Implementing a programming language with a dependent type system

Eduardo Henke[1]

[1] Departamento de Informática e Estatística –
Universidade Federal de Santa Catarina (UFSC)

eduardo.henke@grad.ufsc.br

Abstract. *The main goal of this project is to explore the potential benefits of advanced type systems, in software development, through an implementation of a* dependently typed *programming language. This could help improve the reliability and safety of software by enabling static verification of arbitrary properties about the code. 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.*

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 reason that the semantic analysis part of a language is used, where we can verify if the behavior expected by the developer reflects 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 [Pierce 2002], 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 [Pierce 2004] we can:

- specify invariants (rules that must hold) 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  - in a banking system, during a withdrawal operation, the amount withdrawn cannot be greater than the balance from a bank account[1]:

```
-- given the account balance, an amount and a proof
-- that the withdrawn amount is less than the balance
```

[1] We use a Haskell-style pseudo-code notation to describe this code, explained in more details in subsection 3.2 and in subsection 2.3

```
-- 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[2]:

```
head :
       -- given any type A
       (A : Type)  →
       -- 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, because we can encode logical propositions as types [nLab authors 2022d], and their respective proofs as code (evidence of that type) [nLab authors 2022c], e.g. prove that an operation inserting an element into an ordered list does not change the order of the list, or prove that a mathematical relation is associative:

```
-- proof that addition on Naturals is associative
-- given any three numbers (m, n, p), we can prove that:
-- (m + n) + p = m + (n + p)
plus_assoc : (m n p: Nat)  → ((plus (plus m n) p) = (plus m (plus n
    p)))
plus_assoc = -- ... proof is ommitted
```

[2]In this case, the first two parameters are being passed explicitly, which can be cumbersome to the developer's experience, that is 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

1.1. Existing work

Some of the most well-known dependently-typed languages are *Agda* [Mörtberg and Norell 2022], *Idris* [Team 2022b], *Coq* [Team 2022a] and *Lean* [Team 2022c], all of which allow for static verification and the ability to prove arbitrary properties about the code. These languages are based on different underlying type theories, which shape the foundations of their type systems and the capacity to specify and prove theorems. Additionally, the focus of these languages can vary; for instance, *Agda* is more geared towards programming, while *Coq* has a stronger emphasis on theorem proving.

1.2. Objectives

The general objective of this thesis is to examine the potential advantages of utilizing advanced type systems, such as dependent types, in software development, through the implementation of a dependently-typed programming language. We will provide a deeper understanding of how such languages work behind the scenes, and how they can be used to statically verify properties about a program.

To support this general objective, we will pursue the following specific objectives:

- Investigate existing work on dependently-typed programming languages and advanced type systems
- Develop a dependently-typed programming language as a proof-of-concept implementation. As the focus is on the type system, we will present only the type-checker of such language.
- Use the language to encode business invariants and demonstrate the ability to prove properties about the code, including through the use of static type checking and automated theorem proving.

We believe that achieving these objectives will provide valuable insights into the potential of using advanced type systems in software development.

2. Theoretical Basis

In this section, we are going to provide some background in lambda calculus, inference rules and some type theories.

2.1. Lambda Calculus

When we do not have the tool of abstraction, calculations such as the following seem complex to follow:

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

That is why we can abstract the underlying common concepts and define a function to capture that common abstraction, as defined here:

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

Now we can use the previously defined function[3] to more concisely define the previously cumbersome calculation, as shown here:

```
(factorial 2) + (factorial 7) + (factorial 5)
```

We have 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 [Pierce 2002] and it is a *logical calculus* or *formal system.*

Definition 2.1 (Term)**.** A *term* is an expression in a given formal language that represents a value or a computation.

The terms of the lambda calculus are generated by the following grammar:

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

Figure 1. Lambda calculus grammar

A few example terms of the above grammar:

- x, a simple (unbounded) variable
- λx.x, the identity function
- λx.λy.x, a function receiving argument x, that returns another function receiving argument y, that returns x
- (λx.x) y, applying 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: (λx.x) y in one step evaluates to y, because we are applying the identity function to the variable y. But how does that work for all terms? How can we formalize it?

