$$

$$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \mu x \Rightarrow N : A \to B}\text{ T-}\mu$$

$$\frac{\Gamma \vdash L : A \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash \text{inj}_1 L : A \uplus B}\text{ T-INJ}_1 \qquad \frac{\Gamma \vdash R : B \quad \Gamma \vdash A\ \text{type}}{\Gamma \vdash \text{inj}_2 R : A \uplus B}\text{ T-INJ}_2$$

$$\frac{\Gamma \vdash L : A \uplus B \quad \Gamma, A \vdash M : C \quad \Gamma, B \vdash N : C}{\Gamma \vdash \text{case}\uplus\ L\, [\text{inj}_1\, M \mid \text{inj}_2\, N] : C}\text{ T-CASE}\uplus$$

$$\frac{\Gamma \vdash L : A \quad \Gamma \vdash R : B}{\Gamma \vdash \langle L, R \rangle : A \times B}\text{ T-}\langle\,,\rangle$$

$$\frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \text{proj}_1 L : A}\text{ T-PROJ}_1 \qquad \frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \text{proj}_2 R : B}\text{ T-PROJ}_2$$

Figure 2.4: Typing judgments for our language

```
data _⊢_ : Context → Type → Set where
  `_ : ∀ {Γ A}
    → Γ ∋ A
    → Γ ⊢ A
  ƛ_ : ∀ {Γ A B}
    → Γ , A ⊢ B
    → Γ ⊢ A ⇒ B
  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
    → Γ ⊢ B
  `zero : ∀ {Γ} → Γ ⊢ `ℕ
  `suc_ : ∀ {Γ}
    → Γ ⊢ `ℕ
    → Γ ⊢ `ℕ
  caseℕ : ∀ {Γ A}
    → Γ ⊢ `ℕ
```

```
  → Γ ⊢ A
  → Γ , `ℕ ⊢ A
  → Γ ⊢ A
μ_ : ∀ {Γ A}
  → Γ , A ⊢ A
  → Γ ⊢ A
`inj₁ : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ ⊢ A `⊎ B
`inj₂ : ∀ {Γ A B}
  → Γ ⊢ B
  → Γ ⊢ A `⊎ B
case⊎ : ∀ {Γ A B C}
  → Γ ⊢ A `⊎ B
  → Γ , A ⊢ C
  → Γ , B ⊢ C
  → Γ ⊢ C
`⟨_,_⟩ : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ ⊢ B
  → Γ ⊢ A `× B
`proj₁ : ∀ {Γ A B}
  → Γ ⊢ A `× B
  → Γ ⊢ A
`proj₂ : ∀ {Γ A B}
  → Γ ⊢ A `× B
  → Γ ⊢ B
```

It is remarkable how close the Agda code looks to figure 2.4. The main difference is that we avoid naming terms, and that we choose slightly different names. Let's look at some examples of how to construct programs using these definitions.

Example 2.4.1. A context with two assumptions, and a term referring to the second assumption in the context.

```
_ : ∅ , `ℕ , `ℕ `× `ℕ ⊢ `ℕ
_ = ` (S Z)
```

Example 2.4.2. A function that takes a natural number, and returns a pair consisting of the number and its successor.

```
_ : ∅ ⊢ `ℕ ⇒ `ℕ `× `ℕ
_ = ƛ `⟨ ` Z , `suc ` Z ⟩
```

Example 2.4.3. A function that takes an element of a coproduct of natural numbers, and returns either the first or the second number, whichever is present.

```
_ : ∅ ⊢ `ℕ `⊎ `ℕ ⇒ `ℕ
_ = ƛ case⊎ (` Z) (` Z) (` Z)
```

We now also define a few helper functions to allow us to conveniently refer to variables using de Bruijn indices in our code using the # operator. These are identical to the definitions in [WKS22]. They use a clever construction to ensure that de Bruijn indices are not larger than allowed, but we will not go into the details of how that works.

```
length : Context → ℕ
length ∅ = zero
length (Γ , _) = suc (length Γ)

lookup : {Γ : Context} → {n : ℕ} → (p : n < length Γ) → Type
lookup {(_ , A)} {zero} (s≤s z≤n) = A
lookup {(Γ , _)} {(suc n)} (s≤s p) = lookup p

count : ∀ {Γ} → {n : ℕ} → (p : n < length Γ) → Γ ∋ lookup p
count {_ , _} {zero} (s≤s z≤n) = Z
count {Γ , _} {(suc n)} (s≤s p) = S (count p)

#_ : ∀ {Γ}
  → (n : ℕ)
  → {n∈Γ : True (suc n ≤? length Γ)}
  → Γ ⊢ lookup (toWitness n∈Γ)
#_ n {n∈Γ} = ` count (toWitness n∈Γ)
```

2.5 Substitution

In order to reduce expressions in our language, we need a way of substituting variables when applying functions. Abstraction will cause our context to change, and therefore we need to remap the de Bruijn indices to reflect changes in the context. We define an operator `_[_]` which substitutes a single variable using a more powerful `subst` function, which can handle arbitrary simultaneous substitutions.

This section is again largely similar to what is already described in [WKS22, Properties], and it is described in much greater detail there. Extending it to support our new language constructs is relatively simple, and mainly requires making note of which constructs introduce new bound variables. Both our case rules require one or more new

variables to be bound, and therefore need to be treated specially using context extensions.

```
ext : ∀ {Γ Δ}
  → (∀ {A} →         Γ ∋ A → Δ ∋ A)
  → (∀ {A B} → Γ , B ∋ A → Δ , B ∋ A)
ext ρ Z = Z
ext ρ (S x) = S (ρ x)

rename : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
  → ∀ {B} → Γ ⊢ B → Δ ⊢ B
rename ρ (` x) = ` (ρ x)
rename ρ (ƛ N) = ƛ (rename (ext ρ) N)
rename ρ (L · M) =  (rename ρ L) · (rename ρ M)
rename ρ (`zero) = `zero
rename ρ (`suc M) = `suc (rename ρ M)
rename ρ (caseℕ L M N) = caseℕ (rename ρ L) (rename ρ M) (rename (ext ρ) N)
rename ρ (μ N) = μ (rename (ext ρ) N)
rename ρ (`inj₁ L) = `inj₁ (rename ρ L)
rename ρ (`inj₂ R) = `inj₂ (rename ρ R)
rename ρ (`proj₁ P) = `proj₁ (rename ρ P)
rename ρ (`proj₂ P) = `proj₂ (rename ρ P)
rename ρ (`⟨ L , R ⟩) = `⟨ (rename ρ L) , (rename ρ R) ⟩
rename ρ (case⊎ L M N) = case⊎ (rename ρ L) (rename (ext ρ) M) (rename (ext ρ) N)

exts : ∀ {Γ Δ}
  → (∀ {A} →         Γ ∋ A → Δ ⊢ A)
  → (∀ {A B} → Γ , B ∋ A → Δ , B ⊢ A)
exts σ Z = ` Z
exts σ (S x) = rename S_ (σ x)

subst : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ⊢ A)
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
subst σ (` k) = σ k
subst σ (ƛ N) = ƛ (subst (exts σ) N)
subst σ (L · M) =  (subst σ L) · (subst σ M)
subst σ (`zero) = `zero
subst σ (`suc M) =  `suc (subst σ M)
subst σ (caseℕ L M N) = caseℕ (subst σ L) (subst σ M) (subst (exts σ) N)
```

```
subst σ (μ N) = μ (subst (exts σ) N)
subst σ (`inj₁ L) = `inj₁ (subst σ L)
subst σ (`inj₂ R) = `inj₂ (subst σ R)
subst σ (`proj₁ P) = `proj₁ (subst σ P)
subst σ (`proj₂ P) = `proj₂ (subst σ P)
subst σ (`⟨ L , R ⟩) = `⟨ (subst σ L) , (subst σ R) ⟩
subst σ (case⊎ L M N) = case⊎ (subst σ L) (subst (exts σ) M) (subst (exts σ) N)

_[_] : ∀ {Γ A B}
  → Γ , B ⊢ A
  → Γ ⊢ B
  → Γ ⊢ A
_[_] {Γ} {A} {B} N M = subst {Γ , B} {Γ} σ {A} N
  where
  σ : ∀ {A} → Γ , B ∋ A → Γ ⊢ A
  σ Z = M
  σ (S x) = ` x
```

2.6 Reduction rules

We will now get into how to actually evaluate expressions in our language. To do this, we will need a notion of values to determine when computation should stop. Every expression must either be a value, or there are further reduction rules that can be applied to it. This is a property known as *progress*, which we will cover later on. As such, values define the necessary conditions for our reduction to stop. We define a number of our constructs to be values below.

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where
  V-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B}
    → Value (ƛ N)
  V-zero : ∀ {Γ}
    → Value (`zero {Γ})
  V-suc : ∀ {Γ} {V : Γ ⊢ `ℕ}
    → Value V
    → Value (`suc V)
  V-⟨,⟩ : ∀ {Γ A B} {L : Γ ⊢ A} {R : Γ ⊢ B}
    → Value L
    → Value R
```

```
    → Value (`⟨ L , R ⟩)
  V-inj₁ : ∀ {Γ A B} {VL : Γ ⊢ A}
    → Value VL
    → Value (`inj₁ {_} {_} {B} VL)
  V-inj₂ : ∀ {Γ A B} {VR : Γ ⊢ B}
    → Value VR
    → Value (`inj₂ {_} {A} {_} VR)
```

Notably, in the injective value rules we must explicitly specify the type of the value that is not provided to the constructor. Otherwise, there is no way for Agda to know the full type of the value.

The reduction rules below correspond to and extend the reduction rules given in our presentation of the untyped lambda calculus. In general, β-rules tell us how to reduce a term, and ξ-rules tell us where to perform the reduction. A few of the rules require evidence that a value has been reached before they can be applied, which eliminates ambiguity in reduction order. [WKS22] also provides proof of *confluence*, meaning that reduction is independent of the order in which rules are applied. We don't need to prove that here, since there is only one legal reduction order.

