UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Eduardo Henke

Implementing a programming language with a dependent type system

Florianópolis
16 de novembro de 2022

Eduardo Henke

Implementing a programming language with a dependent type system

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 16 de novembro de 2022.

Prof. Jean Everson Martina, Dr
Coordenador do Curso

Banca Examinadora:

Prof. Alvaro Junio Pereira Franco, Dr.
Orientador
Universidade Federal de Santa Catarina

Li-Yao Xia, Dr.
Coorientador
University of Edinburgh

Prof[a]. Jerusa Marchi, Dr.
Avaliadora
Universidade Federal de Santa Catarina

Prof. Maicon Rafael Zatelli, Dr.
Avaliador
Universidade Federal de Santa Catarina

Li-Yao Xia, Dr.
Avaliador
University of Edinburgh

Eduardo Henke

Implementing a programming language with a dependent type system

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Supervisor: Prof. Alvaro Junio Pereira Franco, Dr.
Co-supervisor: Li-Yao Xia, Dr.

Florianópolis
16 de novembro de 2022

RESUMO

O objetivo principal desse projeto é implementar uma linguagem de programação com um sistema de *tipos dependentes*. Extende-se o cálculo *lambda* com *tipos dependentes*, o que possibilita a escrita de programas que não apenas realizam computações, mas também que permite a prova da corretude de seu comportamento.

Palavras-chave: Tipos dependentes, sistema de tipos, cálculo lambda, linguagem de programação, teoria de tipos.

ABSTRACT

The main goal of this project is to design and implement a *dependently typed* programming language. This work consists in an extension of the lambda calculus with dependent types, which allows us to write programs that not only have the ability to perform computations, but whose correctness can also be proven.

Keywords: Dependent types, type system, lambda calculus, programming language, type theory.

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

Software development is a big area, which it is being progressively valued over time. However, due to the complexity of maintaining and developing large software systems, *bugs* are frequent. Where there is a discrepancy in what the developer expected to happen, with what was written in the software.

Because of this, we can see the importance of being able to guarantee that the software works as we expect, and it is also for this that the semantic analysis part of a language is used, where we can verify if the behavior expected by the developer reflects with what was in fact written.

In the semantic analysis phase, the compiler checks if the code written by the developer is in accordance with the problem modeling. One of the most basic mechanisms for this is *types*, which we use to define which variables are of a given set of possible values.

With a simple type system (**tapl**), we can verify that a variable of type *string* cannot receive a value of type *int*.

With a more advanced type system, i.e. with dependent types (**advancedtapl**) we can:

- specify business invariants that will be statically checked in the code, for example:
  - in a banking system, during a withdrawal operation, the amount withdrawn cannot be greater than the balance from a bank account:

```
-- given the account balance, a amount and a proof
-- that the amount is less than the balance
-- perform the operation
withdraw_from :
      (account : Account) →
      (amount : Nat) →
      (amount <= account.balance) →
      Nat
```

  - in most programs, we can have a list of elements, and we can have a function that returns the first element of the list, but what happens if the list is empty? We can use dependent types to specify that the list cannot be empty[1]:

```
head :
      -- given any type A
      (A : Type) →
```

[1] In this case, the first two parameters are being passed explicitly, which can be cumbersome to the developer's experience, that's why most dependently-typed languages have some *inference* mechanisms, which allows the compiler to infer some parameters being passed, like the element type and length of the vector

```
        -- given any number
        (n : Nat) →
        -- given a vector of type A, with length n+1
        -- (note that this means that even if n is 0,
        --- succ n, will be 1, and the vector will have
        -- at least one element, our language
        -- STATICALLY INVALIDATES all uses of head on an empty list )
        Vec A (succ n) →
        -- return the first element of the vector
        A

-- returns the length of the resulting vector as a type
append : (A : Type) → (n : Nat) → (m : Nat) → Vec A m → Vec A n
    → Vec A (plus m n)
append = ... -- implementation is ommitted
```

- prove properties (and theorems) about the code (e.g. prove that an operation inserting an element into an ordered list does not change the order of the list).

1.1 EXISTING WORK

Famous examples of dependently-typed languages are *Agda* (**agda**), *Idris* (**idris**), *Coq* (**coq**) and *Lean* (**lean**).

1.1.1 Data types

One example of a Agda program is shown below:

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

_+_ : Nat → Nat → Nat
zero + m = m
suc n + m = suc (n + m)
```

We have defined a data type `Nat` that has two constructors, `zero` and `suc` of a `Nat`, which are used to represent the natural numbers. Addition is defined as a function that takes two `Nat` and returns a `Nat`, by recursively removing the `suc` from the first parameter until it reaches `zero`.

We'll show how a vector and a function to get its first element can be defined in Agda:

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

head : {A : Set}{n : Nat} → Vec A (suc n) → A
head (x :: xs) = x
```

We have defined `Vec`, which is a *type constructor*, not a type by itself. We need to apply two arguments for it to be a type, the first one is a *type* representing the vector's element's type and the second one is a *natural number*, representing the vector's length. We now have that `Vec Nat 3` is a type, and that `Vec Nat 2` is also a type, but they are both different from each other.

We have also defined two ways to build *instances* of the `Vec A n` type, the first one is the empty vector, which is represented by the `[]` constructor, it returns an instance of the type `Vec A zero` for any type `A`. And the second one is the `_::_` constructor, which takes an *implicit parameter* `n` representing the length of the vector, a element of type `A`, a vector of type `Vec A n`, and returns a vector of type `Vec A (suc n)`. An example of a vector is `5 :: []`, which has type `Vec Nat (suc zero)`.

The `head` function takes an implicit parameter `A` representing the inner type of the vector, another implicit parameter `n` representing the length of the vector, a vector of type `Vec A (suc n)` and returns an element of type `A`. Because the vector is statically guaranteed to not be empty, Agda can check that the only constructor that can generate an element of type `Vec A (suc n)`, is the `_::_` constructor, with that we can use it to get the first element of the vector.

1.2 THIS WORK

This work aims to design and implement a programming language with such dependent type system. It will provide a deeper understanding of how such languages work behind the scenes. As the focus is on the type system, we will present only the type-checker of such language, that will be used to statically verify properties about a program.