Definition 2.2 (Evaluation)**.** a → b, a term a can evaluate to term 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 the inference rules by having the premises[4] above the bar and the conclusions/consequent below it, like the following inference rule symbolizing the property that adding zero to a number does not change the number:

[3]This concept will be later formalized as "abstraction"

[4]When we do not need any pre-conditions we simply write nothing above the bar

$$\frac{\mathtt{a} \in \mathbb{N}}{\mathtt{a} + 0 \to \mathtt{a}} \text{(Add-Zero)}$$

Definition 2.3 (Value)**.** For a term to be a value, it must have the form of a function, i.e. only terms that are an abstraction ($\lambda \mathtt{x}.\mathtt{t}$) are considered to be values. For ease of notation, we will use $\mathtt{t}$ to denote any term, and $\mathtt{v}$ (and its derivatives) to denote values e.g. $\mathtt{v}_1$.

Definition 2.4 (Substitution)**.** The process of substituting a variable $\mathtt{x}$ with another term $\mathtt{y}$ in a given term $\mathtt{t}$ is denoted by $[\mathtt{x} \mapsto \mathtt{y}]\mathtt{t}$

We will 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 $\mathtt{t}_1'\ \mathtt{t}_2$.

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

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

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

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 ($\mathtt{x}$) to the argument ($\mathtt{v}_2$).[5][6]

Definition 2.5 (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 $\lambda \mathtt{x}.\mathtt{x}$ 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 [Pierce 2002]. It was proved that this model is equivalent to a Turing Machine [Church 1941].

[5]This rule is also called a *beta reduction* [Pierce 2002].

[6]Note that we do not have any preconditions for this rule, whenever we *syntactically* have an application whose argument is a value, we can reduce it.

2.2. Simply-Typed Lambda Calculus

Up to now, we have only had the notion of evaluation of terms, we are able to define computation steps and an engine can run those for us. But without much insight into what *"types"* of terms we are writing we can easily make the underlying evaluation engine try to compute a program that never halts [Pierce 2002], 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 cannot 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* [Pierce 2002].

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`.

We can also annotate various parts of our program that allows this checker (which we will call typechecker from now), to verify the correctness of our 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 do not 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 [Pierce 2004].

Like many dependently-typed languages, we will show the typing rules (Figure 3) and a grammar (Figure 2) with the same *syntax* for the terms and types, however for clarity we will be using lowercase letters for terms and uppercase letters for their types.

$\mathtt{t}, \mathtt{T} ::= \mathtt{x}$		variable
	$\mid \lambda \mathtt{x}.\mathtt{t}$	abstraction
	$\mid \mathtt{t}\ \mathtt{t}$	application
	$\mid (\mathtt{t} : \mathtt{T}) \to \mathtt{T}$	dependent function type
	$\mid \mathbf{Type}$	the "type" of types

Figure 2. Dependently-typed lambda calculus grammar

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

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

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

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

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

Figure 3. Dependently-typed lambda calculus typing rules

Definition 2.6 (Context)**.** A typing context, denoted by 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: $\Gamma, \mathtt{x} : \mathtt{T}$.[7]

The existing typing rule will change from a two-place relation $\mathtt{x} : \mathtt{T}$ to a three-place relation $\Gamma \vdash \mathtt{x} : \mathtt{T}$, meaning that $\mathtt{x}$ has type $\mathtt{T}$ under the context Γ, which will provide assumptions to the types of the free variables[8] in $\mathtt{x}$.

Also, 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[9] is $(\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2$. That means that the function's return type expression $\mathtt{T}_2$ can depend on the function's argument $\mathtt{x}$, 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 rules are explained as follows:

- Rule $\mathtt{T} - \mathtt{Var}$, 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 $\mathtt{T} - \mathtt{Lambda}$, a lambda expression $\lambda \mathtt{x}.\mathtt{t}_2$ has type $(\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2$ 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, and also if $\mathtt{T}_1$ is indeed a type.
- Rule $\mathtt{T} - \mathtt{App}$, a function application $\mathtt{t}_1\ \mathtt{t}_2$ has type $\mathtt{T}_{12}$(with the $\mathtt{x}$ variable substituted by the $\mathtt{t}_2$ parameter) in context Γ, if we can check that $\mathtt{t}_1$ has type $(\mathtt{x} : \mathtt{T}_{11}) \to \mathtt{T}_{12}$ in the context Γ, and if we can check that $\mathtt{t}_2$ has type $\mathtt{T}_{11}$ in the context Γ.
- 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 will add two extensions to our grammar to aid in creating programs:

[7] 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.