```
data _—→_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where
  ξ-·₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A ⇒ B} {M : Γ ⊢ A}
    → L —→ L′
    → L · M —→ L′ · M
  ξ-·₂ : ∀ {Γ A B} {V : Γ ⊢ A ⇒ B} {M M′ : Γ ⊢ A}
    → Value V
    → M —→ M′
    → V · M —→ V · M′
  β-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B} {W : Γ ⊢ A}
    → Value W
    → (ƛ N) · W —→ N [ W ]
  ξ-suc : ∀ {Γ} {M M′ : Γ ⊢ `ℕ}
    → M —→ M′
    → `suc M —→ `suc M′
  ξ-inj₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A}
    → L —→ L′
    → `inj₁ {_} {_} {B} L —→ `inj₁ {_} {_} {B} L′
  ξ-inj₂ : ∀ {Γ A B} {R R′ : Γ ⊢ B}
    → R —→ R′
    → `inj₂ {_} {A} R —→ `inj₂ {_} {A} R′
  ξ-proj₁ : ∀ {Γ A B} {P P′ : Γ ⊢ A `× B}
    → P —→ P′
```

```
  → `proj₁ P —→ `proj₁ P′
ξ-proj₂ : ∀ {Γ A B} {P P′ : Γ ⊢ A `× B}
  → P —→ P′
  → `proj₂ P —→ `proj₂ P′
ξ-⟨,⟩₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A} {R : Γ ⊢ B}
  → L —→ L′
  → `⟨ L , R ⟩ —→ `⟨ L′ , R ⟩
ξ-⟨,⟩₂ : ∀ {Γ A B} {VL : Γ ⊢ A} {R R′ : Γ ⊢ B}
  → Value VL
  → R —→ R′
  → `⟨ VL , R ⟩ —→ `⟨ VL , R′ ⟩
ξ-caseℕ : ∀ {Γ A} {L L′ : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → L —→ L′
  → caseℕ L M N —→ caseℕ L′ M N
ξ-case⊎ : ∀ {Γ A B C} {L L′ : Γ ⊢ A `⊎ B} {M : Γ , A ⊢ C} {N : Γ , B ⊢ C}
  → L —→ L′
  → case⊎ L M N —→ case⊎ L′ M N
β-zero : ∀ {Γ A} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → caseℕ `zero M N —→ M
β-suc : ∀ {Γ A} {V : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → Value V
  → caseℕ (`suc V) M N —→ N [ V ]
β-inj₁ : ∀ {Γ A B C} {VL : Γ ⊢ A} {M : Γ , A ⊢ C} {N : Γ , B ⊢ C}
  → Value VL
  → case⊎ (`inj₁ VL) M N —→ M [ VL ]
β-inj₂ : ∀ {Γ A B C} {VR : Γ ⊢ B} {M : Γ , A ⊢ C} {N : Γ , B ⊢ C}
  → Value VR
  → case⊎ (`inj₂ VR) M N —→ N [ VR ]
β-proj₁ : ∀ {Γ A B} {VL : Γ ⊢ A} {P : Γ ⊢ A `× B}
  → Value VL
  → (`proj₁ P) —→ VL
β-proj₂ : ∀ {Γ A B} {VR : Γ ⊢ B} {P : Γ ⊢ A `× B}
  → Value VR
  → (`proj₂ P) —→ VR
β-μ : ∀ {Γ A} {N : Γ , A ⊢ A}
  → μ N —→ N [ μ N ]
```

With these rules, we can evaluate any expression in our language in a similar fashion as before, albeit with far more rules to consider this time. Luckily, we don't have to do this manually.

2.7 Progress and evaluation

We stated earlier that terms must either be values or they can be further reduced. This is a property known as *progress*. We will now prove that our language satisfies this property using the following definition of progress, which states that we can either take another reduction step or we have a value. Notably, `Progress` is only defined for empty contexts as we must have discharged our assumptions before evaluation.

```
data Progress {A} (M : ∅ ⊢ A) : Set where
  step : ∀ {N : ∅ ⊢ A}
    → M ⟶ N
    → Progress M
  done :
    Value M
    → Progress M
```

There is also another important property called *preservation*, meaning that all reductions preserve the types that held before reduction was applied. This is a property that we have in fact already proven by choosing to represent our language in intrinsic form. Since Agda has type checked our code, and our definitions state that types are preserved, we know that the property holds. If we had chosen an extrinsic presentation, we would have to manually prove this property.

To prove progress, we define a function `progress` that shows how to take an intrinsic term and evaluate it. It pattern matches on all possible cases of our terms, and defines how to take reduction steps as well as when we have reached a value. It is again based on [WKS22, Properties], but extended for our new terms.

```
progress : ∀ {A} → (M : ∅ ⊢ A) → Progress M
progress (` ())
progress (ƛ N) = done V-ƛ
progress (L · M) with progress L
... | step L⟶L′ = step (ξ-·₁ L⟶L′)
... | done V-ƛ with progress M
... | step M⟶M′ = step (ξ-·₂ V-ƛ M⟶M′)
... | done VM = step (β-ƛ VM)
progress (`zero) = done V-zero
progress (`suc M) with progress M
... | step M⟶M′ = step (ξ-suc M⟶M′)
... | done VM = done (V-suc VM)
progress (caseℕ L M N) with progress L
... | step L⟶L′ = step (ξ-caseℕ L⟶L′)
```

```
... | done V-zero = step (β-zero)
... | done (V-suc VL) = step (β-suc VL)
progress (μ N) = step (β-μ)
progress (`inj₁ L) with progress L
... | step L⟶L′ = step (ξ-inj₁ L⟶L′)
... | done VL = done (V-inj₁ VL)
progress (`inj₂ L) with progress L
... | step R⟶R′ = step (ξ-inj₂ R⟶R′)
... | done VR = done (V-inj₂ VR)
progress (case⊎ L M N) with progress L
... | step L⟶L′ = step (ξ-case⊎ L⟶L′)
... | done (V-inj₁ VL) = step (β-inj₁ VL)
... | done (V-inj₂ VR) = step (β-inj₂ VR)
progress (`proj₁ P) with progress P
... | step P⟶P′ = step (ξ-proj₁ P⟶P′)
... | done (V-⟨,⟩ L R) = step (β-proj₁ L)
progress (`proj₂ P) with progress P
... | step P⟶P′ = step (ξ-proj₂ P⟶P′)
... | done (V-⟨,⟩ L R) = step (β-proj₂ R)
progress (`⟨ L , R ⟩) with progress L | progress R
... | step L⟶L′ | step R⟶R′ = step (ξ-⟨,⟩₁ L⟶L′)
... | done VL | step R⟶R′ = step (ξ-⟨,⟩₂ VL R⟶R′)
... | step L⟶L′ | done y = step (ξ-⟨,⟩₁ L⟶L′)
... | done VL | done VR = done (V-⟨,⟩ VL VR)
```

The proof of progress doubles as a method of evaluating terms in our language. Since Agda is a total programming language, meaning that all programs must terminate, and the lambda calculus is not, we actually cannot give a general method for running programs to completion without turning off Agda's termination checker. [WKS22] offers an approach where they specify a maximum limit to the number of reduction steps we are allowed to take, ensuring totality, but we will not repeat it here.

We instead give a minimal example showing that our reduction rules work through one manual application of progress.

Example 2.7.1. Evaluating a trivial program.

```
program : ∅ ⊢ `ℕ
program = (ƛ `suc # 0) · `zero

eval : program ⟶ `suc `zero
eval with progress program
... | step x @ (β-ƛ V-zero) = x
```

With this initial description of the language in place, we are ready to move on to type checking. We will come back to this intrinsic presentation when proving the soundness and completeness of our type checking algorithm.

Chapter 3

Bidirectional typechecking

Bidirectional type checking is a popular algorithm for verifying the type-correctness of programs. According to [DK21, section 1], “the first commonly cited paper on bidirectional typing appeared in 1997 but mentioned that the idea was known as ‘folklore’.” The idea may have been known already in 1988, when John Reynolds is said to have discussed it with Benjamin Pierce and David Turner. [DK21, section 10]

It is a *syntax-directed* approach, meaning that we only need to recurse on the syntax of the language in order to check types. This is a desireable property which makes our implementation read well, since we only need to follow the recursion of cases to know how a type is dealt with.

Given a typing judgment like $\Gamma \vdash e : A$, we can implementing type checking by considering the three meta-variables Γ, e, A to be inputs. If we on the other hand let A be an output, we are implementing type inference or type synthesis. These statuses of meta-variables are known as *modes*. [DK21, section 1]

In bidirectional type checking we alternate between these two modes, that is, *synthesizing* (inferring) and *inheriting* (checking) types for expressions. For example, given the term `zero` we can easily synthesize the type ℕ. However, we cannot always synthesize a type. For example, what is the type of the identity function? It could potentially synthesize to many different function signatures. In this case, we instead have to provide an annotation which we then inherit. This determines which parts of our language need to be explicitly annotated.

Synthesizing is in general harder than inheriting. Damas-Milner type inference, a common algorithm capable of fully synthesizing types for programs, is difficult to extend to more advanced type systems. By alternating between synthesizing and inheriting, we are able to support more complex type systems than Damas-Milner while avoiding fully explicit annotations. [DK21, section 1]

We will now begin mechanizing bidirectional types. This presentation is again heavily based on [WKS22]. We will omit some details for brevity, as they are relatively uninteresting or have already been covered earlier.

```
module thesis.src.Bidi where

-- Imports, fixity declarations, and context types omitted
```

3.1 Terms

Of some interest is the representation we use for term names. Since we are writing a type checker, we need the terms of our language to be untyped and cannot reuse the intrinsic presentation from before. [WKS22] uses string names for terms, but this would become a problem for us later when trying to prove annotatability. We instead use de Bruijn indices again, indexing terms by natural numbers. Many of the proofs differ from [WKS22] due to this representation, where de Bruijn indices are not used in the extrinsic setting.

Here, we provide a way to convert from variable lookups to the numbers we use in the syntax of the language. We create a `Var` record to link between numbered indices and lookups by requiring a proof of their equality.

```
Index : Set
Index = ℕ

index : ∀ {Γ A} → Γ ∋ A → ℕ
index Z = 0
index (S x) = suc (index x)

record Var (Γ : Context) (A : Type) (x : ℕ) : Set where
  constructor [_,_]Var
  field
    ∋x : Γ ∋ A
    idx≡ : index ∋x ≡ x
```

[WKS22] suggests to divide the terms of the language into positive and negative terms.[1] Positive terms are synthesized, and negative terms are inherited. By carefully considering which type a term belongs to, we can minimize the amount of annotations needed when writing expressions in the language. But we also need to consider that certain terms are difficult to synthesize without more advanced work in the type checker.

As an example, if we try to synthesize the type of a left injection to a coproduct, we find that the type of the right injection is unknown. Similarly, a function like `ƛ ` zero` cannot be synthesized without some notion of polymorphism since we cannot know the

[1] Programming language theory also has the concepts of *positive and negative types*, which are unrelated to the use of “positive” and “negative” here.

type of the input argument. For this reason, abstractions and coproducts are inherited and not synthesized.

A few guidelines are given in [WKS22] for deciding which type a term belongs to. We mostly ignore those rules and try to synthesize as much as possible. This leads to slightly more complicated proofs in some cases.

In addition to the terms from before, we also add two special terms `_↑` and `_↓_` which are annotations used to convert between synthesized and inherited terms. As an example of why we need them, our previous example `ƛ ` + "`" + `zero` is in fact invalid without the `_↑` annotation on the zero, since it is a synthesized term and the body of an abstraction is expected to be inherited.

Since the terms refer to each other, they are mutually inductively defined.

```
data Term⁺ : Set
data Term⁻ : Set

data Term⁺ where
  #_                          : Index → Term⁺
  _·_                         : Term⁺ → Term⁻ → Term⁺
  `proj₁                      : Term⁺ → Term⁺
  `proj₂                      : Term⁺ → Term⁺
  `⟨_,_⟩                      : Term⁺ → Term⁺ → Term⁺
  `zero                       : Term⁺
  `suc_                       : Term⁺ → Term⁺
  `caseℕ_[zero⇒_|suc⇒_]       : Term⁺ → Term⁺ → Term⁻ → Term⁺
  `case⊎_[inj₁⇒_|inj₂⇒_]      : Term⁺ → Term⁺ → Term⁻ → Term⁺
  _↓_                         : Term⁻ → Type → Term⁺

data Term⁻ where
  ƛ_                          : Term⁻ → Term⁻
  μ_                          : Term⁻ → Term⁻
  `inj₁                       : Term⁺ → Term⁻
  `inj₂                       : Term⁺ → Term⁻
  _↑                          : Term⁺ → Term⁻
```

3.2 Typing rules

We now define the inductive definitions of when terms synthesize and inherit, in preparation for defining the algorithm itself. For `Term⁺` terms, we define synthesizing judgments of the form $\Gamma \vdash M \uparrow A$ to state that M synthesizes to a type A. Conversely, for

Term⁻ terms, we define inheriting judgments of the form $\Gamma \vdash M \downarrow A$ to state that M inherits or checks as type A.