2 THEORETICAL BASIS

To explain how this thesis is implemented we're going to provide some background in lambda calculus, inference rules and some type theories.

2.1 LAMBDA CALCULUS

When we don't have the tool of abstraction, calculations such as of Figure 1 seem complex to follow. That's why we can abstract the underlying common concepts and define a function to capture that common abstraction(as defined in Figure 2).

```
(3*2*1) + (7*6*5*4*3*2*1) + (5*4*3*2*1)
```

Figure 1 – Calculation of a number

```
factorial = λn . if n == 0 then 0 else n * (factorial (n - 1))
```

Figure 2 – Definition of the factorial function

Now we can use the previously defined function to more concisely define the previously cumbersome calculation, as shown in Figure 3.

```
(factorial 3) + (factorial 7) + (factorial 5)
```

Figure 3 – Simplified calculation of a number

We've captured an essential understanding of the calculation, and abstracted it into a concept that we can later reuse.

Lambda calculus captures the essential mechanisms of a programming language based on a few simple rules (abstraction, application). It was invented by Alonzo Church in the 1920s (**tapl**) and it is a *logical calculus* or *formal system*, whose terms are generated by the following grammar:

t ::= x	variable
| λx.t	abstraction
| t t	application

Figure 4 – Lambda calculus grammar

A few example terms of the above grammar:

- $\mathtt{x}$, a simple(unbounded) variable
- $\lambda\mathtt{x}.\mathtt{x}$, the identity function
- $\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}$
- $(\lambda\mathtt{x}.\mathtt{x})\ \mathtt{y}$, applying $\mathtt{y}$ to the identity function

2.1.1 Evaluation rules

This calculus primary benefit is in its evaluation, or computation. For example, let's take the above mentioned term and evaluate it: $(\lambda\mathtt{x}.\mathtt{x})\ \mathtt{y}$ in one step evaluates to $\mathtt{y}$, because we're applying the identity function to the variable $\mathtt{y}$. But how does that work for all terms? How can we formalize it?

Definition 2.1.1 (Evaluation)**.** $\mathtt{a} \to \mathtt{b}$, a term $\mathtt{a}$ can evaluate to term $\mathtt{b}$

For that we need inference rules. Inference rules are syntactic transformation rules, i.e. they only need to check the form of the term and with that form they transform the original term to something else. We denote them like this, having the premises[1] above the bar and the conclusions/consequent below it:

$$\frac{\mathtt{a} + 0}{\mathtt{a}}(\text{Add-Zero})$$

Definition 2.1.2 (Value)**.** For a term to be a value, it must have the form of a function, i.e. only terms that follow the following structure are considered to be values: $\lambda\mathtt{x}.\mathtt{t}$. We denote them as terms beginning with v, e.g. $\mathtt{v}_1$

Definition 2.1.3 (Substitution)**.** The process of substituting a variable $\mathtt{x}$ for another term $\mathtt{y}$ in a given term $\mathtt{t}$ is denoted by $[\mathtt{x} \mapsto \mathtt{y}]\mathtt{t}$

* * *

We'll be denoting the evaluation of a lambda calculus terms using a system of inference rules written below:

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \to \mathtt{t}_1'\ \mathtt{t}_2}(\text{E-App1})$$

Given a term $\mathtt{t}_1\ \mathtt{t}_2$(applying the argument $\mathtt{t}_2$ to the function $\mathtt{t}_1$), if the term $\mathtt{t}_1$ can evaluate to another term $\mathtt{t}_1'$, we can rewrite the original term to $\mathtt{t}_1'\ \mathtt{t}_2$.

[1] when we don't need any pre-conditions we simply write nothing above the bar

$$\frac{\texttt{t}_2 \rightarrow \texttt{t}_2'}{\texttt{v}_1\ \texttt{t}_2 \rightarrow \texttt{v}_1\ \texttt{t}_2'}\text{(E-App2)}$$

Given a term $\texttt{v}_1\ \texttt{t}_2$(applying the argument $\texttt{t}_2$ to a term which is a value $\texttt{v}_1$), if the term $\texttt{t}_2$ can evaluate to another term $\texttt{t}_2'$, we can rewrite the original term to $\texttt{v}_1\ \texttt{t}_2'$.

$$\frac{}{\lambda\texttt{x}.\texttt{t}_{12}\ \texttt{v}_2 \rightarrow [\texttt{x} \mapsto \texttt{v}_2]\texttt{t}_{12}}\text{(E-AppAbs)}$$

Whenever we have an application and the argument is a value, we can rewrite the original term to the body of the function, while replacing the parameter($\texttt{x}$) to the argument($\texttt{v}_2$).[2,3]

Definition 2.1.4 (Normal form)**.** When we have no more evaluation steps to perform, i.e. none of the above rules can be applied, and the resulting term is a *value* we say that the term is in *normal form*. For example the term $\lambda\texttt{x}.\texttt{x}$ is in normal form.

At first it does not seem that this simple system can compute the same terms that an ordinary programming language can, but it turns out we can encode numbers, data structures(lists, sets, etc.), *if* expressions (**tapl**). It was proved that this model is equivalent to a Turing Machine (**lambda-church**).

2.2 SIMPLY-TYPED LAMBDA CALCULUS

Up until now, we've only had the notion of evaluation of terms, we're able to define computation steps and an engine can run those for us. But without much insight into what *"types"* of terms we're writing we can easily make the underlying evaluation engine try to compute a program that never halts (**tapl**), or if we add to the lambda calculus' rules operations that can only be applied to numbers and feed them with booleans that can also make the engine stuck, i.e. passing a program that can't be computed.

Those problems can be solved if we somehow inspect the original program before evaluating it, and if desired properties can be derived from only the program specification, checking those properties against the code. One way to do it is by the concept of *types* (**tapl**).

Types allow us to define a set of possible values a term may have during runtime. When we talk about a variable having a type of `Nat`(Natural numbers set), it is telling that during runtime that variable can only possess the values `0, 1, 2, ...`. When we define a variable of type `Bool`, the only possible values in runtime are `True, False`.

[2] This rule is also called a *beta reduction* (**tapl**).