[8] Variables that are not bound to any lambda binder

[9] Formally called a *Pi* type

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

Figure 4. 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:

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

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

2.3.1. Definitional Type Equality

Definition 2.7 (Judgement)**.** A judgement is a proposition that is made on a given term. The previously defined *typing rule* is a form of judgement.[nLab authors 2022b]

Definition 2.8 (Definitional equality)**.** Definitional equality is a judgement of the form: $\Gamma \vdash \mathtt{A} = \mathtt{B}$. Classically, definitional equality is called intensional equality[10]. However in this thesis we will define it to mean both intensional **and** computational equality[11] [nLab authors 2022a].

This judgement is defined by the properties stated in Figure 5. Rule $\mathtt{E-Beta}$ ensures that beta-equivalence is contained in this judgement, because terms that evaluate to each other should be equal. Rules $\mathtt{E-Refl}$, $\mathtt{E-Sym}$, and $\mathtt{E-Trans}$ allows this judgement to be considered an equivalence relation [Weirich 2022].

[10] Intensional equality is the relation generated by abbreviatory definitions, changes of bound variables and the principle of substituting equals for equals [nLab authors 2022a]

[11] Computational equality is the relation generated by various reduction rules, e.g. *beta reduction* [nLab authors 2022a]

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

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

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

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

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

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

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

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

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

Figure 5. Inference rules for the definitional type equality judgement

3. Implementation

The rules presented on subsection 2.3 were developed to allow typing proof trees to be built, however they were not developed having in mind *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. The reason for that is because rule $\mathtt{T-Lambda}$ is not syntax-directed, it is not clear what is the type of the argument of the function (we extend the context with it, because it is necessary to typecheck the body of the function). Because of that we need to revise the existing rules and transform them into syntax-directed rules.

3.1. Bidirectional type system

One way to automatically derive proof trees for a given program 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 $\mathtt{x}$ 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 $\mathtt{x}$, a context Γ and a type $\mathtt{T}$, 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 subsection 2.3 rules. Both types of judgements will be used, the type inference rules can use the type checking rules, and vice versa.

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

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

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

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

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

Figure 6. Bidirectional type-inference rules

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

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

Figure 7. Bidirectional type-checking rules

3.2. Modules

We will also introduce the concept of modules and top-level declarations to aid in the development of programs. Where a module consists of a list of declarations that are defined once and used throughout the module:

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

Figure 8. Module grammar

Here are some examples:

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

-- a definition, linking "id" with that term/value
id = λa. λx. 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.

3.3. Data Types

We will also introduce the concept of data types, to aid the programmer in having structured data in their program:

```
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 9. Data type grammar

Definition 3.1 (Telescope)**.** A telescope is a list of declarations, it is called like that because of the scoping behavior of this structure [Weirich 2022]. 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 are 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 are 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 are 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)
```

We type-check type and data-type constructor applications according to the telescope and the arguments provided, e.g.:

```
-- compares the telescope (A : Type) (n : Nat)
-- with the arguments provided: Bool, 1
-- Bool is of type Type, and 1 is of type Nat
v : Vec Bool 1
-- compares the telescope (m : Nat) (A : Type) (v : Vec A m)
-- with the arguments provided: 0, True, Nil
```

```
-- 0 is of type Nat, True is of type A (Bool), and Nil is of type Vec Bool
     0
v = Cons 0 True Nil
```

3.3.1. Pattern-matching

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

Figure 10. Data type with pattern matching grammar

Here are some examples of pattern matching:

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

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

To type-check pattern matching cases, we have to:

- extend the typing context with the pattern variables, e.g. when we see `Succ n' -> body`, we should extend the typing context with `n' : Nat` when we type-check the body:

  ```
  n : Nat
  case n of {
    -- we should know that n' has type Nat,
    --  according to Nat's data definition
    Succ n' → plus n' n',
    ...
  }
  ```