```
data _⊢_↑_ : Context → Term⁺ → Type → Set
data _⊢_↓_ : Context → Term⁻ → Type → Set

data _⊢_↑_ where
  ⊢` : ∀ {Γ A x}
    → (Var Γ A x)
    → Γ ⊢ # x ↑ A
  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ↑ A ⇒ B
    → Γ ⊢ M ↓ A
    → Γ ⊢ L · M ↑ B
  ⊢↓ : ∀ {Γ M A}
    → Γ ⊢ M ↓ A
    → Γ ⊢ (M ↓ A) ↑ A
  ⊢proj₁ : ∀ {Γ P A B}
    → Γ ⊢ P ↑ (A `× B)
    → Γ ⊢ (`proj₁ P) ↑ A
  ⊢proj₂ : ∀ {Γ P A B}
    → Γ ⊢ P ↑ (A `× B)
    → Γ ⊢ (`proj₂ P) ↑ B
  ⊢⟨,⟩ : ∀ {Γ L R A B}
    → Γ ⊢ L ↑ A
    → Γ ⊢ R ↑ B
    → Γ ⊢ `⟨ L , R ⟩ ↑ (A `× B)
  ⊢zero : ∀ {Γ} → Γ ⊢ `zero ↑ `ℕ
  ⊢suc : ∀ {Γ M}
    → Γ ⊢ M ↑ `ℕ
    → Γ ⊢ `suc M ↑ `ℕ
  ⊢caseℕ : ∀ {Γ L M N C}
    → Γ ⊢ L ↑ `ℕ
    → Γ ⊢ M ↑ C
    → Γ , `ℕ ⊢ N ↓ C
    → Γ ⊢ `caseℕ L [zero⇒ M |suc⇒ N ] ↑ C
  ⊢case⊎ : ∀ {Γ A B C L M N}
    → Γ ⊢ L ↑ A `⊎ B
    → Γ , A ⊢ M ↑ C
    → Γ , B ⊢ N ↓ C
```

```
    → Γ ⊢ `case⊎ L [inj₁⇒ M |inj₂⇒ N ] ↑ C

data _⊢_↓_ where
  ⊢ƛ : ∀ {Γ N A B}
    → Γ , A ⊢ N ↓ B
    → Γ ⊢ ƛ N ↓ A ⇒ B
  ⊢μ : ∀ {Γ N A}
    → Γ , A ⊢ N ↓ A
    → Γ ⊢ μ N ↓ A
  ⊢inj₁ : ∀ {Γ L A B}
    → Γ ⊢ L ↑ A
    → Γ ⊢ (`inj₁ L) ↓ A `⊎ B
  ⊢inj₂ : ∀ {Γ R A B}
    → Γ ⊢ R ↑ B
    → Γ ⊢ (`inj₂ R) ↓ A `⊎ B
  ⊢↑ : ∀ {Γ M A B}
    → Γ ⊢ M ↑ A
    → A ≡ B
    → Γ ⊢ (M ↑) ↓ B
```

Example 3.2.1. A synthesis judgment for a variable, using a context consisting of one assumption that there is a variable typed as a natural number.

```
_ : ∅ , `ℕ ⊢ # 0 ↑ `ℕ
_ = ⊢` [ Z , refl ]Var
```

Example 3.2.2. A more complicated example type checking the identity function for natural numbers. Here we need to use the special ↑ constructor in order to make the variable a checked term, as expected by the constructor of the lambda abstraction.

```
_ : ∅ ⊢ (ƛ (# zero) ↑) ↓ `ℕ ⇒ `ℕ
_ = ⊢ƛ (⊢↑ (⊢` [ Z , refl ]Var) refl)
```

3.3 Lemmas

Next we prove some lemmas that will be useful, starting off with a lemma about the decidability of type equality. This allows us to compare types for equality. Agda is smart enough to figure out most of the proofs on its own, and only needs a little bit of help for some of the types. We leave out the cases that are simply `no ()` on the right-hand side.

```
_≟Tp_ : (A B : Type) → Dec (A ≡ B)
`ℕ ≟Tp `ℕ                 = yes refl
(A ⇒ B) ≟Tp (A′ ⇒ B′)
  with A ≟Tp A′ | B ≟Tp B′
... | no A≢ | _           = no λ{refl → A≢ refl}
... | yes _ | no B≢       = no λ{refl → B≢ refl}
... | yes refl | yes refl = yes refl
(A `× B) ≟Tp (A′ `× B′)
  with A ≟Tp A′ | B ≟Tp B′
... | no A≢ | _           = no λ{refl → A≢ refl}
... | yes _ | no B≢       = no λ{refl → B≢ refl}
... | yes refl | yes refl = yes refl
(A `⊎ B) ≟Tp (A′ `⊎ B′)
  with A ≟Tp A′ | B ≟Tp B′
... | no A≢ | _           = no λ{refl → A≢ refl}
... | yes _ | no B≢       = no λ{refl → B≢ refl}
... | yes refl | yes refl = yes refl
```

We then define inversion and injectivity lemmas that allow us to deduce that if two terms are equal then their parts must also be equal, as well as lemmas showing that different types cannot be the same. These lemmas are obvious enough that Agda figures out proofs for them automatically. We leave out the proofs.

```
dom≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → A ≡ A′
rng≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → B ≡ B′
left×≡ : ∀ {A A′ B B′} → A `× B ≡ A′ `× B′ → A ≡ A′
right×≡ : ∀ {A A′ B B′} → A `× B ≡ A′ `× B′ → B ≡ B′
left⊎≡ : ∀ {A B A′ B′} → (A `⊎ B) ≡ (A′ `⊎ B′) → A ≡ A′
right⊎≡ : ∀ {A B A′ B′} → (A `⊎ B) ≡ (A′ `⊎ B′) → B ≡ B′
ℕ≢⇒ : ∀ {A B} → `ℕ ≢ A ⇒ B
ℕ≢× : ∀ {A B} → `ℕ ≢ A `× B
ℕ≢⊎ : ∀ {A B} → `ℕ ≢ A `⊎ B
×≢⇒ : ∀ {A B C D} → (A `× B) ≢ C ⇒ D
⊎≢⇒ : ∀ {A B C D} → (A `⊎ B) ≢ C ⇒ D
⊎≢× : ∀ {A B C D} → (A `⊎ B) ≢ (C `× D)
```

We also provide a lemma showing that two lookups with the same de Bruijn index must have the same type. This proof is slightly different from [WKS22] due to our use of de Bruijn indices.

```
uniq-∋ : ∀ {Γ A B}
  → (∋x : Γ ∋ A)
```

```
  → (∋x′ : Γ ∋ B)
  → (index ∋x ≡ index ∋x′)
  → A ≡ B
uniq-∋ Z Z p = refl
uniq-∋ (S ∋x) (S ∋x′) p = uniq-∋ ∋x ∋x′ (suc-injective p)
```

Next, if A and B are the same types and we have a judgment showing that M inherits A, M also inherits B. [WKS22] does not define this lemma, since they inherit case terms.

```
≡↓ : ∀ {Γ M A B}
  → A ≡ B
  → Γ ⊢ M ↓ A
  → Γ ⊢ M ↓ B
≡↓ A≡B ⊢M rewrite A≡B = ⊢M
```

The next lemma `uniq-↑`, the uniqueness of synthesis, is used quite extensively to prove contradictions when we have two proofs that term synthesizes to different types. If we hadn't carefully selected our synthesized terms earlier, we would run into some trouble here since not all of our types can be synthesized uniquely. For example, a term such as `` `inj₁ `zero `` could synthesize to multiple different coproducts differing in the second injection, as discussed previously.

```
uniq-↑ : ∀ {Γ M A B}
  → Γ ⊢ M ↑ A
  → Γ ⊢ M ↑ B
  → A ≡ B
uniq-↑ (⊢` [ ∋x , idx≡ ]Var) (⊢` [ ∋x′ , idx≡′ ]Var) = uniq-∋ ∋x ∋x′ (trans idx≡ (sym idx≡′))
uniq-↑ (⊢L · ⊢M) (⊢L′ · ⊢M′)         = rng≡ (uniq-↑ ⊢L ⊢L′)
uniq-↑ (⊢↓ ⊢M) (⊢↓ ⊢M′)              = refl
uniq-↑ (⊢proj₁ ⊢P) (⊢proj₁ ⊢P′)      = left×≡ (uniq-↑ ⊢P ⊢P′)
uniq-↑ (⊢proj₂ ⊢P) (⊢proj₂ ⊢P′)      = right×≡ (uniq-↑ ⊢P ⊢P′)
uniq-↑ (⊢⟨,⟩ ⊢L ⊢R) (⊢⟨,⟩ ⊢L′ ⊢R′)
  rewrite uniq-↑ ⊢L ⊢L′
  rewrite uniq-↑ ⊢R ⊢R′              = refl
uniq-↑ ⊢zero ⊢zero                   = refl
uniq-↑ (⊢suc ⊢N) (⊢suc ⊢N′)          = refl
uniq-↑ (⊢caseℕ ⊢L ⊢M ⊢N) (⊢caseℕ ⊢L′ ⊢M′ ⊢N′)
  rewrite uniq-↑ ⊢L ⊢L′              = uniq-↑ ⊢M ⊢M′
uniq-↑ (⊢case⊎ ⊢L ⊢M ⊢N) (⊢case⊎ ⊢L′ ⊢M′ ⊢N′)
  rewrite left⊎≡ (uniq-↑ ⊢L ⊢L′) = uniq-↑ ⊢M ⊢M′
```

We will also need to show that contexts can be extended with new types using `ext∋`, and provide a `lookup` function to show that given an index we may be able to construct a

Var type if the index is not too large. We have again modified this to support de Bruijn indices.

```
ext∋ : ∀ {Γ B x}
  → ¬ (∃[ A ] Var Γ A x)
  → ¬ (∃[ A ] Var (Γ , B) A (suc x))
ext∋ ¬∃ ⟨ A , [ (S ∋x) , idx≡ ]Var ⟩ = ¬∃ ⟨ A , [ ∋x , suc-injective idx≡ ]Var ⟩

lookup : ∀ (Γ : Context) (x : Index)
  → Dec (∃[ A ] Var Γ A x )
lookup ∅ x = no (λ ())
lookup (Γ , B) zero = yes ⟨ B , [ Z , refl ]Var ⟩
lookup (Γ , _) (suc x) with lookup Γ x
... | no ¬∃ = no (ext∋ ¬∃)
... | yes ⟨ A , [ ∋x , idx≡ ]Var ⟩ = yes ⟨ A , [ (S ∋x) , cong suc idx≡ ]Var ⟩
```

The next lemmas are identical to [WKS22]. The first one shows that if we don't have a term of type A, a we cannot apply a function of type $A \Rightarrow B$. The second one shows that a term M cannot both synthesize as type A and have the synthesized term check as B, given that A and B are different types.

```
¬arg : ∀ {Γ A B L M}
  → Γ ⊢ L ↑ A ⇒ B
  → ¬ Γ ⊢ M ↓ A
  → ¬ (∃[ B′ ] Γ ⊢ L · M ↑ B′)
¬arg ⊢L ¬⊢M ⟨ B′ , ⊢L′ · ⊢M′ ⟩ rewrite dom≡ (uniq-↑ ⊢L ⊢L′) = ¬⊢M ⊢M′

¬switch : ∀ {Γ M A B}
  → Γ ⊢ M ↑ A
  → A ≢ B
  → ¬ Γ ⊢ (M ↑) ↓ B
¬switch ⊢M A≢B (⊢↑ ⊢M′ A′≡B) rewrite uniq-↑ ⊢M ⊢M′ = A≢B A′≡B
```

Finally, we provide two helper lemmas that will allow us to substitute equal types in the context of synthesizing and inheriting judgments. These are also additions we had to make in order to support synthesizing case terms.