[3] Note that we don't have any preconditions for this rule, whenever we *syntactically* have an application whose argument is a value, we can reduce it.

We can also annotate various parts of our program that allows this checker(which we'll call typechecker from now), to verify the correctness of our program.

2.2.1 Extension of the calculus

Consider this extended version of lambda calculus:

$$
\begin{array}{lll}
\mathtt{t} ::= \mathtt{x} & & \mathtt{T} ::= \mathtt{Nat} \\
\quad \mid \lambda \mathtt{x} : \mathtt{T} \,.\, \mathtt{t} & & \quad \mid \mathtt{Bool} \\
\quad \mid \mathtt{t}\ \mathtt{t} & & \quad \mid \mathtt{T} \rightarrow \mathtt{T} \\
\quad \mid \mathtt{true} & & \\
\quad \mid \mathtt{false} & \Gamma ::= \emptyset & \\
\quad \mid \mathtt{if\ t\ then\ t\ else\ t} & \quad \mid \Gamma, \mathtt{x} : \mathtt{T} & \\
\quad \mid \mathtt{0} & \mathtt{v} ::= \lambda \mathtt{x} : \mathtt{T} \,.\, \mathtt{t} & \\
\quad \mid \mathtt{succ\ t} & &
\end{array}
$$

Figure 5 – Extended lambda calculus grammar

With its evaluation rules as stated in Figure 6:

- Rule $\mathtt{E-IfTrue}$, whenever we *syntactically* find an if expression, whose condition is `true`, we can evaluate that expression to the first branch(*then*) of the if expression, i.e. $\mathtt{t}_2$.

- Rule $\mathtt{E-IfFalse}$, whenever we *syntactically* find an if expression, whose condition is `false`, we can evaluate that expression to the second branch(*else*) of the if expression, i.e. $\mathtt{t}_3$.

- Rule $\mathtt{E-Succ}$, whenever we *syntactically* find a `succ` expression on a term `t`, we have to check the condition that `t` evaluates to another term $\mathtt{t}'$, if that's the case we can evaluate the `succ t` expression to `succ` $\mathtt{t}'$.

$$\frac{\mathtt{t}_1 \to \mathtt{t}'_1}{\mathtt{t}_1\ \mathtt{t}_2 \to \mathtt{t}'_1\ \mathtt{t}_2}(\text{E-App1})$$

$$\frac{\mathtt{t}_2 \to \mathtt{t}'_2}{\mathtt{v}_1\ \mathtt{t}_2 \to \mathtt{v}_1\ \mathtt{t}'_2}(\text{E-App2})$$

$$\frac{}{\lambda \mathtt{x}.\mathtt{t}_{12}\ \mathtt{v}_2 \to [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}}(\text{E-AppAbs})$$

$$\frac{}{\mathtt{if\ true\ then\ t}_2\ \mathtt{else\ t}_3 \to \mathtt{t}_2}(\text{E-IfTrue})$$

$$\frac{}{\mathtt{if\ false\ then\ t}_2\ \mathtt{else\ t}_3 \to \mathtt{t}_3}(\text{E-IfFalse})$$

$$\frac{\mathtt{t} \to \mathtt{t}'}{\mathtt{succ\ t} \to \mathtt{succ\ t}'}(\text{E-Succ})$$

Figure 6 – Extended lambda calculus evaluation rules

2.2.2 Typing rules

The typing relation for the *extended* part of the extended lambda calculus, written $\mathtt{t} : \mathtt{T}$ is defined by inference rules assigning types to terms as stated in Figure 7:

- Rule $\mathtt{T} - \mathtt{True}$, whenever we *syntactically* find a `true` term, we can assign it the type `Bool`.

- Rule $\mathtt{T} - \mathtt{False}$, whenever we *syntactically* find a `false` term, we can assign it the type `Bool`.

- Rule $\mathtt{T} - \mathtt{If}$, whenever we *syntactically* find an if expression, we need to check the premises that the condition is a `Bool` term, and the two branches are of the same type. If that's the case, we can assign the if expression the type of the two branches.

- Rule $\mathtt{T} - \mathtt{Zero}$, whenever we *syntactically* find a `0` term, we can assign it the type `Nat`.

- Rule $\mathtt{T} - \mathtt{Succ}$, whenever we *syntactically* find a `succ` expression on a term $\mathtt{t}_1$, we have to check the condition that $\mathtt{t}_1$ has type `Nat`, if that's the case we can assign the $\mathtt{succ\ t}_1$ expression the type `Nat`.

$$\frac{}{\texttt{true} : \texttt{Bool}}(\text{T-True})$$

$$\frac{}{\texttt{false} : \texttt{Bool}}(\text{T-False})$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 : \texttt{T}}(\text{T-If})$$

$$\frac{}{\texttt{0} : \texttt{Nat}}(\text{T-Zero})$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ t}_1 : \texttt{Nat}}(\text{T-Succ})$$

Figure 7 – Extended lambda calculus typing rules

Definition 2.2.1 (Well-typed term)**.** A term that we can assign a type to is called a *well-typed* term. For example, the term `if true then 0 else succ 0` is well-typed, because it has type `Nat`, by the following proof tree:

$$\frac{\texttt{true} : \texttt{Bool} \qquad \texttt{0} : \texttt{Nat} \qquad \dfrac{\texttt{0} : \texttt{Nat}}{\texttt{succ 0} : \texttt{Nat}}(\text{T-Succ})}{\texttt{if true then 0 else succ 0} : \texttt{Nat}}(\text{T-If})$$

Terms that are not well-typed, i.e. cannot be assigned a type given the rules above, for example the term `succ true` won't evaluate to a value[4].

* * *

We've shown above how to check the types of the *extended* part of the extended lambda calculus(e.g. for `true`, `0`, `succ`, etc.), but we haven't shown for the *core* part of the lambda calculus(e.g. for $\lambda\texttt{x.t}$, $\texttt{t}_1\ \texttt{t}_2$, etc.). For that we need the concept of a typing context, which will change the typing rules to include a context to aid in this process.