- all cases must conform to the same type, e.g.:

  ```
  nat_to_bool : Nat → Bool
  nat_to_bool = λn. case n of {
  ```

```
  -- the type of the body of this case is Bool,
  -- so the type of the whole case is Bool
  Zero  → False,
  -- the same thing with this case
  Succ n'  → True
}

-- invalid code would have been:
-- case n of { Zero  → False, Succ n'  → n' }
```

- check for exhaustiveness, i.e. we should check that there is no other possible case, otherwise, we should throw an error

```
n : Nat
-- this case is exhaustive, because it covers all possible cases
case n of {
  Zero  → False,
  Succ n'  → True
}

-- this case is not exhaustive, because it does not cover Zero
--   case n of {
--     Succ n'  → True
--   }
```

- unify the scrutinee's type (term being matched) with the type of the pattern, e.g. if we are applying a case expression to a `n` which is a `Nat`, and in the `Zero -> body` case, we should replace the type of `a` from `Nat` to `Zero`, in this branch.

3.4. Project's Code

The project was developed in *Haskell* using the *Unbound* library for variable substitution (*beta reduction*), and was heavily inspired by the *pi-forall* [Weirich 2022] language.

3.4.1. Terms

The syntax of the terms, module and declarations are represented as a data type in *Haskell*. Here we have the data type for the terms, which is a recursive data type, as well as the data type for the types, which is simply an alias to the term data type:

```
1 module Syntax where
2
3 type TName = Unbound.Name Term -- Term names for our AST
4 type TCName = String -- type constructor names
5 type DCName = String -- data constructor names
6
7 -- because types and terms are the same in dependent typing,
8 -- we will alias them
9 type Type = Term
```

```haskell

data Term
      = Type -- type of types
      | Var TName -- variables: x
      | Lam (Unbound.Bind TName Term) -- abstractions: λx.a
      | App Term Term -- application: f x
      | Pi Type (Unbound.Bind TName Type) -- fn types: (x : A) → B
      | Ann Term Type -- "ascription" or "annotated terms": (a: A)
      -- Data-type related terms
      | DCon DCName [Term] -- Just True
      | TCon TCName [Term] -- Maybe Bool
      | Case Term [Match] -- case analysis `case a of matches`
      -- Proof related terms
      | TyEq Type Type -- equality type: (plus 0 n) = n
      | Refl -- equality evidence: refl is of type x = x
      | Subst Term Term
```

Next, we have the data type for the modules and top-level declarations:

```haskell
newtype Module = Module {declarations :: [Decl]}

-- a "top-level definition" of a module
data Decl
      = TypeSig TName Type -- a : A
      | Def TName Term -- a = b
      | DataDef TCName Telescope [ConstructorDef] -- data Bool ...
```

And finally, we have the data type for data types and pattern-matching constructs:

```haskell
-- a data constructor has a name and a telescope of arguments
data ConstructorDef = ConstructorDef DCName Telescope
      deriving (Unbound.Alpha, Unbound.Subst Term)

newtype Telescope = Telescope [Decl]
      deriving (Unbound.Alpha, Unbound.Subst Term)

-- represents a case alternative
newtype Match = Match (Unbound.Bind Pattern Term)
      deriving anyclass (Unbound.Alpha, Unbound.Subst Term)

data Pattern
      = PatCon DCName [Pattern]
      | PatVar TName
      deriving (Eq, Generic, Unbound.Alpha, Unbound.Subst Term)
```

3.4.2. Type-checking

Instead of using a `Maybe` or `Either` type to represent the result of type checking, we will 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

```
1  module Environment where
2
3  import Control.Monad
4  import Control.Monad.Except (ExceptT)
5  import Control.Monad.Reader (ReaderT)
6  import qualified Unbound.Generics.LocallyNameless as Unbound
7
8  type TcMonad = Unbound.FreshMT (ReaderT Env (ExceptT Err IO))
9
10 data Env = Env {ctx :: [Decl]}
11
12 emptyEnv :: Env
13 emptyEnv = Env {ctx = []}
14
15 extendCtx :: Decl  → TcMonad a  → TcMonad a
16 extendCtxs :: [Decl]  → TcMonad a  → TcMonad a
17 lookupTyMaybe :: TName  → TcMonad (Maybe Type)
18 lookupTy :: TName  → TcMonad Type
19 -- 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.

Also the two judgements' signatures will change to:

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

The rules for propositional and definitional equality of terms are defined in the `equate` function, which given two terms will check if they are equal or not, if they are not it will throw an error.

