Type systems
(course notes)

Ambrus Kaposi

Eötvös Loránd University, Budapest, Hungary

akaposi@inf.elte.hu

January 1, 2023

Contents

Introduction

These are the notes for the course on type systems for programming languages given at Eötvös Loránd University (autumn semesters of 2020, 2021, 2022 master's course on computer science). Previous versions of this course used Robert Harper's book [11] (between 2016 autumn and 2019 autumn) and István Csörnyei's books [10, 9] (before 2016). Other excellent books on the topic are [16, 17, 21].

The uniqueness of these notes is that they define (programming) languages as algebraic theories. This is described in Chapter 1 and [12]. We don't yet know how to describe all languages at this level of abstraction, this is one reason for covering much less material than the above-mentioned books. Most of the material is formalised in Agda, it relies on a shallowly embedded version of setoid type theory [3] using rewrite rules [8].

These notes are compiled from literate Agda files and are available at https://bitbucket.org/akaposi/typesystems. They are under continuous development.

Special thanks to Bálint Kocsis and Márk Széles who did the first version of the formalisation during a summer internship in 2020. Thanks to István Donkó who developed materials for the tutorials and contributed significantly to the formalisation. Thanks to the following students who fixed errors and typos: András Kovács, Kálmán Kostenszky, Álmos Sőnfeld, Erik Sulcz, András Vadász. Thanks to the students of the course for their excellent questions.

Chapter 1

A language of closed expressions

Learning goals of this chapter

- Levels of abstraction when defining a programming language: strings, sequences of lexical elements, abstract syntax trees, well-typed syntax, well-typed syntax with equations
- Programs as trees: models, syntax, defining functions by recursion, dependent models, proving properties by induction
- Well-typed syntax, type inference, standard model and the standard interpreter
- Well-typed syntax with equations, normalisation, completeness and stability of normalisation

In this chapter, we study a very simple programming language called NatBool which contains e.g. the following program.

```
if isZero (num 0 + num 1) then false else isZero (num 0)
```

Every program is either a numeric or a boolean expression. Numbers can be formed using `num i` where `i` is a natural number. Booleans are `true` or `false`. We have the usual if-then-else operator, addition and an isZero operator which says whether a number is 0. The above program evaluates in the following steps (each line is a step).

```
if isZero (num 0 + num 1) then false else isZero (num 0)
if isZero (num 1) then false else isZero (num 0)
if false then false else isZero (num 0)
isZero (num 0)
true
```

We will describe this language in several iterations. Each iteration will be a more precise description of the language: strings, sequences of lexical elements, abstract syntax trees, well-typed syntax and well-typed syntax with equations. See Figures 1.1, 1.2.

1.1 String

As a first approximation, we say that a program is a string, that is, a sequence of (ASCII) characters. This is how we write programs on a computer. Any string is a program.

Many strings do not correspond to meaningful programs in our language such as `num 3 - num 2` as we don't have subtraction. Also, there are different strings which represent the same program. For example, `isZero (num 1)` and `isZero  (num 1)` are different as strings but should be the same programs as the extra spaces after `isZero` shouldn't matter.

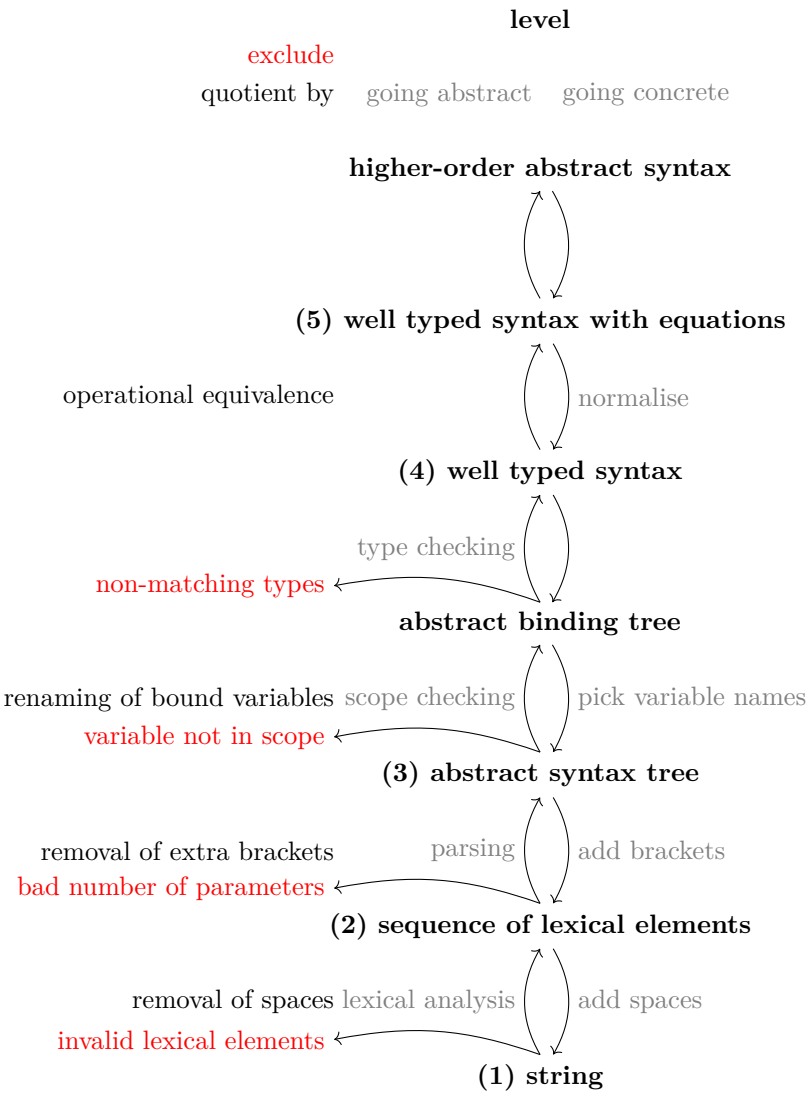
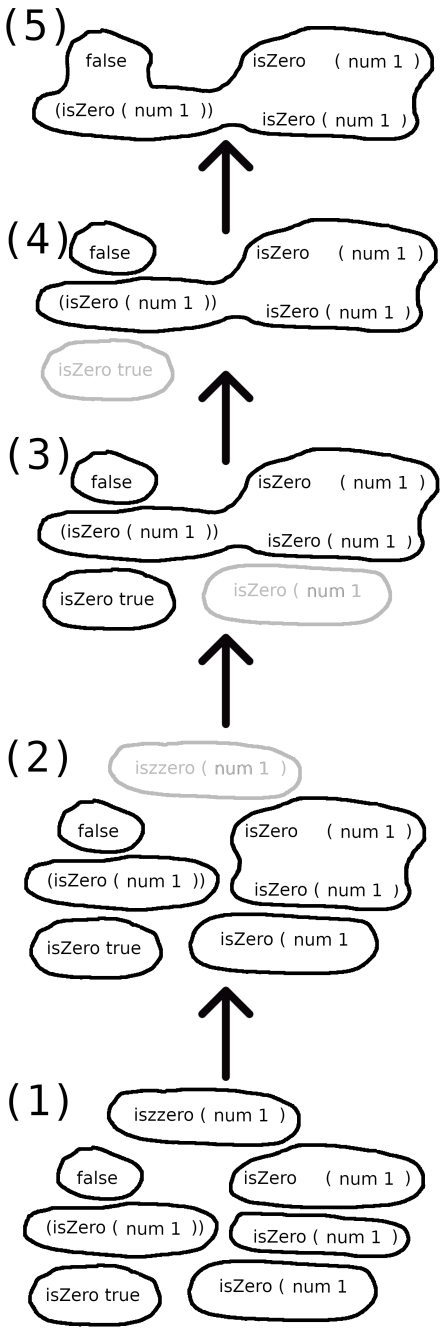


Figure 1.1: Different levels of abstraction when defining a programming language and transformations between levels. At higher levels, certain programs are excluded and others identified. Abstract binding trees are sometimes called well-scoped syntax trees, see Chapter 2. We don't cover higher-order abstract syntax in these notes.



(5) well-typed syntax with equations

Ty	: Set
Tm	: Ty → Set
Bool	: Ty
true	: Tm Bool
false	: Tm Bool
ite	: Tm Bool → Tm A → Tm A → Tm A
num	: ℕ → Tm Nat
isZero	: Tm Nat → Tm Bool
_+_	: Tm Nat → Tm Nat → Tm Nat
$\text{ite}\beta_1$	: ite true u v = u
$\text{ite}\beta_2$	: ite false u v = v
$\text{isZero}\beta_1$	: isZero (num 0) = true
$\text{isZero}\beta_2$	: isZero (num (1+n)) = false
$+\beta$	: num m + num n = num (m + n)

(4) well-typed syntax

Ty	: Set
Tm	: Ty → Set
Bool	: Ty
true	: Tm Bool
false	: Tm Bool
ite	: Tm Bool → Tm A → Tm A → Tm A
num	: ℕ → Tm Nat
isZero	: Tm Nat → Tm Bool
_+_	: Tm Nat → Tm Nat → Tm Nat

(3) abstract syntax tree

Tm	: Set
true	: Tm
false	: Tm
ite	: Tm → Tm → Tm → Tm
num	: ℕ → Tm
isZero	: Tm → Tm
_+_	: Tm → Tm → Tm

(2) list of the following lexical elements:

(,), true, false, if, then, else, num, isZero, +, 0, 1, 2, 3, ...

(1) any string

Figure 1.2: Left: example programs at different levels of abstractions in the NatBool expression language. Each bubble represents a separate program. Right: description of the NatBool expression language at levels (1)–(5).

Instead of describing which strings are meaningful programs and defining an equivalence relation for identifying strings that represent the same program, we will describe programs using a more abstract structure.

1.2 Sequence of lexical elements

We describe NatBool by the following lexical elements.

$$(,),\ \mathsf{true},\ \mathsf{false},\ \mathsf{if},\ \mathsf{then},\ \mathsf{else},\ \mathsf{num},\ \mathsf{isZero},\ +,\ 0,\ 1,\ 2,\ ...$$

The ... part means that any natural number is a lexical element. A program is an arbitrary sequence made of these.

Our example program

$$\mathsf{if\ isZero\ (num\ 0\ +\ num\ 1)\ then\ false\ else\ isZero\ (num\ 0)}$$

is the following sequence.

$$[\mathsf{if},\ \mathsf{isZero},\ (,\ \mathsf{num},\ 0,\ +,\ \mathsf{num},\ 0,\ ),\ \mathsf{then},\ \mathsf{false},\ \mathsf{else},\ \mathsf{isZero},\ (,\ \mathsf{num},\ 0,\ )]$$

Now we have much fewer programs and num 3 - num 2 is not a program anymore because there is no lexical element for -. Any two programs given as strings which differ only in the number of spaces will end up as the same program at this level: isZero (num 1) and isZero (num 1) are both given by the sequence [isZero, (, num, 1,)].

However, we still have meaningless programs, e.g. [(, true] (there is no closing parenthesis) or [num, 1, +] (+ needs two arguments), and so on. Also, there are programs which could be identified, e.g. [(, true,)] and [true] (the parentheses are redundant in the former).

Again, to solve these issues, we move to a higher-level representation of programs.

Note that we can always obtain a string from a sequence of lexical elements by simply printing the sequence. In the other direction, we can implement a lexical analyser (lexer) which turns a string into a sequence of lexical elements or returns an error.

Exercise 1.1 (compulsory)**.** *Mark those strings for which the lexical analyser outputs an error. For those strings that are accepted, write down the list of lexical elements it outputs.*

if true then true else num 0
if true then num 0 else num 0
if num 0 then num 0 else num 0
if if then num 0 else num 0
true + zero
true + num 1
true + isZero
isZero + 0

For convenience, we usually write strings instead of lists of lexical elements. In this case, we specify that we are interpreting the strings as lists of lexical elements.

Exercise 1.2 (compulsory)**.** *Which equations of lists of lexical elements hold?*

num 3 $\stackrel{?}{=}$ num(3)
isZero 0 $\stackrel{?}{=}$ true
if 3 then (false) $\stackrel{?}{=}$ if 3 then (false)
(if 3 then (false)) $\stackrel{?}{=}$ if 3 then (false)
then then if $\stackrel{?}{=}$ then then if

Exercise 1.3. *Write a lexical analyser for NatBool: the input is a string, the output is either a sequence of lexical elements or an error.*

1.3 Abstract syntax tree

We define our language by the following BNF grammar (where N means a natural number):

T ::= true | false | if T then T else T | num N | isZero T | T + T

In Haskell, we would write this as follows (using Int for natural numbers).

```
data Tm = True | False | Ite Tm Tm Tm | Num Int | IsZero Tm | Tm :+: Tm
```

We will use the following Agda notation for the same definition. Here every operator is followed by its *arity*: the number of arguments it takes. We call elements of Tm terms. We write ite instead of if-then-else and +o instead of + for convenience.

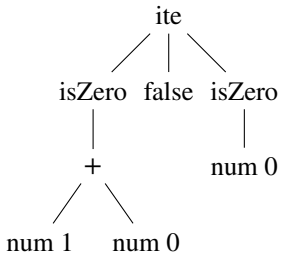
```
data Tm : Set where
  true   : Tm
  false  : Tm
  ite    : Tm → Tm → Tm → Tm
  num    : ℕ → Tm
  isZero : Tm → Tm
  _+o_   : Tm → Tm → Tm
```

In this description, we added the information that if-then-else has three, isZero one and addition two Tm arguments while num has a natural number argument. Programs are now trees which have true, false or num i at their leaves and they can have ternary branching with ite at the branching node, unary branching with isZero at the node or binary branching with + at the node.

Our example program

if isZero (num 0 + num 1) then false else isZero (num 0)

is depicted as follows.

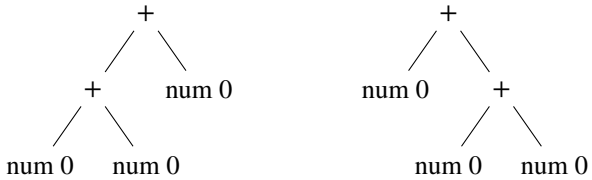


Instead of drawing trees we usually use linear notation to save space:

```
ite (isZero (num 0 +o num 1)) false (isZero (num 0))
```

Programs such as (true and num 1 + which were valid as sequences of lexical elements do not correspond to any tree, and the programs (true) and true correspond to the same tree.

Note that (num 0 + num 0) + num 0 and num 0 + (num 0 + num 0) are different trees.



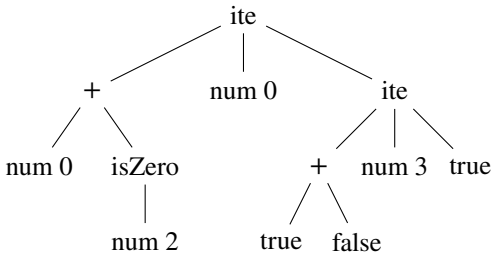
Even if the intuitive meaning of both is the number zero, they are different as trees. An example of parentheses which even changes the intuitive meaning of an expression is $1 + (2 * 3) = 7 \neq 9 = (1 + 2) * 3$ (this example is not in the NatBool language as it has multiplication).

The *meta theory* (meta language) is the language that we use to speak about the *object theory* (object language). Our meta language is Agda (and sometimes English but we can translate all of our English arguments to Agda), our object language is NatBool.

Exercise 1.4 (compulsory)**.** *Draw the trees depicting the following programs:*

(true + true) + true
((true + true) + true) + true
(true + (true + true)) + true
true + ((true + true) + true)
ite (true + true) ((true + true) + true) (true + (true + true))

Exercise 1.5 (compulsory)**.** *Write down the following tree with linear notation:*



Exercise 1.6 (compulsory)**.** *Here are some lists of lexical elements (written as strings). Which ones are accepted by the parser for NatBool?*

isZero (num 1) + (num 1)
isZero (num 1 + num 1)
isZero (num 2) 3
num 1 + isZero
1 + num 1
true + 1

Exercise 1.7 (compulsory)**.** *Write down the following BNF notations in Agda using* data *(as we did for* Tm*).* N *means natural numbers, as before.*

T ::= op0 | op1 T | op2 T T | op3 T T T | op4 T T T T

A ::= a | fb B
B ::= fa A

V ::= vzero | vsuc V
E ::= num N | E < E | E = E | var V
C ::= V := E | while E S | if E then S else S
S ::= empty | C colon S

Exercise 1.8. *Write a parser for NatBool: a program which given a sequence of lexical elements, outputs an element of* Tm *or an error.*

Exercise 1.9 (compulsory)**.** *Which of the following lists of lexical elements are accepted by a parser and produce a* Tm*?*

if true then true else num 0
if true then num 0 else num 0
if num 0 then num 0 else num 0
if num 0 then (num 0 else num 0)
if if then num 0 else num 0
true + false
true + (num 1))
true + (num 1)
(true + isZero)
isZero + num 0

Exercise 1.10 (compulsory)**.** *Which of the following abstract syntax trees are equal to* num 0 + (num 1 + num 1)*?*

num 0 + ((num 1) + (num 1))
(num 0 + ((num 1) + (num 1)))
(num 0 + num 1) + num 1
((num 0 + num 1) + num 1)
num 1 + num 1
num 2
(num 0 + num 0) + (num 1 + num 1)
ite true (num 0 + (num 1 + num 1)) (num 0)
ite true (num 0 + (num 1 + num 1)) (num 0 + (num 1 + num 1))
ite true (num 0 + (num 1 + num 1)) false

1.3.1 Recursion

We call a set A with two elements, a ternary operation on A (an $A \to A \to A \to A$ function), an infinite sequence of As (a $\mathbb{N} \to A$ function), an endofunction (an $A \to A$ function) and a binary operation on A (an $A \to A \to A$ function) a *model* of *NatBoolAST* (sometimes called a NatBoolAST algebra). We fix a notation for this. A NatBoolAST model (or simply model) is a record with the following fields.

```
record Model {ℓ} : Set (lsuc ℓ) where
  field
    Tm    : Set ℓ
    true  : Tm
    false : Tm
    ite   : Tm → Tm → Tm → Tm
    num   : ℕ → Tm
    isZero : Tm → Tm
    _+o_  : Tm → Tm → Tm
```

Note that we used the same names as for the previously defined trees. The trees are called the *syntax* of NatBool. The syntax is also a model, we denote it by I, its components are I.Tm, I.true, I.false, and so on. The syntax has the property that it can be interpreted into any other model (it is an initial model, hence the name I): given a model M, there is a homomorphism (a function that commutes with all operators) called the recursor from I to M which we denote M.⟦_⟧. Other names for the recursor: iterator, fold, catamorphism, nondependent eliminator, interpreter, evaluator.

```
⟦_⟧ : I.Tm → Tm
⟦ I.true        ⟧ = true
⟦ I.false       ⟧ = false
⟦ I.ite t t' t" ⟧ = ite ⟦ t ⟧ ⟦ t' ⟧ ⟦ t" ⟧
⟦ I.num n       ⟧ = num n
⟦ I.isZero t    ⟧ = isZero ⟦ t ⟧
⟦ t I.+o t'     ⟧ = ⟦ t ⟧ +o ⟦ t' ⟧
```

The syntax is also called the free model: we start with I.true, I.false, I.num i (for all is) and then we can freely apply operators on these and then on those terms produced by the operators, and so on, and all syntactic terms are obtained this way.

For example, the NatBoolAST model H (height) is the following.

```
H : Model
H = record
  { Tm    = ℕ
  ; true  = 0
  ; false = 0
  ; ite   = λ n n' n" → 1 + max n (max n' n")
  ; num   = λ n → 0
  ; isZero = λ n → 1 + n
```

```
  ; _+o_  = λ n n' → 1 + max n n'
  }
```

Note that we distinguish two addition operations: +o is the addition operator in a model (H in this case), while + is just addition of (metatheoretic) natural numbers.

If our example term ite (isZero (zero + suc zero)) false (isZero zero) is interpreted in the model H, we obtain the height of the tree.

```
H.ite (H.isZero (H.num 0 H.+o H.num 1))                H.false (H.isZero (H.num 0))  ≡≡
1 + max (H.isZero (H.num 0 H.+o H.num 1))              (max H.false (H.isZero (H.num 0))) ≡≡
1 + max (1 + (H.num 0 H.+o H.num 1))                   (max H.false (H.isZero (H.num 0))) ≡≡
1 + max (1 + (1 + max (H.num 0) (H.num 1)))            (max H.false (H.isZero (H.num 0))) ≡≡
1 + max (1 + (1 + max 0 0))                            (max H.false (H.isZero (H.num 0))) ≡≡
1 + max (1 + (1 + 0))                                  (max H.false (H.isZero (H.num 0))) ≡≡
1 + max 2                                              (max H.false (H.isZero (H.num 0))) ≡≡
1 + max 2                                              (max 0 (H.isZero (H.num 0)))       ≡≡
1 + max 2                                              (max 0 (1 + (H.num 0)))            ≡≡
1 + max 2                                              (max 0 (1 + 1))                    ≡≡
1 + max 2                                              (max 0 2)                          ≡≡
1 + max 2                                              2                                  ≡≡
1 + 2                                                                                     ≡≡
3                                                                                         ∎
```

The metatheoretic set of booleans 𝟚 has two elements tt and ff. We denote logical disjunction by ∨.

We define another model T where terms are elements of 𝟚 and a term is tt if and only if it contains true.

```
T : Model
T = record
  { Tm    = 𝟚
  ; true  = tt
  ; false = ff
  ; ite   = λ t t' t'' → t ∨ t' ∨ t''
  ; num   = λ n → ff
  ; isZero = λ t → t
  ; _+o_  = λ t t' → t ∨ t'
  }
```

Now

```
T.ite (T.isZero (T.num 0 T.+o T.num 1))    T.false    (T.isZero (T.num 0)) ≡≡
      (T.isZero (T.num 0 T.+o T.num 1)) ∨ T.false ∨ (T.isZero (T.num 0)) ≡≡
      (T.num 0 T.+o T.num 1)            ∨ ff      ∨ (T.num 0)            ≡≡
      (T.num 0 ∨    T.num 1)            ∨ ff      ∨ ff                   ≡≡
      (ff      ∨    ff)                 ∨ ff      ∨ ff                   ≡≡
      ff                                                                  ∎
```

and

```
T.ite T.true (T.num 0) T.false ≡≡ T.true ∨ (T.num 0) ∨ T.false ≡≡ tt ∨ ff ∨ ff ≡≡ tt
```

Exercise 1.11 (compulsory). *Define a model where a term is a natural number and if we interpret a syntactic term in the model, we get the number of* trues *in the tree. For example,* false +o num 3 = 0, true +o num 3 = 1, true +o true = 2.

Exercise 1.12 (compulsory). *Define a model which calculates the number of nodes in a tree, e.g.* num 1 = 1, isZero (num 1) = 2, isZero (num 1) + true = 4.

We have that for any model M

```
M.⟦ I.ite (I.isZero (I.num 0 I.+o I.num 1)) I.false (I.isZero (I.num 0)) ⟧ ≡
  M.ite (M.isZero (M.num 0 M.+o M.num 1)) M.false (M.isZero (M.num 0))
```

and similarly for any other syntactic term.

In functional languages like Haskell or Agda, using the recursor on a model is a special case of a recursive definition using pattern matching. For example, H.⟦_⟧ corresponds to a height function defined by pattern matching:

```
height : I.Tm → ℕ
height I.true          = 0
height I.false         = 0
height (I.ite t t' t") = 1 + max (height t) (max (height t') (height t"))
height (I.num n)       = 0
height (I.isZero t)    = 1 + height t
height (t I.+o t')     = 1 + max (height t) (height t')
```

H.⟦_⟧ : I.Tm → ℕ

```
H.⟦ I.true ⟧          ≡ 0
H.⟦ I.false ⟧         ≡ 0
H.⟦ I.ite t t' t" ⟧   ≡ 1 + max H.⟦ t ⟧ (max H.⟦ t' ⟧ H.⟦ t" ⟧)
H.⟦ I.num n ⟧         ≡ 0
H.⟦ I.isZero t ⟧      ≡ 1 + H.⟦ t ⟧
H.⟦ t I.+o t' ⟧       ≡ 1 + max H.⟦ t ⟧ H.⟦ t' ⟧
```

Some Haskell functions defined by pattern matching are not expressible using a model and the recursor. Haskell allows nonterminating recursion or recursion where some cases are not handled, as in the following examples.

```
f : I.Tm → ℕ
f t = f t
```

```
g : I.Tm → ℕ
g I.true  = 3
g I.false = 1
```

In Agda, all functions defined by pattern matching are also expressible using the recursor (or using induction, see below). For details see [14, 1].

The syntactic operators are *disjoint*: for example, I.true is not equal to I.false. We can show this using a model where terms are metatheoretic booleans, true is metatheoretic true, false is metatheoretic false. The other components don't matter, here we just set them to constant false.

```
TF : Model
TF = record
  { Tm     = 𝟚
  ; true   = tt
  ; false  = ff
  ; ite    = λ _ _ _ → ff
  ; num    = λ _ → ff
  ; isZero = λ _ → ff
  ; _+o_   = λ _ _ → ff
  }
```

Now we can use the recursor and congruence of equality to obtain a contradiction from I.true ≡ I.false:

```
true≠false : ¬ (I.true ≡ I.false)
true≠false e = tt≠ff (cong TF.⟦_⟧ e)
```

Exercise 1.13 (compulsory)**.** *Prove that* I.num 0 I.+o I.num 0 *is not equal* I.num 0.

Exercise 1.14 (compulsory)**.** *Prove that* I.isZero (I.num 100) *is not equal* I.isZero (I.num 101).

However, there are models where true = false, e.g. the trivial model where terms are given by the set with one element tt and all operators return this element.

```
Triv : Model
Triv = record
  { Tm    = Lift ⊤
  ; true  = _
  ; false = _
  ; ite   = _
  ; num   = _
  ; isZero = _
  ; _+o_  = _
  }
```

Exercise 1.15. *Create a model where* true = false*, but* true ≠ num 0.

Note that the height of every syntactic term (element of I.Tm) is finite, e.g. there is no tree I.suc (I.suc (I.suc ⋯)) (an infinitely deep tree with I.suc at each node) because it would be mapped to an infinite natural number by H.⟦_⟧ which does not exist. This is not true for arbitrary models: there can be models where Tm has infinite elements, for example, Tm = ℕ → 2 (a term is an infinite sequence of booleans).

Exercise 1.16. *Define a NatBoolAST model where terms are natural numbers, booleans are interpreted in the C style: false is* 0*, true is* 1 *and for if-then-else, anything that is not* 0 *is interpreted as true. We can call this model the degenerate standard model.*

Exercise 1.17 (compulsory)**.** *Is there a model where* Tm = Lift ⊥*?*

Exercise 1.18 (compulsory)**.** *Show that for any two models* M *and* N *there is a model where* Tm = M.Tm × N.Tm.

Exercise 1.19. *Create a model where* Tm = 2, num 0 = tt *and all other operators are* ff*. With the help of this, create a function* isZero : I.Tm → 2*. Then create an optimisation* I.Tm → I.Tm *which maps* I.num 0 + t *to* t *and* t + I.num 0 *to* t.