Definition 2.2.2. Context A typing context Γ is a sequence of variables and their associated types. The empty context is written as $\emptyset$. We can extend an existing context Γ by adding a new variable with its associated type to it: $\Gamma, \texttt{x} : \texttt{T}$.[5]

The existing typing rule will change from a two-place relation $\texttt{x} : \texttt{T}$ to a three-place relation $\Gamma \vdash \texttt{x} : \texttt{T}$, meaning that `x` has type `T` under the context Γ, which will provide assumptions to the types of the free variables[6] in `x`.

[4] we call those terms stuck, in a more formal definition, they are terms in normal form(no more evaluation rules apply) who are not values

[5] we will assume that each of the variables in this list are distinct from each other, so that there will always be at most one assumption about any variable's type.

[6] variables that are not bound to any lambda binder

The typing rules for the *core* part of the lambda calculus, now written as $\Gamma \vdash \mathtt{t} : \mathtt{T}$ are defined in Figure 8:

- Rule $\mathtt{T-Var}$, to check if a variable $\mathtt{x}$ has type $\mathtt{T}$ in the context Γ, we check if $\mathtt{x} : \mathtt{T}$ is in Γ.
- Rule $\mathtt{T-Abs}$, a lambda expression $\lambda \mathtt{x} : \mathtt{T_1}.\mathtt{t_2}$ has type $\mathtt{T_1} \to \mathtt{T_2}$ in context Γ, if when we extend the context Γ to include the variable $\mathtt{x}$ with type $\mathtt{T_1}$, we can check if $\mathtt{t_2}$ has type $\mathtt{T_2}$ in the extended context.
- Rule $\mathtt{T-App}$, a function application $\mathtt{t_1}\ \mathtt{t_2}$ has type $\mathtt{T_{12}}$ in context Γ, if we can check that $\mathtt{t_1}$ has type $\mathtt{T_{11}} \to \mathtt{T_{12}}$ in the context Γ, and if we can check that $\mathtt{t_2}$ has type $\mathtt{T_{11}}$ in the context Γ.

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}}(\text{T-Var})$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T_1}.\ \mathtt{t_2} : \mathtt{T_1} \to \mathtt{T_2}}(\text{T-Abs})$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}} \to \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}}(\text{T-App})$$

Figure 8 – Core lambda calculus typing rules

With that we can now typecheck that the following program is valid:

$$(\lambda \mathtt{x} : \mathtt{Nat}.\mathtt{succ}\ \mathtt{x})\ (\mathtt{succ}\ \mathtt{0})$$

Figure 9 – Well-typed program

$$\frac{\dfrac{\dfrac{\dfrac{\mathtt{x} : \mathtt{Nat} \in \Gamma}{\mathtt{x} : \mathtt{Nat}}(\text{T-Var})}{\mathtt{succ}\ \mathtt{x} : \mathtt{Nat}}(\text{T-Succ})}{\Gamma \vdash (\lambda \mathtt{x} : \mathtt{Nat}.\mathtt{succ}\ \mathtt{x}) : \mathtt{Nat} \to \mathtt{Nat}}(\text{T-Abs}) \qquad \dfrac{\mathtt{0} : \mathtt{Nat}}{(\mathtt{succ}\ \mathtt{0}) : \mathtt{Nat}}(\text{T-Succ})}{(\lambda \mathtt{x} : \mathtt{Nat}.\mathtt{succ}\ \mathtt{x})\ (\mathtt{succ}\ \mathtt{0}) : \mathtt{Nat}}(\text{T-App})$$

Figure 10 – Well-typed program typing proof tree

And we can't build a similar proof tree for the following program, because it is invalid:

$$(\lambda \mathtt{x} : \mathtt{Nat.succ\ x})\ \mathtt{true}$$

Figure 11 – Not well-typed program

2.3 DEPENDENTLY-TYPED LAMBDA CALCULUS

The simply-typed system provides us some basic tools to define types, but we need to be able to define types for more complex terms.

Dependent types allow types to depend on the terms themselves, i.e. we don't have this stark distinction of types and terms. That allows us greater freedom in specifying types, which are a foundation on which our typechecker can verify the correctness of our code (**advancedtapl**).

Like many dependently-typed languages, we'll show the typing rules(Figure 13) and a grammar(Figure 12) with the same *syntax* for the terms and types, however for clarity we'll be using lowercase letters for terms and uppercase letters for their types.

$$\begin{array}{rll}
\mathtt{t}, \mathtt{T} ::= & \mathtt{x} & \text{variable} \\
| & \lambda \mathtt{x.t} & \text{abstraction} \\
| & \mathtt{t\ t} & \text{application} \\
| & (\mathtt{t} : \mathtt{T}) \to \mathtt{T} & \text{dependent function type} \\
| & \mathbf{Type} & \text{the "type" of types}
\end{array}$$

Figure 12 – Dependently-typed lambda calculus grammar

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} (\text{T-Var})$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{y} : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_1 : \mathbf{Type}}{\Gamma \vdash \lambda \mathtt{x.y} : (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2} (\text{T-Lambda})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : (\mathtt{x} : \mathtt{T}_{11}) \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{T}_{12}} (\text{T-App})$$

$$\frac{\Gamma \vdash \mathtt{T}_1 : \mathbf{Type} \qquad \Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{T}_2 : \mathbf{Type}}{\Gamma \vdash (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2 : \mathbf{Type}} (\text{T-Pi})$$

$$\frac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}} (\text{T-Type})$$

Figure 13 – Dependently-typed lambda calculus typing rules

The typing rules $\mathtt{T-Var}$, $\mathtt{T-Lambda}$ and $\mathtt{T-App}$ are very similar to those in the simply-typed lambda calculus, but the main difference is that the function type, has now

a binder variable for the argument. Where before we only had that the type of a function is $\mathtt{T}_1 \to \mathtt{T}_2$, now we have that the type of a function[7] is $(\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2$. That means that the body of the type can depend on the type variable $\mathtt{x}$, e.g. the type of a function that inserts a `Bool` in a length-indexed vector of `Bool`s can be $(\mathtt{n} : \mathtt{Nat}) \to \mathtt{Bool} \to \mathtt{Vec\ Bool}\ (\mathtt{n} + 1)$. The other rules are explained as follows:

- Rule $\mathtt{T} - \mathtt{Pi}$, a Pi type $(\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2$ has type **Type** in context Γ, only if $\mathtt{T}_1$ also has type **Type** in context Γ, and if $\mathtt{T}_2$ has type **Type** in the extended context $\Gamma, \mathtt{x} : \mathtt{T}_1$.
- Rule $\mathtt{T} - \mathtt{Type}$, the term **Type** also has type **Type**.