```
1  equate :: Term  → Term  → TcMonad ()
2  -- two terms are equal, if they are alpha-equivalent
3  -- i.e., by just properly renaming the variables
4  -- they are the same term
```

```
5  equate t1 t2 | aeq t1 t2 = return ()
6  equate t1 t2 = do
7    nf1 ← whnf t1
8    nf2 ← whnf t2
9    case (nf1, nf2) of
10     (Lam bnd1, Lam bnd2) → do
11       -- get the body of each lambda
12       (_, t1, _, t2) ← unbind2Plus bnd1 bnd2
13       -- lambdas are equal, if their bodies are equal
14       equate t1 t2
15     (App f1 x1, App f2 x2) → do
16       equate f1 f2
17       equate x1 x2
18 -- ... rest of the terms ommited for brevity ...
19     (_, _) → err ["Expected", show nf2, "but found", show nf1]
```

- From line 10 to 14, we are checking that a lambda is equal to another lambda if their bodies are equal, according to rule E-Lam in Figure 5
- From line 15 to 17, we are checking that an application is equal to another application if their function and argument are equal, according to rule E-App in Figure 5

Definition 3.2 (Weak-head normal form)**.** A term is said to be in weak-head normal form, when we apply weak-head normalization, which is when the leftmost, outermost reducible expression is always selected for beta-reduction, and the process is halted as soon as the term begins with something other than a lambda abstraction. [Pierce 2004]

An unformal idea of weak-head normalization, is that it is a subset of the normal beta-reduction (computation) rules, so that we compute just enough to observe the structure of the term, e.g. when I do not care about the result of `factorial 100`, but I do care about its structure, if it is a `Nat` or not.

As shown in the above code section, we also need a function to calculate the weak-head normal form of a given term, which is defined as follows in the `whnf` function:

```
1  whnf :: Term → TcMonad Term
2  -- if we are whnf-inf a variable,
3  whnf (Var x) = do
4    -- look up its definition in the context
5    maybeTm ← lookupDefMaybe x
6    case maybeTm of
7      -- and whnf the definition
8      (Just tm) → whnf tm
9      -- or if there is no definition in the context
10     -- return the variable
11     _ → pure (Var x)
12 -- if we are whnf-ing an application,
```

```
whnf (App a b) = do
  v ← whnf a
  -- check the type of the whnf'd function
  case v of
    -- if it is indeed a lambda abstraction
    (Lam bnd) → do
      (x, a') ← unbind bnd
      -- substitute the argument for the bound variable
      whnf (subst x b a')
    -- otherwise, return the application
    _ → return (App v b)
-- ... rest of the terms ommited for brevity ...
whnf tm = return tm
```

Next we will implement the type-checking rules for the terms and types of our language, which are defined in Figure 6.

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

Here we will just define the `inferType` and the `checkType` function:

```
module Typechecker where

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)
```

Next, we will define the `tcTerm` function, which will be used to implement the type-checking rules:

```
tcTerm :: Term → Maybe Type → TcMonad Type
-- Infer-mode
tcTerm (Var x) Nothing = lookupTy x
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 t2 tyA
  return (subst x t2 tyB)
-- ... rest of the cases ommited for brevity ...
tcTerm tm Nothing = err ["Must have a type annotation to check ", show tm]

