Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Language and Compilers

Integers as a Higher Inductive Type

Supervisor:
Dr. Ambrus Kaposi
Associate Professor

Author:
Zoltán Balázs
Computer Science MSc

Budapest, 2024

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

DIPLOMAMUNKA TÉMABEJELENTŐ

Hallgató adatai:
Név: Balázs Zoltán
Neptun kód: HV56L5

Képzési adatok:
Szak: programtervező informatikus, mesterképzés (MA/MSc)
Tagozat : Esti
Belső témavezetővel rendelkezem

Témavezető neve: Kaposi Ambrus Dr.
munkahelyének neve, tanszéke: **ELTE IK, Programozási nyelvek és Fordítóprogramok Tanszék**
munkahelyének címe: **1117, Budapest, Pázmány Péter sétány 1/C.**
beosztás és iskolai végzettsége: **egyetemi docens, programtervező informatikus**

A diplomamunka címe: Integers as a Higher Inductive Type
A diplomamunka témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben diplomamunka témájának leírását)

We define Integers in Homotopy Type Theory as a Higher Inductive Type following Altenkirch and Scoccola (LiCS 2020). We formalize this definition in the proof assistant cubical Agda and compare it with the traditional natural number, normal-form, and initial ring definitions of Integers.
We will formalize the proof that Integers form a commutative ring as well as other basic properties of Integers.

A set and two binary operations (here referred to as addition and multiplication) form a ring if the set and addition form an abelian group, the set is monoid under multiplication, where multiplication distributes over addition. For a ring to be commutative the ring's multiplication operation must also be commutative.
A set and an operation (here referred to as addition) form an abelian group (commutative group) if the operation is associative, the identity element exists, an inverse element exists and the operation is commutative.
A set and an operation (here referred to as multiplication) form a monoid if multiplication is associative and the identity element exists.

Budapest, 2024. 05. 13.

Contents

Chapter 1

Introduction

1.1 Type Definition

In Homotopy Type Theory, we can define integers in numerous ways (one of them is using bi-invertible maps, which is a slightly less elegant version than ours[1], since it involves 2 *pred* constructors, and ditches the *coherence* rule, in favour of being less complicated to prove properties. This bi-invertible map version is also included in the cubical Agda library[2], which will prove useful for us), all with their own pros and cons. Following Paolo Capriotti's idea (presented by Thorsten Altenkirch[3]), our higher inductive type definition will be the following:

```
data ℤh : Set where
  zero : ℤh
  succ : ℤh → ℤh
  pred : ℤh → ℤh
  sec : (z : ℤh) → pred (succ z) ≡ z
  ret : (z : ℤh) → succ (pred z) ≡ z
  coh : (z : ℤh) → congS succ (sec z) ≡ ret (succ z)
```

With this definition, we have the base element *zero*, as well as *succ* and *pred* as constructors, to increment and decrement the integer value respectively. We then postulate that they are inverse to each other with the inclusion of *sec* (*section*, often seen in the Agda cubical library as *predSuc*) and *ret* (*retraction*, often seen as *sucPred* in the same library). With a *coh* (*coherence*) constructor, we define an equivalence of equivalences, this constructor will introduce most of the challenges when trying to work with our integer definition. With the inclusion of this last condition, we also say that succ is a half-adjoint equivalence, something that we can use to our advantage in cubical Agda:

```
isHAℤh : isHAEquiv succ
isHAℤh .isHAEquiv.g    = pred
isHAℤh .isHAEquiv.linv = sec
isHAℤh .isHAEquiv.rinv = ret
isHAℤh .isHAEquiv.com  = coh
```

Using this, we can define a sort of inverse coherence rule, a coherence that inverses the equivalence by applying *pred* to *ret*, and checking that it is equal to passing the *pred* value to *sec*:

```
hoc : (z : ℤh) → congS pred (ret z) ≡ sec (pred z)
hoc = com-op isHAℤh
```

$com - op$ simply uses the given fields to do the work for us, if our type is right, by filling the boundary with *hcomp*. This rule will be useful later on, when defining operations on our integer type. (Specifically when defining negation)

1.2 Commutative Ring

Our question is the following: Is this definition of integers a correct one, is it a set with decidable equality, and if so, do they form a commutative ring? (While this should be fairly obvious, integers do form a commutative ring, with the higher inductive type definition of integers, this hasn't been formally proven yet.) Moreover, what does it mean for integers to form a commutative ring? We will have to prove the following:

- The set and two binary operations (here: addition and multiplication) form a ring:
  - The set and addition form an abelian group:
    - Addition is associative
    - The identity element exists
    - An inverse element exists

    * Addition is commutative
  - The set is monoid under multiplication:
    * Multiplication is associative
    * The indetity element exists
  - Multiplication distributes over addition
- Multiplication is commutative

Before we prove these, we will dive into proving some other useful properties first.

Chapter 2

Set properties

Defining set properties first will make it much easier for us to define operations on our type (iterator), as well make it fairly trivial to prove the needed properties for a set to form a commutative ring (induction property). Let us define the induction property first, to make this easier, we will define a helper property, the induction principle (otherwise known as the eliminator).