```
⊢↑subst : ∀ {Γ M C A A′}
  → A ≡ A′
  → (Γ , A′ ⊢ M ↑ C)
  → (Γ , A ⊢ M ↑ C)
```

```
⊢↑subst refl Γ = Γ

⊢↓subst : ∀ {Γ M C A A′}
  → A ≡ A′
  → (Γ , A′ ⊢ M ↓ C)
  → (Γ , A ⊢ M ↓ C)
⊢↓subst refl Γ = Γ
```

3.4 Synthesizing and inheriting

With all of these lemmas in place, we are ready to get to the proofs of synthesis and inheritance. We show for any given positive term M whether it synthesizes or not, and for any negative term M and type A whether M inherits A. The `synthesize` function uses an existential type, which is the type that it has inferred.

```
synthesize : ∀ (Γ : Context) (M : Term⁺)
  → Dec (∃[ A ] Γ ⊢ M ↑ A )

inherit : ∀ (Γ : Context) (M : Term⁻) (A : Type)
  → Dec (Γ ⊢ M ↓ A)
```

The proofs of `synthesize` and `inherit` are mutually recursive and use all of the lemmas we defined earlier. We will mostly refrain from commenting on them except for when there is something particular to note. The general structure of the proof comes from [WKS22], but it has been enlarged to support more terms and tries to synthesize more.

```
synthesize Γ (# x) with lookup Γ x
... | no ¬∃          = no     λ{ ⟨ A , ⊢` ∋x ⟩ → ¬∃ ⟨ A , ∋x ⟩ }
... | yes ⟨ A , ∋x ⟩ = yes ⟨ A , ⊢` ∋x ⟩
synthesize Γ (L · M) with synthesize Γ L
... | no ¬∃          = no     λ{ ⟨ _ , ⊢L · _  ⟩ → ¬∃ ⟨ _ , ⊢L ⟩ }
... | yes ⟨ `ℕ ,     ⊢L ⟩ = no  λ{ ⟨ _ , ⊢L′ · _ ⟩ → ℕ≢⇒ (uniq-↑ ⊢L ⊢L′) }
... | yes ⟨ A `× B , ⊢L ⟩ = no λ{ ⟨ _ , ⊢L′ · _ ⟩ → ×≢⇒ ((uniq-↑ ⊢L ⊢L′)) }
... | yes ⟨ A `⊎ B , ⊢L ⟩ = no λ{ ⟨ _ , ⊢L′ · _ ⟩ → ⊎≢⇒ ((uniq-↑ ⊢L ⊢L′)) }
... | yes ⟨ A ⇒ B , ⊢L ⟩ with inherit Γ M A
... | no ¬⊢M         = no      (¬arg ⊢L ¬⊢M)
... | yes ⊢M         = yes ⟨ B , ⊢L · ⊢M ⟩
synthesize Γ (M ↓ A) with inherit Γ M A
```

```
... | no ¬∃ = no λ{ ⟨ _ , ⊢↓ ⊢M ⟩ → ¬∃ ⊢M }
... | yes ⊢M          = yes ⟨ A , ⊢↓ ⊢M ⟩
synthesize Γ (`proj₁ P) with synthesize Γ P
... | no ¬∃             = no      λ{ ⟨ A , ⊢proj₁ {_} {_} {_} {B} v ⟩ → ¬∃ ⟨ A `× B , v ⟩}
... | yes ⟨ `ℕ ,     ⊢P ⟩ = no   λ{ ⟨ _ , ⊢proj₁ ⊢P′ ⟩ → ℕ≢× (uniq-↑ ⊢P ⊢P′) }
... | yes ⟨ _ ⇒ _ , ⊢P ⟩ = no   λ{ ⟨ _ , ⊢proj₁ ⊢P′ ⟩ → ×≢⇒ (uniq-↑ ⊢P′ ⊢P) }
... | yes ⟨ _ `⊎ _ , ⊢P ⟩ = no λ{ ⟨ _ , ⊢proj₁ ⊢P′ ⟩ → ⊎≢× (uniq-↑ ⊢P ⊢P′) }
... | yes ⟨ A `× _ , ⊢P ⟩ = yes ⟨ A , ⊢proj₁ ⊢P ⟩
synthesize Γ (`proj₂ P) with synthesize Γ P
... | no ¬∃             = no      λ{ ⟨ B , ⊢proj₂ {_} {_} {A} {_} v ⟩ → ¬∃ ⟨ A `× B , v ⟩}
... | yes ⟨ `ℕ ,     ⊢P ⟩ = no   λ{ ⟨ _ , ⊢proj₂ ⊢P′ ⟩ → ℕ≢× (uniq-↑ ⊢P ⊢P′) }
... | yes ⟨ _ ⇒ _ , ⊢P ⟩ = no   λ{ ⟨ _ , ⊢proj₂ ⊢P′ ⟩ → ×≢⇒ (uniq-↑ ⊢P′ ⊢P) }
... | yes ⟨ _ `⊎ _ , ⊢P ⟩ = no λ{ ⟨ _ , ⊢proj₂ ⊢P′ ⟩ → ⊎≢× (uniq-↑ ⊢P ⊢P′) }
... | yes ⟨ _ `× B , ⊢P ⟩ = yes ⟨ B , ⊢proj₂ ⊢P ⟩
synthesize Γ (`⟨ L , R ⟩) with synthesize Γ L | synthesize Γ R
... | no ¬L          | no ¬R  = no λ{ ⟨ (_ `× B) , ⊢⟨,⟩ _ ⊢R ⟩ → ¬R ⟨ B , ⊢R ⟩ }
... | yes ¬L         | no ¬R  = no λ{ ⟨ (_ `× B) , ⊢⟨,⟩ _ ⊢R ⟩ → ¬R ⟨ B , ⊢R ⟩ }
... | no ¬L          | yes ¬R = no λ{ ⟨ (A `× _) , ⊢⟨,⟩ ⊢L _ ⟩ → ¬L ⟨ A , ⊢L ⟩ }
... | yes ⟨ A , ⊢L ⟩ | yes ⟨ B , ⊢R ⟩ = yes ⟨ A `× B , (⊢⟨,⟩ ⊢L ⊢R) ⟩
synthesize Γ `zero = yes ⟨ `ℕ , ⊢zero ⟩
synthesize Γ (`suc N) with synthesize Γ N
... | no ¬∃        = no λ{ ⟨ .`ℕ , ⊢suc ⊢N ⟩ → ¬∃ ⟨ `ℕ , ⊢N ⟩ }
... | yes ⟨ `ℕ    , ⊢N ⟩ = yes ⟨ `ℕ , (⊢suc ⊢N) ⟩
... | yes ⟨ _ ⇒ _ , ⊢N ⟩ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢⇒ ((uniq-↑ ⊢N′ ⊢N)) }
... | yes ⟨ _ `× _ , ⊢N ⟩ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢× ((uniq-↑ ⊢N′ ⊢N)) }
... | yes ⟨ _ `⊎ _ , ⊢N ⟩ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢⊎ ((uniq-↑ ⊢N′ ⊢N)) }
```

Case constructors synthesize the first branch and check the second branch against the first branch.

```
synthesize Γ `caseℕ L [zero⇒ M |suc⇒ N ] with synthesize Γ L
... | no ¬∃                = no λ{ ⟨ _ , (⊢caseℕ ⊢L _ _) ⟩ → ¬∃ ⟨ `ℕ , ⊢L ⟩ }
... | yes ⟨ _ ⇒ _ , ⊢L ⟩ = no λ{ ⟨ _ , ⊢caseℕ ⊢L′ _ _ ⟩ → ℕ≢⇒ ((uniq-↑ ⊢L′ ⊢L))}
... | yes ⟨ _ `× _ , ⊢L ⟩ = no λ{ ⟨ _ , ⊢caseℕ ⊢L′ _ _ ⟩ → ℕ≢× ((uniq-↑ ⊢L′ ⊢L))}
... | yes ⟨ _ `⊎ _ , ⊢L ⟩ = no λ{ ⟨ _ , ⊢caseℕ ⊢L′ _ _ ⟩ → ℕ≢⊎ ((uniq-↑ ⊢L′ ⊢L))}
... | yes ⟨ `ℕ , ⊢L ⟩ with synthesize Γ M
... | no ¬∃ = no λ{ ⟨ C , ⊢caseℕ _ ⊢M _ ⟩ → ¬∃ ⟨ C , ⊢M ⟩ }
... | yes ⟨ C , ⊢M ⟩ with inherit (Γ , `ℕ) N C
... | no ¬∃ = no λ{ ⟨ _ , ⊢caseℕ _ ⊢M′ ⊢N ⟩ → ¬∃ (≡↓ (uniq-↑ ⊢M′ ⊢M) ⊢N) }
... | yes ⊢N = yes ⟨ _ , ⊢caseℕ ⊢L ⊢M ⊢N ⟩
synthesize Γ `case⊎ L [inj₁⇒ M |inj₂⇒ N ] with synthesize Γ L
```

```
... | no ¬∃              = no λ{ ⟨ _ , (⊢case⊎ ⊢L _ _) ⟩ → ¬∃ ⟨ _ , ⊢L ⟩ }
... | yes ⟨ _ ⇒ _ , ⊢L ⟩ = no λ{ ⟨ _ , ⊢case⊎ ⊢L′ _ _ ⟩ → ⊎≢⇒ ((uniq-↑ ⊢L′ ⊢L))}
... | yes ⟨ _ `× _ , ⊢L ⟩ = no λ{ ⟨ _ , ⊢case⊎ ⊢L′ _ _ ⟩ → ⊎≢× ((uniq-↑ ⊢L′ ⊢L))}
... | yes ⟨ `ℕ , ⊢L ⟩     = no λ{ ⟨ _ , ⊢case⊎ ⊢L′ _ _ ⟩ → ℕ≢⊎ ((uniq-↑ ⊢L ⊢L′))}
... | yes ⟨ A `⊎ B , ⊢L ⟩ with synthesize (Γ , A) M
... | no ¬∃ = no λ{ ⟨ C , ⊢case⊎ ⊢L′ ⊢M _ ⟩ → ¬∃ ⟨ C , ⊢↑subst (left⊎≡ (uniq-↑ ⊢L ⊢L′)) ⊢M ⟩ }
... | yes ⟨ C , ⊢M ⟩ with inherit (Γ , B) N C
... | no ¬∃ = no λ{ ⟨ C′ , ⊢case⊎ ⊢L′ ⊢M′ ⊢N ⟩ → ¬∃ (
        ⊢↓subst
          (right⊎≡ (uniq-↑ ⊢L ⊢L′))
          (≡↓
            (uniq-↑ (⊢↑subst (left⊎≡ (uniq-↑ ⊢L ⊢L′)) ⊢M′) ⊢M)
            ⊢N))}
... | yes ⊢N = yes ⟨ C , ⊢case⊎ ⊢L ⊢M ⊢N ⟩

inherit Γ (ƛ N) `ℕ = no (λ ())
inherit Γ (ƛ N) (A `× B) = no (λ ())
inherit Γ (ƛ N) (A ⇒ B) with inherit (Γ , A) N B
... | no ¬∃ = no (λ{ (⊢ƛ ⊢N) → ¬∃ ⊢N })
... | yes ⊢N = yes (⊢ƛ ⊢N)
inherit Γ (μ N) A with inherit (Γ , A) N A
... | no ¬∃ = no (λ{ (⊢μ ⊢N) → ¬∃ ⊢N })
... | yes ⊢N = yes (⊢μ ⊢N)
inherit Γ (M ↑) B with synthesize Γ M
... | no ¬∃ = no (λ{ (⊢↑ ⊢M _) → ¬∃ ⟨ _ , ⊢M ⟩ })
... | yes ⟨ A , ⊢M ⟩ with A ≟Tp B
... | no   A≢B = no (¬switch ⊢M A≢B)
... | yes A≡B = yes (⊢↑ ⊢M A≡B)
inherit Γ (ƛ a) (b `⊎ b₁) = no λ()
```

Coproducts are inherited, and checking them is a fairly simple matter of comparing the types.