-- Check-mode
tcTerm (Lam bnd) (Just ty@(Pi tyA bnd')) = do
  tcType tyA
  (x, body, _, tyB) ← Unbound.unbind2Plus bnd bnd'
  extendCtx (TypeSig x tyA) (checkType body tyB)
  return ty
tcTerm (Lam _) (Just nf) = err ["Lambda expression should be a function"]
-- ... rest of the cases ommited for brevity ...
-- if there is no specific case for *checking* the type of the term
tcTerm tm (Just ty) = do
  ty' ← inferType tm
  ty `equate` ty'
  return ty'
```

We can describe the type-checking rules as follows:

- In inference mode:
  - At line 3, we are *inferring* the type of a variable, by consulting it from the type-checking monad's context, according to the `I − Var` rule in Figure 6.
  - In the function case at line 7, we are *inferring* the type of a dependent function type (pi type), according to the `I − Pi` rule in Figure 6. We check that the argument type is indeed a type, and then we extend the context with the argument type and check that the body's return type is indeed a type. If both checks are successful, we return the type `Type`.
  - In the function case at line 12, we are *inferring* the type of an application, according to the `I − App − Simple` rule as stated in Figure 6. We check that the function term is indeed a function type (pi type), and then we check that the provided argument has the same type as the function's argument type. Finally, we return the function's body type, with the argument substituted.
- While in type-checking mode, when we have the information of what type this term needs to have:
  - In the function case at line 28, we are *checking* the type of a lambda expression to be a dependent function type, according to the `C − Lambda` rule in Figure 6. We check that the function's argument type is indeed a type, and then check that the lambda's body has the same type as the Pi type body return type. If both checks are successful, we return the checked Pi type.
  - In the last case, we are *checking* the type of a term to be equal to a given type, according to the `C − Infer − Simple` rule in Figure 6. We infer the type of the term, and then check if it is equal to the expected type (using the previously defined *equality among types* function). If both checks are successful, we return the inferred type.

3.5. Examples

The parsing code will be omitted for the sake of brevity (but can be found in the Parser.hs file, in the implementation's repository). We are now going to show some examples of the code in this programming language. First of all we are going to define the `Bool`, `Nat` and `Vec` data types along with `Nat`'s `plus` function:

```
data Bool : Type where {
  False,
  True
}

data Nat : Type where {
  Zero,
  Succ of (Nat)
}

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

data Vec (A : Type) (n : Nat) : Type where {
  Nil of [n = 0],
  Cons of (m : Nat) (A) (Vec A m) [n = Succ m]
}

empty_bool_vec : Vec Bool 0
empty_bool_vec = Nil

bool_vec : Vec Bool 2
bool_vec = Cons 1 False (Cons 0 True Nil)
```

Next, we are showing how to build a map function that works on vectors (note the type of the vectors having the parameterized length, on the map function type, showing dependent-types being used), we are buiding a vector and making a proof (static check) of its value:

```
map : (A B : Type) → (n : Nat) → (f : (A → B)) → Vec A n → Vec B n
map = λA B n f v. case v of {
  Nil → Nil,
  Cons n' h t → Cons n' (f h) (map A B n' f t)
}

nat_vec : Vec Nat 2
nat_vec = map Bool Nat 2 λ(b. case b of {False → 10, True → 20})
    bool_vec

-- proof that it is a vector like [10, 20]
p_nat_vec : nat_vec = (Cons 1 10 (Cons 0 20 Nil))
```

```
p_nat_vec = refl
```

Another use of dependent-types can be shown with the `concat` function, showing the returned vector having the length of the sum of the input vectors:

```
concat : (A : Type) → (m n: Nat) → Vec A m → Vec A n → Vec A (plus m
    n)
concat = λA m n a b. case a of {
  Nil → b,
  Cons m' h t → Cons (plus m' n) h (concat A m' n t b)
}
```

The aforementioned `head` function can also be written in our language, along with a proof showing that the returned element is indeed the first element of the vector, and a comment stating a impossible term (the `head` of an empty vector)):