Exercise 1.20 (compulsory)**.** *Consider a subset of NatBool, the programming language Nat:*

```
Tm : Set
zero : Tm
suc : Tm → Tm
_+_ : Tm → Tm → Tm
```

Define a model St *such that* St.⟦_⟧ : I.Tm → ℕ *is an interpreter for this language (where* I *is the syntax for this language). E.g.* St.⟦num 1 + num 3⟧ = 4.

1.3.2 Induction

In addition to recursion, the syntax supports induction which is a generalisation (dependent variant) of recursion. Induction can be stated by saying that for any *dependent model* D there is a *dependent homomorphism* from I to D. A *dependent model* (dependent algebra, displayed algebra/model) is given by the following components.

```
record DepModel {ℓ} : Set (lsuc ℓ) where
  field
    Tm•    : I.Tm → Set ℓ
```

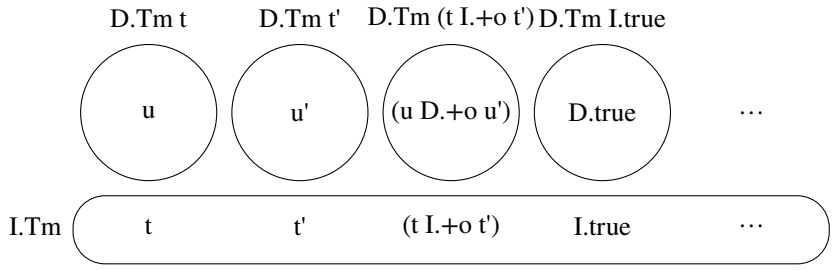


Figure 1.3: Depiction of a dependent model D. There is a separate set D.Tm t for every syntactic term t.

```
true•    : Tm• I.true
false•   : Tm• I.false
ite•     : ∀{t u v} → Tm• t → Tm• u → Tm• v → Tm• (I.ite t u v)
num•     : (n : ℕ) → Tm• (I.num n)
isZero•  : ∀{t} → Tm• t → Tm• (I.isZero t)
_+o•_    : ∀{u v} → Tm• u → Tm• v → Tm• (u I.+o v)
```

In a dependent model, instead of a set of terms, there is a family of sets indexed over I.Tm, that is, there is a set Tm• t for each element t : I.Tm (See Figure 1.3). Then we have elements of these families at each syntactic operator as index. So, we have a Tm• I.true, a Tm• I.false, … , a Tm• (t I.+o t'). For those operators with parameters, we also have induction hypotheses, for example, we have Tm• t and Tm• t' as inputs for _+o_.

The dependent homomorphism from the syntax is called *induction* (induction principle, eliminator). This is a dependent function which for an input t : I.Tm outputs a Tm• t. Moreover, as in the non-dependent case, it maps each syntactic operator to its counterpart in the dependent model.

```
⟦_⟧ : (t : I.Tm) → Tm• t
⟦ I.true       ⟧ = true•
⟦ I.false      ⟧ = false•
⟦ I.ite t t' t" ⟧ = ite• ⟦ t ⟧ ⟦ t' ⟧ ⟦ t" ⟧
⟦ I.num n      ⟧ = num• n
⟦ I.isZero t   ⟧ = isZero• ⟦ t ⟧
⟦ t I.+o t'    ⟧ = ⟦ t ⟧ +o• ⟦ t' ⟧
```

A special case of the dependent model is when Tm• t = Lift (P t) for some P : I.Tm → Prop, that is, Tm is a predicate (unary relation) on syntactic terms. In this case, the other components say that each syntactic operator respects the predicate. For example, if the predicate holds for t and t', then it also holds for t I.+o t'. Cf. natural number models (a set with an element and an endofunction) and induction on natural numbers, see Section 1.3.3. All inductively defined sets come with a notion of model, recursion, dependent model and induction. Other well-known examples are lists, binary trees.

Consider the following model which describes the number of leaves.

```
L1 : Model
L1 = record
  { Tm   = ℕ
  ; true   = 1
  ; false  = 1
  ; ite    = λ t t' t" → t + t' + t"
  ; num    = λ _ → 1
  ; isZero = λ t → t
  ; _+o_ = _+_
```

```
  }
module L1 = Model L1
```

Here is another model for describing the number of leaves using a different logic: At a binary branching, the number of leaves increases by one, at a ternary branching it increases by two.

```
L2 : Model
L2 = record
  { Tm    = ℕ
  ; true  = 0
  ; false = 0
  ; ite   = λ t t' t'' → 2 + t + t' + t''
  ; num   = λ _ → 0
  ; isZero = λ t → t
  ; _+o_ = λ t t' → 1 + t + t'
  }
module L2 = Model L2
```

Now we prove that counting the leaves of a syntactic term using L1 is equal to counting the leaves using L2 and adding one. For this end, we define the following dependent model

```
D : DepModel
D = record
  { Tm• = λ t → Lift (L1.⟦ t ⟧ ≡ 1 + L2.⟦ t ⟧)
  ; true• = mk refl
  ; false• = mk refl
  ; ite• = λ { {t}{u}{v} (mk t•) (mk u•) (mk v•) → mk (
   L1.⟦ t I.+o u I.+o v ⟧
                                        ≡⟨ refl ⟩
   L1.⟦ t ⟧ + L1.⟦ u ⟧ + L1.⟦ v ⟧
                                        ≡⟨ cong₃ (λ x y z → x + y + z) t• u• v• ⟩
   (1 + L2.⟦ t ⟧) + (1 + L2.⟦ u ⟧) + (1 + L2.⟦ v ⟧)
                                        ≡⟨ ass+ {1 + L2.⟦ t ⟧} ⟩
   (1 + L2.⟦ t ⟧) + ((1 + L2.⟦ u ⟧) + (1 + L2.⟦ v ⟧))
                                        ≡⟨ cong ((1 + L2.⟦ t ⟧) +_) (ass+ {1 + L2.⟦ u ⟧} ⁻¹) ⟩
   (1 + L2.⟦ t ⟧) + ((1 + L2.⟦ u ⟧ + 1) + L2.⟦ v ⟧)
                                        ≡⟨ cong (λ z → (1 + L2.⟦ t ⟧) + (z + L2.⟦ v ⟧))
                                              (+comm {1 + L2.⟦ u ⟧}) ⟩
   (1 + L2.⟦ t ⟧) + (2 + L2.⟦ u ⟧ + L2.⟦ v ⟧)
                                        ≡⟨ refl ⟩
   (1 + L2.⟦ t ⟧) + (2 + (L2.⟦ u ⟧ + L2.⟦ v ⟧))
                                        ≡⟨ ass+ {1 + L2.⟦ t ⟧} ⁻¹ ⟩
   ((1 + L2.⟦ t ⟧) + 2) + (L2.⟦ u ⟧ + L2.⟦ v ⟧)
                                        ≡⟨ cong (_+ (L2.⟦ u ⟧ + L2.⟦ v ⟧))
                                              (+comm {1 + L2.⟦ t ⟧}{2}) ⟩
   3 + L2.⟦ t ⟧ + (L2.⟦ u ⟧ + L2.⟦ v ⟧)
                                        ≡⟨ ass+ {3 + L2.⟦ t ⟧} ⁻¹ ⟩
   3 + L2.⟦ t ⟧ + L2.⟦ u ⟧ + L2.⟦ v ⟧
                                        ≡⟨ refl ⟩
   1 + L2.⟦ I.ite t u v ⟧
                                        ∎)}
  ; num• = λ _ → mk refl
  ; isZero• = λ t• → t•
  ; _+o•_ = λ { {u}{v} (mk u•) (mk v•) → mk (
   L1.⟦ u I.+o v ⟧
                                        ≡⟨ refl ⟩
   L1.⟦ u ⟧ + L1.⟦ v ⟧
```

```
    1 + L2.⟦ u ⟧ + L1.⟦ v ⟧                    ≡⟨ cong (_+ L1.⟦ v ⟧) u• ⟩

    (1 + L2.⟦ u ⟧) + (1 + L2.⟦ v ⟧)            ≡⟨ cong (λ z → 1 + L2.⟦ u ⟧ + z) v• ⟩

    ((1 + L2.⟦ u ⟧) + 1) + L2.⟦ v ⟧            ≡⟨ ass+ {1 + L2.⟦ u ⟧} ⁻¹ ⟩

    (1 + (L2.⟦ u ⟧ + 1)) + L2.⟦ v ⟧            ≡⟨ cong (_+ L2.⟦ v ⟧) (ass+ {1}{_}{1} ⁻¹) ⟩

                                               ≡⟨ cong (λ z → 1 + z + L2.⟦ v ⟧)
    (1 + (1 + L2.⟦ u ⟧)) + L2.⟦ v ⟧                      (+comm {L2.⟦ u ⟧}{1}) ⟩

    2 + (L2.⟦ u ⟧ + L2.⟦ v ⟧)                  ≡⟨ refl ⟩

    1 + (L2.⟦ u I.+o v ⟧)                      ≡⟨ refl ⟩

                                               ∎)}
  }
module D = DepModel D
```

Interpreting into the dependent model D gives our proof:

```
L1=L2 : ∀ t → L1.⟦ t ⟧ ≡ 1 + L2.⟦ t ⟧
L1=L2 t = un D.⟦ t ⟧
```

Exercise 1.21 (recommended). *Show that the number of* true*s in a syntactic term is less than or equal to* 3 *to the power the height of the tree. For this, define a dependent model* D *where* D.Tm t *has an inhabitant exactly when* N.⟦t⟧ *is less than or equal to* 3 ^ (H.⟦t⟧). *Here* N *is the model defined in Exercise 1.11 and* H *is the model defined for calculating the height of a term.*

D.Tm t := N.⟦ t ⟧ ≤ 3 ^ (H.⟦ t ⟧)
D.true : N.⟦ I.true ⟧ = 1 ≤ 1 = 3 ^ 0 = 3 ^ (H.⟦ I.true ⟧)
D.false : N.⟦ I.false ⟧ = 0 ≤ 1 = 3 ^ 0 = 3 ^ (H.⟦ I.false ⟧)
D.ite t^D $t^{D'}$ $t^{D''}$:
N.⟦ I.ite t t' t'' ⟧ =
N.⟦t⟧ + N.⟦t'⟧ + N.⟦t''⟧ ≤(t^D,$t^{D'}$,$t^{D''}$)
3 ^ (H.⟦t⟧) + 3 ^ (H.⟦t'⟧) + 3 ^ (H.⟦t''⟧) ≤
3 ^ (H.⟦t⟧) + 3 ^ (H.⟦t'⟧) + 3 ^ (H.⟦t''⟧) ≤
3 * 3 ^ (max H.⟦t⟧ (max H.⟦t'⟧ H.⟦t''⟧)) =
3 ^ (1 + max H.⟦t⟧ (max H.⟦t'⟧ H.⟦t''⟧)) =
3 ^ (H.⟦ I.ite t t' t'' ⟧)
D.num n : N.⟦ I.num n ⟧ = 0 ≤ 1 = 3 ^ 0 = 3 ^ (H.⟦ I.num n ⟧)
D.isZero t^D : N.⟦ I.isZero t ⟧ = N.⟦ t ⟧ ≤(t^D) 3 ^ (H.⟦ t ⟧) ≤ 3 ^ (1 + H.⟦ t ⟧) = 3 ^ (H.⟦ I.isZero t ⟧)
t^D D.+o $t^{D'}$:
N.⟦ t I.+o t' ⟧ =
N.⟦ t ⟧ + N.⟦ t' ⟧ ≤(t^D,$t^{D'}$)
3 ^ (H.⟦ t ⟧) + 3 ^ (H.⟦ t' ⟧) ≤
2 * 3 ^ (max H.⟦ t ⟧ H.⟦ t' ⟧) ≤
3 * 3 ^ (max H.⟦ t ⟧ H.⟦ t' ⟧) ≤
3 ^ (1 + max H.⟦ t ⟧ H.⟦ t' ⟧) =
3 ^ (H.⟦ t I.+o t' ⟧)

Exercise 1.22. *Show that every model can be turned into a dependent model, and derive the recursor using the induction principle.*

Exercise 1.23 (recommended). *Prove that* H.⟦ t ⟧ ≤ S.⟦ t ⟧ *where* S *is the model defined in exercise 1.12.*

Exercise 1.24 (recommended)**.** *Prove that* I.isZero *is injective: if* I.isZero t ≡ I.isZero t', *then* t ≡ t'. *Define a model* M *where* M.isZero *is not injective.*

Exercise 1.25. *Define a model which counts the number of* true*s and* false*s in a term. Define a model which counts the number of* true*s in a term. Show that interpreting a syntactic term in the first one always gives a greater or equal number than interpreting a term in the second one.*

1.3.3 Another example of recursion and induction: natural numbers

A model of naturals is a set Nat with an element Zero and an endofunction Suc.

```
record Model {ℓ} : Set (lsuc ℓ) where
  field
    Nat  : Set ℓ
    Zero : Nat
    Suc  : Nat → Nat
```

There is a model I given by actual natural numbers.

```
I : Model
I = record { Nat = ℕ ; Zero = 0 ; Suc = 1 +_ }
```

For any other model M, we have a function from I.Nat to M.Nat which respects the two operations.

```
M.⟦_⟧ : I.Nat → M.Nat
M.⟦ I.Zero ⟧ = M.Zero
M.⟦ I.Suc n ⟧ = M.Suc M.⟦ n ⟧
```

We define the following model where Nat is syntactic natural numbers.

```
M : Model
M = record { Nat = I.Nat ; Zero = I.Suc I.Zero ; Suc = λ n → I.Suc (I.Suc n) }
```

Interpretation into M is the function $n \mapsto 2 * n + 1$:

```
testM0 : M.⟦ 0 ⟧ ≡ 1
testM1 : M.⟦ 1 ⟧ ≡ 3
testM2 : M.⟦ 2 ⟧ ≡ 5
```

Now we define a model where Nat is endofunctions on I.Nat, Zero is the identity function, and Suc is post-composition with I.Suc.

```
A : Model
A = record { Nat = I.Nat → I.Nat ; Zero = λ n → n ; Suc = λ f → I.Suc ∘ f }
```

Interpretation into A is the function that maps n into the function which adds n to a number:

```
testA0 : A.⟦ 0 ⟧ ≡ λ n → n
testA1 : A.⟦ 1 ⟧ ≡ I.Suc
testA2 : A.⟦ 2 ⟧ ≡ I.Suc ∘ I.Suc
testA3 : A.⟦ 3 ⟧ ≡ I.Suc ∘ I.Suc ∘ I.Suc
```

Thus we can define addition of natural numbers as follows.

```
_+'_ : I.Nat → I.Nat → I.Nat
_+'_ = A.⟦_⟧


test1+3 : 1 +' 3 ≡ 4
test3+2 : 3 +' 2 ≡ 5
```

A dependent model is the data for induction on natural numbers.

```
record DepModel {ℓ} : Set (lsuc ℓ) where
  field
    Nat•  : I.Nat → Set ℓ
    Zero• : Nat• I.Zero
    Suc•  : {n : I.Nat} → Nat• n → Nat• (I.Suc n)
```

For example, we prove associativity of the above addition by the following dependent model. The Nat component says what we want to prove for each number, the Zero component is the base case, the Suc components is the inductive case.

```
Ass : (n o : I.Nat) → DepModel
Ass n o = record
  { Nat•  = λ m → Lift ((m +' n) +' o ≡ m +' (n +' o))
  ; Zero• = mk refl
  ; Suc•  = λ ih → mk (cong suc (un ih))
  }
```

In the base holds by reflexivity, in the inductive case we simply use the induction hypothesis. Now we obtain the proof of associativity by interpreting into the dependent model Ass.

```
ass : (m n o : I.Nat) → (m +' n) +' o ≡ m +' (n +' o)
ass m n o = un (Assno.⟦_⟧ m)
  where
    module Assno = DepModel (Ass n o)
```

Exercise 1.26 (recommended)**.** *Show that* 0 *is right unit for addition using another dependent model.*

Exercise 1.27 (recommended)**.** *Show that* + *is commutative. You will need two separate dependent models.*

1.4 Well-typed syntax

The abstract syntax tree description of NatBool still contains meaningless programs, e.g. isZero true or true + num 2. The problem is that isZero expects a natural number and not a boolean and + expects two natural numbers. The solution is to add the type information to the terms, that is, whether a term is a boolean or a natural number.

NatBoolWT (well-typed) contains two sorts: Ty and Tm, and the latter sort is indexed by the first one. A NatBoolWT model is a record with the following components.

```
record Model {ℓ ℓ'} : Set (lsuc (ℓ ⊔ ℓ')) where
  field
    Ty     : Set ℓ
    Tm     : Ty → Set ℓ'
    Nat    : Ty
    Bool   : Ty
    true   : Tm Bool
    false  : Tm Bool
    ite    : {A : Ty} → Tm Bool → Tm A → Tm A → Tm A
    num    : ℕ → Tm Nat
    isZero : Tm Nat → Tm Bool
    _+o_   : Tm Nat → Tm Nat → Tm Nat
```

There is no set of terms anymore, but there is a set of Nat-terms and a set of Bool-terms, separately. Each operator expects a term of the correct type as input and returns a term of

the appropriate type. For example, the first argument of ite has to be a boolean and the second and third arguments have to be of the same type. The type of ite t u v is the same as the type of the second and third arguments. This type is an *implicit argument* of ite which we omit for readability (the full type would be (A : Ty) → Tm Bool → Tm A → Tm A → Tm A, now it is {A : Ty} → Tm Bool → Tm A → Tm A → Tm A).

We could have used two sorts, TmBool and TmNat instead however that would not generalise to function types (Chapter 3).

The syntax of NatBoolWT is still called I and its recursor consists of two functions (one for each sort, denoted by ⟦_⟧T and ⟦_⟧t) which preserve all operations.

```
⟦_⟧T : I.Ty → Ty
⟦_⟧t  : ∀{A} → I.Tm A → Tm ⟦ A ⟧T
⟦ I.Nat       ⟧T = Nat
⟦ I.Bool      ⟧T = Bool
⟦ I.true      ⟧t = true
⟦ I.false     ⟧t = false
⟦ I.ite t u v ⟧t = ite ⟦ t ⟧t ⟦ u ⟧t ⟦ v ⟧t
⟦ I.num n     ⟧t = num n
⟦ I.isZero t  ⟧t = isZero ⟦ t ⟧t
⟦ u I.+o v    ⟧t = ⟦ u ⟧t +o ⟦ v ⟧t
```

We also have a notion of dependent model and induction just as for NatBoolAST.

```
record DepModel {ℓ ℓ'} : Set (lsuc (ℓ ⊔ ℓ')) where
  field
    Ty•     : I.Ty → Set ℓ
    Tm•     : ∀{A} → Ty• A → I.Tm A → Set ℓ'
    Nat•    : Ty• I.Nat
    Bool•   : Ty• I.Bool
    true•   : Tm• Bool• I.true
    false•  : Tm• Bool• I.false
    ite•    : ∀{A t u v}{A• : Ty• A} → Tm• Bool• t → Tm• A• u → Tm• A• v →
              Tm• A• (I.ite t u v)
    num•    : (n : ℕ) → Tm• Nat• (I.num n)
    isZero• : ∀{t} → Tm• Nat• t → Tm• Bool• (I.isZero t)
    _+o•_   : ∀{u v} → Tm• Nat• u → Tm• Nat• v → Tm• Nat• (u I.+o v)
  ⟦_⟧T : (A : I.Ty) → Ty• A
  ⟦_⟧t  : ∀{A}(t : I.Tm A) → Tm• ⟦ A ⟧T t
  ⟦ I.Nat       ⟧T = Nat•
  ⟦ I.Bool      ⟧T = Bool•
  ⟦ I.true      ⟧t = true•
  ⟦ I.false     ⟧t = false•
  ⟦ I.ite t u v ⟧t = ite• ⟦ t ⟧t ⟦ u ⟧t ⟦ v ⟧t
  ⟦ I.num n     ⟧t = num• n
  ⟦ I.isZero t  ⟧t = isZero• ⟦ t ⟧t
  ⟦ u I.+o v    ⟧t = ⟦ u ⟧t +o• ⟦ v ⟧t
```

Exercise 1.28 (recommended)**.** *Implement the two different leaf count functions and using a dependent model show that they give the same result. See the previous section.*

1.4.1 Type inference

The goal of type inference is to turn a syntactic NatBoolAST term into a well-typed (NatBoolWT) term. To achieve this, we try to build a NatBoolAST model using NatBoolWT terms. This is not really possible because e.g. NatBoolAST requires us to add any two terms using _+_, but the addition that we have only works for terms of type Nat. Terms in our NatBoolAST algebra are either a pair of a NatBoolWT type and a term of that type or a failure. We use Maybe to express

possible failure: Just (A, t) and Nothing are both in the set Maybe (Σ I.Ty I.Tm), the latter expresses failure. Failure propagates through the operations ite, isZero and _+_.

```
Inf : NatBoolAST.Model
Inf = record
  { Tm    = Maybe (Σ I.Ty I.Tm)
  ; true  = Just (I.Bool , I.true)
  ; false = Just (I.Bool , I.false)
  ; ite   = ite
  ; num   = λ n → Just (I.Nat , I.num n)
  ; isZero = isZero
  ; _+o_  = _+o_
  }
  where
    ite : Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm)
    ite (Just (I.Bool , t)) (Just (I.Nat  , u)) (Just (I.Nat  , v)) = Just (I.Nat , I.ite t u v)
    ite (Just (I.Bool , t)) (Just (I.Bool , u)) (Just (I.Bool , v)) = Just (I.Bool , I.ite t u v)
    ite _                   _                   _                   = Nothing

    isZero : Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm)
    isZero (Just (I.Nat , t)) = Just (I.Bool , I.isZero t)
    isZero _                  = Nothing

    _+o_ : Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm) → Maybe (Σ I.Ty I.Tm)
    Just (I.Nat , t) +o Just (I.Nat , t') = Just (I.Nat , (t I.+o t'))
    _                +o _                 = Nothing
module Inf = NatBoolAST.Model Inf
```

Interpretation into this model gives us type inference.

```
infer : NatBoolAST.I.Tm → Maybe (Σ I.Ty I.Tm)
infer = Inf.⟦_⟧
```

Exercise 1.29 (compulsory)**.** *Here are some lists of lexical elements (written as strings). They are all accepted by the parser. Which ones are rejected by type inference? For the ones which are accepted, write down the NatBoolWT terms which are produced.*

if true then true else num 0
if true then num 0 else num 0
if num 0 then num 0 else num 0
if num 0 then num 0 else true
true + zero
true + num 1
true + isZero false
true + isZero (num 0)

Exercise 1.30 (compulsory)**.** *Here are some NatBoolAST terms. Which ones are rejected by type inference? For the ones which are accepted, write down the NatBoolWT terms which are produced.*

ite true true (num 0)
ite true (num 0) (num 0)
ite (num 0) (num 0) (num 0)
true +o zero
true +o num 1
true +o isZero (num 1)
isZero (num 1) +o num 0

1.4.2 Standard interpretation

We define the standard model (set model, denotational semantics). The idea is that all operators in our object language are given by their metatheoretic counterparts.

```
St : Model
St = record
  { Ty    = Set
  ; Tm    = λ T → T
  ; Nat   = ℕ
  ; Bool  = 2
  ; true  = tt
  ; false = ff
  ; ite   = if_then_else_
  ; num   = λ n → n
  ; isZero = λ { zero → tt ; _ → ff }
  ; _+o_  = _+_
  }
module St = Model St
```

Using the recursor we get an interpreter (the metacircular interpreter, evaluator, normaliser) for our language:

```
norm : ∀{A} → I.Tm A → St.⟦ A ⟧T
norm = St.⟦_⟧t
```

We were not able to define the standard interpreter for NatBoolAST because there we had meaningless programs. For example, we would have had to interpret both **Bool** and **Nat** by the same set (c.f. Exercise 1.16).

1.4.3 Typing relation

The traditional way of defining the well-typed syntax is by defining a binary relation between syntactic types and syntactic NatBoolAST terms (see e.g. [11]). This relation is defined inductively as follows.

```
module I' = NatBoolAST.I
data _⊗_ : I'.Tm → I.Ty → Prop where
  true  :
                        ----------------------
                        I'.true ⊗ I.Bool
  false :
                        ----------------------
                        I'.false ⊗ I.Bool
  ite   : ∀{t u v A} →
                        t ⊗ I.Bool →
                        u ⊗ A      →
                        v ⊗ A      →
                        ----------------------
                        I'.ite t u v ⊗ A
  num   : ∀{n} →
                        ----------------------
                        I'.num n ⊗ I.Nat
  isZero : ∀{t} →
                        t ⊗ I.Nat  →
                        ----------------------
                        I'.isZero t ⊗ I.Bool
```

```
_+o_ : ∀{u v} →
                    u ∋ I.Nat  →
                    v ∋ I.Nat  →
                    ----------------------
                    (u I'.+o v) ∋ I.Nat
```

Exercise 1.31. *Show that there is an isomorphism between* I.Tm A *and* Σ I'.Tm (_∋ A).

We defined well-typed terms directly because we do not want to talk about non-well-typed terms in the same way as we are not interested in programs with non-matching brackets.

1.5 Well-typed syntax with equations

In the well-typed syntax, we still had too many terms in a sense: for example, the terms I.true and I.isZero (I.num 0) were different, even if they should be equal in a sensible semantics. The information which terms should be the same (usually this is called operational semantics) was missing. Now we add this information in the form of equalities.

A NatBool model (without any qualifiers) consists of the following components. The only difference from NatBoolWT is the addition of the five equations.

```
record Model {ℓ ℓ'} : Set (lsuc (ℓ ⊔ ℓ')) where
  field
    Ty       : Set ℓ
    Tm       : Ty → Set ℓ'
    Nat      : Ty
    Bool     : Ty
    true     : Tm Bool
    false    : Tm Bool
    ite      : {A : Ty} → Tm Bool → Tm A → Tm A → Tm A
    num      : ℕ → Tm Nat
    isZero   : Tm Nat → Tm Bool
    _+o_     : Tm Nat → Tm Nat → Tm Nat
    iteβ₁    : ∀{A}{u v : Tm A} → ite true u v ≡ u
    iteβ₂    : ∀{A}{u v : Tm A} → ite false u v ≡ v
    isZeroβ₁ : isZero (num 0) ≡ true
    isZeroβ₂ : {n : ℕ} → isZero (num (1 + n)) ≡ false
    +β       : {m n : ℕ} → num m +o num n ≡ num (m + n)
```

The recursor is given as before.

```
⟦_⟧T       : I.Ty → Ty
⟦ I.Nat ⟧T  = Nat
⟦ I.Bool ⟧T = Bool

postulate
  ⟦_⟧t     : {A : I.Ty} → I.Tm A → Tm ⟦ A ⟧T
  ⟦true⟧   :          ⟦ I.true          ⟧t ≈ true
  ⟦false⟧  :          ⟦ I.false         ⟧t ≈ false
  ⟦ite⟧    : {A : I.Ty}{t : I.Tm I.Bool}{u v : I.Tm A} →
                      ⟦ I.ite t u v ⟧t ≈ ite ⟦ t ⟧t ⟦ u ⟧t ⟦ v ⟧t
  ⟦num⟧    : {n : ℕ} → ⟦ I.num n   ⟧t ≈ num n
  ⟦isZero⟧ : ∀{t}    → ⟦ I.isZero t ⟧t ≈ isZero ⟦ t ⟧t
  ⟦+o⟧     : ∀{u v}  → ⟦ u I.+o v   ⟧t ≈ ⟦ u ⟧t +o ⟦ v ⟧t
```

We also have dependent models and an eliminator.