```
inherit Γ (`inj₁ a) `ℕ        = no λ()
inherit Γ (`inj₁ a) (x ⇒ x₁) = no λ()
inherit Γ (`inj₁ a) (x `× x₁) = no λ()
inherit Γ (`inj₁ L) (A `⊎ B) with synthesize Γ L
... | no ¬E = no (λ{ (⊢inj₁ L′) → ¬E ⟨ A , L′ ⟩ })
... | yes ⟨ A′ , ⊢A ⟩ with A ≟Tp A′
... | no A′≢A                  = no λ{ (⊢inj₁ ⊢A′) → A′≢A (uniq-↑ ⊢A′ ⊢A )}
... | yes A′≡A rewrite A′≡A = yes (⊢inj₁ ⊢A)
```

```
inherit Γ (`inj₂ a) `ℕ = no λ()
inherit Γ (`inj₂ a) (x ⇒ x₁) = no λ()
inherit Γ (`inj₂ a) (x `× x₁) = no λ()
inherit Γ (`inj₂ R) (A `⊎ B) with synthesize Γ R
... | no ¬E = no (λ{ (⊢inj₂ R′) → ¬E ⟨ B , R′ ⟩ })
... | yes ⟨ B′ , ⊢B ⟩ with B ≟Tp B′
... | no B′≢B                  = no λ{ (⊢inj₂ ⊢B′) → B′≢B (uniq-↑ ⊢B′ ⊢B )}
... | yes B′≡B rewrite B′≡B = yes (⊢inj₂ ⊢B)
```

We now give some examples demonstrating that the algorithm works.

Example 3.4.1. Synthesizing the type of the natural number 1.

```
_ : synthesize ∅ (`suc `zero) ≡ yes ⟨ `ℕ , ⊢suc ⊢zero ⟩
_ = refl
```

Example 3.4.2. A coproduct case expression, taking in an element of the type $(\mathbb{N} \times \mathbb{N}) \uplus \mathbb{N}$ and returning a natural number.

```
_ :   synthesize ∅
        (`case⊎ ((`inj₂ `zero) ↓ ((`ℕ `× `ℕ) `⊎ `ℕ))
          [inj₁⇒ `proj₁ (# 0)
          |inj₂⇒ (# 0) ↑ ])
    ≡ yes ⟨
      `ℕ ,
      ⊢case⊎
        (⊢↓ (⊢inj₂ ⊢zero))
        (⊢proj₁ (⊢` [ Z , refl ]Var))
        (⊢↑ (⊢` [ Z , refl ]Var) refl) ⟩
_ = refl
```

Example 3.4.3. A function which swaps the elements of a pair.

```
_ : inherit ∅
      (ƛ (`⟨ `proj₂ (# 0) , `proj₁ (# 0) ⟩ ↑)) (`ℕ `× `ℕ ⇒ `ℕ `× `ℕ)
    ≡ yes (⊢ƛ (⊢↑ (⊢⟨,⟩ (⊢proj₂ (⊢` [ Z , refl ]Var)) (⊢proj₁ (⊢` [ Z , refl ]Var))) refl))
_ = refl
```

3.5 Soundness

We now prove that given synthesizing and inheriting judgments from our language, we can map them to intrinsically typed judgments. This proves the soundness of our our typing judgments: that every program that type checks is well-typed. We make use of the fact that the intrinsic representation we described earlier is well-typed by definition. The proofs are relatively straightforward and translate between the different constructs in the most obvious way.

```
‖_‖Tp : Type → Intr.Type
‖ `ℕ ‖Tp     = Intr.`ℕ
‖ A ⇒ B ‖Tp = ‖ A ‖Tp Intr.⇒ ‖ B ‖Tp
‖ A `× B ‖Tp = ‖ A ‖Tp Intr.`× ‖ B ‖Tp
‖ A `⊎ B ‖Tp = ‖ A ‖Tp Intr.`⊎ ‖ B ‖Tp

‖_‖Cx : Context → Intr.Context
‖ ∅ ‖Cx = Intr.∅
‖ Γ , A ‖Cx = ‖ Γ ‖Cx Intr., ‖ A ‖Tp

‖_‖∋ : ∀ {Γ A} → Γ ∋ A → ‖ Γ ‖Cx Intr.∋ ‖ A ‖Tp
‖ Z ‖∋ = Intr.Z
‖ S ∋x ‖∋ = Intr.S ‖ ∋x ‖∋

‖_‖⁺ : ∀ {Γ M A} → Γ ⊢ M ↑ A → ‖ Γ ‖Cx Intr.⊢ ‖ A ‖Tp
‖_‖⁻ : ∀ {Γ M A} → Γ ⊢ M ↓ A → ‖ Γ ‖Cx Intr.⊢ ‖ A ‖Tp

‖ ⊢` [ ⊢x , _ ]Var ‖⁺ = Intr.` ‖ ⊢x ‖∋
‖ ⊢L · ⊢M ‖⁺       = ‖ ⊢L ‖⁺ Intr.· ‖ ⊢M ‖⁻
‖ ⊢↓ ⊢M ‖⁺         = ‖ ⊢M ‖⁻
‖ ⊢zero ‖⁺         = Intr.`zero
‖ ⊢suc ⊢M ‖⁺       = Intr.`suc ‖ ⊢M ‖⁺
‖ ⊢proj₁ ⊢P ‖⁺  = Intr.`proj₁ ‖ ⊢P ‖⁺
‖ ⊢proj₂ ⊢P ‖⁺  = Intr.`proj₂ ‖ ⊢P ‖⁺
‖ ⊢⟨,⟩ ⊢L ⊢R ‖⁺ = Intr.`⟨ ‖ ⊢L ‖⁺ , ‖ ⊢R ‖⁺ ⟩
‖ ⊢caseℕ ⊢L ⊢M ⊢N ‖⁺ = Intr.caseℕ ‖ ⊢L ‖⁺ ‖ ⊢M ‖⁺ ‖ ⊢N ‖⁻
‖ ⊢case⊎ ⊢L ⊢M ⊢N ‖⁺ = Intr.case⊎ ‖ ⊢L ‖⁺ ‖ ⊢M ‖⁺ ‖ ⊢N ‖⁻

‖ ⊢ƛ ⊢N ‖⁻         = Intr.ƛ ‖ ⊢N ‖⁻
‖ ⊢μ ⊢M ‖⁻         = Intr.μ ‖ ⊢M ‖⁻
‖ ⊢↑ ⊢M refl ‖⁻ = ‖ ⊢M ‖⁺
‖ ⊢inj₁ ⊢L ‖⁻    = Intr.`inj₁ ‖ ⊢L ‖⁺
‖ ⊢inj₂ ⊢R ‖⁻    = Intr.`inj₂ ‖ ⊢R ‖⁺
```

3.6 Annotatability

As can be seen from our previous examples, annotations show up in our code through the `_↑_` and `_↑` constructors. The `_↑` annotations could plausibly be inserted automatically as needed by a more advanced type checker, but the inheriting annotations using `_↑_` would have to be specified manually. [DK21, p. 6] lists four criteria for such annotations in the design of a bidirectional type checker: they should be lightweight, predictable, stable, and legible. In our design, we've mainly tried to keep annotations minimal.

In practice, since we have made most of our terms synthesized terms, our language requires a small amount of such annotations, primarily for function signatures. It is common practice even in languages using a Damas-Milner type checker, such as OCaml, to write out function signatures explicitly even though they are not required. This tends to aid readability and produce clearer error messages. We are therefore not losing much by requiring such annotations.

Having proven soundness, we now also want to show that we can annotate any intrinsically typed term, proving that any well-typed program can be annotated with types. This corresponds to the completeness of our type system, that every well-typed program can be annotated in our type system. [WKS22] does not prove annotatability, but the proof is somewhat similar to the proof of soundness. We use `⟪_⟫` to denote completeness, which is an arbitrary choice. To start with, we need to translate between the types and contexts of the intrinsic and extrinsic views.

```
⟪_⟫Tp : Intr.Type → Type
⟪ M Intr.⇒ N ⟫Tp = ⟪ M ⟫Tp ⇒ ⟪ N ⟫Tp
⟪ Intr.`ℕ ⟫Tp = `ℕ
⟪ A Intr.`× B ⟫Tp = ⟪ A ⟫Tp `× ⟪ B ⟫Tp
⟪ A Intr.`⊎ B ⟫Tp = ⟪ A ⟫Tp `⊎ ⟪ B ⟫Tp

⟪_⟫Cx : Intr.Context → Context
⟪ Intr.∅ ⟫Cx = ∅
⟪ Γ Intr., A ⟫Cx = ⟪ Γ ⟫Cx , ⟪ A ⟫Tp

⟪_⟫∋ : ∀ {Γ A} → Γ Intr.∋ A → ⟪ Γ ⟫Cx ∋ ⟪ A ⟫Tp
⟪ Intr.Z ⟫∋ = Z
⟪ Intr.S ∋x ⟫∋ = S ⟪ ∋x ⟫∋
```

To prove annotatability, we will first need a way to translate from intrinsically typed terms to untyped positive and negative terms.