We'll add two extensions to our grammar to aid in creating programs:

$$
\begin{array}{rll}
\mathtt{t}, \mathtt{T} ::= & \ldots & \\
& \mid \mathtt{t} : \mathtt{T} & \text{type annotation} \\
& \mid \mathtt{name} = \mathtt{t} & \text{assigning a name to a term}
\end{array}
$$

Figure 14 – Dependently-typed lambda calculus syntax sugar

Which respectively mean:

- An expression can be annotated with a type, e.g. $\mathtt{x} : \mathtt{Nat}$, and that will trigger the typechecker to check that $\Gamma \vdash \mathtt{x} : \mathtt{Nat}$ given the underlying context.
- A name can be assigned to a term, e.g. $\mathtt{id} = \lambda \mathtt{x}.\mathtt{x}$.

* * *

With that we can write this polymorphic identity function program annotated with its type and its associated typing proof tree:

$$
\begin{aligned}
&\mathtt{id} : (\mathtt{x} : \mathbf{Type}) \to (\mathtt{y} : \mathtt{x}) \to \mathtt{x} \\
&\mathtt{id} = \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}
\end{aligned}
$$

We can derive a proof tree proving that the type of `id` is in fact what was annotated:

$$
\dfrac{\Gamma, \mathtt{x} : \mathbf{Type} \vdash \dfrac{\dfrac{\mathtt{x} : \mathbf{Type} \in \Gamma}{\mathtt{x} : \mathbf{Type}}(\text{T-Var}) \qquad \dfrac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}}(\text{T-Type})}{\Gamma, \mathtt{y} : \mathtt{x} \vdash \mathtt{y} : \mathtt{x}}(\text{T-Lambda})}{\Gamma \vdash \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y} : (\mathtt{x} : \mathbf{Type}) \to (\mathtt{y} : \mathtt{x}) \to \mathtt{x}}(\text{T-Lambda})
$$

The goal of this project is to derive the typing proof tree automatically.

[7] formally called a *Pi* type

2.3.1 Definitional Type Equality

2.3.1.1 Motivation

In languages with dependent types, it is often necessary to equate types that are not merely alpha-equivalent[8]. This is because more expressions need to type check in these languages. For example, a type of length-indexed vector might be `Vec A n`, where `A` is the type of the vector's elements, and `n` is the length of the vector. We could have a safe head operation that would allow us to access the first element of the vector, as long as it isn't empty and a append operation that would allow us to append elements to the vector.

```
head : (A : Type) → (n : Nat) → Vec A (succ n) → A
head = ... -- implementation is ommitted

-- returns the length of the resulting vector as a type
append : (A : Type) → (n : Nat) → (m : Nat) → Vec A m → Vec A n → Vec A (
    plus m n)
append = ... -- implementation is ommitted
```

Observe that the following program would compile with the existing theory:

```
v' : Vec Bool (succ 0)
v' = Cons True VNil

h : Bool
h = head Bool 0 v'
```

Because the type of v' is `Vec Bool (succ 0)` matches exactly what the `head` function expected:

```
head : (A : Type) → (n : Nat) → Vec A (succ n) → A
head Bool : (n : Nat) → Vec Bool (succ n) → Bool
head Bool 0 : Vec Bool (succ 0) → Bool
head Bool 0 v' : Bool
```

However the application of `head Bool 1 (append Bool 1 1 v' v')`, wouldn't type-check, observe why:

```
v' : Vec Bool (succ 0)
append : (A : Type) → (n : Nat) → (m : Nat) → Vec A m → Vec A n → Vec A (
    plus m n)
```

[8] alpha-equivalence is the property of two terms being equal are equivalent for all purposes if their only difference is the renaming of bound variables, e.g. λx.x is alpha-equivalent to λy.y (**nlab:alpha-equivalence**)

```
append Bool 1 1 : Vec Bool 1 → Vec Bool 1 → Vec Bool (plus 1 1)
append Bool 1 1 v' v' : Vec Bool (plus 1 1)

head : (A : Type) → (n : Nat) → Vec A (succ n) → A
head Bool : (n : Nat) → Vec Bool (succ n) → Bool
head Bool 1 : Vec Bool (succ 1) → Bool

-- would not type check because head expects:
--   (Vec Bool (succ 1))
-- but we have:
--   (Vec Bool (plus 1 1))
head Bool 1 (append Bool 1 1 v' v') : ???
```

It seems alpha-equivalence is not enough to type check this program, we need to be able to equate `Vec Bool (succ 1)` and `Vec Bool (plus 1 1)`. And that seems to require some number of steps of computation to be able to do so. Definitional type equality is the tool we need to also equate those types of terms.

2.3.1.2 Definition

Definition 2.3.1 (Definitional equality)**.** Definitional equality is a judgement of the form: $\Gamma \vdash A = B$. Classically, definitional equality is called intensional equality[9]. However in this paper we'll define it to mean both intensional **and** computational equality[10] (**nlab:equality**).

This judgement is defined by the properties stated in Figure 15. Rule $E - Beta$ ensures that beta-equivalence is contained in this judgement, because terms that evaluate to each other should be equal. Rules $E{-}Refl$, $E{-}Sym$, and $E{-}Trans$ allows this judgement to be considered an equivalence relation (**oplss**).