```
head : (A : Type) → (n : Nat) → Vec A (Succ n) → A
head = λA n v. case v of {
  Cons _ h _ → h
}

p_head : (head Nat 1 nat_vec) = 10
p_head = refl

-- the following lines do not type-check
-- because when we pass 0 as argument
-- the length of the expected vec argument is (Succ 0), which
-- does not match with the length of the actual vector (which is 0)
-- p_head_empty : (head Bool 0 empty_bool_vec) = False
-- p_head_empty = refl
```

3.6. Evaluation

In this section, we presented the implementation of a proof-of-concept programming language with dependent types. We demonstrated the use of dependent types to specify invariants and prove properties about code, and showed how these capabilities can improve code reliability and maintainability. We provided several examples of code written in our language, including functions that manipulate vectors and perform basic operations like mapping and concatenation.

Overall, our implementation illustrates the potential benefits of advanced type systems like dependent types, and demonstrates their ability to enable static checking of invariants and the encoding of logical propositions as types and proofs as code. While our proof-of-concept language is not intended for production use, it serves as a useful case study for exploring the capabilities and potential of dependent types in software development.

3.6.1. Results

In comparing our proof-of-concept language to *Agda*, a well-established dependently-typed language, it is evident that our language lacks many of the advanced features

and capabilities of *Agda*. Our language does not support features like *universes*, which allow for the organization and hierarchy of types and their associated terms, and *implicit parameters*, which allow for the implicit passing of arguments in function definitions. Additionally, our language does not support *coinductive types*, which allow for the definition of infinite data structures and the representation of infinite processes. These features are present in *Agda* and can be useful for a wide range of applications. It is clear that our language is limited in its capabilities and may not be suitable for all types of projects that may benefit from the use of dependent types.

4. Conclusion

In this thesis, we aimed to examine the potential advantages of utilizing advanced type systems, such as dependent types, in software development through the implementation of a dependently-typed programming language. Our specific objectives included investigating existing work on dependently-typed languages, developing a proof-of-concept language, using the language to encode business invariants and demonstrate the ability to prove properties about the code.

To support these objectives, we presented a dependently-typed programming language with a type-checker and a parser, written in Haskell. The language is based on the lambda calculus and has a simple syntax, with a type system that supports dependent types. While the language is not intended for production use, it serves as a proof-of-concept and a starting point for learning about dependently-typed programming languages.

Through the implementation and analysis of this language, we were able to gain a deeper understanding of how advanced type systems such as dependent types work and their potential benefits for improving the reliability and safety of software. However, it is important to note that the use of dependent types also comes with trade-offs and limitations, including the added burden of having to prove properties about the code and the potential for increased complexity in the codebase and development process. While the benefits of dependent types may outweigh the costs in certain cases, such as for critical or safety-sensitive systems, it is important to carefully evaluate the trade-offs and limitations for each specific application.

Overall, our findings suggest that dependent types have the potential to significantly improve the reliability and safety of software, but more research and development is needed to fully realize this potential in practice. The language and related resources can be found at `https://github.com/eduhenke/dep-tt`.

References

[Church 1941] Church, A. (1941). The calculi of lambda-conversion.

[Mörtberg and Norell 2022] Mörtberg, A. and Norell, U. (2022). Agda. `https://agda.readthedocs.io/en/v2.6.2.2/`. [Online; accessed 25-October-2022].

[nLab authors 2022a] nLab authors (2022a). equality. `http://ncatlab.org/nlab/show/equality`. Revision 27.

[nLab authors 2022b] nLab authors (2022b). judgment. http://ncatlab.org/nlab/show/judgment. Revision 22.

[nLab authors 2022c] nLab authors (2022c). proofs as programs. https://ncatlab.org/nlab/show/proofs+as+programs. Revision 5.

[nLab authors 2022d] nLab authors (2022d). propositions as types. https://ncatlab.org/nlab/show/propositions+as+types. Revision 41.

[Pierce 2002] Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press, 1st edition.

[Pierce 2004] Pierce, B. C. (2004). *Advanced Topics in Types and Programming Languages*. The MIT Press.

[Team 2022a] Team, T. C. D. (2022a). Coq. https://coq.inria.fr/. [Online; accessed 25-October-2022].

[Team 2022b] Team, T. I. D. (2022b). Idris. https://www.idris-lang.org/. [Online; accessed 25-October-2022].

[Team 2022c] Team, T. L. D. (2022c). Lean. https://leanprover.github.io/. [Online; accessed 25-October-2022].

[Weirich 2022] Weirich, S. (2022). Implementing Dependent Types in pi-forall. https://github.com/sweirich/pi-forall/blob/2022/doc/oplss.pdf. [Online; accessed 25-July-2022].