```
⟪_⟫⁺ : ∀ {Γ} → {A : Intr.Type} → Γ Intr.⊢ A → Term⁺
⟪_⟫⁻ : ∀ {Γ} → {A : Intr.Type} → Γ Intr.⊢ A → Term⁻
```

```
⟪ Intr.` x ⟫⁺ = # index ⟪ x ⟫∋
⟪ L Intr.· M ⟫⁺ = ⟪ L ⟫⁺ · ⟪ M ⟫⁻
⟪ Intr.`zero ⟫⁺ = `zero
⟪ Intr.`suc N ⟫⁺ = `suc ⟪ N ⟫⁺
⟪ Intr.caseℕ L M N ⟫⁺ = `caseℕ ⟪ L ⟫⁺ [zero⇒ ⟪ M ⟫⁺ |suc⇒ ⟪ N ⟫⁻ ]
⟪ Intr.case⊎ L M N ⟫⁺ = `case⊎ ⟪ L ⟫⁺ [inj₁⇒ ⟪ M ⟫⁺ |inj₂⇒ ⟪ N ⟫⁻ ]
⟪ Intr.`⟨ L , R ⟩ ⟫⁺ = `⟨ ⟪ L ⟫⁺ , ⟪ R ⟫⁺ ⟩
⟪ Intr.`proj₁ P ⟫⁺ = `proj₁ ⟪ P ⟫⁺
⟪ Intr.`proj₂ P ⟫⁺ = `proj₂ ⟪ P ⟫⁺
⟪_⟫⁺ {_} {A} (Intr.ƛ N) = (ƛ ⟪ N ⟫⁻) ↓ ⟪ A ⟫Tp
⟪_⟫⁺ {_} {A} (Intr.μ N) = (μ ⟪ N ⟫⁻) ↓ ⟪ A ⟫Tp
⟪_⟫⁺ {_} {A} (Intr.`inj₁ L) = `inj₁ ⟪ L ⟫⁺ ↓ ⟪ A ⟫Tp
⟪_⟫⁺ {_} {A} (Intr.`inj₂ R) = `inj₂ ⟪ R ⟫⁺ ↓ ⟪ A ⟫Tp

⟪ Intr.` x ⟫⁻ = (# index ⟪ x ⟫∋) ↑
⟪ L Intr.· M ⟫⁻ = (⟪ L ⟫⁺ · ⟪ M ⟫⁻) ↑
⟪ Intr.`zero ⟫⁻ = `zero ↑
⟪ Intr.`suc N ⟫⁻ = `suc ⟪ N ⟫⁺ ↑
⟪ Intr.caseℕ L M N ⟫⁻ = `caseℕ ⟪ L ⟫⁺ [zero⇒ ⟪ M ⟫⁺ |suc⇒ ⟪ N ⟫⁻ ] ↑
⟪ Intr.case⊎ L M N ⟫⁻ = `case⊎ ⟪ L ⟫⁺ [inj₁⇒ ⟪ M ⟫⁺ |inj₂⇒ ⟪ N ⟫⁻ ] ↑
⟪ Intr.`⟨ L , R ⟩ ⟫⁻ = `⟨ ⟪ L ⟫⁺ , ⟪ R ⟫⁺ ⟩ ↑
⟪ Intr.`proj₁ P ⟫⁻ = `proj₁ ⟪ P ⟫⁺ ↑
⟪ Intr.`proj₂ P ⟫⁻ = `proj₂ ⟪ P ⟫⁺ ↑
⟪ Intr.ƛ N ⟫⁻ = ƛ ⟪ N ⟫⁻
⟪ Intr.μ N ⟫⁻ = μ ⟪ N ⟫⁻
⟪ Intr.`inj₁ L ⟫⁻ = `inj₁ ⟪ L ⟫⁺
⟪ Intr.`inj₂ R ⟫⁻ = `inj₂ ⟪ R ⟫⁺
```

We can then use these definitions to state the full synthesizing and inheriting judgments a given intrinsic term can have. Since we have the freedom to convert between synthesizing and inheriting, we are able to define these functions for both positive and negative terms.

```
⟪_⟫↑ : ∀ {Γ} → {A : Intr.Type} → (M : Γ Intr.⊢ A) → ⟪ Γ ⟫Cx ⊢ ⟪ M ⟫⁺ ↑ ⟪ A ⟫Tp
⟪_⟫↓ : ∀ {Γ} → {A : Intr.Type} → (M : Γ Intr.⊢ A) → ⟪ Γ ⟫Cx ⊢ ⟪ M ⟫⁻ ↓ ⟪ A ⟫Tp

⟪ Intr.` x ⟫↑ = ⊢` [ ⟪ x ⟫∋ , refl ]Var
⟪ L Intr.· M ⟫↑ = ⟪ L ⟫↑ · ⟪ M ⟫↓
⟪ Intr.`zero ⟫↑ = ⊢zero
⟪ Intr.`suc M ⟫↑ = ⊢suc ⟪ M ⟫↑
⟪ Intr.caseℕ L M N ⟫↑ = ⊢caseℕ ⟪ L ⟫↑ ⟪ M ⟫↑ ⟪ N ⟫↓
```

```
⟪ Intr.case⊎ L M N ⟫↑ = ⊢case⊎ ⟪ L ⟫↑ ⟪ M ⟫↑ ⟪ N ⟫↓
⟪ Intr.`⟨ L , R ⟩ ⟫↑ = ⊢⟨,⟩ ⟪ L ⟫↑ ⟪ R ⟫↑
⟪ Intr.`proj₁ P ⟫↑ = ⊢proj₁ ⟪ P ⟫↑
⟪ Intr.`proj₂ P ⟫↑ = ⊢proj₂ ⟪ P ⟫↑
⟪ Intr.ƛ N ⟫↑ = ⊢↓ (⊢ƛ ⟪ N ⟫↓)
⟪ Intr.μ N ⟫↑ = ⊢↓ (⊢μ ⟪ N ⟫↓)
⟪ Intr.`inj₁ L ⟫↑ = ⊢↓ (⊢inj₁ ⟪ L ⟫↑)
⟪ Intr.`inj₂ R ⟫↑ = ⊢↓ (⊢inj₂ ⟪ R ⟫↑)

⟪ Intr.` x ⟫↓ = ⊢↑ (⊢` [ ⟪ x ⟫∋ , refl ]Var) refl
⟪ Intr.ƛ M ⟫↓ = ⊢ƛ ⟪ M ⟫↓
⟪ L Intr.· M ⟫↓ = ⊢↑ (⟪ L ⟫↑ · ⟪ M ⟫↓) refl
⟪ Intr.`zero ⟫↓ = ⊢↑ ⊢zero refl
⟪ Intr.`suc N ⟫↓ = ⊢↑ (⊢suc ⟪ N ⟫↑) refl
⟪ Intr.caseℕ L M N ⟫↓ = ⊢↑ (⊢caseℕ ⟪ L ⟫↑ ⟪ M ⟫↑ ⟪ N ⟫↓) refl
⟪ Intr.μ N ⟫↓ = ⊢μ ⟪ N ⟫↓
⟪ Intr.`inj₁ M ⟫↓ = ⊢inj₁ ⟪ M ⟫↑
⟪ Intr.`inj₂ M ⟫↓ = ⊢inj₂ ⟪ M ⟫↑
⟪ Intr.case⊎ L M N ⟫↓ = ⊢↑ (⊢case⊎ ⟪ L ⟫↑ ⟪ M ⟫↑ ⟪ N ⟫↓) refl
⟪ Intr.`⟨ L , R ⟩ ⟫↓ = ⊢↑ (⊢⟨,⟩ ⟪ L ⟫↑ ⟪ R ⟫↑) refl
⟪ Intr.`proj₁ P ⟫↓ = ⊢↑ (⊢proj₁ ⟪ P ⟫↑) refl
⟪ Intr.`proj₂ P ⟫↓ = ⊢↑ (⊢proj₂ ⟪ P ⟫↑) refl
```

That concludes our proofs. We have given a provably correct implementation of bidirectional type checking, and shown that it is both sound and complete. In the final chapter, we will look into extending this algorithm for the more complex case of polymorphism.

Chapter 4

Polymorphism

We now venture beyond the scope of [WKS22] by introducing *polymorphism*. In our previous language, functions were limited to operate on unique types. However, the behavior of many functions can generalize to arbitrary types. As a trivial example, consider the identity function $x \mapsto x$. In a monomorphic language this function could be typed as $\mathbb{N} \to \mathbb{N}$, but this typing would restrict our identity function to operate on natural numbers only. With polymorphism we are able to give a more general typing of this function.

While in the usual lambda calculus we have contexts of term variables, conventionally denoted by Γ, we will now add an additional context of type variables which we will conventionally call Δ. We then introduce a new construct Λ which represents a type-level abstraction, akin to the term-level λ. The Λ abstraction lets us define functions that generalize over families of types, and its type is given by universal quantification over a type variable.

For brevity we will not extend the whole language, but rather show how bidirectional typechecking might be extended to a language similar to the polymorphic typed lambda calculus, also known as System F. [Har16, p. 141]

Our implementation here supports *higher-rank polymorphism*, where the polymorphic quantifier can be nested at arbitrary places in function signatures. An example of this is $f : \forall\alpha.(\forall\beta.\beta \to \alpha) \to \alpha$. In programming languages like Haskell, Λ abstractions are inferred automatically which allows for cleaner syntax, but this type of inference is restricted to *prefix polymorphism* [DK21, section 5] where all quantifiers must come at the start of the signature. We avoid problems surrounding this by always requiring explicit polymorphic abstractions, even though this is a fairly cumbersome limitation. A more advanced type checking algorithm may want to loosen this requirement.

As an aside, a polymorphic function type restricts the space of functions that can inhabit the type, leading to the notion of free theorems and parametricity: from just looking at the signature of a polymorphic function, we can deduce properties that the function will satisfy. This concept was originally described by Reynolds [Rey83], and

explored further in the paper Theorems for Free by Philip Wadler [Wad89], who happens to be a coauthor of the book much of this thesis was based on. We will not explore any of this here, but leave it as an avenue for further exploration to the interested reader.

```
module thesis.src.Poly where

-- Fixity declarations omitted
```

To represent polymorphism, we will need a notion of type variables. Type variables have a *kind*, which can be seen as a type of types. Our language will only have one kind, representing an arbitrary type, but a more advanced language could have higher-kinded types. We will not delve further into kinds. We start by defining a kind data type, with a single constructor `*`. This is a trivial type definition, but makes our code read a little bit better.

```
data Kind : Set where
  * : Kind
```

Just like we stored our typing judgments in a context, we will place our type variable judgments in a context as well. This definition is nearly equivalent to our earlier definition of contexts.

```
data TyContext : Set where
  ∅ : TyContext
  _,_ : TyContext → Kind → TyContext

data _∋_ : TyContext → Kind → Set where
  Z : ∀ {Δ A}
    → Δ , A ∋ A
  S_ : ∀ {Δ A B}
    → Δ ∋ A
    → Δ , B ∋ A
```

Now we will differ from the presentation in Chapter 3. With the introduction of a type context, we will need types to depend on a context Δ of type variables. That is to say, we will need judgments stating that for a given type A and a context Δ we have $\Delta \vdash A$ type. We define these judgments below. Our definitions are minimal, and serve as a proof of concept rather than a full extension of the past chapter.

Of particular note is the `` `∀ `` constructor, representing the universal quantifier. It is special since it is the only type that uses two different type contexts. We can think of it as discharging an assumed type variable through abstraction, similar to how we defined lambda abstractions before.

The second type of note is the `` `var `` constructor, which is largely equivalent to a variable term but for types instead.

```
data _⊢type : TyContext → Set where
  `ℕ : ∀ {Δ} → Δ ⊢type
  _⇒_ : ∀ {Δ}
    → Δ ⊢type
    → Δ ⊢type
    → Δ ⊢type
  `∀_ : ∀ {Δ : TyContext}
    → (Δ , *) ⊢type
    → Δ ⊢type
  `var : ∀ {Δ A}
    → Δ ∋ A
    → Δ ⊢type
```

Example 4.0.1. The type of the polymorphic identity function $\forall\alpha.\alpha \to \alpha$:

```
_ : ∅ ⊢type
_ = `∀ `var Z ⇒ `var Z
```

Next, we need to redefine our term context. Since the term context contains judgments about the types of terms and types depend on the type context, it follows that the term context (like our types) must also depend on a type context. That is, we need a context judgment of the form

$$\Delta \vdash \Gamma \text{ context}$$

where Δ is a context of type variables.

```
data _⊢ctx (Δ : TyContext) : Set where
  ∅ : Δ ⊢ctx
  _,_ : Δ ⊢ctx → Δ ⊢type → Δ ⊢ctx

data _∣_∋_ : (Δ : TyContext) → Δ ⊢ctx → Δ ⊢type → Set where
  Z : ∀ {Δ Γ A}
    → Δ ∣ Γ , A ∋ A
  S_ : ∀ {Δ Γ A B}
    → Δ ∣ Γ ∋ A
    → Δ ∣ Γ , B ∋ A
```

We will now extend the substitution machinery from the previous chapter to support our new definitions. The details are not noteworthy, as the definitions can be adapted quite naturally, so we leave out the proofs.