[9] intensional equality is the relation generated by abbreviatory definitions, changes of bound variables and the principle of substituting equals for equals (**nlab:equality**)

[10] computational equality is the relation generated by various reduction rules, e.g. *beta reduction* (**nlab:equality**)

$$\frac{}{\Gamma \vdash (\lambda \mathtt{x.a})\ \mathtt{b} = [\mathtt{x} \mapsto \mathtt{b}]\mathtt{a}}(\text{E-Beta})$$

$$\frac{}{\Gamma \vdash \mathtt{A} = \mathtt{A}}(\text{E-Refl})$$

$$\frac{\Gamma \vdash \mathtt{A} = \mathtt{B}}{\Gamma \vdash \mathtt{B} = \mathtt{A}}(\text{E-Sym})$$

$$\frac{\Gamma \vdash \mathtt{A}_1 = \mathtt{A}_2 \qquad \Gamma \vdash \mathtt{A}_2 = \mathtt{A}_3}{\Gamma \vdash \mathtt{A}_1 = \mathtt{A}_3}(\text{E-Trans})$$

$$\frac{\Gamma \vdash \mathtt{A}_1 = \mathtt{A}_2 \qquad \Gamma, \mathtt{x} : \mathtt{A}_1 \vdash \mathtt{B}_1 = \mathtt{B}_2}{\Gamma \vdash (\mathtt{x} : \mathtt{A}_1) \to \mathtt{B}_1 : (\mathtt{x} : \mathtt{A}_2) \to \mathtt{B}_2}(\text{E-Pi})$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{A}_1 \vdash \mathtt{b}_1 = \mathtt{b}_2}{\Gamma \vdash \lambda \mathtt{x}.\mathtt{b}_1 : \lambda \mathtt{x}.\mathtt{b}_2}(\text{E-Lam})$$

$$\frac{\Gamma \vdash \mathtt{a}_1 = \mathtt{a}_2 \qquad \Gamma \vdash \mathtt{b}_1 = \mathtt{b}_2}{\Gamma \vdash \mathtt{a}_1\ \mathtt{b}_1 = \mathtt{a}_2\ \mathtt{b}_2}(\text{E-App})$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{b} : \mathtt{B} \qquad \Gamma \vdash \mathtt{a}_1 = \mathtt{a}_2}{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{a}_1]\mathtt{b} = [\mathtt{x} \mapsto \mathtt{a}_2]\mathtt{b}}(\text{E-Lift})$$

$$\frac{\Gamma \vdash \mathtt{a}_1 = \mathtt{a}_2}{\Gamma \vdash (\mathtt{a}_1 : \mathtt{A}) = \mathtt{a}_2}(\text{E-Annot})$$

Figure 15 – Inference rules for the definitional type equality judgement

With those rules we can now successfully type-check the program, because `plus 1 1` will evaluate to `succ 1`.

3 IMPLEMENTATION

The rules presented on section 2.3 were developed to allow typing proof trees to be built, however they were not developed with saying *how* these proof trees can be built, i.e. these rules are not syntax-directed, we can not devise a decidable algorithm based on these rules alone. Because of that we need to revise the existing rules and transform them into syntax-directed rules.

3.1 BIDIRECTIONAL TYPE SYSTEM

Definition 3.1.1 (Judgement)**.** A judgement is a proposition that is made on a given term. The previously defined *typing rule* is a form of judgement.(**nlab:judgment**)

One way to do that is to define a bidirectional type system, which is based on two types of judgements:

- *Type inference*, denoted as $\Gamma \vdash \mathtt{x} \Rightarrow \mathtt{T}$ which given a term `x` and a context Γ, will *infer*(return) the type of that term.

```
inferType :: Context → Term → Maybe Type
```

- *Type checking*, denoted as $\Gamma \vdash \mathtt{x} \Leftarrow \mathtt{T}$ which given a term `x`, a context Γ and a type `T`, will *check* that the term is of the given type.

```
checkType :: Context → Term → Type → Bool
```

We now can develop inference rules that represent the algorithmically-feasible version of the 2.3 rules. Both types of judgements will be used, the type inference rules can use the type checking rules, and vice versa.

$$\frac{\mathtt{x} : \mathtt{A} \in \Gamma}{\Gamma \vdash \mathtt{x} \Rightarrow \mathtt{A}}\text{(I-Var)}$$

$$\frac{\Gamma \vdash \mathtt{a} \Rightarrow (\mathtt{x} : \mathtt{A}) \to \mathtt{B} \qquad \Gamma \vdash \mathtt{b} \Leftarrow \mathtt{A}}{\Gamma \vdash \mathtt{a\ b} \Rightarrow [\mathtt{x} \mapsto \mathtt{b}]\mathtt{B}}\text{(I-App-Simple)}$$

$$\frac{\Gamma \vdash \mathtt{A} \Leftarrow \mathbf{Type} \qquad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{B} \Leftarrow \mathbf{Type}}{\Gamma \vdash (\mathtt{x} : \mathtt{A}) \to \mathtt{B} \Rightarrow \mathbf{Type}}\text{(I-Pi)}$$

$$\frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type}}\text{(I-Type)}$$

$$\frac{\Gamma \vdash \mathtt{a} \Leftarrow \mathtt{A}}{\Gamma \vdash (\mathtt{a} : \mathtt{A}) \Rightarrow \mathtt{A}}\text{(I-Annot)}$$

Figure 16 – Bidirectional type-inference rules

$$\frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{a} \Leftarrow \mathtt{B} \qquad \Gamma \vdash \mathtt{A} \Leftarrow \mathbf{Type}}{\Gamma \vdash \lambda \mathtt{x.a} \Leftarrow (\mathtt{x} : \mathtt{A}) \to \mathtt{B}}(\text{C-Lambda})$$

$$\frac{\Gamma \vdash \mathtt{a} \Rightarrow \mathtt{A}}{\Gamma \vdash \mathtt{a} \Leftarrow \mathtt{A}}(\text{C-Infer-Simple})$$

Figure 17 – Bidirectional type-checking rules

3.2 MODULES

A module consists of a list of top-level declarations, that are defined once and used throughout the module:

$$\begin{array}{rll} \mathtt{Decl} ::= & \mathtt{t} : \mathtt{T} & \text{type-signature} \\ \mid & \mathtt{t} = \mathtt{t} & \text{definition} \end{array}$$

Figure 18 – Module grammar

Here's some examples:

```
-- a type signature, linking "id" with that type
id : (a : Type) → a → a

-- a definition, linking "id" with that term/value
id = a. x. x
```