Now we can reproduce the evaluation of the example program from the beginning of Chapter 1 as an equality between terms.

```
ite (isZero (num 0 +o num 1)) false (isZero (num 0))
  ≡⟨ cong (λ x → ite (isZero x) false (isZero (num 0))) +β ⟩
ite (isZero (num 1))          false (isZero (num 0))
  ≡⟨ cong (λ x → ite x false (isZero (num 0))) isZeroβ₂ ⟩
ite false                     false (isZero (num 0))
  ≡⟨ iteβ₂ ⟩
isZero (num 0)
  ≡⟨ isZeroβ₁ ⟩
true
  ∎
```

Two terms which result in the same boolean or number are equal.

The type inference algorithm given for NatBoolWT also works for NatBool. We can't calculate the height of a term anymore as before because of the equalities. Every function has to preserve the equalities. For example, we need that height (isZero (num 0)) = height true. If we view these terms as syntax trees, their height is 1 and 0, respectively.

1.5.1 Standard model, trivial model

The term ite (isZero (num 0 +o num 1)) false (isZero (num 0)) was built using the operations of our language. Every model supports these operations, hence the term can be built in any model. The equality ite (isZero (num 0 +o num 1)) false (isZero (num 0)) = true was built using the equations in our language which hold in every model. Hence the equality also holds in every model. Such a term or equation is called *derivable*. A derivable term can be constructed in any model, a derivable equation holds in every model. In contrast, there are properties that we can only prove using induction: these hold in the syntax, but might not hold for every model. For example, the fact that for any t : Tm Bool we have t = true or t = false holds in the syntax but not in an arbitrary model (this is a consequence of normalisation, see below). Such properties are called *admissible*.

Exercise 1.32 (recommended)**.** *Construct a model in which it does not hold that for a* t : Tm Bool *we have* t = true *or* t = false.

Another property of the syntax is that true ≠ false. We can show this using the standard model.

The operators in the standard model are defined exactly as for NatBoolWT, but we also have to prove the five equalities iteβ₁, …, +β: they all hold by definition.

```
St : Model
St = record
  { Ty       = Set
  ; Tm       = λ A → A
  ; Nat      = ℕ
  ; Bool     = 𝟚
  ; true     = tt
  ; false    = ff
  ; ite      = if_then_else_
  ; num      = λ n → n
  ; isZero   = iteℕ tt (λ _ → ff)
  ; _+o_     = _+_
  ; iteβ₁    = refl
  ; iteβ₂    = refl
  ; isZeroβ₁ = refl
  ; isZeroβ₂ = refl
  ; +β       = refl
  }
module St = Model St
```

From the standard model we obtain *equational consistency* of our language, that is, there are terms which are not equal. This means that we didn't add too many equalities when defining the

language. In the syntax, if true was equal to false, then their standard interpretations would also be equal, but then tt would be equal to ff.

```
true≠false : ¬ (I.true ≡ I.false)
true≠false e = tt≠ff (cong St.⟦_⟧t e)
```

If we are not careful when defining the equations of the language, it is easy to lose equational consistency.

Exercise 1.33 (compulsory)**.** *Show that if* true = false *in a model, then any two* u, v : Tm Nat *are equal in that model.*

A consequence is that if a compiler compiles true and false to the same code, then it compiles all natural numbers to the same code.

As for any language, we can define a trivial model.

```
Triv : Model
Triv = record
  { Ty       = Lift ⊤
  ; Tm       = λ _ → Lift ⊤
  ; Nat      = mk triv
  ; Bool     = mk triv
  ; true     = mk triv
  ; false    = _
  ; ite      = _
  ; num      = _
  ; isZero   = _
  ; _+o_     = _
  ; isZeroβ₁ = refl
  ; isZeroβ₂ = refl
  ; +β       = refl
  ; iteβ₁    = refl
  ; iteβ₂    = refl
  }
module Triv = Model Triv
```

In the trivial model, any two terms are equal, hence all the equalities that can be described in the language, hold. For example, in this model we not only have true = false, but also Nat = Bool and true = num 5 (this equality only makes sense because we know that Nat = Bool). We also have that for any t : Tm Bool, t = true or t = false, trivially.

1.5.2 Normalisation

Normalisation essentially says that the language can also be defined without equations. That is, there is a subset of syntactic terms called *normal forms* which are pairwise distinct and every syntactic term is equal to a normal form. Syntactic terms are quotiented by the equations of the language and normalisation implies that there is a function which selects a representative from each equivalence class. For NatBool, there are two normal forms of type Bool (true and false) and countably many normal forms of type Nat (num n for each n : ℕ). Thus we simply define normal forms of type A by interpreting the A in the standard model, together with an injection ⌜_⌝ into syntactic terms.

```
Nf : I.Ty → Set
Nf A = St.⟦ A ⟧T

⌜_⌝ : ∀ {A} → St.⟦ A ⟧T → I.Tm A
⌜_⌝ {I.Bool} tt = I.true
⌜_⌝ {I.Bool} ff = I.false
⌜_⌝ {I.Nat} = I.num
```

The normalisation function is simply interpretation into the standard model:

```
norm : {A : I.Ty} → I.Tm A → Nf A
norm = St.⟦_⟧t
```

Completeness of normalisation says that every term is equal to its normal form (followed by quote). In our case, this means that every boolean term (element of I.Tm I.Bool) is either true or false and every natural number term is a finite application of I.sucs on I.zero. We first show that the injection ⌜_⌝ commutes with the operations ite, isZero and _+_.

```
⌜ite⌝ : ∀{b A}{u v : St.⟦ A ⟧T} → ⌜ if b then u else v ⌝ ≡ I.ite {A} ⌜ b ⌝ ⌜ u ⌝ ⌜ v ⌝
⌜ite⌝ {tt}{A} = I.iteβ₁ ⁻¹
⌜ite⌝ {ff}{A} = I.iteβ₂ ⁻¹

⌜isZero⌝ : ∀{n} → ⌜ iteℕ tt (λ _ → ff) n ⌝ ≡ I.isZero ⌜ n ⌝
⌜isZero⌝ {zero}  = I.isZeroβ₁ ⁻¹
⌜isZero⌝ {suc n} = I.isZeroβ₂ ⁻¹

⌜+⌝ : ∀{m n} → ⌜ m + n ⌝ ≡ ⌜ m ⌝ I.+o ⌜ n ⌝
⌜+⌝ {m}{n} = I.+β ⁻¹
```

Now we can directly prove completeness of normalisation by induction on terms.

```
Comp : DepModel
Comp = record
  { Ty•       = λ _ → Lift ⊤
  ; Tm•       = λ _ t → Lift (⌜ St.⟦ t ⟧t ⌝ ≡ t)
  ; Nat•      = _
  ; Bool•     = _
  ; num•      = λ n → mk refl
  ; isZero•   = λ {t} t• → mk (⌜isZero⌝ ■ cong I.isZero (un t•))
  ; _+o•_     = λ {u}{v} u• v• → mk (
                  (⌜+⌝ {St.⟦ u ⟧t}{St.⟦ v ⟧t}) ■
                  (cong₂ I._+o_ (un u•) (un v•)))
  ; true•     = mk refl
  ; false•    = mk refl
  ; ite•      = λ {A}{A•}{t}{u}{v} t• u• v• → mk (
                  (⌜ite⌝ {St.⟦ t ⟧t}{A}{St.⟦ u ⟧t}{St.⟦ v ⟧t}) ■
                  (cong₃ I.ite (un t•) (un u•) (un v•)))
  ; iteβ₁•    = refl
  ; iteβ₂•    = refl
  ; isZeroβ₁• = refl
  ; isZeroβ₂• = refl
  ; +β•       = refl
  }
module Comp = DepModel Comp

comp : {A : I.Ty}{t : I.Tm A} → ⌜ norm {A} t ⌝ ≡ t
comp {t = t} = un Comp.⟦ t ⟧t
```

Stability of normalisation means that there is no junk in normal forms, for every normal form there is a term which normalises to it.

```
stab : {A : I.Ty}{t : St.⟦ A ⟧T} → norm {A} ⌜ t ⌝ ≡ t
stab {I.Nat} {t} = refl
stab {I.Bool} {tt} = refl
stab {I.Bool} {ff} = refl
```

Exercise 1.34. *Define normalisation and prove its completeness using a modified standard model where* Bool = Maybe 2. *Show that stability does not hold.*

Exercise 1.35 (recommended)**.** *Show that in the syntax we have* ite t u u = u *for any* t, u, *and construct a model in which this doesn't hold.*

We used all the equations of NatBool in the completeness proof. If we take out e.g. the rule isZeroβ_2 from the definition of the language, we cannot prove completeness anymore because we have elements of I.Tm I.Bool which are neither I.true, nor I.false. The following (NatBool without isZeroβ_2) model distinguishes true, false and isZero (num 1).

```
module NatBoolIncomplete where
  Ty       : Set₁
  Tm       : Ty → Set
  Nat      : Ty
  Bool     : Ty
  num      : ℕ → Tm Nat
  isZero   : Tm Nat → Tm Bool
  _+o_     : Tm Nat → Tm Nat → Tm Nat
  true     : Tm Bool
  false    : Tm Bool
  ite      : ∀{A} → Tm Bool → Tm A → Tm A → Tm A
  isZeroβ₁ : isZero (num 0) ≡ true
  +β       : ∀{m n} → num m +o num n ≡ num (m + n)
  iteβ₁    : ∀{A}{u v : Tm A} → ite true  u v ≡ u
  iteβ₂    : ∀{A}{u v : Tm A} → ite false u v ≡ v

  Ty               = Set
  Tm A             = A
  Nat              = ℕ
  Bool             = Maybe 2
  num              = λ n → n
  isZero zero      = Just tt
  isZero (suc _)   = Nothing
  _+o_             = _+_
  true             = Just tt
  false            = Just ff
  ite Nothing u v  = u
  ite (Just tt) u v  = u
  ite (Just ff) u v  = v
  isZeroβ₁         = refl
  +β {m}{n}        = refl
  iteβ₁ {A}{u}{v}  = refl
  iteβ₂ {A}{u}{v}  = refl

  true≠false     : ¬ (true  ≡ false)
  true≠isZero1   : ¬ (true  ≡ isZero (num 1))
  false≠isZero1  : ¬ (false ≡ isZero (num 1))
```

Exercise 1.36 (recommended)**.** *Define a height function on terms which returns the height of the normal form of the term.*

A model of a monoid over A has the following components.

```
C   : Set
f   : A → C
_∘_ : C → C → C
id  : C
ass : (a ∘ b) ∘ c ≡ a ∘ (b ∘ c)
idl : id ∘ a ≡ a
idr : a ∘ id ≡ a
```

The initial model is called the free monoid over A.

Exercise 1.37 (recommended)**.** *Define normalisation and prove its completeness and stability for the free monoid over a set* A*. Normal forms are simply lists of* A*.*

1.5.3 Transition relation

The traditional way of defining equality of syntactic terms is using transition relations on syntactic NatBoolAST terms (see e.g. [11]).

```
module SmallStep where
  import NatBoolAST
  module I' = NatBoolAST.I
  data _↦_ : I'.Tm → I'.Tm → Prop where
    isZeroβ₁     :                 I'.isZero (I'.num 0) ↦ I'.true
    isZeroβ₂     : ∀{n} →          I'.isZero (I'.num (1 + n)) ↦ I'.false
    +β           : ∀{m n} → (I'.num m I'.+o I'.num n) ↦ I'.num (m + n)
    iteβ₁        : ∀{u v} → I'.ite I'.true  u v ↦ u
    iteβ₂        : ∀{u v} → I'.ite I'.false u v ↦ v

    isZero-cong : ∀{t t'}     → t ↦ t' → I'.isZero t   ↦ I'.isZero t'
    +-cong₁     : ∀{u v u'}   → u ↦ u' → (u I'.+o v)   ↦ (u' I'.+o v)
    +-cong₂     : ∀{u v v'}   → v ↦ v' → (u I'.+o v)   ↦ (u I'.+o v')
    ite-cong    : ∀{t t' u v} → t ↦ t' → (I'.ite t u v) ↦ (I'.ite t' u v)
```

The idea is that if there is a $p : t \mapsto t'$, this means that the program t evaluates to t' in one step. Running a program means that we have a sequence of evaluation steps: $t \mapsto t' \mapsto t'' \mapsto \cdots$

There is a rewriting rule corresponding to each equation in the notion of NatBool-model. Then there are congruence rules (order rules) which express which parameters and when should be evaluated. In our example rewriting system above rewriting can happen anywhere except for I'.ite it only happens in the first parameter.

Taking the reflexive transitive closure of $\mapsto$ we obtain the multi-step rewriting relation $\mapsto^*$, taking the symmetric closure of $\mapsto^*$ we get the conversion relation (definitional equality) $\sim$ which in the case of NatBool, corresponds to equality of syntactic terms.

Exercise 1.38. *Prove that* $t \equiv t'$ *is logically equivalent to* $t \sim t'$ *for all* t *and* t'*.*

For other languages, this is not necessarily the case. In general, the transition system contains more information than the description with equations. Transition is directed and is not necessarily a congruence. We will only present languages where there is no such additional information.

1.5.4 Another example of normalisation: integers

In this subsection, we define integers as the initial model of an algebraic theory. We illustrate the concepts of model, syntax, recursion, dependent model, induction, normalisation, completeness and stability of normalisation. The representation of integers is from [**?**].

An integer model has five components, a set, a zero element, a successor operation, a predecessor operation and two equalities saying that successor after or before predecessor is the identity.

```
record Model {ℓ} : Set (lsuc ℓ) where
  field
    Z       : Set ℓ
    Zero    : Z
    Suc     : Z → Z
    Pred    : Z → Z
    SucPred : (i : Z) → Suc (Pred i) ≡ i
    PredSuc : (i : Z) → Pred (Suc i) ≡ i
```

We have a recursor, that is a homomorphism from the initial model to any model.

```
postulate
  [[_]]    : I.Z → Z
  [[Zero]] :          [[ I.Zero   ]] ≈ Zero
  [[Suc]]  : ∀{i} → [[ I.Suc i  ]] ≈ Suc [[ i ]]
  [[Pred]] : ∀{i} → [[ I.Pred i ]] ≈ Pred [[ i ]]
```

Integers themselves are given by the initial integer model.

```
ℤ = I.Z
```

There are multiple representations of each number, e.g. -1 is given by Pred Zero, Pred (Suc (Pred Zero)), and so on.

Examples of equalities which hold in any integer model.

```
module examples {ℓ}(M : Model {ℓ}) where
  open Model M

  eq1 : Pred (Suc (Pred Zero)) ≡ Pred Zero
  eq1 = PredSuc (Pred Zero)

  eq2 : Pred (Suc (Suc (Pred Zero))) ≡ Zero
  eq2 = Pred (Suc (Suc (Pred Zero)))
                                    ≡⟨ PredSuc (Suc (Pred Zero)) ⟩
        Suc (Pred Zero)
                                    ≡⟨ SucPred Zero ⟩
        Zero                        ∎
```

Dependent models and the eliminator.

The model M i where Zero is i, everything else comes from the syntax.

```
M : I.Z → Model
M i = record
  { Z       = I.Z
  ; Zero    = i
  ; Suc     = I.Suc
  ; Pred    = I.Pred
  ; SucPred = I.SucPred
  ; PredSuc = I.PredSuc
  }
module M i = Model (M i)
```

Now we have addition j +ℤ i which replaces I.Zero inside j by i:

```
_+ℤ_ : I.Z → I.Z → I.Z
j +ℤ i = M.[[_]] i j
```

```
test+ : {i : I.Z} → I.Suc (I.Suc (I.Pred I.Zero)) +ℤ i ≡ I.Suc (I.Suc (I.Pred i))
test+ {i} = refl

test+' : {i : I.Z} → I.Pred (I.Suc (I.Suc (I.Pred I.Zero))) +ℤ i ≡ I.Pred (I.Suc (I.Suc (I.Pred i)))
test+' {i} = refl
```

Integers were defined using a language with equations. Normal forms of integers are defined by another language without equations. Normalisation means that the syntaxes of these two languages are in bijection: there is a bijection between **I.Z** and **Nf** (we don't use other models of the language of normal forms, we only work with its syntax **Nf**).

Normal forms and examples.

```
data Nf : Set where
  -Suc  : ℕ → Nf
  Zero  : Nf
  +Suc : ℕ → Nf

minusThree minusFive plusSix : Nf
minusThree = -Suc 2
minusFive  = -Suc 4
plusSix    = +Suc 5

SucNf : Nf → Nf
SucNf (-Suc zero)      = Zero
SucNf (-Suc (suc n))   = -Suc n
SucNf Zero             = +Suc 0
SucNf (+Suc n)         = +Suc (suc n)

PredNf : Nf → Nf
PredNf (-Suc n)        = -Suc (suc n)
PredNf Zero            = -Suc 0
PredNf (+Suc zero)     = Zero
PredNf (+Suc (suc n)) = +Suc n
```

Normalisation.

```
N : Model
N = record
  { Z    = Nf
  ; Zero = Zero
  ; Suc  = SucNf
  ; Pred = PredNf
  ; SucPred = λ { (-Suc zero)     → refl
                ; (-Suc (suc n))  → refl
                ; Zero            → refl
                ; (+Suc zero)     → refl
                ; (+Suc (suc n)) → refl
                }
  ; PredSuc = λ { (-Suc zero)     → refl
                ; (-Suc (suc n))  → refl
                ; Zero            → refl
                ; (+Suc zero)     → refl
                ; (+Suc (suc n)) → refl }
  }
module N = Model N

norm : I.Z → Nf
norm = N.⟦_⟧
```

Embedding of normal forms into integers:

```
⌜_⌝ : Nf → I.Z
⌜ -Suc zero     ⌝ = I.Pred I.Zero
⌜ -Suc (suc n)  ⌝ = I.Pred ⌜ -Suc n ⌝
⌜ Zero          ⌝ = I.Zero
⌜ +Suc zero     ⌝ = I.Suc I.Zero
⌜ +Suc (suc n)  ⌝ = I.Suc ⌜ +Suc n ⌝
```

A unit test for normalisation:

```
testnorm : ⌜ norm (I.Pred (I.Pred (I.Suc (I.Pred (I.Pred (I.Pred (I.Suc I.Zero))))))) ⌝ ≡
           I.Pred (I.Pred (I.Pred I.Zero))
testnorm = refl
```

Stability of normalisation:

```
stab : (v : Nf) → norm ⌜ v ⌝ ≡ v
stab (-Suc zero)     = refl
stab (-Suc (suc n))  = cong PredNf (stab (-Suc n))
stab Zero            = refl
stab (+Suc zero)     = refl
stab (+Suc (suc n)) = cong SucNf (stab (+Suc n))
```

Helper lemmas for completeness:

```
⌜Suc⌝ : (v : Nf) → ⌜ SucNf v ⌝ ≡ I.Suc ⌜ v ⌝
⌜Suc⌝ (-Suc zero)     = I.SucPred I.Zero ⁻¹
⌜Suc⌝ (-Suc (suc n))  = I.SucPred ⌜ -Suc n ⌝ ⁻¹
⌜Suc⌝ Zero            = refl
⌜Suc⌝ (+Suc n)        = refl

⌜Pred⌝ : (v : Nf) → ⌜ PredNf v ⌝ ≡ I.Pred ⌜ v ⌝
⌜Pred⌝ (-Suc n)       = refl
⌜Pred⌝ Zero           = refl
⌜Pred⌝ (+Suc zero)    = I.PredSuc I.Zero ⁻¹
⌜Pred⌝ (+Suc (suc n)) = I.PredSuc ⌜ +Suc n ⌝ ⁻¹
```

Completeness of normalisation:

```
Comp : DepModel
Comp = record
  { Z•    = λ i → Lift (⌜ norm i ⌝ ≡ i)
  ; Zero• = mk refl
  ; Suc•  = λ {i} i• → mk (⌜Suc⌝ N.⟦ i ⟧ ■ cong I.Suc (un i•))
  ; Pred• = λ {i} i• → mk (⌜Pred⌝ N.⟦ i ⟧ ■ cong I.Pred (un i•))
  ; SucPred• = λ _ → refl
  ; PredSuc• = λ _ → refl
  }
module Comp = DepModel Comp

comp : (i : I.Z) → ⌜ norm i ⌝ ≡ i
comp i = un (Comp.⟦ i ⟧)
```

Chapter 2

Variables

Learning goals of this chapter

Binding, variables, scope, free and bound variables, De Bruijn indices. Abstract binding trees. Contexts, substitutions, local definitions. Constructors, destructors, computation and uniqueness rules. Main arguments of a destructor. Neutral terms and normal forms, normalisation by induction on terms. Decidability of equality of terms from normalisation.

In this chapter we extend our expression language with variables and let definitions. For example, we will be able to write let x := 1 + 2 in x + x which will be equal to $(1 + 2) + (1 + 2) = 3 + 3 = 6$. We will first describe the language at the level of abstract binding trees, then using well-typed syntax with equations.

What we call variable is more precisely called name or identifier. Note that this notion of variable is not the same as a mutable reference which is another concept sometimes called variable in programming languages.

2.1 Abstract binding trees

Variables (names) can be *bound* by symbols which we call *binders* and each binder has a *scope*. For example, in the expression $\sum_{x=1}^{10} x^2$, the variable x is bound by the binder $\sum$ in the scope of the binder which is the expression x^2 in this case. In $\lim_{x \mapsto \infty} 2^{-x}$, lim binds x in 2^{-x}. In $\int_0^{10} 2x^2 + x - 1 dx$, $\int \ldots dx$ binds x in $2x^2 + x - 1$. In $\forall x, x + 3 = 3 + x$, $\forall$ binds x in $x + 3 = 3 + x$. In int f(int i) { return(i+1); }, the function definition binds i in the function body (the parts between { and }). In let x := 1 + 2 in x + x, x is bound by let in $x + x$.

At the level of abstract syntax trees, we add variable names and let expressions:

```
N ::= x | y | z | ...
T ::= N | let N := T in T | true | false | ite T T T | ...
```

Multiple binders can be used in one expression:

```
(let x := t in x + x) + (let y := t' in y + y)
```

The names of the binders have no significance. The above term should be equal to

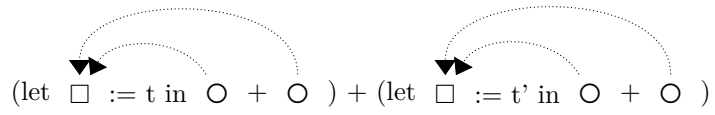
```
(let y := t in y + y) + (let x := t' in x + x)
```

or

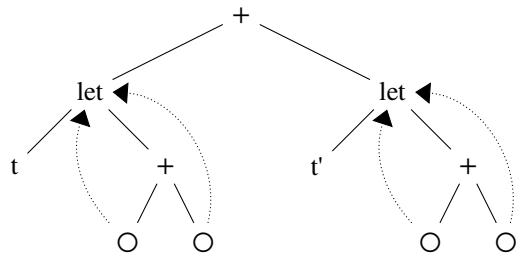
```
(let z := t in z + z) + (let z := t' in z + z)
```

for any t, t'. These are different terms at the level of abstract syntax trees, however, they are equal at the level of *abstract binding trees* (ABTs). ABT is between levels (3) and (4) of Chapter 1, see Figure 1.1. Variables are replaced by pointers to the binding. The above terms become the

following.



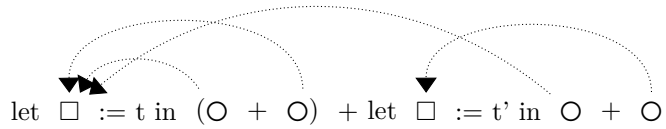
In tree notation:



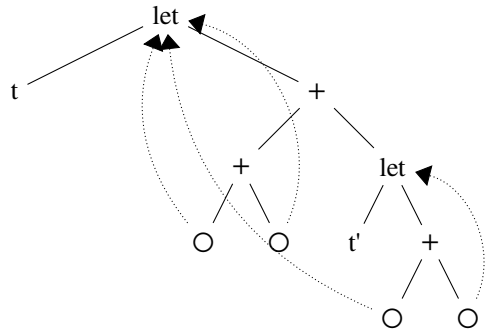
The scope of let is the longest possible, so the scope of both lets end at the end of the term. Another example:

let x := t in (x + x) + let y := t' in x + y

Binding structure:



In tree notation:



The term (let x := t in x + x) + (let y := t' in x + y) is different from both of the above terms, it contains a *free* variable x in x + y. Free variables are not in the scope of a binder which binds the same name. A term is called *closed* if there are no free variables in it, *open* if it contains at least one free variable. Every open term can be closed by adding enough binders at the front.

Exercise 2.1 (compulsory)**.** *In the following ABT-level syntactic terms, circle the free variables. From bound variables, draw a pointer to the binder.*

x + y

x + let x := 3 in y + x

x + let x := y in y + x

x + let x := x' + (x + let x' := z in x' + x)

3 + let x := x + (x' + let x' := 2 in x' + x)

Exercise 2.2 (compulsory)**.** *Decide whether the following ABT-level syntactic terms are equal.*

$x + y \stackrel{?}{=} x + z$
$(x + \text{let } x := 3 \text{ in } x + y) \stackrel{?}{=} (x + \text{let } y := 3 \text{ in } y + y)$
$(x + \text{let } x := 3 \text{ in } x + y) \stackrel{?}{=} (x + \text{let } y := 3 \text{ in } y + x)$
$(x + \text{let } x := 3 \text{ in } x + y) \stackrel{?}{=} (x + \text{let } x' := 3 \text{ in } x' + y)$
$(x + \text{let } x := 3 \text{ in let } y := 4 \text{ in } x + y) \stackrel{?}{=} (x + \text{let } y := 3 \text{ in let } x := 4 \text{ in } x + y)$
$(x + \text{let } x := 3 \text{ in let } y := 4 \text{ in } x + y) \stackrel{?}{=} (x + \text{let } y := 3 \text{ in let } x := 4 \text{ in } y + x)$

Exercise 2.3 (compulsory)**.** *Decide whether the following ABT-level syntactic terms are open or closed.*

$\text{let } x := 3 \text{ in } x + x$
$\text{let } x := y \text{ in } x + x$
$\text{let } x := y \text{ in } x + y$
$\text{let } y := x \text{ in } x + y$
$\text{let } y := x \text{ in } x + x$

One way to make abstract binding trees precise is to add the following equation:

$(\text{let } x{:=}t \text{ in } u) = (\text{let } y{:=}t \text{ in } u[x \mapsto y])$

where $u[x \mapsto y]$ means that we replace all occurrences of x by y in u. However one has to be careful:

$(\text{let } x{:=}t \text{ in } x + y) \neq (\text{let } y{:=}t \text{ in } y + y) = (\text{let } y{:=}t \text{ in } (x + y)[x \mapsto y])$

The left-hand side term is open, the right-hand side is closed, so they cannot have the same binding tree. In the equation above it has to be stated that y is fresh for u (u does not contain y). See e.g. [11, Section 1.2] for details on that approach.