```
ext : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
  → (∀ {A B} → Γ , B ∋ A → Δ , B ∋ A)

rename : ∀ {Δ Δ′}
  → (∀ {A} → Δ ∋ A → Δ′ ∋ A)
  → (Δ ⊢type → Δ′ ⊢type)

exts : ∀ {Δ Δ′}
  → (∀ {A} → Δ ∋ A → Δ′ ⊢type)
  → (∀ {A B} → Δ , B ∋ A → Δ′ , B ⊢type)

subst : ∀ {Δ Δ′}
  → (∀ {A} → Δ ∋ A → Δ′ ⊢type)
  → Δ ⊢type
  → Δ′ ⊢type

substOne : ∀ {Δ}
  → Δ , * ⊢type
  → Δ ⊢type
  → Δ ⊢type
```

We will also need a notion of weakening types and term contexts. Given that a type A or a context Γ is valid in the type context Δ, we will show that they are also valid in the more general type context $\Delta , *$.

```
weakTy : ∀ {Δ} → Δ ⊢type → (Δ , *) ⊢type
weakTy ty = rename S_ ty

weakCtx : ∀ {Δ} → Δ ⊢ctx → (Δ , *) ⊢ctx
weakCtx ∅ = ∅
weakCtx (Γ , ty) = weakCtx Γ , weakTy ty
```

4.1 Intrinsic presentation

We now present a minimal version of our language in intrinsic form. This is mainly instructive, and we will not make use of these definitions at all in our implementation of bidirectional type checking.

```
module Intrinsic where
  infix 4 _|_⊢_
```

```
infix 5 ƛ_
infix 5 Λ_

data _|_⊢_ : ∀ {Δ Γ} → TyContext → Γ ⊢ctx → Δ ⊢type → Set where
  `_ : ∀ {Δ Γ A}
    → Δ | Γ ∋ A
    → Δ | Γ ⊢ A
  `zero : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx}
    → Δ | Γ ⊢ `ℕ {Δ}
  `suc : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx}
    → Δ | Γ ⊢ `ℕ {Δ}
    → Δ | Γ ⊢ `ℕ {Δ}
  _·_ : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx} {A : Δ ⊢type} {B : Δ ⊢type}
    → Δ | Γ ⊢ A ⇒ B
    → Δ | Γ ⊢ A
    → Δ | Γ ⊢ B
  ƛ_ : ∀ {Δ A B} {Γ : Δ ⊢ctx}
    → Δ | Γ , A ⊢ B
    → Δ | Γ ⊢ A ⇒ B
  Λ_ : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx} {B : (Δ , *) ⊢type}
    → Δ , * | weakCtx Γ ⊢ B
    → Δ | Γ ⊢ `∀ B
  _[_] : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx} {B : (Δ , *) ⊢type}
    → Δ | Γ ⊢ `∀ B
    → (A : Δ ⊢type)
    → Δ | Γ ⊢ substOne B A
```

This allows us to express polymorphic functions as shown by the examples below.

Example 4.1.1. A doubly polymorphic function `f`.

```
f : ∅ | ∅ ⊢ `∀ `∀ `var (S Z) ⇒ `var (S Z)
f = Λ Λ ƛ ` Z
```

Example 4.1.2. Instantiating `f` with the type of natural numbers.

```
_ : ∅ | ∅ ⊢ `∀ (`ℕ ⇒ `ℕ)
_ = f [ `ℕ ]
```

4.2 Type checking preliminaries

We now want to show that we can adapt our implementation of bidirectional type checking to support polymorphism. Many of the details are trivially modified to support type

contexts, and we will not remark on them unless they are of particular note. Here, we mostly repeat previous definitions modified for use with type contexts.

```
module Bidirectional where

-- Fixity declarations and imports omitted


Index : Set
Index = ℕ

index : ∀ {Δ Γ A} → Δ ∣ Γ ∋ A → ℕ
index Z = 0
index (S x) = suc (index x)

record Var (Δ : TyContext) (Γ : Δ ⊢ctx) (A : Δ ⊢type) (x : ℕ) : Set where
  constructor [_,_]Var
  field
    ∋x : Δ ∣ Γ ∋ A
    idx≡ : index ∋x ≡ x

ext∋ : ∀ {Δ Γ B x}
  → ¬ (∃[ A ] Var Δ Γ A x)
  → ¬ (∃[ A ] Var Δ (Γ , B) A (suc x))
ext∋ ¬∃ ⟨ A , [ (S ∋x) , idx≡ ]Var ⟩ = ¬∃ ⟨ A , [ ∋x , suc-injective idx≡ ]Var ⟩

lookup : ∀ {Δ} → (Γ : Δ ⊢ctx ) (x : Index)
  → Dec (∃[ A ] Var Δ Γ A x )
lookup ∅ x = no (λ ())
lookup (Γ , B) zero = yes ⟨ B , [ Z , refl ]Var ⟩
lookup (Γ , _) (suc x) with lookup Γ x
... | no ¬∃ = no (ext∋ ¬∃)
... | yes ⟨ A , vx ⟩ = yes ⟨ A , [ (S (Var.∋x vx)) , cong suc (Var.idx≡ vx) ]Var ⟩
```

4.3 Terms and typing judgments

As before, we will proceed by categorizing our terms as synthesized or inherited terms. In this language, we keep the abstractions inherited and everything else synthesized.

Since types now feature in our untyped term language, we need some way of representing types as terms without explicit reference to a type context. We choose to denote

types appearing as part of terms using the empty type context. This is a limitation since it forbids the use of type variables in substitutions. An improved version of this type checker might implement an indexing approach, like we've done for term variables, to look up type variables in a context.

```
data Term⁺ : Set
data Term⁻ : Set

data Term⁺ where
  #_    : Index → Term⁺
  `zero : Term⁺
  `suc_ : Term⁺ → Term⁺
  _·_   : Term⁺ → Term⁻ → Term⁺
  _[_]  : Term⁺ → ∅ ⊢type → Term⁺
  _↓_   : Term⁻ → ∅ ⊢type → Term⁺

data Term⁻ where
  ƛ_    : Term⁻ → Term⁻
  Λ_    : Term⁻ → Term⁻
  _↑    : Term⁺ → Term⁻
```

Given that we represent term-level types as types in the empty context, we will need another lemma for weakening such types to arbitrary depths. This allows us to easily map such types into any given type context.

```
weakEmptyTy : ∀ {Δ}
  → ∅ ⊢type
  → Δ ⊢type
weakEmptyTy {∅} A = A
weakEmptyTy {Δ , _} A = rename S_ (weakEmptyTy {Δ} A)
```

We will now define our synthesizing and inheriting judgments as before. They are largely analogous to their monomorphic equivalents, with some differences to support polymorphism. Here we make use of our weakening lemmas to support type substitution and type abstraction.

```
data _|_⊢_↑_ : (Δ : TyContext) → Δ ⊢ctx → Term⁺ → Δ ⊢type → Set
data _|_⊢_↓_ : (Δ : TyContext) → Δ ⊢ctx → Term⁻ → Δ ⊢type → Set

data _|_⊢_↑_ where
  ⊢` : ∀ {Δ : TyContext} {Γ : Δ ⊢ctx} {A : Δ ⊢type} {x : ℕ}
    → (Var Δ Γ A x)
```

```
    → Δ ∣ Γ ⊢ # x ↑ A
  ⊢zero : ∀ {Δ Γ}
    → Δ ∣ Γ ⊢ `zero ↑ `ℕ
  ⊢suc : ∀ {Δ Γ M}
    → Δ ∣ Γ ⊢ M ↑ `ℕ
    → Δ ∣ Γ ⊢ `suc M ↑ `ℕ
  _·_ : ∀ {Δ Γ L M A B}
    → Δ ∣ Γ ⊢ L ↑ A ⇒ B
    → Δ ∣ Γ ⊢ M ↓ A
    → Δ ∣ Γ ⊢ L · M ↑ B
  ⊢[] : ∀ {Δ Γ M B A}
    → Δ ∣ Γ ⊢ M ↑ `∀ B
    → Δ ∣ Γ ⊢ M [ A ] ↑ substOne B (weakEmptyTy A)
  ⊢↓ : ∀ {Δ Γ M A}
    → Δ ∣ Γ ⊢ M ↓ weakEmptyTy A
    → Δ ∣ Γ ⊢ (M ↓ A) ↑ weakEmptyTy A

data _∣_⊢_↓_ where
  ⊢ƛ : ∀ {Δ Γ A B N}
    → Δ ∣ Γ , A ⊢ N ↓ B
    → Δ ∣ Γ ⊢ ƛ N ↓ A ⇒ B
  ⊢Λ : ∀ {Δ Γ M B}
    → Δ ⊢ctx
    → Δ , * ∣ weakCtx Γ ⊢ M ↓ B
    → Δ ∣ Γ ⊢ Λ M ↓ `∀ B
  ⊢↑ : ∀ {Δ Γ M A B}
    → Δ ∣ Γ ⊢ M ↑ A
    → A ≡ B
    → Δ ∣ Γ ⊢ (M ↑) ↓ B
```

Example 4.3.1. A typing derivation for the polymorphic identity function.

```
_ : ∅ ∣ ∅ ⊢ Λ (ƛ ((# 0) ↑)) ↓ (`∀ `var Z ⇒ `var Z)
_ = ⊢Λ ∅ (⊢ƛ (⊢↑ (⊢` [ Z , refl ]Var) refl))
```

4.4 Lemmas

Next, we will again need some lemmas in similar fashion to what we needed before. They are similar to before, and we leave out some of the less interesting proofs.

```
ℕ≢⇒ : ∀ {Δ A B} → `ℕ {Δ} ≢ A ⇒ B
ℕ≢var : ∀ {Δ A N} → `ℕ {Δ} ≢ `var {Δ} {A} N
⇒≢∀ : ∀ {Δ A B C} → (_⇒_ {Δ} A B) ≢ `∀ C
⇒≢var : ∀ {Δ A B C D} → (_⇒_ {Δ} A B) ≢ `var {Δ} {C} D
var≢∀ : ∀ {Δ A B C} → (`var {Δ} {A} B) ≢ `∀ C
∀≡ : ∀ {Δ A B} → `∀_ {Δ} A ≡ `∀ B → A ≡ B
dom≡ : ∀ {Δ} {A A′ B B′ : Δ ⊢type} → A ⇒ B ≡ A′ ⇒ B′ → A ≡ A′
rng≡ : ∀ {Δ} {A A′ B B′ : Δ ⊢type} → A ⇒ B ≡ A′ ⇒ B′ → B ≡ B′

uniq-∋ : ∀ {Δ Γ A B}
  → (∋x : Δ | Γ ∋ A)
  → (∋x′ : Δ | Γ ∋ B)
  → (index ∋x ≡ index ∋x′)
  → A ≡ B
uniq-∋ Z Z p = refl
uniq-∋ (S ∋x) (S ∋x′) p = uniq-∋ ∋x ∋x′ (suc-injective p)

uniq-↑ : ∀ {Δ Γ M A B}
  → Δ | Γ ⊢ M ↑ A
  → Δ | Γ ⊢ M ↑ B
  → A ≡ B
uniq-↑ (⊢` [ ∋x , idx≡ ]Var) (⊢` [ ∋x′ , idx≡′ ]Var) = uniq-∋ ∋x ∋x′ (trans idx≡ (sym idx≡′))
uniq-↑ (⊢↓ ⊢M) (⊢↓ ⊢M′) = refl
uniq-↑ ⊢zero ⊢zero = refl
uniq-↑ (⊢suc ⊢N) (⊢suc ⊢N′) = refl
uniq-↑ (⊢[] ⊢M) (⊢[] ⊢M′) rewrite ∀≡ (uniq-↑ ⊢M ⊢M′) = refl
uniq-↑ (⊢L · ⊢M) (⊢L′ · ⊢M′) = rng≡ (uniq-↑ ⊢L ⊢L′)

_≟Tp_ : ∀ {Δ} → (A B : Δ ⊢type) → Dec (A ≡ B)

¬arg : ∀ {Δ Γ A B L M}
  → Δ | Γ ⊢ L ↑ A ⇒ B
  → ¬ Δ | Γ ⊢ M ↓ A
  → ¬ (∃[ B′ ] Δ | Γ ⊢ L · M ↑ B′)
¬arg ⊢L ¬∃ ⟨ B′ , ⊢L′ · ⊢M′ ⟩ rewrite dom≡ (uniq-↑ ⊢L ⊢L′) = ¬∃ ⊢M′

¬switch : ∀ {Δ Γ M A B}
  → Δ | Γ ⊢ M ↑ A
  → A ≢ B
  → ¬ Δ | Γ ⊢ (M ↑) ↓ B
¬switch ⊢M A≢B (⊢↑ ⊢M′ A′≡B) rewrite uniq-↑ ⊢M ⊢M′ = A≢B A′≡B
```

4.5 Synthesizing and inheriting

Finally, we have reached the prize: bidirectional type checking with support for polymorphism. The hard part was to come up with the right definitions, after which the implementation follows naturally. We extend the `synthesize` and `inherit` functions to take an additional type context argument, and proceed to fill in the cases using the lemmas defined earlier.