3.2.1 Type-checking

To type-check a module we use the `checkType` function when we have an associated type-signature for a given name, and we use the `inferType` function when we do not have a type-signature for a given name. Also we put the existing declarations in the scope of all subsequent type-checks.

```
Decl ::= ...
       | data TConName Telescope : Type where ConstructorDef+    data definition

ConstructorDef ::= DConName Telescope
        TConName ::= identifier              type constructor name
        DConName ::= identifier              data constructor name
       Telescope ::= Decl*

t, T ::= ...
       | TConName t*   constructing a data type by applying a list of arguments
       | DConName t*                                  instance of a data type
```

Figure 19 – Data type grammar

Definition 3.3.1 (Telescope)**.** A telescope is a list of declarations, it is called like that because of the scoping behavior of this structure (**oplss**). The scope of each variable merges with all of the subsequent ones, e.g.:

```
-- notice the scoping behavior of this structure
(A : Type) (n : Nat) (v : Vec A n)
-- here we're showing that the telescope can use both
-- a type annotation, and a definition, which symbolizes
-- a constraint being put upon any variable, in this
-- case, it requires n to be of value Zero
(n : Nat) [n = Zero]
```

Here we're showing an example, by first defining some data types, then using it as terms:

```
data Bool : Type where {
      False, -- False is a case from Bool, it is a data type constructor
      True   -- as well as True
} -- Bool is a data definition with an empty telescope

data Nat : Type where {
      Zero,
      -- Succ is a data type constructor with
      -- a Telescope of one argument(another Nat number)
      Succ of (Nat)
}
```

```
data List (A : Type) : Type where {
      Nil,
      -- Cons is a data type constructor with
      -- a Telescope of two arguments(element of A and a List of A)
      Cons of (A) (List A)
}

-- t is annotated with a type, constructed by applying a
-- list of arguments(Bool) to the type constructor(List)
t : List Bool
-- t is defined using a data type constructor(Cons) by applying
-- a list of arguments(True, Nil) to it
t = Cons True Nil
```

Next, we're showing that the telescope can have a constraint on the previous values, with a *Vector* data type:

```
data Vec (A : Type) (n : Nat) : Type where {
  -- Nil is a data type constructor with
  -- a constraint that n(provided by the type) must be zero
  Nil of [n = Zero],
  -- Cons is a data type constructor with a Telescope of
  -- three arguments(Nat m, element of A, and a Vec of A with length m)
  -- and one constraint that the Vector built by Cons,
  -- must have length m+1, m being the underlying vector
  -- which this Cons was built upon
  Cons of (m : Nat) (A) (Vec A m) [n = Succ m]
}

-- t is annotated with a type, constructed by applying a
-- list of arguments(Bool) to the type constructor(List)
t : Vec Bool 2
-- t is defined using a data type constructor(Cons) by applying
-- a list of arguments(True, Nil) to it
t = Cons 1 True (Cons 0 False Nil)
```

3.3.0.1 Type-checking

- type and data constructor applications are checked according to the telescope

3.3.1 Pattern-matching

$$
\begin{aligned}
\texttt{t}, \texttt{T} ::= & \ \ldots \\
& \mid \textit{case}\ \texttt{t}\ \textit{of}\ \texttt{Case+} \quad \text{pattern matching of a term} \\
\texttt{Case} ::= & \ \texttt{Pattern} \ \rightarrow \ \texttt{t} \\
\texttt{Pattern} ::= & \ \texttt{DConName PatVar+} \\
& \mid \texttt{PatVar} \\
\texttt{PatVar} ::= & \ \texttt{identifier}
\end{aligned}
$$

Figure 20 – Data type with pattern matching grammar

Here's some examples of pattern matching:

```
not : Bool → Bool
not = b. case b of {
  False → True,
  True → False
}

plus : Nat → Nat → Nat
plus = a. b. case a of {
  Zero → b,
  Succ a' → Succ (plus a' b)
}
```

3.3.1.1 Type-checking

- context extension by pattern variables - unification of scrutinee and case - check type of body of case - exhaustivity check

3.4 CODE

The project was developed in *Haskell* using the *Unbound* library for variable substitution(*beta reduction*).

3.4.1 Terms

The terms are represent as a data type.

```
import qualified Unbound.Generics.LocallyNameless as Unbound
```

```
-- Term names for our AST
type TName = Unbound.Name Term

data Term
= Type -- type of types
| Var TName -- variables: x
| Lam (Unbound.Bind TName Term) -- abstractions:  x.a
| App Term Arg
| Pi Type (Unbound.Bind TName Type) -- function types: (x : A)  → B
| Ann Term Type -- "ascription" or "annotated terms": (a: A)
deriving (Show, Generic)

data Arg = Arg {unArg :: Term}
  deriving (Show, Generic, Unbound.Alpha, Unbound.Subst Term)

-- because types and terms are the same in dependent typing,
-- we'll alias them
type Type = Term

data Decl
  = TypeSig TName Type -- a : A
  | Def TName Term  -- a = b
  deriving (Show)

-- a file
data Module = Module
  { declarations :: [Decl]
  }
  deriving (Show)
```

3.4.2 Type-checking

3.4.2.1 Type-checking monad

Instead of using a `Maybe` or `Either` type to represent the result of type checking, we'll use a monad to encapsulate the behaviour of:

- failing to type-check and returning an error message
- getting and setting the state of the typing context

- generating fresh variable names

```
import Control.Monad
import Control.Monad.Except (ExceptT)
import Control.Monad.Reader (ReaderT)
import qualified Unbound.Generics.LocallyNameless as Unbound

type TcMonad = Unbound.FreshMT (ReaderT Env (ExceptT Err IO))

data Env = Env {ctx :: [Decl]}

emptyEnv :: Env
emptyEnv = Env {ctx = []}

extendCtx :: Decl  → TcMonad a  → TcMonad a
extendCtxs :: [Decl]  → TcMonad a  → TcMonad a
lookupTyMaybe :: TName  → TcMonad (Maybe Type)
lookupTy :: TName  → TcMonad Type
-- implementation ...
```

We now have some helper functions that can be used to implement the type-checking rules, e.g. given a name of a variable we can use `lookupTy` to get its type, if it fails to find that variable in the context, it will stop the type-checking process and issue an error.