Another approach to define abstract binding trees takes it seriously that the names of bound variables don't matter and does not write names at all. Instead, we write natural numbers (De Bruijn indices [4]). Examples:

$(\text{let } x{:=}t \text{ in } x + x) + (\text{let } y{:=}t' \text{ in } y + y)$ $\quad$ $(\text{let } t \text{ in } v0 + v0) + (\text{let } t' \text{ in } v0 + v0)$
$\text{let } x{:=}t \text{ in } (x + x) + \text{let } y{:=}t' \text{ in } x + y$ $\quad$ $\text{let } t \text{ in } (v0 + v0) + \text{let } t' \text{ in } v1 + v0$

v0 is a reference to the nearest binder, v1 is a reference to the next one, and so on.

Now terms are indexed by natural numbers expressing the maximal number of free variables in them.

- Tm 0 is the set of closed terms (programs).
- Tm 1 is the set of terms with maximum one free variable.
- Tm 2 is the set of terms with maximum two free variables.
- Tm 3 is the set of terms with maximum three free variables.
- …

Binders are those operators which decrease this number: e.g. let takes a Tm (1 + n) in which it binds the variable and returns a Tm n. Examples:

$$\text{if false then num 0 else num 3} : \text{Tm } 0$$

$$\text{let t } \underbrace{(v0 + v0)}_{:\text{ Tm } 1} + \text{let t' } \underbrace{(v0 + v0)}_{:\text{ Tm } 1} : \text{Tm } 0$$

$$\underbrace{\text{let t } ((v0 + v0) + \text{let t' } \underbrace{(v1 + v0)}_{:\text{ Tm } 2})}_{:\text{ Tm } 1} : \text{Tm } 0$$

We have a separate set Var n of variables which are included in terms. We write def instead of let.

```
module I where
  infixl 7 _+o_
  data Var : ℕ → Set where
    vz      : ∀{n} → Var (suc n)
    vs      : ∀{n} → Var n → Var (suc n)

  data Tm : ℕ → Set where
    var     : ∀{n} → Var n → Tm n
    def     : ∀{n} → Tm n → Tm (suc n) → Tm n

    true    : ∀{n} → Tm n
    false   : ∀{n} → Tm n
    ite     : ∀{n} → Tm n → Tm n → Tm n → Tm n
    num     : ∀{n} → ℕ → Tm n
    isZero  : ∀{n} → Tm n → Tm n
    _+o_    : ∀{n} → Tm n → Tm n → Tm n
```

Some abbreviations:

```
v0 : {n : ℕ} → Tm (1 + n)
v0 = var vz
v1 : {n : ℕ} → Tm (2 + n)
v1 = var (vs vz)
v2 : {n : ℕ} → Tm (3 + n)
v2 = var (vs (vs vz))
v3 : {n : ℕ} → Tm (4 + n)
v3 = var (vs (vs (vs vz)))
```

The DefABT model is as following:

```
record Model {ℓ ℓ'} : Set (lsuc (ℓ ⊔ ℓ')) where
  infixl 7 _+o_
  field
    Var : ℕ → Set ℓ
    vz : ∀{n} → Var (suc n)
    vs : ∀{n} → Var n → Var (suc n)

    Tm     : ℕ → Set ℓ'
    var    : ∀{n} → Var n → Tm n
    def    : ∀{n} → Tm n → Tm (suc n) → Tm n
    true   : ∀{n} → Tm n
    false  : ∀{n} → Tm n
    ite    : ∀{n} → Tm n → Tm n → Tm n → Tm n
    num    : ∀{n} → ℕ → Tm n
    isZero : ∀{n} → Tm n → Tm n
    _+o_   : ∀{n} → Tm n → Tm n → Tm n

  ⟦_⟧v : ∀{n} → I.Var n → Var n
  ⟦ I.vz ⟧v = vz
  ⟦ I.vs v ⟧v = vs ⟦ v ⟧v

  ⟦_⟧t : ∀{n} → I.Tm n → Tm n
  ⟦ I.var x ⟧t = var ⟦ x ⟧v
  ⟦ I.def x t ⟧t = def ⟦ x ⟧t ⟦ t ⟧t
  ⟦ I.true ⟧t = true
```

```
⟦ I.false ⟧t = false
⟦ I.ite t tr fa ⟧t = ite ⟦ t ⟧t ⟦ tr ⟧t ⟦ fa ⟧t
⟦ I.num x ⟧t = num x
⟦ I.isZero t ⟧t = isZero ⟦ t ⟧t
⟦ l I.+o r ⟧t = ⟦ l ⟧t +o ⟦ r ⟧t
```

Formal version of the term (let x:=1+2 in x+x) + (let y:=3+4 in y+y).

```
t = def (num 1 +o num 2) (v0 +o v0) +o def (num 3 +o num 4) (v0 +o v0)
```

Formal version of the term let x:=1+2 in ((x+x) + let y:=3+4 in x+y).

```
t' = def (num 1 +o num 2) ((v0 +o v0) +o def (num 3 +o num 4) (v1 +o v0))
```

Exercise 2.4 (compulsory). *Rewrite the following (closed) terms with De Bruijn notation.*

```
let x:=1 in x + let y:=x+1 in y + let z:=x+y in (x+z)+(y+x)
(let x:=1 in x) + let y:=1 in y + let z:=1+y in z+(y+1)
(let x:=1 in x + let y:=x+1 in y) + let z:=1 in z+z
(let x:=1 in x) + (let y:=1 in y) + let z:=1 in z+z
```

Exercise 2.5 (compulsory). *Rewrite the following (closed) terms with variable name notation.*

```
t1 = def true (v0 +o def v0 (v0 +o v1))
t2 = def true (def false (ite v0 v0 v1))
t3 = true +o def true (false +o def v0 (v1 +o v0))
t4 = def true (def false (def true (def false ((v0 +o v1) +o (v2 +o v3)))))
```

Exercise 2.6 (recommended). *Write a function which returns the number of occurrences of each variable in a term:*

```
countVars : {n : ℕ} → Tm n → Vec ℕ n
```

For example it should work as follows:

```
countVarsTest1 : countVars {1} ((v0 +o v0) +o v0) ≡ 3 :: []
countVarsTest2 : countVars {2} ((v1 +o v1) +o v0) ≡ 1 :: 2 :: []
countVarsTest3 : countVars {2} ((v0 +o v0) +o v1) ≡ 2 :: 1 :: []
countVarsTest4 : countVars {3} ((v2 +o v0) +o v1) ≡ 1 :: 1 :: 1 :: []
```

Note that we don't have variable names, so there is no function which extracts the names of the free variables from a term.

We also have dependent models and an eliminator.

2.2 Well-typed syntax

As a warmup for the real description of the language Def, we look at its well-typed description which does not include equations.

To obtain well-typed terms, we have to keep track not only of the maximal number of free variables that can occur in a term but also of their types. Thus we have a new sort of *contexts* which is a list of types. The empty context ◇ denotes no free variables, the context ◇ ▷ A ▷ B ▷ C has length three and the variables v0, v1, v2 have types C, B and A, respectively. Variables (and terms) are now indexed by their context and their type. The well-typed De Bruijn indices are recovered as follows.

```
v0 : {Γ : Con}{A : Ty}          → Tm (Γ ▷ A) A
v0 = var vz
```

```
v1 : {Γ : Con}{A B : Ty}          → Tm (Γ ▷ A ▷ B) A
v1 = var (vs vz)
v2 : {Γ : Con}{A B C : Ty}        → Tm (Γ ▷ A ▷ B ▷ C) A
v2 = var (vs (vs vz))
v3 : {Γ : Con}{A B C D : Ty}      → Tm (Γ ▷ A ▷ B ▷ C ▷ D) A
v3 = var (vs (vs (vs vz)))
v4 : {Γ : Con}{A B C D E : Ty}    → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E) A
v4 = var (vs (vs (vs (vs vz))))
v5 : {Γ : Con}{A B C D E F : Ty}  → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F) A
v5 = var (vs (vs (vs (vs (vs vz)))))
v6 : {Γ : Con}{A B C D E F G : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F ▷ G) A
v6 = var (vs (vs (vs (vs (vs (vs vz))))))
v7 : {Γ : Con}{A B C D E F G H : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F ▷ G ▷ H) A
v7 = var (vs (vs (vs (vs (vs (vs (vs vz)))))))
```

A Def-wt model has the following components.

```
record Model {i j k l} : Set (lsuc (i ⊔ j ⊔ k ⊔ l)) where
  infixl 5 _▷_
  infixl 6 _+o_
  field
    Ty     : Set i
    Nat    : Ty
    Bool   : Ty

    Con    : Set j
    ◇      : Con
    _▷_    : Con → Ty → Con

    Var    : Con → Ty → Set k
    vz     : ∀{Γ A} → Var (Γ ▷ A) A
    vs     : ∀{Γ A B} → Var Γ A → Var (Γ ▷ B) A

    Tm     : Con → Ty → Set l
    var    : ∀{Γ A}   → Var Γ A → Tm Γ A
    def    : ∀{Γ A B} → Tm Γ A → Tm (Γ ▷ A) B → Tm Γ B
    true   : ∀{Γ}     → Tm Γ Bool
    false  : ∀{Γ}     → Tm Γ Bool
    ite    : ∀{Γ A}   → Tm Γ Bool → Tm Γ A → Tm Γ A → Tm Γ A
    num    : ∀{Γ}     → ℕ → Tm Γ Nat
    isZero : ∀{Γ}     → Tm Γ Nat → Tm Γ Bool
    _+o_   : ∀{Γ}     → Tm Γ Nat → Tm Γ Nat → Tm Γ Nat
```

Examples:

$$\text{let } \underbrace{(\text{isZero (num 0)})}_{:\text{Tm} \diamond \text{Bool}} \ \underbrace{(\text{if v0 then num 0 else num 1})}_{:\text{Tm} (\diamond \triangleright \text{Bool}) \text{Nat}} : \text{Tm} \diamond \text{Nat}$$

$$\text{let} \ \underbrace{\text{num 2}}_{:\text{Tm} \diamond \text{Nat}} \ \underbrace{(\text{let} \ \underbrace{\text{v0}}_{\text{Tm} (\diamond \triangleright \text{Nat}) \text{Nat}} \ \underbrace{(\text{isZero (v0 + v1)})}_{\text{Tm} (\diamond \triangleright \text{Nat} \triangleright \text{Nat}) \text{Bool}})}_{\text{Tm} (\diamond \triangleright \text{Nat}) \text{Bool}} : \text{Tm} \diamond \text{Bool}$$

2.3 Well-typed syntax with equations

If we have local definitions, then in addition to the equations such as e.g. isZero (num 0) = true, we need equations like the following.

def t (v0 + v0) = t + t
def t (def t' (v0 + v1)) = t' + t
def t (def t' (v1 + (v0 + (v1 + num 3)))) = t + (t' + (t + num 3))

In general, we would like to have def t t' = t'[id,t] where t'[id,t] is the same as t', but all v0s are replaced by t, all v1s are replaced by v0s, all v2s are replaced by v1s, and so on. The nicest way to express this is to add a new sort of *substitutions*:

Sub : Con → Con → Set

If $\Gamma = \diamond \triangleright A_1 \triangleright A_2 \triangleright ... \triangleright A_n$, then a γ : Sub Δ Γ is a list of terms $\gamma = (t_1 , t_2 , ... t_n)$ where

t_1 : Tm Δ A_1
t_2 : Tm Δ A_2
...
t_n : Tm Δ A_n

We call Sub Δ Γ a substitution from Δ to Γ. A Sub Δ Γ contains enough information to turn a term defined in context Γ into a term defined in context Δ. This is witnessed by a new operator called *instantiation*:

_[_] : Tm Γ A → Sub Δ Γ → Tm Δ A

There is an identity substitution id : Sub Γ Γ which doesn't do anything, i.e. t [id] = t and with this we can express the equation for let expressions:

def t u = u [id , t]

Substitutions can be composed: $\gamma \circledcirc \delta$: Sub Θ Γ if γ : Sub Δ Γ and δ : Sub Θ Δ. The v0 De Bruijn index is now called q : Tm (Γ ▷ A) A and we have a substitution p : Sub (Γ ▷ A) Γ which forgets about the last element in the context: p ⊚ (σ , t) = p. The rest of the variables can be defined: v1 = q [p], v2 = q [p] [p], v3 = q [p] [p] [p], and so on.

```
record Model {i j k l} : Set (lsuc (i ⊔ j ⊔ k ⊔ l)) where
```

A Def-model consists of the following three groups of components:

- The substitution calculus (this is also called simply typed category with families (sCwF [6])):

```
  Con : Set i
  Sub : Con → Con → Set j
  _⊚_ : ∀{Γ Δ Θ} → Sub Δ Γ → Sub Θ Δ → Sub Θ Γ
  ass : ∀{Γ Δ Θ Ξ}{γ : Sub Δ Γ}{δ : Sub Θ Δ}{θ : Sub Ξ Θ} →
        (γ ⊚ δ) ⊚ θ ≡ γ ⊚ (δ ⊚ θ)
  id  : ∀{Γ} → Sub Γ Γ
  idl : ∀{Γ Δ}{γ : Sub Δ Γ} → id ⊚ γ ≡ γ
  idr : ∀{Γ Δ}{γ : Sub Δ Γ} → γ ⊚ id ≡ γ

  ◇   : Con
  ε   : ∀{Γ} → Sub Γ ◇
  ◇η  : ∀{Γ}{σ : Sub Γ ◇} → σ ≡ ε

  Ty  : Set k

  Tm  : Con → Ty → Set l
  _[_] : ∀{Γ Δ A} → Tm Γ A → Sub Δ Γ → Tm Δ A
  [∘] : ∀{Γ Δ Θ A}{t : Tm Γ A}{γ : Sub Δ Γ}{δ : Sub Θ Δ} →
        t [ γ ⊚ δ ] ≡ t [ γ ] [ δ ]
  [id] : ∀{Γ A}{t : Tm Γ A} → t [ id ] ≡ t
  _▷_ : Con → Ty → Con
  _,o_ : ∀{Γ Δ A} → Sub Δ Γ → Tm Δ A → Sub Δ (Γ ▷ A)
```

```
p     : ∀{Γ A} → Sub (Γ ▷ A) Γ
q     : ∀{Γ A} → Tm (Γ ▷ A) A
▷β₁   : ∀{Γ Δ A}{γ : Sub Δ Γ}{t : Tm Δ A} → p ∘ (γ ,o t) ≡ γ
▷β₂   : ∀{Γ Δ A}{γ : Sub Δ Γ}{t : Tm Δ A} → q [ γ ,o t ] ≡ t
▷η    : ∀{Γ Δ A}{γa : Sub Δ (Γ ▷ A)} → p ∘ γa ,o q [ γa ] ≡ γa
```

- Booleans:

```
Bool   : Ty
true   : ∀{Γ} → Tm Γ Bool
false  : ∀{Γ} → Tm Γ Bool
ite    : ∀{Γ A} → Tm Γ Bool → Tm Γ A → Tm Γ A → Tm Γ A
iteβ₁  : ∀{Γ A u v} → ite {Γ}{A} true u v ≡ u
iteβ₂  : ∀{Γ A u v} → ite {Γ}{A} false u v ≡ v
true[] : ∀{Γ Δ}{γ : Sub Δ Γ} → true [ γ ] ≡ true
false[] : ∀{Γ Δ}{γ : Sub Δ Γ} → false [ γ ] ≡ false
ite[]  : ∀{Γ A t u v Δ}{γ : Sub Δ Γ} → (ite {Γ}{A} t u v) [ γ ] ≡ ite (t [ γ ]) (u [ γ ]) (v [ γ ])
```

- Natural numbers:

```
Nat      : Ty
num      : ∀{Γ} → ℕ → Tm Γ Nat
isZero   : ∀{Γ} → Tm Γ Nat → Tm Γ Bool
_+o_     : ∀{Γ} → Tm Γ Nat → Tm Γ Nat → Tm Γ Nat
isZeroβ₁ : ∀{Γ} → isZero (num {Γ} 0) ≡ true
isZeroβ₂ : ∀{Γ n} → isZero (num {Γ} (1 + n)) ≡ false
+β       : ∀{Γ m n} → num {Γ} m +o num n ≡ num (m + n)
num[]    : ∀{Γ n Δ}{γ : Sub Δ Γ} → num n [ γ ] ≡ num n
isZero[] : ∀{Γ t Δ}{γ : Sub Δ Γ} → isZero t [ γ ] ≡ isZero (t [ γ ])
+[]      : ∀{Γ u v Δ}{γ : Sub Δ Γ} → (u +o v) [ γ ] ≡ (u [ γ ]) +o (v [ γ ])
```

We define let (def) as the following abbreviation.

```
def : ∀{Γ A B} → Tm Γ A → Tm (Γ ▷ A) B → Tm Γ B
def t u = u [ id ,o t ]
```

Variables are typed De Bruijn indices:

```
v0 : ∀{Γ A}         → Tm (Γ ▷ A) A
v0 = q
v1 : ∀{Γ A B}       → Tm (Γ ▷ A ▷ B) A
v1 = q [ p ]
v2 : ∀{Γ A B C}     → Tm (Γ ▷ A ▷ B ▷ C) A
v2 = q [ p ∘ p ]
v3 : ∀{Γ A B C D} → Tm (Γ ▷ A ▷ B ▷ C ▷ D) A
v3 = q [ p ∘ p ∘ p ]
v4 : {Γ : Con} → {A B C D E : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E) A
v4 = q [ p ∘ p ∘ p ∘ p ]
v5 : {Γ : Con} → {A B C D E F : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F) A
v5 = q [ p ∘ p ∘ p ∘ p ∘ p ]
v6 : {Γ : Con} → {A B C D E F G : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F ▷ G) A
v6 = q [ p ∘ p ∘ p ∘ p ∘ p ∘ p ]
v7 : {Γ : Con} → {A B C D E F G H : Ty} → Tm (Γ ▷ A ▷ B ▷ C ▷ D ▷ E ▷ F ▷ G ▷ H) A
v7 = q [ p ∘ p ∘ p ∘ p ∘ p ∘ p ∘ p ]
```

Some equalities that hold in any model:

```
▷η' : ∀{Γ A} → p ,o q ≡ id {Γ ▷ A}
▷η' {Γ}{A} =
  p ,o q
    ≡⟨ cong {A = Sub (Γ ▷ A) Γ × Tm (Γ ▷ A) A} (λ w → π₁ w ,o π₂ w) (idr ,×= [id]) ⁻¹ ⟩
  p ⊚ id ,o q [ id ]
    ≡⟨ ▷η ⟩
  id
    ∎

,∘ : ∀{Γ Δ Θ A}{γ : Sub Δ Γ}{t : Tm Δ A}{δ : Sub Θ Δ} →
  (γ ,o t) ⊚ δ ≡ γ ⊚ δ ,o t [ δ ]
,∘ {Γ}{Δ}{Θ}{A}{γ}{t}{δ} =
  (γ ,o t) ⊚ δ
    ≡⟨ ▷η ⁻¹ ⟩
  (p ⊚ ((γ ,o t) ⊚ δ) ,o q [ (γ ,o t) ⊚ δ ])
    ≡⟨ cong {A = Sub Θ Γ × Tm Θ A} (λ w → π₁ w ,o π₂ w) (ass ⁻¹ ,×= [∘]) ⟩
  ((p ⊚ (γ ,o t)) ⊚ δ ,o q [ γ ,o t ] [ δ ])
    ≡⟨ cong {A = Sub Θ Γ × Tm Θ A} (λ w → π₁ w ,o π₂ w)
          (cong (_⊚ δ) ▷β₁ ,×= cong (_[ δ ]) ▷β₂) ⟩
  γ ⊚ δ ,o t [ δ ]
    ∎

^∘ : ∀{Γ Δ}{γ : Sub Δ Γ}{A}{Θ}{δ : Sub Θ Δ}{t : Tm Θ A} →
  (γ ⊚ p ,o q) ⊚ (δ ,o t) ≡ (γ ⊚ δ ,o t)
^∘ {Γ}{Δ}{γ}{A}{Θ}{δ}{t} =
  (γ ⊚ p ,o q) ⊚ (δ ,o t)
    ≡⟨ ,∘ ⟩
  (γ ⊚ p ⊚ (δ ,o t) ,o q [ δ ,o t ])
    ≡⟨ cong (λ x → (x ,o q [ δ ,o t ])) ass ⟩
  (γ ⊚ (p ⊚ (δ ,o t)) ,o q [ δ ,o t ])
    ≡⟨ cong (λ x → (γ ⊚ x ,o q [ δ ,o t ])) ▷β₁ ⟩
  (γ ⊚ δ ,o q [ δ ,o t ])
    ≡⟨ cong (λ x → γ ⊚ δ ,o x) ▷β₂ ⟩
  (γ ⊚ δ ,o t)
    ∎
```

The term let x:= false in if x then true else isZero (if x then (num 0) else (num 1)) is equal to false in any model.

```
def false (ite q true (isZero (ite q (num 0) (num 1))))
  ≡⟨ refl ⟩
ite q true (isZero (ite q (num 0) (num 1))) [ id ,o false ]
  ≡⟨ ite[] ⟩
ite (q [ id ,o false ]) (true [ id ,o false ]) (isZero (ite q (num 0) (num 1)) [ id ,o false ])
  ≡⟨ cong {A = Tm ⋄ Bool}
        (λ x → ite x (true [ id ,o false ]) (isZero (ite q (num 0) (num 1)) [ id ,o false ]))
        ▷β₂ ⟩
ite false (true [ id ,o false ]) (isZero (ite q (num 0) (num 1)) [ id ,o false ])
  ≡⟨ iteβ₂ ⟩
isZero (ite q (num 0) (num 1)) [ id ,o false ]
  ≡⟨ isZero[] ⟩
isZero (ite q (num 0) (num 1) [ id ,o false ])
  ≡⟨ cong isZero ite[] ⟩
isZero (ite (q [ id ,o false ]) (num 0 [ id ,o false ]) (num 1 [ id ,o false ]))
  ≡⟨ cong {A = Tm ⋄ Bool}
        (λ x → isZero (ite x (num 0 [ id ,o false ]) (num 1 [ id ,o false ])))
        ▷β₂ ⟩
```

```
isZero (ite false (num 0 [ id ,o false ]) (num 1 [ id ,o false ]))
  ≡⟨ cong isZero iteβ₂ ⟩
isZero (num 1 [ id ,o false ])
  ≡⟨ cong isZero num[] ⟩
isZero (num 1)
  ≡⟨ isZeroβ₂ ⟩
false
  ∎
```

Exercise 2.7 (compulsory). *Prove* (def t u) [γ] ≡ def (t [γ]) (u [γ ⊙ p ,o q]) *for any* t, u, γ *in any model.*

A *program* of type A is an element of I.Tm ◇ A. That is what we can write "inside" the language. We could have introduced true and false only in the empty context:

```
module _
  (true◇ : Tm ◇ Bool)
  (false◇ : Tm ◇ Bool)
  where
```

Then the general versions could be defined using instantiation with ε and they obey the substitution rule:

```
true'    : ∀{Γ} → Tm Γ Bool
true'    = true◇ [ ε ]
false'   : ∀{Γ} → Tm Γ Bool
false'   = false◇ [ ε ]
true'[]  : ∀{Γ Δ}{γ : Sub Δ Γ} → true' [ γ ] ≡ true'
true'[]  {Γ}{Δ}{γ} = [◦] ⁻¹ ■ cong (true◇ [_]) ◇η
false'[] : ∀{Γ Δ}{γ : Sub Δ Γ} → false' [ γ ] ≡ false'
false'[] {Γ}{Δ}{γ} = [◦] ⁻¹ ■ cong (false◇ [_]) ◇η
```

2.3.1 Standard model

In the standard model, types are sets as before, contexts are iterated products of their constituent types, so they are also sets. Terms are not simply elements of their types because they depend on a context. This dependency is modelled using functions: a term of type A in context Γ is a function Γ → A. Substitutions are also functions.

```
St : Model
St = record
  { Con     = Set
  ; Sub     = λ Δ Γ → Δ → Γ
  ; _⊙_     = λ γ δ θ* → γ (δ θ*)
  ; ass     = refl
  ; id      = λ γ* → γ*
  ; idl     = refl
  ; idr     = refl

  ; ◇       = Lift ⊤
  ; ε       = _
  ; ◇η      = refl

  ; Ty      = Set

  ; Tm      = λ Γ A → Γ → A
```

```
  ; _[_]      = λ a γ δ* → a (γ δ*)
  ; [∘]       = refl
  ; [id]      = refl
  ; _▷_       = _×_
  ; _,o_      = λ γ t δ* → γ δ* , t δ*
  ; p         = π₁
  ; q         = π₂
  ; ▷β₁       = refl
  ; ▷β₂       = refl
  ; ▷η        = refl

  ; Bool      = 𝟚
  ; true      = λ _ → tt
  ; false     = λ _ → ff
  ; ite       = λ t u v γ* → if t γ* then u γ* else v γ*
  ; iteβ₁     = refl
  ; iteβ₂     = refl
  ; true[]    = refl
  ; false[]   = refl
  ; ite[]     = refl

  ; Nat       = ℕ
  ; num       = λ n γ* → n
  ; isZero    = λ t γ* → iteℕ tt (λ _ → ff) (t γ*)
  ; _+o_      = λ m n γ* → m γ* + n γ*
  ; isZeroβ₁  = refl
  ; isZeroβ₂  = refl
  ; +β        = refl
  ; num[]     = refl
  ; isZero[]  = refl
  ; +[]       = refl
  }
module St = Model St
```

A term in the empty context reproduces the interpretation of terms in the standard model of NatBool, e.g. Tm ◇ Nat = ⊤ → ℕ which is equivalent to having a natural number. All equations hold by definition in the standard model: we can just prove them by refl.

The standard model can be used to show that two closed terms are not equal in the syntax. If they were equal, then their interpretations into the standard model would be also equal.

```
not-equal : ¬ (
  I.ite (I.isZero (I.num 2)) (I.num 3) (I.num 12) ≡
  I.ite (I.isZero (I.num 0 I.+o I.num 0)) (I.num 11) (I.num {I.◇} 3))
not-equal e = 12≠11 (cong (λ t → St.⟦ t ⟧t (mk triv)) e)
  where
    12≠11 : ¬ (12 ≡ 11)
    12≠11 ()
```

For open terms, the standard model is not good enough for the same purpose. From normalisation, we will obtain that isZero (q {◇} +o num 2) is not equal in the syntax to false, but in the standard model they are both interpreted by the constant false function.

```
ex-open : St.⟦ I.isZero (I.q {I.◇} I.+o I.num 2) ⟧t ≡ St.⟦ I.false {I.◇ I.▷ I.Nat} ⟧t
ex-open = funext λ { (_ , n) → cong (iteℕ tt (λ _ → ff)) (+comm {n}{2}) }
```

2.3.2 Type inference

TODO

2.3.3 Normalisation

Before defining normal forms we have to group the operators specific to Bool and Nat as follows:

- constructors (introduction rules): operators which introduce elements of the type (true and false for Bool; num for Nat),
- destructors (elimination rules, eliminators): operators that can be used to eliminate an element of the type (ite for Bool; isZero and _+o_ for Nat),
- computation (β) rules: equations that explain what happens if a destructor is applied to a constructor (iteβ$_1$ and iteβ$_2$ for Bool; isZeroβ$_1$, isZeroβ$_2$ and +β for Nat),
- uniqueness (η) rules: equations that explain what happens if a constructor is applied to a destructor (we don't have any),
- substitution rules: explain how instantiation _[_] interacts with the operators (true[], false[], ite[], num[], isZero[], +[]).

