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
28 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, 28 de novembro de 2022.

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
28 de novembro de 2022

RESUMO

O objetivo principal desse projeto é implementar uma linguagem de programação com um sistema de *tipos dependentes*. O cálculo *lambda* é expandido com *tipos dependentes*, o que possibilita a escrita de programas que realizam computações e também permitem 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 FIGURES

CONTENTS

1 IMPLEMENTATION

The rules presented on section **??** 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 $\mathtt{T} - \mathtt{Lambda}$ 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.

1.1 BIDIRECTIONAL TYPE SYSTEM

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 $\mathtt{x}$ 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 **??** 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 1 – Bidirectional type-inference rules

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

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

Figure 2 – Bidirectional type-checking rules

1.2 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:

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

Figure 3 – 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
```

1.2.1 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.

1.3 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 4 – Data type grammar

Definition 1.3.1 (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)
```

1.3.0.1 Type-checking

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
```

1.3.1 Pattern-matching

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

Figure 5 – 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)
}
```

1.3.1.1 Type-checking

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.

1.4 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.

1.4.1 Terms

The syntax of the terms, module and declarations are represented as a data type in *Haskell*:

```haskell
module Syntax where

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

type TCName = String -- type constructor names

type DCName = String -- data constructor names

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

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) -- function 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
     deriving (Generic)

-- Modules
newtype Module = Module {declarations :: [Decl]}
     deriving (Show)

-- a "top-level definition" of a module
data Decl
```

```haskell
    = TypeSig TName Type -- a : A
    | Def TName Term -- a = b
    | DataDef TCName Telescope [ConstructorDef] -- data Bool ...
    deriving (Generic, Unbound.Alpha, Unbound.Subst Term)

-- Data-types

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

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

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

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

1.4.2 Type-checking

1.4.2.1 Type-checking monad

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

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

```haskell
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.

1.4.2.2 Equality and weak-head normal form

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.

```haskell
equate :: Term  → Term  → TcMonad ()
-- two terms are equal, if they are alpha-equivalent
-- i.e., by just properly renaming the variables
-- they are the same term
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
      -- get the body of each lambda
      (_, t1, _, t2)  ← unbind2Plus bnd1 bnd2
      -- lambdas are equal, if their bodies are equal
      equate t1 t2
    -- application is equal when:
    -- - both functions are equal and
  -- - both args are equal
  (App f1 x1, App f2 x2)  → do
    equate f1 f2
    equate x1 x2
  -- ... rest of the terms ommited for brevity ...
  (_, _)  → err ["Expected", show nf2, "but found", show nf1]
```

Definition 1.4.1 (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:

```
whnf :: Term  → TcMonad Term
-- if we are whnf-inf a variable,
whnf (Var x) = do
  -- look up its definition in the context
  maybeTm  ← lookupDefMaybe x
  case maybeTm of
    -- and whnf the definition
    (Just tm)  → whnf tm
    -- or if there is no definition in the context
    -- return the variable
    _  → pure (Var x)
-- 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
```

1.4.2.3 *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 will 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
-- Infer-mode
-- looks up the var type from context
tcTerm (Var x) Nothing = lookupTy x
-- when we reach an annotation, checks the type of the term
-- to be the annotated type
tcTerm (Ann tm ty) Nothing = do
```

```
  checkType tm ty
  return ty
tcTerm (Pi tyA bnd) Nothing = do
  (x, tyB) ← unbind bnd
  -- check that the argument type is indeed a type
  tcType tyA
  -- extend the context with x:A, then check
  -- that the body's return type is indeed a type
  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
  -- checks if the "function term" is indeed a function and
  -- gets the bounded arg var, the argument and body's type
  (x, tyA, tyB) ← ensurePi nf1
  -- checks that the provided argument t2, has the same type as the
  -- function's argument type
  checkType t2 tyA
  -- returns the function's body type, with the argument substituted
  return (subst x t2 tyB)
-- ... rest of the terms 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
  -- verifies that the function's argument type is indeed a type
  tcType tyA
  -- unbinds the function's argument, body and body's return type
  (x, body, _, tyB) ← Unbound.unbind2Plus bnd bnd'
  -- checks that the function's body has the same type as the
  -- function's body's return type (provided by the Pi type)
  extendCtx (TypeSig x tyA) (checkType body tyB)
  return ty
tcTerm (Lam _) (Just nf) = err ["Lambda expression should have a function type, not", sho
-- ... rest of the terms ommited for brevity ...
-- if there is no specific case for *checking* the type of the term
tcTerm tm (Just ty) = do
  -- infer its type
  ty' ← inferType tm
  -- check if the inferred type is equal to the expected type
  ty `equate` ty'
  -- return the inferred type
  return ty'
```

1.5 EXAMPLES

The parsing code will be omitted for the sake of brevity. 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
```

REFERENCES

PIERCE, Benjamin C. **Advanced Topics in Types and Programming Languages**. [S.l.]: The MIT Press, 2004. ISBN 0262162288.

WEIRICH, Stephanie. **Implementing Dependent Types in pi-forall**. [S.l.: s.n.], 2022. `https://github.com/sweirich/pi-forall/blob/2022/doc/oplss.pdf`. [Online; accessed 25-July-2022].