3.4.2.2 Equality and weak-head normal form

The rules for propositional and definitional equality of terms, along with the weak-head normal forms are defined as follows:

```
equate :: Term  → Term  → TcMonad ()
equate t1 t2 | aeq t1 t2 = return ()
equate t1 t2 = do
  nf1  ← whnf t1
  nf2  ← whnf t2
  case (nf1, nf2) of
    (Lam bnd1, Lam bnd2)  → do
      (_, t1, _, t2)  ← unbind2Plus bnd1 bnd2
      equate t1 t2
    (App a1 a2, App b1 b2)  → do
      equate a1 b1
```

```
      equate (unArg a2) (unArg b2)
    (Pi ty1 bnd1, Pi ty2 bnd2)  → do
      equate ty1 ty2
      (_, t1, _, t2)  ← unbind2Plus bnd1 bnd2
      equate t1 t2
    (Type, Type)  → return ()
    (Var x, Var y) | x == y  → return ()
    (_, _)  → err ["Expected", show nf2, "but found", show nf1]

whnf :: Term  → TcMonad Term
-- strangely WHNF-Var has a different implementation
-- than what was provided in the paper
whnf (Var x) = do
  maybeTm  ← lookupDefMaybe x
  pure (fromMaybe (Var x) maybeTm)
whnf (App a b) = do
  v  ← whnf a
  case v of
    -- WHNF-Lam-Beta
    (Lam bnd)  → do
      (x, a')  ← unbind bnd
      whnf (subst x (unArg b) a')
    -- WHNF-Lam-Cong
    _  → return (App v b)
whnf (Ann tm _) = return tm
-- WHNF-Type, WHNF-Lam, WHNF-Var, WHNF-Pi
whnf tm = return tm
```

3.4.2.3 Type-checking rules

Also the two judgements' signatures will change to:

```
-- given a term return its type(or an error message)
inferType :: Term  → TcMonad Type
-- given a term and a type, check if the term is of the given type
-- if it is, return (), otherwise an error message
checkType :: Term  → Type  → TcMonad ()
```

We'll use a **`tcTerm`** function that will be used to centralize the two judgements. If the second parameter is **`Nothing`**, it will enter into *inference* mode, otherwise it will enter into *type-checking* mode.

```
inferType :: Term → TcMonad Type
inferType t = tcTerm t Nothing

checkType :: Term → Type → TcMonad ()
checkType tm ty = do
  -- Whenever we call checkType we should call it
  -- with a term that has already been reduced to
  -- normal form. This will allow rule c-lam to
  -- match against a literal function type.
  nf ← whnf ty
  ty' ← tcTerm tm (Just nf)

-- | Make sure that the term is a type (i.e. has type 'Type')
tcType :: Term → TcMonad ()
tcType tm = void (checkType tm Type)

tcTerm :: Term → Maybe Type → TcMonad Type
tcTerm (Var x) Nothing = lookupTy x
tcTerm Type Nothing = return Type
tcTerm (Ann tm ty) Nothing = do
  checkType tm ty
  return ty
tcTerm (Pi tyA bnd) Nothing = do
  (x, tyB) ← unbind bnd
  tcType tyA
  extendCtx (TypeSig x tyA) (tcType tyB)
  return Type
tcTerm (App t1 t2) Nothing = do
  ty1 ← inferType t1
  let ensurePi :: Type → TcMonad (TName, Type, Type)
      ensurePi (Ann a _) = ensurePi a
      ensurePi (Pi tyA bnd) = do
        (x, tyB) ← unbind bnd
        return (x, tyA, tyB)
      ensurePi ty = err ["Expected a function type, but found ", show ty]
  nf1 ← whnf ty1
  (x, tyA, tyB) ← ensurePi nf1
  checkType (unArg t2) tyA
  return (subst x (unArg t2) tyB)
```

```
tcTerm (Lam bnd) (Just ty@(Pi tyA bnd')) = do
  tcType tyA
  -- warning: you can't use unbind two times in a row here,
  -- because the variables in the type part and the
  -- term part won't coincide then
  (x, body, _, tyB) ← Unbound.unbind2Plus bnd bnd'
  extendCtx (TypeSig x tyA) (checkType body tyB)
  return ty
tcTerm (Lam _) (Just nf) =
  err ["Lambda expression should have a function type, not ", show nf]
tcTerm tm (Just ty) = do
  ty' ← inferType tm
  ty `equate` ty'
  return ty'
tcTerm tm Nothing =
  err ["Must have a type annotation to check ", show tm]

-- Create a Def if either side normalizes to a single variable
def :: Term → Term → TcMonad [Decl]
def t1 t2 = do
  nf1 ← whnf t1
  nf2 ← whnf t2
  case (nf1, nf2) of
    (Var x, _) → return [Def x nf2]
    (_, Var x) → return [Def x nf1]
    _ → return []
```

3.5 EXAMPLES

The parsing code will be omitted for the sake of brevity. We can now verify that a proof that the logical **and** operator commutes[1]:

```
bool : Type
bool = (x : Type) → x → x → x

true : bool
true = x. y. z. y

false : bool
```

[1] note that to build this proof, definitional equality is needed

```
false = x. y. z. z

cond : bool → (x:Type) → x → x → x
cond = b. b

and : Type → Type → Type
and = p. q. (c: Type) → (p → q → c) → c

conj : (p: Type) → (q:Type) → p → q → and p q
conj = p .q. x .y. c. f. f x y

proj1 : (p: Type) → (q:Type) → and p q → p
proj1 = p. q. a. a p (x. y. x)

proj2 : (p: Type) → (q:Type) → and p q → q
proj2 = p. q. a. a q (x. y. y)

and_commutes : (p:Type) → (q:Type) → and p q → and q p
and_commutes = p. q. a. conj q p (proj2 p q a) (proj1 p q a)
```

When we run our type-checker, we see that it successfully type-checks the proof. However, if we for example, change the type of the `and_commutes` to:

```
and_commutes : (p:Type) → (q:Type) → and p q → and p p
```

We get an error from our type-checker: `Expected Var q but found Var p`