Perhaps surprisingly, isZero is a destructor (of Nat) instead of a constructor (of Bool). This is because it computes depending on its input which is a Nat.

Each destructor has some *main arguments* (scrutinees). These are the arguments which are in constructor forms in the computation rule. The main argument of ite is the first one, the single arugment of isZero is a main argument, and both arguments of _+o_ are main. Once we know the main arguemnts, we can compute further using the β rule.

Variables, neutral terms and normal forms are defined as follows.

```
data Var : Con → Ty → Set where
  vz : ∀{Γ A} → Var (Γ ▷ A) A
  vs : ∀{Γ A B} → Var Γ A → Var (Γ ▷ B) A

data Ne (Γ : Con) : Ty → Set
data Nf (Γ : Con) : Ty → Set

data Ne Γ where
  var       : ∀{A} → Var Γ A → Ne Γ A
  iteNe     : ∀{A} → Ne Γ Bool → Nf Γ A → Nf Γ A → Ne Γ A
  isZeroNe  : Ne Γ Nat → Ne Γ Bool
  _+NeNe_ : Ne Γ Nat → Ne Γ Nat → Ne Γ Nat
  _+NeNf_ : Ne Γ Nat → ℕ → Ne Γ Nat
  _+NfNe_ : ℕ → Ne Γ Nat → Ne Γ Nat

data Nf Γ where
  neu       : ∀{A} → Ne Γ A → Nf Γ A
  trueNf    : Nf Γ Bool
  falseNf   : Nf Γ Bool
  numNf     : (n : ℕ) → Nf Γ Nat
```

Variables are typed De Bruijn indices (exactly as in the well-typed syntax in Section 2.2). Neutral terms and normal forms are defined mutually. Neutral terms are either variables or a destructor with a neutral main argument. For example, ite q true true is a neutral term because we don't know which computation rule (iteβ$_1$ or iteβ$_2$) to apply. Normal forms are either neutral terms or constructors. All non-main arguments of destructors and arguments of constructors have to be normal (we don't have any of the latter). The destructor _+_ has two main arguments, at least one of them has to be neutral. That is why we have three cases: both are neutral, the left is normal (that is, a natural number because that is the only parameter of num), the right is normal. For example, num 3 +o q and num 3 +o ite q (num 1) (num 2) are neutral (and hence normal), but num 3 +o num 2 is not normal because by +β it is equal to num 5.

Normal forms are embedded into the syntax.

```
⌜_⌝V : ∀{Γ A} → Var Γ A → Tm Γ A
⌜ vz ⌝V = q
⌜ vs x ⌝V = ⌜ x ⌝V [ p ]

⌜_⌝Ne : ∀{Γ A} → Ne Γ A → Tm Γ A
⌜_⌝Nf : ∀{Γ A} → Nf Γ A → Tm Γ A
⌜ var x ⌝Ne = ⌜ x ⌝V
⌜ iteNe t a a' ⌝Ne = ite ⌜ t ⌝Ne ⌜ a ⌝Nf ⌜ a' ⌝Nf
⌜ isZeroNe t ⌝Ne = isZero ⌜ t ⌝Ne
⌜ t +NeNe t' ⌝Ne = ⌜ t ⌝Ne +o ⌜ t' ⌝Ne
⌜ t +NeNf n' ⌝Ne = ⌜ t ⌝Ne +o num n'
⌜ n +NfNe t' ⌝Ne = num n +o ⌜ t' ⌝Ne
⌜ neu t ⌝Nf = ⌜ t ⌝Ne
⌜ trueNf ⌝Nf = true
⌜ falseNf ⌝Nf = false
⌜ numNf n ⌝Nf = num n
```

In contrast with NatBool (Section 1.5.2) where we were able to normalise using the standard model, for Def we have to define a separate model for normalisation. Terms in this model will be normal forms and substitutions will be maps from variables to normal forms. We first define these.

```
Nfs : Con → Con → Set
Nfs Δ Γ = ∀{A} → Var Γ A → Nf Δ A
⌜_⌝Nfs : ∀{Δ Γ} → Nfs Δ Γ → Sub Δ Γ
⌜_⌝Nfs {Γ = ◇} _ = ε
⌜_⌝Nfs {Γ = Γ ▷ A} γa = ⌜ (λ x → γa (vs x)) ⌝Nfs ,o ⌜ γa vz ⌝Nf
```

Now we define the interpretation of the destructors on normal forms. They return the neutral term when given a neutral argument, and follow the computation rule when given a constructor.

```
iteNf : ∀{Γ A} → Nf Γ Bool → Nf Γ A → Nf Γ A → Nf Γ A
iteNf (neu t) a a' = neu (iteNe t a a')
iteNf trueNf a a' = a
iteNf falseNf a a' = a'

_+NfNf_ : ∀{Γ} → Nf Γ Nat → Nf Γ Nat → Nf Γ Nat
neu tn +NfNf neu tn' = neu (tn +NeNe tn')
neu tn +NfNf numNf n = neu (tn +NeNf n)
numNf n +NfNf neu tn = neu (n +NfNe tn)
numNf n +NfNf numNf n' = numNf (n + n')

isZeroNf : ∀{Γ} → Nf Γ Nat → Nf Γ Bool
isZeroNf (neu n) = neu (isZeroNe n)
isZeroNf (numNf zero) = trueNf
isZeroNf (numNf (suc n)) = falseNf
```

Instantiation of normal forms by normal substitutions.

```
_[_]V : ∀{Γ A} → Var Γ A → ∀{Δ} → Nfs Δ Γ → Nf Δ A
x [ γa ]V = γa x

_[_]Ne : ∀{Γ A} → Ne Γ A → ∀{Δ} → Nfs Δ Γ → Nf Δ A
_[_]Nf : ∀{Γ A} → Nf Γ A → ∀{Δ} → Nfs Δ Γ → Nf Δ A
var x [ γ ]Ne = x [ γ ]V
iteNe t a a' [ γ ]Ne = iteNf (t [ γ ]Ne) (a [ γ ]Nf) (a' [ γ ]Nf)
(t +NeNe t') [ γ ]Ne = (t [ γ ]Ne) +NfNf (t' [ γ ]Ne)
(t +NeNf n') [ γ ]Ne = (t [ γ ]Ne) +NfNf numNf n'
(n +NfNe t') [ γ ]Ne = numNf n +NfNf (t' [ γ ]Ne)
```

```
isZeroNe t [ γ ]Ne = isZeroNf (t [ γ ]Ne)
neu a [ γ ]Nf = a [ γ ]Ne
trueNf [ γ ]Nf = trueNf
falseNf [ γ ]Nf = falseNf
numNf n [ γ ]Nf = numNf n
```

Substitution laws for normal destructors are proved by case analysis on the main arguments.

```
iteNf[] : ∀{Γ A t u v Δ}{γ : Nfs Δ Γ} →
  (iteNf {Γ}{A} t u v) [ γ ]Nf ≡ iteNf (t [ γ ]Nf) (u [ γ ]Nf) (v [ γ ]Nf)


+NfNf[] : ∀{Γ u v Δ}{γ : Nfs Δ Γ} → (u +NfNf v) [ γ ]Nf ≡ (u [ γ ]Nf) +NfNf (v [ γ ]Nf)


isZeroNf[] : ∀{Γ t Δ}{γ : Nfs Δ Γ} → isZeroNf t [ γ ]Nf ≡ isZeroNf (t [ γ ]Nf)
```

Composition of normal substitutions.

```
_◎Nf_ : ∀{Γ Δ Θ} → Nfs Δ Γ → Nfs Θ Δ → Nfs Θ Γ
(γ ◎Nf δ) x = γ x [ δ ]Nf
```

Composition laws for neutral terms and normal forms are proved by induction on t.

```
[∘]Ne : ∀{Γ Δ Θ A}{t : Ne Γ A}{γ : Nfs Δ Γ}{δ : Nfs Θ Δ} →
  t [ γ ◎Nf δ ]Ne ≡ t [ γ ]Ne [ δ ]Nf
[∘]Nf : ∀{Γ Δ Θ A}{t : Nf Γ A}{γ : Nfs Δ Γ}{δ : Nfs Θ Δ} →
  t [ γ ◎Nf δ ]Nf ≡ t [ γ ]Nf [ δ ]Nf
```

The identity normal substitution is just the identity function followed by var and neu.

```
idNf : ∀{Γ} → Nfs Γ Γ
idNf x = neu (var x)
```

The identity laws are proven by induction on t.

```
[id]Ne : ∀{Γ A}{t : Ne Γ A} → t [ idNf ]Ne ≡ neu t
[id]Nf : ∀{Γ A}{t : Nf Γ A} → t [ idNf ]Nf ≡ t
```

Extending a normal substitution with a normal form:

```
_,Nf_ : ∀{Γ Δ A} → Nfs Δ Γ → Nf Δ A → Nfs Δ (Γ ▷ A)
(γ ,Nf a) vz = a
(γ ,Nf a) (vs x) = γ x
```

The normal p substitution is just vs followed by var and neu. Now we build the model for normalisation.

```
N : ParaModel
N = record
  { Con• = Lift ⊤
  ; Sub• = λ Δ Γ → Nfs (π₁ Δ) (π₁ Γ)
  ; _◎•_ = λ γ δ → π₂ γ ◎Nf π₂ δ
  ; ass• = λ where {γ = γ} → funexti λ A → funext λ x → [∘]Nf {t = π₂ γ x} ⁻¹
  ; id• = idNf
  ; idl• = refl
  ; idr• = λ where {γ = γ} → funexti λ A → funext λ x → [id]Nf {t = π₂ γ x}
  ; ◇• = _
  ; ε• = λ ()
  ; ◇η• = funexti λ A → funext λ ()
```

```
  ; Ty• = Lift ⊤
  ; Tm• = λ Γ A → Nf (π₁ Γ) (π₁ A)
  ; _[_]• = λ a γ → π₂ a [ π₂ γ ]Nf
  ; [∘]• = λ where {t = t} → [∘]Nf {t = π₂ t}
  ; [id]• = [id]Nf
  ; _▷•_ = _
  ; _,o•_ = λ γ a → π₂ γ ,Nf π₂ a
  ; p• = pNf
  ; q• = neu (var vz)
  ; ▷β₁• = refl
  ; ▷β₂• = refl
  ; ▷η• = funexti λ A → funext λ { vz → refl ; (vs x) → refl }
  ; Bool• = _
  ; true• = trueNf
  ; false• = falseNf
  ; ite• = λ t a a' → iteNf (π₂ t) (π₂ a) (π₂ a')
  ; iteβ₁• = refl
  ; iteβ₂• = refl
  ; true[]• = refl
  ; false[]• = refl
  ; ite[]• = λ where {t = t} → iteNf[] {t = π₂ t}
  ; Nat• = _
  ; num• = numNf
  ; isZero• = λ t → isZeroNf (π₂ t)
  ; _+o•_ = λ t t' → π₂ t +NfNf π₂ t'
  ; isZeroβ₁• = refl
  ; isZeroβ₂• = refl
  ; +β• = refl
  ; num[]• = refl
  ; isZero[]• = λ where {t = t} → isZeroNf[] {t = π₂ t}
  ; +[]• = λ where {u = u} → +NfNf[] {u = π₂ u}
  }
module N = ParaModel N
```

Normalisation is interpretation into this model.

```
norm : ∀{Γ A} → Tm Γ A → Nf Γ A
norm = N.[[_]]t
```

Some examples of normalisation:

```
ex1 : norm (ite (isZero (num 1 +o num 2)) q false) ≡ falseNf {◇ ▷ Bool}
ex1 = refl

ex2 : norm (def (num 1) ((q +o q) +o ite false q q)) ≡ numNf {◇} 3
ex2 = refl
```

We used the standard model to show that two closed terms are not equal in the syntax. We can also use normalisation for the same purpose:

```
not-equal' : ¬ (
  I.ite (I.isZero (I.num 2)) (I.num 3) (I.num 12) ≡
  I.ite (I.isZero (I.num 0 I.+o I.num 0)) (I.num 11) (I.num {I.◇} 3))
not-equal' e = 12≠11 (cong norm e)
  where
    12≠11 : ¬ (numNf 12 ≡ numNf 11)
    12≠11 ()
```

And unlike the standard interpretation, normalisation can be used to prove inequality of open terms:

```
ex-open' : ¬ (I.isZero (I.q {I.◇} I.+o I.num 2) ≡ I.false)
ex-open' e with cong norm e
... | ()
```

For completeness, we first prove that the operations in the normalisation model commute with embedding into the syntax.

```
⌜ite⌝ : ∀{Γ A}{t : Nf Γ Bool}{a a' : Nf Γ A} →
  ⌜ iteNf t a a' ⌝Nf ≡ ite ⌜ t ⌝Nf ⌜ a ⌝Nf ⌜ a' ⌝Nf


⌜isZero⌝ : ∀{Γ}{t : Nf Γ Nat} → ⌜ isZeroNf t ⌝Nf ≡ isZero ⌜ t ⌝Nf


⌜+⌝ : ∀{Γ}{t t' : Nf Γ Nat} → ⌜ t +NfNf t' ⌝Nf ≡ ⌜ t ⌝Nf +o ⌜ t' ⌝Nf


⌜[]V⌝ : ∀{Γ Δ A}{x : Var Γ A}{γ : Nfs Δ Γ} → ⌜ x [ γ ]V ⌝Nf ≡ ⌜ x ⌝V [ ⌜ γ ⌝Nfs ]


⌜[]Ne⌝ : ∀{Γ Δ A}{a : Ne Γ A}{γ : Nfs Δ Γ} → ⌜ a [ γ ]Ne ⌝Nf ≡ ⌜ a ⌝Ne [ ⌜ γ ⌝Nfs ]
⌜[]Nf⌝ : ∀{Γ Δ A}{a : Nf Γ A}{γ : Nfs Δ Γ} → ⌜ a [ γ ]Nf ⌝Nf ≡ ⌜ a ⌝Nf [ ⌜ γ ⌝Nfs ]


⌜∘⌝ : ∀{Γ Δ Θ}{γ : Nfs Δ Γ}{δ : Nfs Θ Δ} → ⌜ γ ◎Nf δ ⌝Nfs ≡ ⌜ γ ⌝Nfs ◎ ⌜ δ ⌝Nfs
```

To show that pNf commutes with embedding, we first define renamings (substitutions only made up of variables) and by induction on γ, we prove that putting a vs on every variable is the same as precomposing the renaming by p.

```
Vars : Con → Con → Set
Vars Δ Γ = ∀{A} → Var Γ A → Var Δ A
⌜_⌝Vs : ∀{Δ Γ} → Vars Δ Γ → Sub Δ Γ
⌜_⌝Vs {Γ = ◇} _ = ε
⌜_⌝Vs {Γ = Γ ▷ A} γa = ⌜ (λ x → γa (vs x)) ⌝Vs ,o ⌜ γa vz ⌝V

⌜⌝∘p : ∀{Δ Γ A}{γ : Vars Δ Γ} → ⌜ (λ x → vs (γ x)) ⌝Vs ≡ (⌜ γ ⌝Vs ◎ p {Δ}{A})
```

By induction on Γ, we prove the following.

```
⌜vs⌝ : ∀{Γ A} → ⌜ vs ⌝Vs ≡ p {Γ}{A}


⌜⌝=⌜⌝ : ∀{Γ Δ}{γ : Vars Δ Γ} → ⌜ γ ⌝Vs ≡ ⌜ (λ x → neu (var (γ x)) ) ⌝Nfs
```

Now we have that p comomutes with embedding.

```
⌜pNf⌝ : ∀{Γ A} → ⌜ pNf ⌝Nfs ≡ p {Γ}{A}
⌜pNf⌝ = ⌜⌝=⌜⌝ {γ = vs} ⁻¹ ■ ⌜vs⌝
```

By induction on Γ, we prove that id commutes with embedding.

```
⌜id⌝ : ∀{Γ} → ⌜ idNf {Γ} ⌝Nfs ≡ id
```

Now we build the dependent model for completeness.

```
C : DepModel
C = record
  { Con• = λ _ → Lift ⊤
```

```
  ; Sub• = λ {Δ} _ {Γ} _ γ → Lift (⌜ N.⟦ γ ⟧s ⌝Nfs ≡ γ)
  ; _◎•_ = λ γ= δ= → mk (⌜∘⌝ ■ cong₂ _◎_ (un γ=) (un δ=))
  ; ass• = refl
  ; id• = mk ⌜id⌝
  ; idl• = refl
  ; idr• = refl
  ; ◇• = _
  ; ε• = mk refl
  ; ◇η• = refl
  ; Ty• = λ _ → Lift ⊤
  ; Tm• = λ {Γ}{A} _ _ a → Lift (⌜ N.⟦ a ⟧t ⌝Nf ≡ a)
  ; _[_]• = λ where {t = a} a= γ= → mk (⌜[]Nf⌝ {a = N.⟦ a ⟧t} ■ cong₂ _[_] (un a=) (un γ=))
  ; [∘]• = refl
  ; [id]• = refl
  ; _▷•_ = _
  ; _,o•_ = λ where {γ = γ}{a} γ= a= → mk (cong₂ _,o_ (un γ=) (un a=))
  ; p• = mk ⌜pNf⌝
  ; q• = mk refl
  ; ▷β₁• = refl
  ; ▷β₂• = refl
  ; ▷η• = refl
  ; Bool• = _
  ; true• = mk refl
  ; false• = mk refl
  ; ite• = λ { {t = t}{a}{a'} t= a= a'= →
    mk (⌜ite⌝ {t = N.⟦ t ⟧t} ■ cong₃ ite (un t=) (un a=) (un a'=)) }
  ; iteβ₁• = refl
  ; iteβ₂• = refl
  ; true[]• = refl
  ; false[]• = refl
  ; ite[]• = refl
  ; Nat• = _
  ; num• = λ _ → mk refl
  ; isZero• = λ where {t = t} t= → mk (⌜isZero⌝ {t = N.⟦ t ⟧t} ■ cong isZero (un t=))
  ; _+o•_ = λ { {u = t}{t'} t= t'= →
    mk (⌜+⌝ {t = N.⟦ t ⟧t}{t' = N.⟦ t' ⟧t} ■ cong₂ _+o_ (un t=) (un t'=)) }
  ; isZeroβ₁• = refl
  ; isZeroβ₂• = refl
  ; +β• = refl
  ; num[]• = refl
  ; isZero[]• = refl
  ; +[]• = refl
  }
module C = DepModel C

compl : ∀{Γ A}(t : Tm Γ A) → ⌜ norm t ⌝Nf ≡ t
compl t = un C.⟦ t ⟧t
```

We can use completeness to show that two terms are equal without doing all the complex equational reasoning.

```
e4 : ite true (num 3 +o num 2) (num 0) ≡ num 1 +o (num 2 +o (num 1 +o num {◇} 1))
e4 =
  ite true (num 3 +o num 2) (num 0)
                                    ≡⟨ compl (ite true (num 3 +o num 2) (num 0)) ⁻¹ ⟩
  num 5
                                    ≡⟨ compl (num 1 +o (num 2 +o (num 1 +o num 1))) ⟩
```

```
num 1 +o (num 2 +o (num 1 +o num 1))
                                    ∎
```

Stability of normalisation (there is no junk in normal forms) is proved by induction on variables, neutral terms and normal forms.

```
stabV : ∀{Γ A}(x : Var Γ A) → norm ⌜ x ⌝V ≡ neu (var x)
```

```
stabNe : ∀{Γ A}(t : Ne Γ A) → norm ⌜ t ⌝Ne ≡ neu t
stabNf : ∀{Γ A}(t : Nf Γ A) → norm ⌜ t ⌝Nf ≡ t
```

Decidability of equality of normal forms. We show how we do it for natural numbers, the rest of the proofs are analogous (see the source code).

```
_≟ℕ_ : (m n : ℕ) → Dec (Lift (m ≡ n))
zero ≟ℕ zero     = ι₁ (mk refl)
zero ≟ℕ suc n    = ι₂ (mk λ { (mk ()) })
suc m ≟ℕ zero    = ι₂ (mk λ { (mk ()) })
suc m ≟ℕ suc n with m ≟ℕ n
... | ι₁ (mk e)  = ι₁ (mk (cong suc e))
... | ι₂ f       = ι₂ (mk λ { (mk refl) → un f (mk refl)})
```

```
_≟Ty_ : (A B : Ty) → Dec (Lift (A ≡ B))
```

```
_≟Var_ : ∀{Γ A₀ A₁}(x₀ : Var Γ A₀)(x₁ : Var Γ A₁) →
  Dec (Σ (Lift (A₀ ≡ A₁)) λ e → Lift (transp (Var Γ) (un e) x₀ ≡ x₁))
```

```
_≟Ne_ : ∀{Γ A₀ A₁}(a₀ : Ne Γ A₀)(a₁ : Ne Γ A₁) →
  Dec (Σ (Lift (A₀ ≡ A₁)) λ e → Lift (transp (Ne Γ) (un e) a₀ ≡ a₁))
_≟Nf_ : ∀{Γ A}(a₀ a₁ : Nf Γ A) → Dec (Lift (a₀ ≡ a₁))
```

We obtain decidability of equality for all terms by checking whether the normal forms are equal. If they are, then by completeness, we have an equality between the two terms. If they are not equal, then if the terms were equal, then their normal forms would be also equal, which is a contradiction.

```
_≟Tm_ : ∀{Γ A}(t₀ t₁ : Tm Γ A) → Dec (Lift (t₀ ≡ t₁))
t₀ ≟Tm t₁ with norm t₀ ≟Nf norm t₁
... | ι₁ (mk e) = ι₁ (mk (compl t₀ ⁻¹ ■ cong ⌜_⌝Nf e ■ compl t₁))
... | ι₂ f = ι₂ (mk λ t₀₁ → un f (mk (cong norm (un t₀₁))))
```

Now it is very easy to prove equalities in the syntax. If we know that a witness of decidability is ι_1 by definition, then we can extract the witness from decidability of equality.

```
ex3 : isZero (q +o num 2) [ ε ,o num {◇} 0 ] ≡ false
ex3 = extract (isZero (q +o num 2) [ ε ,o num {◇} 0 ] ≟Tm false)
```

If we know that it is ι_2 by definition, we can use extract' to obtain falsity:

```
ex-open'' : ¬ (isZero (q {◇} +o num 2) ≡ false)
ex-open'' = extract' (isZero (q {◇} +o num 2) ≟Tm false)
```

TODO: show that "constructors are injective" also follows from normalisation.

Chapter 3

Function space

Learning goals of this chapter

Rules for function space. Uniqueness rule. Currying. Relationship of metatheoretic and object theoretic function space. Bidirectional typechecking. Tait's method for normalisation.

An STT model is a Def model which has the following additional operators and equations.

```
_⇒_  : Ty → Ty → Ty
lam  : ∀{Γ A B} → Tm (Γ ▷ A) B → Tm Γ (A ⇒ B)
_$_  : ∀{Γ A B} → Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
⇒β   : ∀{Γ A B}{t : Tm (Γ ▷ A) B}{u : Tm Γ A} → lam t $ u ≡ t [ id ,o u ]
⇒η   : ∀{Γ A B}{t : Tm Γ (A ⇒ B)} → lam (t [ p ] $ q) ≡ t
lam[] : ∀{Γ A B}{t : Tm (Γ ▷ A) B}{Δ}{γ : Sub Δ Γ} →
        (lam t) [ γ ] ≡ lam (t [ γ ⊚ p ,o q ])
$[]  : ∀{Γ A B}{t : Tm Γ (A ⇒ B)}{u : Tm Γ A}{Δ}{γ : Sub Δ Γ} →
        (t $ u) [ γ ] ≡ t [ γ ] $ u [ γ ]
```

The operator _⇒_ is right-associative, e.g. A ⇒ B ⇒ C = A ⇒ (B ⇒ C). In an STT model, types are binary trees with Nat or Bool at the leaves. lam is a binder, it binds a variable (of type A) in its first (and only) explicit argument.

In STT, there are three ways to form types: Bool, Nat, and for any two types A and B, we also have a type A ⇒ B. We group the operators and equations for _⇒_:

- type introduction: _⇒_,
- constructor (called abstraction): lam,
- destructor (called application): _$_,
- computation: ⇒β,
- uniqueness: ⇒η,
- substitution: lam[], $[].