```
synthesize : ∀ (Δ : TyContext) (Γ : Δ ⊢ctx) (M : Term⁺)
  → Dec (∃[ A ] Δ │ Γ ⊢ M ↑ A)

inherit : ∀ (Δ : TyContext) (Γ : Δ ⊢ctx) (M : Term⁻) (A : Δ ⊢type)
  → Dec (Δ │ Γ ⊢ M ↓ A)

synthesize Δ Γ `zero = yes ⟨ `ℕ , ⊢zero ⟩
synthesize Δ Γ (# x) with lookup Γ x
... | no ¬∃ = no λ{ ⟨ A , ⊢` ∋x ⟩ → ¬∃ ⟨ A , ∋x ⟩ }
... | yes ⟨ A , ∋x ⟩ = yes ⟨ A , ⊢` ∋x ⟩
synthesize Δ Γ (`suc N) with synthesize Δ Γ N
... | no ¬∃ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N ⟩ → ¬∃ ⟨ `ℕ , ⊢N ⟩ }
... | yes ⟨ `ℕ , ⊢N ⟩ = yes ⟨ `ℕ , (⊢suc ⊢N) ⟩
... | yes ⟨ `∀_ N , ⊢N ⟩ = no λ{ ⟨ `ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢∀ (uniq-↑ ⊢N′ ⊢N) }
... | yes ⟨ `var _ , ⊢N ⟩ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢var (uniq-↑ ⊢N′ ⊢N) }
... | yes ⟨ _ ⇒ _ , ⊢N ⟩ = no λ{ ⟨ .`ℕ , ⊢suc ⊢N′ ⟩ → ℕ≢⇒ (uniq-↑ ⊢N′ ⊢N) }
synthesize Δ Γ (L · M) with synthesize Δ Γ L
... | no ¬∃ = no λ{ ⟨ _ , ⊢L  · _  ⟩ → ¬∃ ⟨ _ , ⊢L ⟩ }
... | yes ⟨ `ℕ ,    ⊢L ⟩ = no λ{ ⟨ _ , ⊢L′ · _ ⟩ → ℕ≢⇒ (uniq-↑ ⊢L ⊢L′) }
... | yes ⟨ `∀ N ,  ⊢L ⟩ = no λ{ ⟨ _ , ⊢L′ · _ ⟩ → ⇒≢∀ (uniq-↑ ⊢L′ ⊢L) }
... | yes ⟨ `var x , ⊢L ⟩ = no λ{ ⟨ _ , ⊢L′ · _ ⟩ → ⇒≢var (uniq-↑ ⊢L′ ⊢L) }
... | yes ⟨ A ⇒ B , ⊢L ⟩ with inherit Δ Γ M A
... | no     ¬∃ = no (¬arg ⊢L ¬∃)
... | yes ⊢M = yes ⟨ B , ⊢L · ⊢M ⟩
synthesize Δ Γ (N [ A ]) with synthesize Δ Γ N
... | no ¬∃ = no λ{ ⟨ _ , ⊢[] ⊢N ⟩ → ¬∃ ⟨ _ , ⊢N ⟩ }
... | yes ⟨ `ℕ , ⊢N ⟩ = no λ{ ⟨ _ , ⊢[] ⊢N′ ⟩ → ℕ≢∀ (uniq-↑ ⊢N ⊢N′) }
... | yes ⟨ _ ⇒ _ , ⊢N ⟩ = no λ{ ⟨ _ , ⊢[] ⊢N′ ⟩ → ⇒≢∀ (uniq-↑ ⊢N ⊢N′) }
... | yes ⟨ `∀ _ , ⊢N ⟩ = yes ⟨ _ , ⊢[] ⊢N ⟩
... | yes ⟨ `var _ , ⊢N ⟩ = no λ{ ⟨ _ , ⊢[] ⊢N′ ⟩ → var≢∀ (uniq-↑ ⊢N ⊢N′) }
synthesize Δ Γ (M ↓ A) with inherit Δ Γ M (weakEmptyTy A)
... | no ¬∃ = no λ{ ⟨ _ , ⊢↓ ⊢M ⟩ → ¬∃ ⊢M }
... | yes ⊢M = yes ⟨ _ , ⊢↓ ⊢M ⟩
```

```
inherit Δ Γ (ƛ x) `ℕ = no λ ()
inherit Δ Γ (ƛ N) (A ⇒ B) with inherit Δ (Γ , A) N B
... | no ¬∃ = no (λ{ (⊢ƛ ⊢N) → ¬∃ ⊢N })
... | yes ⊢N = yes (⊢ƛ ⊢N)
inherit Δ Γ (ƛ x) (`∀ A) = no λ ()
inherit Δ Γ (ƛ x) (`var x₁) = no λ ()
inherit Δ Γ (Λ N) `ℕ = no λ ()
inherit Δ Γ (Λ N) (_ ⇒ _) = no λ ()
inherit Δ Γ (Λ N) (`var _) = no λ ()
inherit Δ Γ (Λ N) (`∀ A) with inherit (Δ , *) (weakCtx Γ) N A
... | no ¬∃ = no λ{ (⊢Λ _ ⊢N) → ¬∃ ⊢N }
... | yes (⊢ƛ N) = yes (⊢Λ Γ (⊢ƛ N))
... | yes (⊢Λ Γ′ ⊢N) = yes (⊢Λ Γ (⊢Λ Γ′ ⊢N))
... | yes (⊢↑ ⊢N A≡A′) = yes (⊢Λ Γ (⊢↑ ⊢N A≡A′))
inherit Δ Γ (M ↑) B with synthesize Δ Γ M
... | no ¬∃ = no (λ{ (⊢↑ ⊢M _) → ¬∃ ⟨ _ , ⊢M ⟩ })
... | yes ⟨ A , ⊢M ⟩ with A ≟Tp B
... | no A≢B = no    (¬switch ⊢M A≢B)
... | yes A≡B = yes (⊢↑ ⊢M A≡B)
```

To show that our implementation does what is expected, we give a few examples.

Example 4.5.1. The polymorphic lambda function instantiated with the natural numbers synthesizes as expected. Note that we must explicitly annotate the Λ-abstraction.

```
_ : synthesize ∅ ∅
      (((Λ (ƛ (# 0 ↑))) ↓ (`∀ `var Z ⇒ `var Z)) [ `ℕ ])
    ≡ yes ⟨ `ℕ ⇒ `ℕ , ⊢[] (⊢↓ (⊢Λ ∅ (⊢ƛ (⊢↑ (⊢` [ Z , refl ]Var) refl)))) ⟩
_ = refl
```

Example 4.5.2. The same function, without being instantiated, checks as a polymorphic function.

```
_ : inherit ∅ ∅
      (Λ (ƛ ((# 0) ↑))) (`∀ `var Z ⇒ `var Z)
      ≡ yes (⊢Λ ∅ (⊢ƛ (⊢↑ (⊢` [ Z , refl ]Var) refl)))
_ = refl
```

We will not attempt to give any proofs of soundness or completeness, to keep the presentation short. It is likely possible to prove both of these properties for this type system.

With this, we have shown how to extend the earlier algorithm to support polymorphism. Though the example is minimal, adding the remaining terms that we previously defined would likely be easy.

Chapter 5

Conclusion

Using the simply typed lambda calculus as a starting point, we implemented a more advanced language and proved that progress and preservation hold for this language. We then gave an implementation of bidirectional typechecking for the language, proving that it is sound and complete. Finally, we demonstrated how the algorithm can be extended to support parametric polymorphism in a minimal language.

Bidirectional type checking is a simple and flexible approach to type checking which supports vastly more complicated languages than the ones we have defined here. Agda itself uses a variation of this algorithm to implement its dependent type system, which should serve as a testament to its power.

All proofs were mechanized using Agda, a proof assistant based on dependent types. This ensures that all proofs are correct, an unfamiliar affordance coming from pen-and-paper mathematics.

The intersection of mathematics and programming language theory is an exciting area of study, and this thesis demonstrates how mathematics can lend rigor to the craft of programming.

Bibliography

[Mar72] Per Martin-Löf. *An intuitionistic theory of types.* 1972.

[Mar82] Per Martin-Löf. "Constructive mathematics and computer programming". In: *Studies in Logic and the Foundations of Mathematics.* Vol. 104. Elsevier, 1982, pp. 153–175.

[Rey83] John C Reynolds. "Types, abstraction and parametric polymorphism". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres.* 1983, pp. 513–523.

[TD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, Vol 1.* ISSN. Elsevier Science, 1988. ISBN: 9780080570884. URL: https://books.google.se/books?id=-tc2qp0-2bsC.

[Wad89] Philip Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture.* 1989, pp. 347–359.

[Cha09] Kaustuv Chaudhuri. *de Bruijn index illustration.* https://commons.wikimedia.org/wiki/File:De_Bruijn_index_illustration_1.svg. [Online; accessed 10-March-2023]. 2009.

[Nor09] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures.* Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 230–266. ISBN: 978-3-642-04652-0. DOI: 10.1007/978-3-642-04652-0_5. URL: https://doi.org/10.1007/978-3-642-04652-0_5.

[MZ13] Paul-André Melliès and Noam Zeilberger. *Type refinement and monoidal closed bifibrations.* 2013. arXiv: 1310.0263 [cs.LO].

[Har16] Robert Harper. *Practical foundations for programming languages, second edition.* Mar. 2016, pp. 1–476. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[DK21] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021), pp. 1–38. DOI: 10.1145/3450952. URL: https://doi.org/10.1145%2F3450952.

[BPI22] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. "Constructive Mathematics". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[WKS22] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/22.08/, Aug. 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

[Agd23] Agda. *Welcome to Agda's documentation! — Agda 2.6.3 documentation*. 2023. URL: https://agda.readthedocs.io/en/v2.6.3/.

[DP23] Peter Dybjer and Erik Palmgren. "Intuitionistic Type Theory". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Spring 2023. Metaphysics Research Lab, Stanford University, 2023.