The function that adds 2 to its input is given as follows.

```
add2 : Tm ◇ (Nat ⇒ Nat)
add2 = lam (v0 +o num 2) -- λx.x+2
```

Note that isZero does not have a function type (in fact, we introduced isZero before _⇒_). But we can define a function which acts like isZero as follows.

```
isZero' : ∀{Γ} → Tm Γ (Nat ⇒ Bool)
isZero' = lam (isZero v0) -- λx.isZero x
```

It obeys a computation rule similar to $\mathsf{isZero}\beta_1$ which is admissible:

```
isZero'β₁ : ∀{Γ} → isZero' {Γ} $ num 0 ≡ true
isZero'β₁ =
  isZero' $ num 0
    ≡⟨ refl ⟩
  lam (isZero v0) $ num 0
    ≡⟨ ⇒β ⟩
  isZero v0 [ id ,o num 0 ]
    ≡⟨ isZero[] ⟩
  isZero (v0 [ id ,o num 0 ])
    ≡⟨ cong isZero ▷β₂ ⟩
  isZero (num 0)
    ≡⟨ isZeroβ₁ ⟩
  true
    ∎
```

Exercise 3.1 (compulsory)**.** *Prove the other computation rule:*

```
isZero'β₂ : ∀{Γ} → isZero' {Γ} $ num 1 ≡ false
```

The function that constantly returns true is the following. Its domain type can be anything.

```
constTrue : ∀{A} → Tm ◇ (A ⇒ Bool)
constTrue = lam true -- λx.true
```

A constant function is really constant:

```
constIsConst : ∀{Γ A B}{t : Tm Γ B}{u : Tm Γ A} → lam (t [ p ]) $ u ≡ t
constIsConst {t = t}{u = u} =
  lam (t [ p ]) $ u
    ≡⟨ ⇒β ⟩
  t [ p ] [ id ,o u ]
    ≡⟨ [∘] ⁻¹ ⟩
  t [ p ◎ (id ,o u) ]
    ≡⟨ cong (t [_]) ▷β₁ ⟩
  t [ id ]
    ≡⟨ [id] ⟩
  t
    ∎
```

Functions with multiple parameters can be defined using currying. A function whose input is a Nat and a Bool and whose output is a Nat is the same as a function whose domain is Nat and whose codomain is Bool ⇒ Nat:

```
curried : Tm ◇ (Nat ⇒ (Bool ⇒ Nat))
curried = lam (lam (v1 +o ite v0 (num 1) (num 2))) -- λ x y . x + if y then 1 else 2
```

The type of the following term also contains two arrows but it is parenthesised differently. It is a higher-order function where the domain is a function type.

```
higher : Tm ◇ ((Bool ⇒ Nat) ⇒ Nat)
higher = lam (v0 $ isZero (v0 $ true)) -- λ f . f $ (isZero (f $ true))
```

Exercise 3.2 (compulsory). *Define the function* apply3 : Tm ◇ ((Bool ⇒ Bool) ⇒ Bool ⇒ Bool) *which applies the function given as the first parameter three times to the boolean given as the second parameter.*

Exercise 3.3 (compulsory). *What is the type of the following terms (in the empty context)?*

```
lam (isZero q)
lam (lam (ite q (isZero (q [ p ])) false))
lam (lam ((num 1 +o ((q [ p ]) $ q))))
lam (lam ((num 1 +o (q $ (q [ p ])))))
```

Note the difference between the types of isZero and isZero':

```
isZero  : Tm ◇ Nat → Tm ◇ Bool
isZero' : Tm ◇ (Nat ⇒ Bool)
```

One is a metatheoretic function relating terms, the other is a function in our object theory. We show that the latter is stronger than the former: the types

```
EXT = Σsp (∀ Γ → Tm Γ A → Tm Γ B) λ f → ∀{Γ Δ γ a} → f Γ a [ γ ] ≡ f Δ (a [ γ ])
INT = Tm ◇ (A ⇒ B)
```

are isomorphic (for any A, B in any model).

```
toINT : EXT → INT
toINT f = lam (π₁ f (◇ ▷ A) q)
toEXT : INT → EXT
toEXT t = (λ Γ a → t [ ε ] $ a) , λ {Γ}{Δ}{γ}{a} →
  (t [ ε ] $ a) [ γ ]
                    ≡⟨ $[] ⟩
  t [ ε ] [ γ ] $ a [ γ ]
                    ≡⟨ cong (_$ a [ γ ]) ([∘] ⁻¹) ⟩
  t [ ε ⊚ γ ] $ a [ γ ]
                    ≡⟨ cong (λ x → t [ x ] $ a [ γ ]) ◇η ⟩
  t [ ε ] $ a [ γ ]
                    ∎
extRoundtrip : (f : EXT) → toEXT (toINT f) ≡ f
extRoundtrip f = (funext λ Γ → funext λ a →
  lam (π₁ f (◇ ▷ A) q) [ ε ] $ a
                                    ≡⟨ cong (_$ a) lam[] ⟩
  lam (π₁ f (◇ ▷ A) q [ ε ⊚ p ,o q ]) $ a
                                    ≡⟨ cong (λ x → lam x $ a) (π₂ f) ⟩
  lam (π₁ f (Γ ▷ A) (q [ ε ⊚ p ,o q ])) $ a
                                    ≡⟨ cong (λ x → lam (π₁ f (Γ ▷ A) x) $ a) ▷β₂ ⟩
  lam (π₁ f (Γ ▷ A) q) $ a
                                    ≡⟨ ⇒β ⟩
  π₁ f (Γ ▷ A) q [ id ,o a ]
                                    ≡⟨ π₂ f ⟩
  π₁ f Γ (q [ id ,o a ])
                                    ≡⟨ cong (π₁ f Γ) ▷β₂ ⟩
  π₁ f Γ a
                                    ∎) ,=-
intRoundtrip : (t : INT) → toINT (toEXT t) ≡ t
intRoundtrip t = cong (λ γ → lam (t [ γ ] $ q)) (◇η ⁻¹) ■ ⇒η
```

Exercise 3.4 (recommended). *Show that for any model,* Tm Γ (A ⇒ B) *is isomorphic to* Σ (∀{Δ}(γ : Sub Δ Γ) → Tm Δ A → Tm Δ B) λ f → ∀{γ a δ} → f γ a [δ] ≡ f (γ ∘ δ) (a [δ])

3.1 Standard model

The new components in the standard model:

```
; _⇒_  = λ A B → A → B
; lam  = λ t γ* α* → t (γ* , α*)
; _$_  = λ t u γ* → t γ* (u γ*)
; ⇒β   = refl
; ⇒η   = refl
; lam[] = refl
; $[]   = refl
```

Exercise 3.5. *Extend the syntax with an isOne* isOne : Tm ⋄ Nat → Tm ⋄ Bool. *Hint: interpret the input term in the standard model, calculate the result in the metatheory. Is an internal* fib : Tm ⋄ (Nat ⇒ Bool) *definable in STT?*

3.2 Type checking and inference

TODO

3.3 Normalisation

Normal forms are like in Def, but there is a twist.

```
data Var : Con → Ty → Set where
  vz : ∀{Γ A} → Var (Γ ▷ A) A
  vs : ∀{Γ A B} → Var Γ A → Var (Γ ▷ B) A

data Ne (Γ : Con) : Ty → Set
data Nf (Γ : Con) : Ty → Set

data Ne Γ where
  var       : ∀{A} → Var Γ A → Ne Γ A
  _$Ne_     : ∀{A B} → Ne Γ (A ⇒ B) → Nf Γ A → Ne Γ B
  iteNe     : ∀{A} → Ne Γ Bool → Nf Γ A → Nf Γ A → Ne Γ A
  isZeroNe  : Ne Γ Nat → Ne Γ Bool
  _+NeNe_ : Ne Γ Nat → Ne Γ Nat → Ne Γ Nat
  _+NeNf_ : Ne Γ Nat → ℕ → Ne Γ Nat
  _+NfNe_ : ℕ → Ne Γ Nat → Ne Γ Nat

data Nf Γ where
  neuBool   : Ne Γ Bool → Nf Γ Bool
  neuNat    : Ne Γ Nat → Nf Γ Nat
  lamNf     : ∀{A B} → Nf (Γ ▷ A) B → Nf Γ (A ⇒ B)
  trueNf    : Nf Γ Bool
  falseNf   : Nf Γ Bool
  numNf     : (n : ℕ) → Nf Γ Nat


⌜_⌝V : ∀{Γ A} → Var Γ A → Tm Γ A


⌜_⌝Ne : ∀{Γ A} → Ne Γ A → Tm Γ A
⌜_⌝Nf : ∀{Γ A} → Nf Γ A → Tm Γ A
```

Only neutral terms of base types (Bool and Nat) are included in normal forms (this was also true in Def because there were no other types). Without this restriction, we would have two different normal forms for a variabe of a function type: q which is equal to lam (q [p]) $ q. We have to do this whenever we have uniqueness rules.

Exercise 3.6. *Write down the normal form of the following terms.*

```
q                : Tm (◇ ▷ Nat ⇒ Bool) (Nat ⇒ Bool)
q                : Tm (◇ ▷ Bool) Bool
q                : Tm (◇ ▷ Nat) Nat
num 1 +o q       : Tm ◇ (◇ ▷ Nat) Nat
num 1 +o num 2   : Tm ◇ Nat
isZero (q + num 1) : Tm (◇ ▷ Nat) Bool
isZero (num 2)   : Tm ◇ Bool
```

Recursive normalisation as for Def does not work as for STT. This is because when defining normal application we need instantiation and when defining instantiation, we need normal application and it is not obvious that such mutual definition terminates. [1]

```
_$Nf_  : Nf Γ (A ⇒ B) → Nf Γ A → Nf Γ B
_[_]Ne : Ne Γ A → Sub Δ Γ → Nf Δ A

lamNf t $Nf a = t [ id ,o a ]Nf

(t $Ne a) [ γ ]Ne = t [ γ ]Ne $Nf a [ γ ]Ne
```

However there is a simple direct method for proving normalisation which was introduced by Tait [19]. This is also called the method of logical predicates (logical relations) and normalisation by evaluation. Instead of proving normalisation by induction on terms, we do induction on types as well. For each type, we define a predicate on terms of that type. For contexts we define a predicate on substitutions into that type by iterating the predicate for the types in the codomain of the substitution. For terms $\mathbf{Tm}\,\Gamma\,A$, we show by induction that they respect the predicate: if the predicate holds for a substitution into Γ, then the predicate holds at A holds for the term instantiated by this substitution. Similarly, substitutions respect the predicate. If the predicate holds for a term, then it is in normal form.

Here we show a simplified version of normalisation called *canonicity*. This only gives normalisation for terms of type **Bool** and **Nat** in the empty context. We show some parts of the proof. TODO: one equation left.

```
Canon : DepModel
Canon = record
  { Con• = λ Γ → Sub ◇ Γ → Set
  ; Sub• = λ Δ• Γ• γ → ∀{δ◇} → Δ• δ◇ → Γ• (γ ◎ δ◇)
  ; _◎•_ = λ {_}{_}{_}{Γ•} γ• δ• θ* → transp Γ• (ass ⁻¹) (γ• (δ• θ*))

  ; id• = λ {_}{Γ•} γ* → transp Γ• (idl ⁻¹) γ*

  ; ◇• = λ _ → Lift ⊤
  ; ε• = _

  ; Tm• = λ {Γ} Γ• A• a → ∀{γ◇} → Γ• γ◇ → A• (a [ γ◇ ])
  ; _[_]• = λ where {A• = A•} a• γ• {δ◇} δ* → transp A• [∘] (a• (γ• δ*))

  ; _▷•_ = λ Γ• A• γa◇ → Γ• (p ◎ γa◇) × A• (q [ γa◇ ])
  ; _,o•_ = λ { {Γ• = Γ•}{A• = A•} γ• a• δ* →
    transp Γ• (▷β₁ ⁻¹ ■ cong (p ◎_) (,∘ ⁻¹)) (γ• δ*) ,
    transp A• (▷β₂ ⁻¹ ■ cong (q [_]) (,∘ ⁻¹)) (a• δ*) }
  ; p• = π₁
  ; q• = π₂
```

[1] It does terminate and normalisation can be defined this way, this is called hereditary substitution [13].

```
; Bool• = λ t → Lift (t ≡ true) ⊎ Lift (t ≡ false)
; true• = λ _ → ι₁ (mk true[])
; false• = λ _ → ι₂ (mk false[])
; ite• = λ {Γ}{A}{Γ•}{A•}{t}{u}{v} t• u• v• {γ◇} γ* → case (t• γ*)
    (λ e → transp A• (iteβ₁ ⁻¹ ■ cong (λ z → ite z (u [ γ◇ ]) (v [ γ◇ ])) (un e) ⁻¹ ■ ite[] ⁻¹)
                     (u• γ*))
    (λ e → transp A• (iteβ₂ ⁻¹ ■ cong (λ z → ite z (u [ γ◇ ]) (v [ γ◇ ])) (un e) ⁻¹ ■ ite[] ⁻¹)
                     (v• γ*))

; _⇒•_ = λ {A}{B} A• B• t → {a : Tm ◇ A} → A• a → B• (t $ a)
; lam• = λ {Γ}{A}{B}{Γ•}{A•}{B•}{t} t• {γ◇} γ* {a◇} a* →
    transp B• (⇒β' ⁻¹) (t• {γ◇ ,o a◇} (transp Γ• (▷β₁ ⁻¹) γ* , transp A• (▷β₂ ⁻¹) a*))
; _$•_ = λ {Γ}{A}{B}{Γ•}{A•}{B•}{t}{u} t• u• {γ◇} γ* → transp B• ($[] ⁻¹) (t• γ* (u• γ*))

; Nat• = λ t → Σ ℕ λ n → Lift (t ≡ ⌜ numNf n ⌝Nf)
; num• = λ n _ → n , mk num[]
; isZero• = λ { {t = t} t• {γ◇} γ* → indℕ
    (λ n → t [ γ◇ ] ≡ num n →
      Lift (isZero t [ γ◇ ] ≡ true) ⊎ Lift (isZero t [ γ◇ ] ≡ false))
    (λ e → ι₁ (mk (isZero[] ■ cong isZero e ■ isZeroβ₁)))
    (λ _ _ e → ι₂ (mk (isZero[] ■ cong isZero e ■ isZeroβ₂)))
    (π₁ (t• γ*))
    (un (π₂ (t• γ*))) }
; _+o•_ = λ u• v• γ* →
  π₁ (u• γ*) + π₁ (v• γ*) ,
  mk (+[] ■ (cong₂ _+o_ (un (π₂ (u• γ*))) (un (π₂ (v• γ*))) ■ +β))
```

TODO: show full normalisation.

Chapter 4

Product and sum types

Learning goals of this chapter

Nullary and binary products. Nullary and binary sums, enumerations, booleans encoded as sums, option type. More equalities in the standard model than in the syntax.

TODO: named records, curry, uncurry, isomorphisms of finite types, normalisation (hard for sums).

A Fin model consists of a substitution calculus (sCwF, see Section 2.3), components for function space and components for the following four new type formers.

- Binary products:

$$
\begin{array}{ll}
\_\times o\_ & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\langle \_,\_ \rangle & : \forall\{\Gamma\ A\ B\} \to \mathsf{Tm}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ B \to \mathsf{Tm}\ \Gamma\ (A \times o\ B) \\
\mathsf{fst} & : \forall\{\Gamma\ A\ B\} \to \mathsf{Tm}\ \Gamma\ (A \times o\ B) \to \mathsf{Tm}\ \Gamma\ A \\
\mathsf{snd} & : \forall\{\Gamma\ A\ B\} \to \mathsf{Tm}\ \Gamma\ (A \times o\ B) \to \mathsf{Tm}\ \Gamma\ B \\
\times\beta_1 & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ \Gamma\ B\} \to \mathsf{fst}\ \langle\ u\ ,\ v\ \rangle \equiv u \\
\times\beta_2 & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ \Gamma\ B\} \to \mathsf{snd}\ \langle\ u\ ,\ v\ \rangle \equiv v \\
\times\eta & : \forall\{\Gamma\ A\ B\}\{t : \mathsf{Tm}\ \Gamma\ (A \times o\ B)\} \to \langle\ \mathsf{fst}\ t\ ,\ \mathsf{snd}\ t\ \rangle \equiv t \\
,[] & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ \Gamma\ B\}\{\Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \\
 & \quad \langle\ u\ ,\ v\ \rangle\ [\ \gamma\ ] \equiv \langle\ u\ [\ \gamma\ ]\ ,\ v\ [\ \gamma\ ]\ \rangle
\end{array}
$$

  A term of the binary product type A ×o B (other notations: A × B, record { a : A , b : B}) is an ordered pair of a term of type A and another of type B. The constructor is pairing, and there are two destructors: first and second projections. The computation rules say what happens if we project out the first or second element of a pair, the uniqueness rule says that any term of the product type is a pair. We only need a substitution rule for the constructor, the substitution rules for the destructors can be proven, see below. The rules for binary products can be summarised by the following isomorphism: Tm Γ (A ×o B) ≅ Tm Γ A × Tm Γ B. The left to right direction is given by fst and snd, the right to left direction by ⟨_,_⟩, the proofs that the roundtrips are identity are ×β and ×η.

- Nullary products:

$$
\begin{array}{ll}
\mathsf{Unit} & : \mathsf{Ty} \\
\mathsf{trivial} & : \forall\{\Gamma\} \to \mathsf{Tm}\ \Gamma\ \mathsf{Unit} \\
\mathsf{Unit}\eta & : \forall\{\Gamma\}\{u : \mathsf{Tm}\ \Gamma\ \mathsf{Unit}\} \to u \equiv \mathsf{trivial}
\end{array}
$$

  The nullary product is the unit (top, ()) type with only one constructor and no destructor: this shows that there is no information in a term of type Unit, there is no way to use such a term. This is why in some programming languages this type is called void – when a function does not return a value, its return type is unit. The summarising isomorphism: Tm Γ Unit ≅ ⊤.

- Binary sums:

```
_+o_  : Ty → Ty → Ty
inl   : ∀{Γ A B} → Tm Γ A → Tm Γ (A +o B)
inr   : ∀{Γ A B} → Tm Γ B → Tm Γ (A +o B)
caseo : ∀{Γ A B C} → Tm (Γ ▷ A) C → Tm (Γ ▷ B) C → Tm (Γ ▷ A +o B) C
+β₁   : ∀{Γ A B C}{u : Tm (Γ ▷ A) C}{v : Tm (Γ ▷ B) C}{t : Tm Γ A} →
        caseo u v [ id ,o inl t ] ≡ u [ id ,o t ]
+β₂   : ∀{Γ A B C}{u : Tm (Γ ▷ A) C}{v : Tm (Γ ▷ B) C}{t : Tm Γ B} →
        caseo u v [ id ,o inr t ] ≡ v [ id ,o t ]
+η    : ∀{Γ A B C}{t : Tm (Γ ▷ A +o B) C} →
        caseo (t [ p ,o inl q ]) (t [ p ,o inr q ]) ≡ t
inl[] : ∀{Γ A B}{t : Tm Γ A}{Δ}{γ : Sub Δ Γ} →
        (inl {B = B} t) [ γ ] ≡ inl (t [ γ ])
inr[] : ∀{Γ A B}{t : Tm Γ B}{Δ}{γ : Sub Δ Γ} →
        (inr {A = A} t) [ γ ] ≡ inr (t [ γ ])
case[] : ∀{Γ A B C}{u : Tm (Γ ▷ A) C}{v : Tm (Γ ▷ B) C}{Δ}{γ : Sub Δ Γ} →
        (caseo u v) [ γ ⊚ p ,o q ] ≡
        caseo (u [ γ ⊚ p ,o q ]) (v [ γ ⊚ p ,o q ])
```

  caseo binds a variable of type A in its first explicit argument and a variable of type B in its second explicit argument. They are specified by the following isomorphism.

  (inl q, inr q) : Tm (Γ ▷ A +o B) C ≅ Tm (Γ ▷ A) C × Tm (Γ ▷ B) C : case

  TODO: prove this: f ∘ case u v = case (f∘u) (f∘v).

- Nullary sums:

```
Empty   : Ty
absurd  : ∀{Γ A} → Tm (Γ ▷ Empty) A
Emptyη : ∀{Γ A}{t : Tm (Γ ▷ Empty) A} → t ≡ absurd
```

  The isomorphism: Tm (Γ ▷ Empty) A ≅ ⊤.

The following are provable in any algebra:

```
fst[] : ∀{Γ Δ A B}{t : Tm Γ (A ×o B)}{γ : Sub Δ Γ} →
  (fst t) [ γ ] ≡ fst (t [ γ ])
fst[] {Γ}{Δ}{A}{B}{t}{γ} =
  fst t [ γ ]
                    ≡⟨ ×β₁ ⁻¹ ⟩
  fst ⟨ fst t [ γ ] , snd t [ γ ] ⟩
                    ≡⟨ cong fst (,[] ⁻¹) ⟩
  fst (⟨ fst t , snd t ⟩ [ γ ])
                    ≡⟨ cong (λ x → fst (x [ γ ])) ×η ⟩
  fst (t [ γ ])
                    ∎

snd[] : ∀{Γ Δ A B}{t : Tm Γ (A ×o B)}{γ : Sub Δ Γ} →
  (snd t) [ γ ] ≡ snd (t [ γ ])
snd[] {Γ}{Δ}{A}{B}{t}{γ} =
  snd t [ γ ]
                    ≡⟨ ×β₂ ⁻¹ ⟩
  snd ⟨ fst t [ γ ] , snd t [ γ ] ⟩
                    ≡⟨ cong snd (,[] ⁻¹) ⟩
  snd (⟨ fst t , snd t ⟩ [ γ ])
                    ≡⟨ cong (λ x → snd (x [ γ ])) ×η ⟩
```

```
  snd (t [ γ ])
                ∎

trivial[] : ∀{Γ Δ}{γ : Sub Δ Γ} → trivial [ γ ] ≡ trivial
trivial[] = Unitη

+β₁' : ∀{Γ A B C}{u : Tm (Γ ▷ A) C}{v : Tm (Γ ▷ B) C}{Δ}{γ : Sub Δ Γ}{t : Tm Δ A} →
  caseo u v [ γ ,o inl t ] ≡ u [ γ ,o t ]
+β₁' {Γ}{A}{B}{C}{u}{v}{Δ}{γ}{t} =
  caseo u v [ γ ,o inl t ]
    ≡⟨ cong (λ x → caseo u v [ x ,o inl t ]) (idr ⁻¹) ⟩
  caseo u v [ (γ ⊚ id ,o inl t) ]
    ≡⟨ cong (caseo u v [_]) (^∘ ⁻¹) ⟩
  caseo u v [ (γ ⊚ p ,o q) ⊚ (id ,o inl t) ]
    ≡⟨ [∘] ⟩
  caseo u v [ γ ⊚ p ,o q ] [ id ,o inl t ]
    ≡⟨ cong (_[ id ,o inl t ]) case[] ⟩
  caseo (u [ γ ⊚ p ,o q ]) (v [ γ ⊚ p ,o q ]) [ id ,o inl t ]
    ≡⟨ +β₁ ⟩
  u [ γ ⊚ p ,o q ] [ id ,o t ]
    ≡⟨ [∘] ⁻¹ ⟩
  u [ (γ ⊚ p ,o q) ⊚ (id ,o t) ]
    ≡⟨ cong (u [_]) ^∘ ⟩
  u [ γ ⊚ id ,o t ]
    ≡⟨ cong (λ x → u [ x ,o t ]) idr ⟩
  u [ γ ,o t ]
    ∎

+β₂' : ∀{Γ A B C}{u : Tm (Γ ▷ A) C}{v : Tm (Γ ▷ B) C}{Δ}{γ : Sub Δ Γ}{t : Tm Δ B} →
  caseo u v [ γ ,o inr t ] ≡ v [ γ ,o t ]
+β₂' {Γ}{A}{B}{C}{u}{v}{Δ}{γ}{t} =
  caseo u v [ γ ,o inr t ]
    ≡⟨ cong (λ x → caseo u v [ x ,o inr t ]) (idr ⁻¹) ⟩
  caseo u v [ (γ ⊚ id ,o inr t) ]
    ≡⟨ cong (caseo u v [_]) (^∘ ⁻¹) ⟩
  caseo u v [ (γ ⊚ p ,o q) ⊚ (id ,o inr t) ]
    ≡⟨ [∘] ⟩
  caseo u v [ γ ⊚ p ,o q ] [ id ,o inr t ]
    ≡⟨ cong (_[ id ,o inr t ]) case[] ⟩
  caseo (u [ γ ⊚ p ,o q ]) (v [ γ ⊚ p ,o q ]) [ id ,o inr t ]
    ≡⟨ +β₂ ⟩
  v [ γ ⊚ p ,o q ] [ id ,o t ]
    ≡⟨ [∘] ⁻¹ ⟩
  v [ (γ ⊚ p ,o q) ⊚ (id ,o t) ]
    ≡⟨ cong (v [_]) ^∘ ⟩
  v [ γ ⊚ id ,o t ]
    ≡⟨ cong (λ x → v [ x ,o t ]) idr ⟩
  v [ γ ,o t ]
    ∎

absurd[] : ∀{Γ A}{Δ}{γ : Sub Δ Γ} →
  absurd {Γ}{A} [ γ ⊚ p ,o q ] ≡ absurd
absurd[] = Emptyη
```

4.1 Applications

Enumerations can be defined using the unit and sum types. For example, the type with three elements:

```
Three : Ty
Three = Unit +o (Unit +o Unit)
```

Its three elements:

```
three0 three1 three2 : ∀{Γ} → Tm Γ Three
three0 = inl trivial
three1 = inr (inl trivial)
three2 = inr (inr trivial)
```

Elimination principle of Three.

Exercise 4.1 (recommended)**.** *The second and third rules are exercises.*

```
caseThree : ∀{Γ A} → Tm Γ Three → Tm Γ A → Tm Γ A → Tm Γ A → Tm Γ A
caseThree t u v w = caseo (u [ p ]) (caseo (v [ p ]) (w [ p ])) [ id ,o t ]
Threeβ0 : ∀{Γ A}{u v w : Tm Γ A} → caseThree three0 u v w ≡ u
Threeβ1 : ∀{Γ A}{u v w : Tm Γ A} → caseThree three1 u v w ≡ v
Threeβ2 : ∀{Γ A}{u v w : Tm Γ A} → caseThree three2 u v w ≡ w
Threeβ0 {Γ}{a}{u}{v}{w} =
  caseThree three0 u v w
                          ≡⟨ refl ⟩
  caseo (u [ p ]) (caseo (v [ p ]) (w [ p ])) [ id ,o inl trivial ]
                          ≡⟨ +β₁ ⟩
  u [ p ] [ id ,o trivial ]
                          ≡⟨ [∘] ⁻¹ ⟩
  u [ p ⊚ (id ,o trivial) ]
                          ≡⟨ cong (u [_]) ▷β₁ ⟩
  u [ id ]
                          ≡⟨ [id] ⟩
  u
                          ∎
```

Addition modulo 3:

```
plus3 : Tm ⋄ (Three ⇒ Three ⇒ Three)
plus3 = lam (lam (caseThree v1
  {- v1=0 -} v0
  {- v1=1 -} (caseThree v0 three1 three2 three0)
  {- v1=2 -} (caseThree v0 three2 three0 three1)))

plus3test00 : plus3 $ three0 $ three0 ≡ three0
plus3test11 : plus3 $ three1 $ three1 ≡ three2
plus3test12 : plus3 $ three1 $ three2 ≡ three0
plus3test00 = exercisep
plus3test11 = exercisep
plus3test12 = exercisep
```

We define booleans. The usual eliminator (if-then-else) is defined using case.

```
Bool : Ty
Bool = Unit +o Unit
```

```
true : ∀{Γ} → Tm Γ Bool
true = inl trivial

false : ∀{Γ} → Tm Γ Bool
false = inr trivial

ite : ∀{Γ A} → Tm Γ Bool → Tm Γ A → Tm Γ A → Tm Γ A
ite b u v = caseo (u [ p ]) (v [ p ]) [ id ,o b ]

Boolβ₁ : ∀{Γ A}{u v : Tm Γ A} → ite true u v ≡ u
Boolβ₁ {Γ}{A}{u}{v} =
  ite true u v
    ≡⟨ refl ⟩
  caseo (u [ p ]) (v [ p ]) [ id ,o inl trivial ]
    ≡⟨ +β₁ ⟩
  u [ p ] [ id ,o trivial ]
    ≡⟨ [∘] ⁻¹ ⟩
  u [ p ⊚ (id ,o trivial) ]
    ≡⟨ cong {A = Sub _ _} (u [_]) ▷β₁ ⟩
  u [ id ]
    ≡⟨ [id] ⟩
  u
    ∎
Boolβ₂ : ∀{Γ A}{u v : Tm Γ A} → ite false u v ≡ v
Boolβ₂ {Γ}{A}{u}{v} =
  ite false u v
    ≡⟨ refl ⟩
  caseo (u [ p ]) (v [ p ]) [ id ,o inr trivial ]
    ≡⟨ +β₂ ⟩
  v [ p ] [ id ,o trivial ]
    ≡⟨ [∘] ⁻¹ ⟩
  v [ p ⊚ (id ,o trivial) ]
    ≡⟨ cong (v [_]) ▷β₁ ⟩
  v [ id ]
    ≡⟨ [id] ⟩
  v
    ∎
```

With Bool and binary products we can simulate homogeneous binary sums:

```
2*_ : Ty → Ty
2* A = Bool ×o A
inl' inr' : ∀{Γ A} → Tm Γ A → Tm Γ (2* A)
inl' t = ⟨ true , t ⟩
inr' t = ⟨ false , t ⟩
case' : ∀{Γ A C} → Tm Γ (2* A) → Tm (Γ ▷ A) C → Tm (Γ ▷ A) C → Tm Γ C
case' t u v = ite (fst t) (u [ id ,o snd t ]) (v [ id ,o snd t ])
2*β₁ : ∀{Γ A C}{t : Tm Γ A}{u v : Tm (Γ ▷ A) C} → case' (inl' t) u v ≡ u [ id ,o t ]
2*β₁ {Γ}{A}{C}{t}{u}{v} =
  case' (inl' t) u v
    ≡⟨ refl ⟩
  ite (fst ⟨ true , t ⟩) (u [ id ,o snd (inl' t) ]) (v [ id ,o snd (inl' t) ])
    ≡⟨ cong (λ x → ite x (u [ id ,o snd (inl' t) ]) (v [ id ,o snd (inl' t) ])) ×β₁ ⟩
  ite true (u [ id ,o snd (inl' t) ]) (v [ id ,o snd (inl' t) ])
    ≡⟨ Boolβ₁ ⟩
  u [ id ,o snd (inl' t) ]
    ≡⟨ refl ⟩
```

```
    u [ id ,o snd ⟨ true , t ⟩ ]
      ≡⟨ cong (λ x → u [ id ,o x ]) ×β₂ ⟩
    u [ id ,o t ]
      ∎
2*β₂ : ∀{Γ A C}{t : Tm Γ A}{u v : Tm (Γ ▷ A) C} → case' (inr' t) u v ≡ v [ id ,o t ]
2*β₂ = exercisep
```

The option type former (sometimes called maybe) adds an extra element to a type:

```
Opt : Ty → Ty
Opt A = A +o Unit
none : ∀{Γ A} → Tm Γ (Opt A)
none = inr trivial
some : ∀{Γ A} → Tm Γ A → Tm Γ (Opt A)
some t = inl t
```

A type with n elements:

```
Fino : ℕ → Ty
Fino zero = Unit
Fino (suc n) = Unit +o Fino n
```

Exercise 4.2. *For each* n*, define the elimination principle of* Fino n*.*

For a natural number n, we define n-ary products which are parameterised by n types.

```
Tys : ℕ → Set k
Tys zero     = Raise (Lift ⊤)
Tys (suc n)  = Ty × Tys n

Prod : {n : ℕ} → Tys n → Ty
Prod {zero} _          = Unit
Prod {suc n} (A , As) = A ×o Prod As
```

The constructor takes n terms and turns them into a term.

```
Tms : Con → {n : ℕ} → Tys n → Set l
Tms Γ {zero} _          = Raise (Lift ⊤)
Tms Γ {suc n} (A , As) = Tm Γ A × Tms Γ As

tpl : ∀{Γ n}{As : Tys n} → Tms Γ As → Tm Γ (Prod As)
tpl {Γ} {zero}  _       = trivial
tpl {Γ} {suc n} (t , ts) = ⟨ t , tpl ts ⟩
```

We have n different projections.

```
_!_ : ∀{n} → Tys n → Fin n → Ty
(A , As) ! zero  = A
(A , As) ! suc i = As ! i

prj : ∀{Γ n}{As : Tys n}(i : Fin n) → Tm Γ (Prod As) → Tm Γ (As ! i)
prj zero    t = fst t
prj (suc i) t = prj i (snd t)
```

Exercise 4.3. *State and prove the computation rules and the substitution laws.*

Exercise 4.4. *Define* n*-ary coproducts.*

4.2 Algebraic properties of types

Types form a commutative semiring with exponentials, up to isomorphism.

```
record _≅_ (A B : Ty) : Set l where
  field
    f : Tm (◇ ▷ A) B
    g : Tm (◇ ▷ B) A
    fg : f [ p ,o g ] ≡ q
    gf : g [ p ,o f ] ≡ q
open _≅_
infix 4 _≅_

+idr : ∀{A} → A +o Empty ≅ A
f (+idr {A}) = caseo q absurd
g (+idr {A}) = inl q
fg (+idr {A}) =
  caseo q absurd [ p ,o inl q ]
    ≡⟨ +β₁' ⟩
  q [ p ,o q ]
    ≡⟨ ▷β₂ ⟩
  q
    ∎
gf (+idr {A}) =
  inl q [ p ,o caseo q absurd ]
    ≡⟨ inl[] ⟩
  inl (q [ p ,o caseo q absurd ])
    ≡⟨ cong inl ▷β₂ ⟩
  inl (caseo q absurd)
    ≡⟨ +η ⁻¹ ⟩
  caseo (inl (caseo q absurd) [ p ,o inl q ]) (inl (caseo q absurd) [ p ,o inr q ])
    ≡⟨ cong (λ x → caseo x (inl (caseo q absurd) [ p ,o inr q ])) inl[] ⟩
  caseo (inl (caseo q absurd [ p ,o inl q ])) (inl (caseo q absurd) [ p ,o inr q ])
    ≡⟨ cong (λ x → caseo (inl (caseo q absurd [ p ,o inl q ])) x) inl[] ⟩
  caseo (inl (caseo q absurd [ p ,o inl q ])) (inl (caseo q absurd [ p ,o inr q ]))
    ≡⟨ cong (λ x → caseo (inl {A = A}{Empty} x) (inl (caseo q absurd [ p ,o inr q ]))) +β₁' ⟩
  caseo (inl (q [ p ,o q ])) (inl (caseo q absurd [ p ,o inr q ]))
    ≡⟨ cong (λ x → caseo (inl {A = A}{Empty} x) (inl (caseo q absurd [ p ,o inr q ]))) ▷β₂ ⟩
  caseo (inl q)                (inl (caseo q absurd [ p ,o inr q ]))
    ≡⟨ cong (λ x → caseo (inl q) x) Emptyη ⟩
  caseo (inl q) absurd
    ≡⟨ cong (λ x → caseo (inl q) x) (Emptyη ⁻¹) ⟩
  caseo (inl q) (inr q)
    ≡⟨ cong (λ x → caseo (inl q) x) (▷β₂ ⁻¹) ⟩
  caseo (inl q) (q [ p ,o inr q ])
    ≡⟨ cong (λ x → caseo x (q [ p ,o inr q ])) (▷β₂ ⁻¹) ⟩
  caseo (q [ p ,o inl q ]) (q [ p ,o inr q ])
    ≡⟨ +η ⟩
  q
    ∎
{-
 +comm : ∀{A B} → A +o B ≅ B +o A
 f (+comm {A} {B}) = caseo (inr q) (inl q)
 g (+comm {A} {B}) = caseo (inr q) (inl q)
 fg (+comm {A} {B}) = {!!}
 gf (+comm {A} {B}) = {!!}
-}
```

Exercise 4.5 (compulsory). *Show that for any two types* A, B, *there is an isomorphism between* A +o B *and* B +o A.

The following exercise is a type theoretic version of the algebraic equation $(a+b)^2 = a^2 + 2*a*b + b^2$.

Exercise 4.6 (compulsory). *Show that for any two types* A, B, *there is an isomorphism between* (Unit +o Unit) ⇒ (A +o B) *and* ((Unit +o Unit) ⇒ A +o (Unit +o Unit) ×o A ×o B) +o (Unit +o Unit) ⇒ B. *At least define the first two components (*f *and* g*).*

4.3 Standard model

The new components:

```
; _×o_     = _×_
; ⟨_,_⟩    = λ a b γ* → (a γ* , b γ*)
; fst      = λ t γ* → π₁ (t γ*)
; snd      = λ t γ* → π₂ (t γ*)
; ×β₁      = λ {Γ}{A}{B}{u}{v} → refl {A = Γ → A}
; ×β₂      = λ {Γ}{A}{B}{u}{v} → refl {A = Γ → B}
; ×η       = λ {Γ}{A}{B}{t} → refl {A = Γ → A × B}
; ,[]      = λ {Γ}{A}{B}{u}{v}{Δ}{γ} → refl {A = Δ → A × B}

; Unit     = Lift ⊤
; trivial  = λ γ* → mk triv
; Unitη    = λ {Γ}{u} → refl {A = Γ → Lift ⊤}

; _+o_     = _⊎_
; inl      = λ a γ* → ι₁ (a γ*)
; inr      = λ b γ* → ι₂ (b γ*)
; caseo    = λ u v γw* → case (π₂ γw*) (λ a* → u (π₁ γw* , a*)) λ b* → v (π₁ γw* , b*)
; +β₁      = refl
; +β₂      = refl
; +η       = λ {Γ}{A}{B}{C}{t} → funext λ γw → un (ind⊎
              (λ x → Lift (case x (λ a* → t (π₁ γw , ι₁ a*)) (λ b* → t (π₁ γw , ι₂ b*)) ≡ t (π₁ γw , x)))
              (λ a → mk (refl {A = C}))
              (λ b → mk (refl {A = C}))
              (π₂ γw))
; inl[]    = refl
; inr[]    = refl
; case[]   = refl

; Empty    = Lift ⊥
; absurd   = λ γb → exfalso (un (π₂ γb))
; Emptyη   = funext λ γa → exfalsop (un (π₂ γa))
```

Some equalities which hold in the standard model are not present in the syntax, for example:

```
e1 : Con ≡ Ty
e2 : ∀{Γ Δ} → Sub Δ Γ ≡ Tm Δ Γ
e3 : ∀{Γ A Δ} → _≡_ {A = Sub Δ (Γ ▷ A) → Sub Δ Γ} (p ∘_) fst
e4 : ∀{Γ A Δ} → _≡_ {A = Sub Δ (Γ ▷ A) → Tm Δ A} (q [_]) snd
e5 : ∀{A} → Tm ◇ (Unit ×o A) ≡ Sub ◇ (◇ ▷ A)
```

4.4 Normalisation

TODO: what are normal forms?

Chapter 5

Inductive types

Learning goals of this chapter

Inductive types by examples: natural numbers, lists, binary trees. Type formers, constructors, iterator, recursor, defining functions by iteration. Definability, examples of definable and non definable functions.

TODO: definablility. Equality is coarse: recNat zero (suc q) q ≠ q. We have several representations of the same function.

TODO: iterator is like a for loop where the body of the loop cannot refer to the loop variable. Recursor is like for loop. Fixpoint combinator is like while loop.

TODO: example: syntax as an inductive type.

TODO: redo nullary and binary products, sums as inductive types. See an exercise below.

TODO: add an infinitary example.

TODO: all functions defined by the iterator are terminating.

TODO: add mutual inductive types.

An inductive type is specified by its constructors. Its eliminator and computation rules are determined by the constructors: the eliminator says that for any type C (called motive) and elements of that type which have the shape of the constructors (these are called methods), there is a function (called iterator, recursor, fold, catamorphism) from the inductive type to C. The computation rules say that if we apply the iterator to a constructor, we obtain the corresponding element.

We have already seen some inductive types: Bool had two constructors true and false, and its iterator was called if-then-else, it said that for any type C and two elements of C, there is a function from Bool to C. For any two types A and B, A +o B was an inductive type with constructors inl and inr. Its iterator was called caseo, it said that for any type C and elements A → C and B → C, there is a function from A +o B to C.

Natural numbers are also an inductive type: the constructors are zero and suc, the latter can be written in several different equivalent ways:

suc : Tm Γ Nat → Tm Γ Nat
suc : Tm Γ (Nat ⇒ Nat)
suc : Tm (Γ ▷ Nat) Nat

When specifying the arguments of the iterator, we use the last version which does not refer to metatheoretic or object theoretic function space. So the iterator says that given a type C, a term in Tm Γ C and a term in Tm (Γ ▷ C) C, we get a function from Tm Γ Nat to Tm Γ C.

A language has natural numbers if it extends the substitution calculus with the following operators and equations:

```
Nat   : Ty
zeroo : ∀{Γ} → Tm Γ Nat
suco  : ∀{Γ} → Tm Γ Nat → Tm Γ Nat
```

$$
\begin{array}{ll}
\mathsf{iteNat} & : \forall\{\Gamma\ A\} \to \mathsf{Tm}\ \Gamma\ A \to \mathsf{Tm}\ (\Gamma \triangleright A)\ A \to \mathsf{Tm}\ \Gamma\ \mathsf{Nat} \to \mathsf{Tm}\ \Gamma\ A \\
\mathsf{Nat}\beta_1 & : \forall\{\Gamma\ A\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A)\ A\} \to \mathsf{iteNat}\ u\ v\ \mathsf{zeroo} \equiv u \\
\mathsf{Nat}\beta_2 & : \forall\{\Gamma\ A\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A)\ A\}\{t : \mathsf{Tm}\ \Gamma\ \mathsf{Nat}\} \to \\
& \quad \mathsf{iteNat}\ u\ v\ (\mathsf{suco}\ t) \equiv v\ [\ \mathsf{id}\ ,\mathsf{o}\ \mathsf{iteNat}\ u\ v\ t\ ] \\
\mathsf{zero}[] & : \forall\{\Gamma\ \Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \mathsf{zeroo}\ [\ \gamma\ ] \equiv \mathsf{zeroo} \\
\mathsf{suc}[] & : \forall\{\Gamma\}\{t : \mathsf{Tm}\ \Gamma\ \mathsf{Nat}\}\{\Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to (\mathsf{suco}\ t)\ [\ \gamma\ ] \equiv \mathsf{suco}\ (t\ [\ \gamma\ ]) \\
\mathsf{iteNat}[] & : \forall\{\Gamma\ A\}\{u : \mathsf{Tm}\ \Gamma\ A\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A)\ A\}\{t : \mathsf{Tm}\ \Gamma\ \mathsf{Nat}\}\{\Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \\
& \quad \mathsf{iteNat}\ u\ v\ t\ [\ \gamma\ ] \equiv \mathsf{iteNat}\ (u\ [\ \gamma\ ])\ (v\ [\ \gamma \circledcirc \mathsf{p}\ ,\mathsf{o}\ \mathsf{q}\ ])\ (t\ [\ \gamma\ ])
\end{array}
$$

iteNat binds a variable of type A in its second argument. The computation rules $\mathsf{Nat}\beta_1$ and $\mathsf{Nat}\beta_2$ express that iteNat u v t works as follows: it replaces all sucos in t by what is specified by v, and replaces zeroo by u. For example, iteNat u v (suco (suco (suco zero))) is equal to v [id ,o v [id ,o v [id ,o u]]] (which can be thought about as v (v (v u)) but we use substitution to express the applications).

Gödel's System T is the name of the language which has function space ⇒ and natural numbers as above.

The iterator is sometimes called recursor or primitive recursor. We distinguish the two. The recursor for natural numbers does not only receive the result of the recursive call, but also the number.

recNat : ∀{Γ A} → Tm Γ A → Tm (Γ ▷ Nat ▷ A) A → Tm Γ Nat → Tm Γ A
$\mathsf{recNat}\beta_1$: recNat u v zeroo ≡ u
$\mathsf{recNat}\beta_2$: recNat u v (suco t) ≡ v [id ,o t ,o recNat u v t]

For example, with the iterator, it is not possible to define a function pred : Tm (◇ ▷ Nat) Nat s.t. pred [id ,o suco t] ≡ t for any t (note that t might be a variable e.g. q) [15].

Exercise 5.1 (recommended)**.** *Define the recursor for natural numbers with the help of the iterator. It does not need to (and will not) satisfy* $\mathsf{recNat}\beta_2$*. TODO: elaborate.*

In the case of Bool (or other non-recursive inductive types), there is no difference between the iterator and the recursor.

Our previous definition of Nat (in Section 2.3) was not an inductive type because it did not have an iterator or recursor, only two special cases of the iterator: addition and isZero. Both of these can be defined using the iterator, see below.

Another inductive type is that of lists. For any type A, we have a type List A. It has two constructors nil : Tm Γ (List A) and cons which again can be expressed in three different ways. We will use the first one when specifying the constructor and the third one when specifying the arguments of the iterator.

cons : Tm Γ A → Tm Γ (List A) → Tm Γ (List A)
cons : Tm Γ (A ⇒ List A ⇒ List A)
cons : Tm (Γ ▷ A ▷ List A) (List A)

A language has lists if it extends the substitution calculus with the following operators and equations:

$$
\begin{array}{ll}
\mathsf{List} & : \mathsf{Ty} \to \mathsf{Ty} \\
\mathsf{nil} & : \forall\{\Gamma\ A\} \to \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A) \\
\mathsf{cons} & : \forall\{\Gamma\ A\} \to \mathsf{Tm}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A) \to \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A) \\
\mathsf{iteList} & : \forall\{\Gamma\ A\ B\} \to \mathsf{Tm}\ \Gamma\ B \to \mathsf{Tm}\ (\Gamma \triangleright A \triangleright B)\ B \to \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A) \to \mathsf{Tm}\ \Gamma\ B \\
\mathsf{List}\beta_1 & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ B\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A \triangleright B)\ B\} \to \mathsf{iteList}\ u\ v\ \mathsf{nil} \equiv u \\
\mathsf{List}\beta_2 & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ B\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A \triangleright B)\ B\}\{t_1 : \mathsf{Tm}\ \Gamma\ A\}\{t : \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A)\} \to \\
& \quad \mathsf{iteList}\ u\ v\ (\mathsf{cons}\ t_1\ t) \equiv (v\ [\ \mathsf{id}\ ,\mathsf{o}\ t_1\ ,\mathsf{o}\ \mathsf{iteList}\ u\ v\ t\ ]) \\
\mathsf{nil}[] & : \forall\{\Gamma\ A\ \Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \mathsf{nil}\ \{\Gamma\}\{A\}\ [\ \gamma\ ] \equiv \mathsf{nil}\ \{\Delta\}\{A\} \\
\mathsf{cons}[] & : \forall\{\Gamma\ A\}\{t_1 : \mathsf{Tm}\ \Gamma\ A\}\{t : \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A)\}\{\Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \\
& \quad (\mathsf{cons}\ t_1\ t)\ [\ \gamma\ ] \equiv \mathsf{cons}\ (t_1\ [\ \gamma\ ])\ (t\ [\ \gamma\ ]) \\
\mathsf{iteList}[] & : \forall\{\Gamma\ A\ B\}\{u : \mathsf{Tm}\ \Gamma\ B\}\{v : \mathsf{Tm}\ (\Gamma \triangleright A \triangleright B)\ B\}\{t : \mathsf{Tm}\ \Gamma\ (\mathsf{List}\ A)\}\{\Delta\}\{\gamma : \mathsf{Sub}\ \Delta\ \Gamma\} \to \\
& \quad \mathsf{iteList}\ u\ v\ t\ [\ \gamma\ ] \equiv \mathsf{iteList}\ (u\ [\ \gamma\ ])\ (v\ [\ (\gamma \circledcirc \mathsf{p}\ ,\mathsf{o}\ \mathsf{q}) \circledcirc \mathsf{p}\ ,\mathsf{o}\ \mathsf{q}\ ])\ (t\ [\ \gamma\ ])
\end{array}
$$

iteList binds a variable of type A and another variable of type B in its second explicit argument. In iteList u v t, u expresses what to do when t = nil t', v expresses what to do when t = cons t' t''.

Another inductive type is that of binary trees with information at the leaves. For any type A, Ty A is a type, the constructors are leaf and node. A language has these binary trees if it has the following operators and equations:

```
Tree      : Ty → Ty
leaf      : ∀{Γ A} → Tm Γ A → Tm Γ (Tree A)
node      : ∀{Γ A} → Tm Γ (Tree A) → Tm Γ (Tree A) → Tm Γ (Tree A)
iteTree   : ∀{Γ A B} → Tm (Γ ▷ A) B → Tm (Γ ▷ B ▷ B) B → Tm Γ (Tree A) → Tm Γ B
Treeβ1    : ∀{Γ A B}{l : Tm (Γ ▷ A) B}{n : Tm (Γ ▷ B ▷ B) B}{a : Tm Γ A} →
            iteTree l n (leaf a) ≡ l [ id ,o a ]
Treeβ2    : ∀{Γ A B}{l : Tm (Γ ▷ A) B}{n : Tm (Γ ▷ B ▷ B) B}{ll rr : Tm Γ (Tree A)} →
            iteTree l n (node ll rr) ≡ n [ id ,o iteTree l n ll ,o iteTree l n rr ]
leaf[]    : ∀{Γ A}{a : Tm Γ A}{Δ}{γ : Sub Δ Γ} → (leaf a) [ γ ] ≡ leaf (a [ γ ])
node[]    : ∀{Γ A}{ll rr : Tm Γ (Tree A)}{Δ}{γ : Sub Δ Γ} →
            (node ll rr) [ γ ] ≡ node (ll [ γ ]) (rr [ γ ])
iteTree[] : ∀{Γ A B}{l : Tm (Γ ▷ A) B}{n : Tm (Γ ▷ B ▷ B) B}{t : Tm Γ (Tree A)}
            {Δ}{γ : Sub Δ Γ} →
            iteTree l n t [ γ ] ≡
            iteTree (l [ (γ ⊚ p) ,o q ]) (n [ (γ ⊚ p ⊚ p) ,o (q [ p ]) ,o q ]) (t [ γ ])
```

iteTree binds a variable of type A in its first explicit argument and two variables of type B in its second explicit argument.

With the iterator, we can define the following functions on natural numbers. Addition with variable names is written plus = λ x y . iteNat y (z . suc z) x which means that we have a function with two inputs x and y, we do recursion on the first input (x), and in the input, we replace zeros by y, and sucs by suc. The same information is expressed in languages with pattern matching as follows.

```
plus zero     y = y
plus (suc x') y = suc (plus x' y)
```

The first line corresponds to the first argument of iteNat (i.e. y), the second line corresponds to the second argument in which z is bound (z . suc z) where z refers to the recursive call which we write (plus x' y) using pattern matching notation.

In our formal syntax, we write this as follows.

```
plus : Tm ◇ (Nat ⇒ Nat ⇒ Nat)
plus = lam (lam (iteNat v0 (suco v0) v1))
```

The isZero operation can be defined as follows.

```
isZero : Tm ◇ (Nat ⇒ Bool)
isZero = lam (iteNat true false v0)
```

The zero case is simply true, the successor case is false regardless of the result of the recursive call.

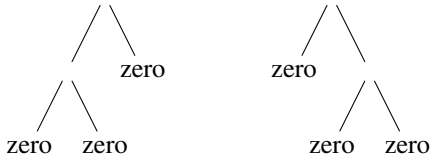
Examples for lists:

```
isnil : {A : Ty} → Tm ◇ (List A ⇒ Bool)
isnil = lam (iteList true false v0)

concat : {A : Ty} → Tm ◇ (List A ⇒ List A ⇒ List A)
concat = lam (lam (iteList v0 (cons v1 v0) v1))
```

isnil and concat are polymorphic functions in the sense that they work for any type A but the polymorphism happens in our metalanguage and not in our object language.

The following two trees in Tm ◇ (Tree Nat)



are defined as

```
treeEx1 = node (node (leaf zeroo) (leaf zeroo)) (leaf zeroo)
treeEx2 = node (leaf zeroo) (node (leaf zeroo) (leaf zeroo))
```

The following function adds all the numbers in the leaves of a tree.

```
sum : Tm ⋄ (Tree Nat ⇒ Nat)
sum = lam (iteTree v0 (plus [ p ⊙ p ⊙ p ] $ v1 $ v0) v0)
```

With variable names, we would write this as sum = λ x . iteTree (z . z) (z_1 z_2 . plus $ z_1 $ z_2) x. In the formal syntax, we had to weaken plus by p ∘ p ∘ p because it was defined in the empty context above and now we used it in a context with three free variables (one bound by lam, the other two by iteTree).

The following function turns a binary tree into a list:

```
flatten : ∀{A} → Tm ⋄ (Tree A ⇒ List A)
flatten = lam (iteTree (cons v0 nil) (concat [ p ⊙ p ⊙ p ] $ v1 $ v0) v0)
```

Exercise 5.2 (compulsory)**.** *List the rules for inductively defined trees with ternary branching, a boolean at each node and a natural number at each leaf.*

Exercise 5.3 (compulsory)**.** *List the rules for inductively defined trees with two kinds of nullary, one kind of binary and three kinds of ternary branching. There is no extra information at leaves or nodes.*

Exercise 5.4 (compulsory)**.** *List the rules for inductively defined trees no information at the leaves and infinity (*Nat*-ary) branching.*

Exercise 5.5 (compulsory)**.** *For types* A*,* B*, list the rules for the inductive type with only leaves (and no nodes) that contain an* A *and a* B*. Derive all its rules from binary products of the previous section. Which rule of binary products cannot be derived from this inductive type?*

Exercise 5.6 (compulsory)**.** *For types* A*,* B*, list the rules for the inductive type with two kinds of leaves and no nodes. One kind of leaf contains an* A*, the other kind of leaf contains a* B*. Derive all the rules from binary sums of the previous section. Which rule of binary sums cannot be derived from this inductive type?*

5.1 Standard model

We define lists and trees using Agda's inductive types.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
iteList : {A B : Set} → B → (A → B → B) → List A → B
iteList b f [] = b
iteList b f (a :: as) = f a (iteList b f as)

data Tree (A : Set) : Set where
  leaf : A → Tree A
  node : Tree A → Tree A → Tree A
```

```
iteTree : {A B : Set} → (A → B) → (B → B → B) → Tree A → B
iteTree f g (leaf a) = f a
iteTree f g (node t t') = g (iteTree f g t) (iteTree f g t')
```

The new components in the standard model:

```
; Nat      = ℕ
; zeroo    = λ _ → zero
; suco     = λ t γ* → suc (t γ*)
; iteNat   = λ u v t γ* → iteℕ (u γ*) (λ x → v (γ* , x)) (t γ*)
; Natβ₁    = refl
; Natβ₂    = refl
; zero[]   = refl
; suc[]    = refl
; iteNat[] = refl

; List     = List
; nil      = λ _ → []
; cons     = λ u v γ* → u γ* :: v γ*
; iteList  = λ u v t γ* → iteList (u γ*) (λ x y → v ((γ* , x) , y)) (t γ*)
; Listβ₁   = refl
; Listβ₂   = refl
; nil[]    = refl
; cons[]   = refl
; iteList[] = refl

; Tree     = Tree
; leaf     = λ t γ* → leaf (t γ*)
; node     = λ u v γ* → node (u γ*) (v γ*)
; iteTree  = λ {Γ}{A}{B} u v t γ* → iteTree (λ x → u (γ* , x)) (λ x y → v ((γ* , x) , y)) (t γ*)
; Treeβ₁   = refl
; Treeβ₂   = refl
; leaf[]   = refl
; node[]   = refl
; iteTree[] = refl
```

As we explained in Subsection 1.5.1, from the standard model we obtain that $\mathsf{true} \neq \mathsf{false}$ and $\mathsf{zero} \neq \mathsf{suc}\ \mathsf{t}$ in the syntax for any t.

```
true≠false : ¬ (I.true {I.◇} ≡ I.false)
true≠false = λ e → tt≠ff (cong (λ t → St.⟦ t ⟧t _) e)

zeroo≠suco : ∀{t} → ¬ (I.zeroo {I.◇} ≡ I.suco t)
zeroo≠suco = λ e → zero≠suc (cong (λ t → St.⟦ t ⟧t _) e)

postulate
  t≠suct : ∀{t} → ¬ (t ≡ I.suco {I.◇} t)
```

TODO: show that these hold in any context.

5.2 Definable functions on natural numbers

The Ackermann function is not definable using first-order primitive recursion (that is, in Def + natural numbers as a $\mathsf{Tm}\ (\diamond \triangleright \mathsf{Nat})\ \mathsf{Nat}$). But it is definable in our language because we have higher-order functions [11, Section 9.3]. Here is its definition in Agda:

```
ack : (x y : ℕ) -> ℕ
ack zero n = 1 + n
```

```
ack (suc m) zero = ack m 1
ack (suc m) (suc n) = ack m (ack (suc m) n)
```

Natural numbers are included in terms.

```
⌜_⌝ : ℕ → Tm ⋄ Nat
⌜ zero ⌝ = zeroo
⌜ suc n ⌝ = suco ⌜ n ⌝
```

A function on natural numbers is definable if there is a term that, when applied to a natural number, gives the same result as the function.

```
Definable : (ℕ → ℕ) → Set
Definable f = Σsp (Tm ⋄ (Nat ⇒ Nat)) λ t → (n : ℕ) → t $ ⌜ n ⌝ ≡ ⌜ f n ⌝
```

For example, the times two function is definable.

```
definable2* : Definable (iteℕ zero (λ x → suc (suc x)))
definable2* = (lam (iteNat zeroo (suco (suco q)) q)) , λ n →
  lam (iteNat zeroo (suco (suco q)) q) $ ⌜ n ⌝
      ≡⟨ ⇒β ⟩
  iteNat zeroo (suco (suco q)) q [ id ,o ⌜ n ⌝ ]
      ≡⟨ iteNat[] ⟩
  iteNat (zeroo [ id ,o ⌜ n ⌝ ]) (suco (suco q) [ (id ,o ⌜ n ⌝) ◎ p ,o q ]) (q [ id ,o ⌜ n ⌝ ])
      ≡⟨ cong (λ x → iteNat x (suco (suco q) [ (id ,o ⌜ n ⌝) ◎ p ,o q ]) (q [ id ,o ⌜ n ⌝ ])) zero[] ⟩
  iteNat zeroo (suco (suco q) [ (id ,o ⌜ n ⌝) ◎ p ,o q ]) (q [ id ,o ⌜ n ⌝ ])
      ≡⟨ cong (λ x → iteNat zeroo x (q [ id ,o ⌜ n ⌝ ])) suc[] ⟩
  iteNat zeroo (suco (suco q [ (id ,o ⌜ n ⌝) ◎ p ,o q ])) (q [ id ,o ⌜ n ⌝ ])
      ≡⟨ cong (λ x → iteNat zeroo (suco x) (q [ id ,o ⌜ n ⌝ ])) suc[] ⟩
  iteNat zeroo (suco (suco (q [ (id ,o ⌜ n ⌝) ◎ p ,o q ]))) (q [ id ,o ⌜ n ⌝ ])
      ≡⟨ cong (λ x → iteNat zeroo (suco (suco x)) (q [ id ,o ⌜ n ⌝ ])) ▷β₂ ⟩
  iteNat zeroo (suco (suco q)) (q [ id ,o ⌜ n ⌝ ])
      ≡⟨ cong (iteNat zeroo (suco (suco q))) ▷β₂ ⟩
  iteNat zeroo (suco (suco q)) ⌜ n ⌝
      ≡⟨ ⌜rec⌝ n ⁻¹ ⟩
    ⌜ iteℕ zero (λ x → suc (suc x)) n ⌝
      ∎
  where
    ⌜rec⌝ : (n : ℕ) → ⌜ iteℕ zero (λ x → suc (suc x)) n ⌝ ≡ iteNat zeroo (suco (suco q)) ⌜ n ⌝
    ⌜rec⌝ zero = Natβ₁ ⁻¹
    ⌜rec⌝ (suc n) =
      suco (suco ⌜ iteℕ zero (λ x → suc (suc x)) n ⌝)
        ≡⟨ cong {A = Tm _ _}(λ x → suco (suco x)) (⌜rec⌝ n) ⟩
      suco (suco (iteNat zeroo (suco (suco q)) ⌜ n ⌝))
        ≡⟨ cong {A = Tm _ _}(λ x → suco (suco x)) (▷β₂ ⁻¹) ⟩
      suco (suco (q [ id ,o iteNat zeroo (suco (suco q)) ⌜ n ⌝ ]))
        ≡⟨ cong suco (suc[] ⁻¹) ⟩
      suco (suco q [ id ,o iteNat zeroo (suco (suco q)) ⌜ n ⌝ ])
        ≡⟨ suc[] ⁻¹ ⟩
      suco (suco q) [ id ,o iteNat zeroo (suco (suco q)) ⌜ n ⌝ ]
        ≡⟨ Natβ₂ ⁻¹ ⟩
      iteNat zeroo (suco (suco q)) (suco ⌜ n ⌝)
        ∎
```

Here is definability for two-argument functions. TODO: using an isomorphism $\mathbb{N} \cong \mathbb{N} \times \mathbb{N}$ and its internal version $\mathrm{Nat} \cong \mathrm{Nat} \times \mathrm{Nat}$, we can show a $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$ function is definable iff. its $\mathbb{N} \Rightarrow \mathbb{N}$ encoded variant is definable.

```
Definable' : (ℕ → ℕ → ℕ) → Set
Definable' f = Σ (Tm ◇ (Nat ⇒ Nat ⇒ Nat)) λ t → (m n : ℕ) → Lift (t $ ⌜ m ⌝ $ ⌜ n ⌝ ≡ ⌜ f m n ⌝)
```

It is (relatively) easy to encode terms as natural numbers, for now we only assume a code function.

```
postulate
  code : Tm ◇ (Nat ⇒ Nat) → ℕ
```

The evaluator of the language can be defined as interpretation into the standard model, for simplicity we only assume it here. Here we use a special case which takes a code for a Nat-endofunction and a natural number. TODO: use normalisation and its completeness together with decode-encode iso to define the evaluator and prove its property. Is canonicity enough?

```
eval : ℕ → ℕ → ℕ
evalProp : (t : Tm ◇ (Nat ⇒ Nat))(n : ℕ) → t $ ⌜ n ⌝ ≡ ⌜ eval (code t) n ⌝
```

The evaluator is not definable. The proof uses Cantor's diagonal argument.

```
evalNotDefinable : Definable' eval → Lift ⊥
evalNotDefinable (t , f) = mk (t≠suct (
  ⌜ eval (code u) (code u) ⌝
    ≡⟨ evalProp u (code u) ⁻¹ ⟩
  u $ ⌜ code u ⌝
    ≡⟨ ⇒β ⟩
  suco (t [ p ] $ q $ q) [ id ,o ⌜ code u ⌝ ]
    ≡⟨ suc[] ⟩
  suco ((t [ p ] $ q $ q) [ id ,o ⌜ code u ⌝ ])
    ≡⟨ cong suco $[] ⟩
  suco (((t [ p ] $ q) [ id ,o ⌜ code u ⌝ ] $ q [ id ,o ⌜ code u ⌝ ]))
    ≡⟨ cong (λ x → suco (x $ q [ id ,o ⌜ code u ⌝ ])) $[] ⟩
  suco ((t [ p ] [ id ,o ⌜ code u ⌝ ] $ q [ id ,o ⌜ code u ⌝ ] $ q [ id ,o ⌜ code u ⌝ ]))
    ≡⟨ cong (λ x → suco ((t [ p ] [ id ,o ⌜ code u ⌝ ] $ x $ x))) ▷β₂ ⟩
  suco ((t [ p ] [ id ,o ⌜ code u ⌝ ] $ ⌜ code u ⌝ $ ⌜ code u ⌝))
    ≡⟨ cong (λ x → suco ((x $ ⌜ code u ⌝ $ ⌜ code u ⌝))) ([∘] ⁻¹) ⟩
  suco ((t [ p ◎ (id ,o ⌜ code u ⌝) ] $ ⌜ code u ⌝ $ ⌜ code u ⌝))
    ≡⟨ cong (λ x → suco ((t [ x ] $ ⌜ code u ⌝ $ ⌜ code u ⌝))) ▷β₁ ⟩
  suco ((t [ id ] $ ⌜ code u ⌝ $ ⌜ code u ⌝))
    ≡⟨ cong (λ x → suco ((x $ ⌜ code u ⌝ $ ⌜ code u ⌝))) [id] ⟩
  suco ((t $ ⌜ code u ⌝ $ ⌜ code u ⌝))
    ≡⟨ cong suco (un (f (code u) (code u))) ⟩
  suco ⌜ eval (code u) (code u) ⌝
    ∎))
  where
    u : Tm ◇ (Nat ⇒ Nat)
    u = lam (suco (t [ p ] $ q $ q))
```

TODO: define $\mathbb{N} \times \mathbb{N} \cong \mathbb{N}$ and use the ordinary definition of definability.

Summary of non-definability: if there is a surjection from ℕ to Tm ◇ (Nat ⇒ Nat) and the latter is isomorphic to $\mathbb{N} \to \mathbb{N}$, then there is a surjection from ℕ to $\mathbb{N} \to \mathbb{N}$.

Chapter 6

Coinductive types

Learning goals of this chapter

Coinductive types by example: streams, machines.

TODO: products as coinductive types.

TODO: functions as coinductive types, streams as functions. [2]

Coinductive types are dual to inductive types. Inductive types are specified by their constructors, coinductive types are specified by their destructors. Their constructor is determined by the destructors: if we have a type **C** (the seed) together with terms which have the shape of the destructors (but using **C** instead of the coinductive type), we obtain a function (called corecursor, generator, anamorphism) from the seed to the coinductive type. The seed is called like that because the generator makes the coinductive type "grow" out of it. While elements of inductive type are trees of finite depth, elements of coinductive types are potentially infinitely deep trees. Functions out of inductive types are all terminating, functions producing elements of coinductive types are *productive*.

The usual example is infinite lists or streams. There are two ways to destruct a stream: look at its first element (head), or forget its first element (tail). To generate a stream, we need to fix a seed type **C** which represents the internal state of the stream; we have to provide an **A** from any **C**; we have to be able to step the internal state when forgetting the first element; and finally, we need a state to start with. The computation rules say what happens if a destructor is applied to the constructor (generator): we use the corresponding method with the actual internal state, and corecursively call the generator on the new state, if needed.

```
Stream      : Ty → Ty
head        : ∀{Γ A} → Tm Γ (Stream A) → Tm Γ A
tail        : ∀{Γ A} → Tm Γ (Stream A) → Tm Γ (Stream A)
genStream   : ∀{Γ A C} → Tm (Γ ▷ C) A → Tm (Γ ▷ C) C → Tm Γ C → Tm Γ (Stream A)
Streamβ₁    : ∀{Γ A C}{u : Tm (Γ ▷ C) A}{v : Tm (Γ ▷ C) C}{t : Tm Γ C} →
              head (genStream u v t) ≡ u [ id ,o t ]
Streamβ₂    : ∀{Γ A C}{u : Tm (Γ ▷ C) A}{v : Tm (Γ ▷ C) C}{t : Tm Γ C} →
              tail (genStream u v t) ≡ genStream u v (v [ id ,o t ])
head[]      : ∀{Γ A}{as : Tm Γ (Stream A)}{Δ}{γ : Sub Δ Γ} →
              head as [ γ ] ≡ head (as [ γ ])
tail[]      : ∀{Γ A}{as : Tm Γ (Stream A)}{Δ}{γ : Sub Δ Γ} → tail as [ γ ] ≡ tail (as [ γ ])
genStream[] : ∀{Γ A C}{u : Tm (Γ ▷ C) A}{v : Tm (Γ ▷ C) C}{t : Tm Γ C}{Δ}
              {γ : Sub Δ Γ} →
              genStream u v t [ γ ] ≡ genStream (u [ γ ⊙ p ,o q ]) (v [ γ ⊙ p ,o q ]) (t [ γ ])
```

genStream binds a variable of type **C** in both its first end second explicit arguments.

State machines, virtual machines, Turing machines, servers, operating systems are all coinductive types. An example simple machine has three operations: (i) we can put in a natural number, (ii) we can press a button on it, (iii) we can ask it to output a natural number.

```
Machine       : Ty
put           : ∀{Γ} → Tm Γ Machine → Tm Γ Nat → Tm Γ Machine
set           : ∀{Γ} → Tm Γ Machine → Tm Γ Machine
get           : ∀{Γ} → Tm Γ Machine → Tm Γ Nat
genMachine    : ∀{Γ C} → Tm (Γ ▷ C ▷ Nat) C → Tm (Γ ▷ C) C → Tm (Γ ▷ C) Nat →
                Tm Γ C → Tm Γ Machine
Machineβ1     : ∀{Γ C}{u : Tm (Γ ▷ C ▷ Nat) C}{v : Tm (Γ ▷ C) C}{w : Tm (Γ ▷ C) Nat}
                {t : Tm Γ C}{t' : Tm Γ Nat} →
                put (genMachine u v w t) t' ≡ genMachine u v w (u [ id ,o t ,o t' ])
Machineβ2     : ∀{Γ C}{u : Tm (Γ ▷ C ▷ Nat) C}{v : Tm (Γ ▷ C) C}{w : Tm (Γ ▷ C) Nat}
                {t : Tm Γ C} →
                set (genMachine u v w t) ≡ genMachine u v w (v [ id ,o t ])
Machineβ3     : ∀{Γ C}{u : Tm (Γ ▷ C ▷ Nat) C}{v : Tm (Γ ▷ C) C}{w : Tm (Γ ▷ C) Nat}
                {t : Tm Γ C} →
                get (genMachine u v w t) ≡ w [ id ,o t ]
put[]         : ∀{Γ}{t : Tm Γ Machine}{u : Tm Γ Nat}{Δ}{γ : Sub Δ Γ} →
                put t u [ γ ] ≡ put (t [ γ ]) (u [ γ ])
set[]         : ∀{Γ}{t : Tm Γ Machine}{Δ}{γ : Sub Δ Γ} → set t [ γ ] ≡ set (t [ γ ])
get[]         : ∀{Γ}{t : Tm Γ Machine}{Δ}{γ : Sub Δ Γ} → get t [ γ ] ≡ get (t [ γ ])
genMachine[] : ∀{Γ C}{u : Tm (Γ ▷ C ▷ Nat) C}{v : Tm (Γ ▷ C) C}{w : Tm (Γ ▷ C) Nat}
                {t : Tm Γ C}{Δ}{γ : Sub Δ Γ} →
                genMachine u v w t [ γ ] ≡
                genMachine (u [ (γ ⊚ p ,o q) ⊚ p ,o q ]) (v [ γ ⊚ p ,o q ]) (w [ γ ⊚ p ,o q ]) (t [ γ ])
```

genMachine binds a variable of type C and a variable of type Nat in its first explicit argument, and binds a variable of type C in its second and third explicit arguments.

The stream which contains the number 0,1,2,...:

```
numbers : Tm ⋄ (Stream Nat)
numbers = genStream v0 (suco v0) zeroo

testNumbers0 : head numbers ≡ zeroo
testNumbers0 =
  head numbers
    ≡⟨ refl {A = Tm _ _} ⟩
  head (genStream q (suco q) zeroo)
    ≡⟨ Streamβ1 ⟩
  q [ id ,o zeroo ]
    ≡⟨ ▷β2 ⟩
  zeroo
    ∎

testNumbers2 : head (tail (tail numbers)) ≡ suco (suco zeroo)
testNumbers2 = exercisep
```

A machine which adds the numbers that we input unless we press the button, then it adds one when inputting a number. If we toggle the button again, it will start adding the numbers again. The number it outputs is the current sum.

```
aMachine : Tm ⋄ Machine
aMachine = genMachine ⟨ fst v1 , plus [ ε ] $ snd v1 $ v0 ⟩
  ⟨ iteBool false true (fst v0) , snd v0 ⟩ (snd v0) ⟨ true , zeroo ⟩
```

Exercise 6.1 (compulsory). *For types* A, B, *define the coinductive type of machines from which we can get out an* A *and a* B. *List all the rules. Derive these rules from those of binary products.*

Exercise 6.2 (compulsory). *Define the coinductive types of machines for which there are two different ways to input a natural number and if we press a button, they output a natural number.*

Implement a machine which has two numbers in its inner state and outputs one or the other alternating. If we input a number in the first way, the machine adds that to its first inner number. If we input a number in the second way, the machine adds that to its second inner number.

6.1 Standard model

We define streams and machines using Agda's coinductive types.

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
open Stream
genStream : {A C : Set} → (C → A) → (C → C) → C → Stream A
head (genStream f g c) = f c
tail (genStream f g c) = genStream f g (g c)

record Machine : Set where
  coinductive
  field
    put : ℕ → Machine
    set : Machine
    get : ℕ
open Machine
genMachine : {C : Set} → (C → ℕ → C) → (C → C) → (C → ℕ) → C → Machine
put (genMachine f g h c) n = genMachine f g h (f c n)
set (genMachine f g h c) = genMachine f g h (g c)
get (genMachine f g h c) = h c
```

The new components in the standard model:

```
; Stream       = Stream
; head         = λ t γ* → head (t γ*)
; tail         = λ t γ* → tail (t γ*)
; genStream    = λ u v t γ* → genStream (λ x → u (γ* , x)) (λ x → v (γ* , x)) (t γ*)
; Streamβ₁     = refl
; Streamβ₂     = refl
; head[]       = refl
; tail[]       = refl
; genStream[]  = refl

; Machine      = Machine
; put          = λ t u γ* → put (t γ*) (u γ*)
; set          = λ t γ* → set (t γ*)
; get          = λ t γ* → get (t γ*)
; genMachine   = λ u v w t γ* → genMachine (λ x y → u ((γ* , x) , y))
                  (λ x → v (γ* , x)) (λ x → w (γ* , x)) (t γ*)
; Machineβ₁    = refl
; Machineβ₂    = refl
; Machineβ₃    = refl
; put[]        = refl
; set[]        = refl
; get[]        = refl
; genMachine[] = refl
```

Chapter 7

SK combinator calculus

Moses Schönfinkel's combinator calculus [18] was the earliest description of universal computation, before Church's lambda calculus [7] and Turing machines [20]. It avoids the usage of variables which makes its definition much simpler, compare it with the substitution calculus of Section 2.3 corresponding to lambda calculus. However writing (and reading!) programs in SK is much harder than using lambda calculus. Here we present the simply typed variant of combinator calculus with one base type ι.

TODO: conversion from lambda terms.

An SK model consists of two sorts (types and term), two type operations, three term operations and two term equations.

```
record Model {i j} : Set (lsuc (i ⊔ j)) where
  infixr 5 _⇒_
  infixl 5 _$_

  field
    Ty  : Set i
    ι   : Ty
    _⇒_ : Ty → Ty → Ty
    Tm  : Ty → Set j
    _$_ : ∀{A B}     → Tm (A ⇒ B) → Tm A → Tm B
    K   : ∀{A B}     → Tm (A ⇒ B ⇒ A)
    S   : ∀{A B C} → Tm ((A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C)
    Kβ  : ∀{A B}{t : Tm A}{u : Tm B} → K $ t $ u ≡ t
    Sβ  : ∀{A B C}{t : Tm (A ⇒ B ⇒ C)}{u : Tm (A ⇒ B)}{v : Tm A} →
          S $ t $ u $ v ≡ t $ v $ (u $ v)
```

The I combinator can be derived (here we call it I' to distinguish from the initial model).

```
I' : ∀{A} → Tm (A ⇒ A)
I' {A} = S $ K $ K {A}{ι}

Iβ : ∀{A}{u : Tm A} → I' $ u ≡ u
Iβ {A}{u} =
  I' $ u
                ≡⟨ refl ⟩
  S $ K $ K $ u
                ≡⟨ Sβ ⟩
  K $ u $ (K $ u)
                ≡⟨ Kβ ⟩
  u
                ∎
```

Another famous combinator is B (which we call B').

```
B' : ∀{A B C} → Tm ((B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C)
B' = S $ (K $ S) $ K

Bβ : ∀{A B C}{t : Tm (B ⇒ C)}{u : Tm (A ⇒ B)}{v : Tm A} → B' $ t $ u $ v ≡ t $ (u $ v)
Bβ {t = t}{u = u}{v = v} =
  B' $ t $ u $ v
                                ≡⟨ refl ⟩
  S $ (K $ S) $ K $ t $ u $ v
                                ≡⟨ cong {A = Tm _} (λ x → x $ u $ v) Sβ ⟩
  K $ S $ t $ (K $ t) $ u $ v
                                ≡⟨ cong {A = Tm _} (λ x → x $ (K $ t) $ u $ v) Kβ ⟩
  S $ (K $ t) $ u $ v
                                ≡⟨ Sβ ⟩
  K $ t $ v $ (u $ v)
                                ≡⟨ cong {A = Tm _} (λ x → x $ (u $ v)) Kβ ⟩
  t $ (u $ v)
                                ∎
```

The standard model.

```
St : Model
St = record
  { Ty  = Set
  ; Tm  = λ A → A
  ; ι   = Lift ⊤
  ; _⇒_ = λ A B → A → B
  ; _$_ = λ f x → f x
  ; K   = λ x y → x
  ; S   = λ x y z → x z (y z)
  ; Kβ  = refl
  ; Sβ  = refl
  }
module St = Model St
```

Normal forms are given as an inductive predicate over terms (as opposed to Subsection 1.5.2 where they are defined as sets together with a functions into terms – the two representations are related by the family–map correspondance, see [5, p. 221]).

A term is in normal form if it is a partial application of either K or S. A full application of K has two arguments, a full application of S has three arguments (and they are equal to some other term using the equations Kβ and Sβ).

```
open I
data Nf : (A : I.Ty) → I.Tm A → Set where
  K₀ : ∀{A B} → Nf (A ⇒ B ⇒ A) K
  K₁ : ∀{A B t} → Nf A t → Nf (B ⇒ A) (K $ t)
  S₀ : ∀{A B C} → Nf ((A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C) S
  S₁ : ∀{A B C t} → Nf (A ⇒ B ⇒ C) t → Nf ((A ⇒ B) ⇒ A ⇒ C) (S $ t)
  S₂ : ∀{A B C t u} → Nf (A ⇒ B ⇒ C) t → Nf (A ⇒ B) u → Nf (A ⇒ C) (S $ t $ u)
```

The normalisation dependent model: for each type, we have a predicate over terms of that type together with a function which says that if the predicate holds for a term, then it is in normal form. For each term we have a witness of the predicate.

For ι, the predicate is always false. For a function t the predicate says that if the predicate holds for an input u then it holds for the output t I.$ u, and t has to be in normal form.

```
-- naive normalisation doesn't work:
{-
Norm' : DepModel
```

```
Norm' = record
       { Ty = λ _ → Lift ⊤
       ; ι = mk _
       ; _⇒_ = _
       ; Tm = λ {A} _ → Nf A
       ; _$_ = λ {
         K₀ → K₁ ;
         (K₁ nt) nu → {! nt!} ; -- this is OK
         S₀ → S₁ ;
         (S₁ x) → S₂ x ;
         (S₂ {A}{B}{C} nt nu) nv → {!!} }  -- S $ t $ u₁ $ u = t $ u $ (u₁ $ u)
       ; K = K₀
       ; S = S₀
       ; Kβ = {!!}
       ; Sβ = {!!}
       }
-}

Norm : DepModel
Norm = record
  { Ty•  = λ A → Σ (I.Tm A → Set) λ P → {t : I.Tm A} → P t → Nf A t
  ; Tm•  = π₁
  ; ι•   = (λ _ → Lift ⊥) , λ where ()
  ; _⇒•_ = λ {A}{B} A• B• →
           (λ t → ({u : I.Tm A} → π₁ A• u → π₁ B• (t I.$ u)) × Nf (A I.⇒ B) t) , π₂
  ; _$•_ = λ t u → π₁ t u
  ; K•   = λ {A}{B}{A•}{B•} →
           (λ t → (λ u → transp (π₁ A•) (I.Kβ ⁻¹) t) , K₁ (π₂ A• t)) , K₀
  ; S•   = λ {_}{_}{_}{_}{_}{C•} →
           (λ t → (λ u → (λ v → transp (π₁ C•) (I.Sβ ⁻¹) (π₁ (π₁ t v) (π₁ u v))
                         ) , S₂ (π₂ t) (π₂ u)
                  ) , S₁ (π₂ t)
           ) , S₀
  ; Kβ•  = λ {A}{B}{A•}{B•}{t}{u}{t•}{u•} → transptransp (π₁ A•) (Kβ ⁻¹)
  ; Sβ•  = λ {A}{B}{C}{A•}{B•}{C•}{t}{u}{v}{t•}{u•}{v•} → transptransp (π₁ C•)(Sβ ⁻¹){Sβ}
  }
```

Note that Norm makes use of all the five ways to form normal forms and both equations in our language. To normalise a term of type A, we first interpret the type in the Norm dependent model. This results in a pair the second component of which is a function which produces a normal form from any term for which the predicate holds. Interpreting the term in Norm gives us a witness of the predicate. We put together these as follows.

```
norm : ∀{A}(t : I.Tm A) → Nf A t
norm {A} t = π₂ Norm.⟦ A ⟧T Norm.⟦ t ⟧t
```

Because normal forms are defined as a predicate, the function norm includes completeness. Stability is proven by induction on normal forms.

```
stab : ∀{A t}(n : Nf A t) → norm t ≡ n
stab K₀        = refl
stab (K₁ n)    = cong K₁ (stab n)
stab S₀        = refl
stab (S₁ n)    = cong S₁ (stab n)
stab (S₂ n n') = cong (λ w → S₂ (π₁ w) (π₂ w)) (stab n ,×= stab n')
```

TODO: peldak normalizalasra, kijavitani a hibat, hogy neha vegtelen ciklusba kerulunk
TODO: decidability of equality for normal forms

Chapter 8

Summary

A table with the rules for different type formers of the languages defined in the previous chapters.

type formation	constructor	destructor	computation	uniqueness	
Bool	true, false	ite	iteβ_1, iteβ_2		Chapter 1
Nat	num	isZero, _+o_	isZeroβ_1, isZeroβ_2, +β		
_⇒_	lam	_$_	⇒β	⇒η	Chapter 3
_×o_	⟨_,_⟩	fst, snd	×β_1, ×β_2	×η	Chapter 4
Unit	trivial			Unitη	
_+o_	inl, inr	caseo	+β_1, +β_2	+η	
Empty		aburd		Emptyη	
Nat	zeroo, suco	iteNat	Natβ_1, Natβ_2		Chapter 5
List	nil, cons	iteList	Listβ_1, Listβ_2		
Tree	leaf, node	iteTree	Treeβ_1 Treeβ_2		
Stream	genStream	head, tail	Streamβ_1, Streamβ_2		Chapter 6
Machine	genMachine	put, set, get	Machineβ_1, Machineβ_2, Machineβ_3		

Bibliography

[1] Andreas Abel. Foetus – termination checker for simple functional programs. Programming Lab Report, 1998.

[2] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[3] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.

[4] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[5] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[6] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019.

[7] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345, 1936.

[8] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In Silvia Ghilezan and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.

[9] Zoltán Csörnyei. *Lambda-kalkulus. A funkcionális programozás alapjai*. Typotex, 2007.

[10] Zoltán Csörnyei. *Bevezetés a típusrendszerek elméletébe*. ELTE Eötvös Kiadó, 2012.

[11] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.

[12] Ambrus Kaposi. Levels of abstraction when defining type theory in type theory. Talk given at Conference "Celebrating 90 Years of Gödel's Incompleteness Theorems" in Nürtingen, Germany. Slides: https://akaposi.github.io/pres_godel90.pdf. Video: https://youtu.be/8qRrN6ewSYw, July 2021.

[13] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 3–10. ACM, 2010.

[14] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 1999.

[15] Michel Parigot. On the representation of data in lambda-calculus. In Egon Börger, Hans Kleine Büning, and Michael M. Richter, editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *Lecture Notes in Computer Science*, pages 309–321. Springer, 1989.

[16] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[17] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2021. Version 6.1, http://softwarefoundations.cis.upenn.edu.

[18] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3):305–316, Sep 1924.

[19] William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.

[20] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[21] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020. http://plfa.inf.ed.ac.uk/20.07.