2.1 Induction principle (eliminator)

Defining the induction principle will be fairly easy, given that we have a correct type definition:

```
ℤₕ-ind :
  ∀ {ℓ} {P : ℤₕ → Type ℓ}
  → (P-zero : P zero)
  → (P-succ : ∀ z → P z → P (succ z))
  → (P-pred : ∀ z → P z → P (pred z))
  → (P-sec : ∀ z → (pz : P z) →
          PathP
            (λ i → P (sec z i))
            (P-pred (succ z) (P-succ z pz))
            pz)
  → (P-ret : ∀ z → (pz : P z) →
          PathP
            (λ i → P (ret z i))
```

```
            (P-succ (pred z) (P-pred z pz))
            pz)
  → (P-coh : ∀ z → (pz : P z) →
          SquareP
            (λ i j → P (coh z i j))
            (congP (λ i → P-succ (sec z i)) (P-sec z pz))
            (P-ret (succ z) (P-succ z pz))
            refl
            refl)
  → (z : ℤh)
  → P z
```

This eliminator will allow us to !!EXPAND!! The definition is fairly trivial, we will just need to pattern match on the given integer and use recursion:

```
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh zero        = P-zero
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh (succ z)    = P-succ z
↪   (ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh z)
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh (pred z)    = P-pred z
↪   (ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh z)
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh (sec z i)   = P-sec z
↪   (ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh z) i
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh (ret z i)   = P-ret z
↪   (ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh z) i
ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh (coh z i j) = P-coh z
↪   (ℤh-ind P-zero P-succ P-pred P-sec P-ret P-coh z) i j
```

2.2 Induction property

With the induction principle, defining the induction property is even easier. The induction property will allow us to only prove the commutative ring properties for the base element and the 0-dimensional constructors (*succ* and *pred*), for the 1- and 2-dimensional constructors (*sec*, *ret* and *coh*) the proofs will be derived:

```
ℤh-ind-prop :
  ∀ {ℓ} {P : ℤh → Type ℓ}
  → (∀ z → isProp (P z))
  → P zero
  → (∀ z → P z → P (succ z))
  → (∀ z → P z → P (pred z))
  → (z : ℤh)
  → P z
ℤh-ind-prop {P = P} P-isProp P-zero P-succ P-pred =
  ℤh-ind
    P-zero
    P-succ
    P-pred
    (λ z pz → toPathP (P-isProp z _ _))
    (λ z pz → toPathP (P-isProp z _ _))
    (λ z pz → isProp→SquareP (λ i j → P-isProp (coh z i j)) _ _ _ _)
```

(Note: We use the fact that Agda can infer the needed arguments for $P-isProp$, we can also manually give these parameters, but this would only lengthen our definition. See the source file for the manually given parameters.)

2.3 Iterator

While the induction property is useful for allowing us to use induction when proving properties, the iterator property will make it easier for us to define operations on our type. While it would be possible to manually pattern match, we would have a hard time to give the needed boundaries in the 1- and 2-dimensional cases, especially in the case of multiplication.

Our iterator will !!NEEDED!!

```
ℤh-ite :
  ∀ {ℓ} {A : Type ℓ}
  → A
  → A ≃ A
  → ℤh
```

```
  → A
ℤₕ-ite {A = A} a e =
  let
    (s , isHA) = equiv→HAEquiv e
  in
    ℤₕ-ind
      {P = λ _ → A}
      a
      (λ _ → s)
      (λ _ → g isHA)
      (λ _ → linv isHA)
      (λ _ → rinv isHA)
      (λ _ → com isHA)
```

2.4 IsSet

This will be the first quite labours property to define. To make our live easier in the future, we will have to prove that our definition of integers actually form a set. To prove this, we will prove that our definition of integers is isomorphic with the standard definition of integers in cubical Agda:

```
data ℤ : Type₀ where
  pos    : (n : ℕ) → ℤ
  negsuc : (n : ℕ) → ℤ
```

To do this, we will need to define 4 functions:

- We can convert our type to the standard integer definition
- We can convert from the standard integer definition to our type
- Converting the standard integer definition to our type, and back to the standard integer definition results in the exact same value
- Converting our type to the standard integer definition, and back to our type results in the exact same value

Let us begin with proving these.

2.4.1 Converting our type to the standard definition

!!NEEDED!!

```
ℤₕ-ℤ : ℤₕ → ℤ
ℤₕ-ℤ zero         = pos zero
ℤₕ-ℤ (succ x)     = sucℤ (ℤₕ-ℤ x)
ℤₕ-ℤ (pred x)     = predℤ (ℤₕ-ℤ x)
ℤₕ-ℤ (sec x i)    = predSuc (ℤₕ-ℤ x) i
ℤₕ-ℤ (ret x i)    = sucPred (ℤₕ-ℤ x) i
ℤₕ-ℤ (coh x i j) = isSetℤ
  (sucℤ (predℤ (sucℤ (ℤₕ-ℤ x))))
  (sucℤ (ℤₕ-ℤ x))
  (congS sucℤ (predSuc (ℤₕ-ℤ x)))
  (sucPred (sucℤ (ℤₕ-ℤ x)))
  i j
```

2.4.2 Converting the standard definition to our type

!!NEEDED!!

```
ℤ-ℤₕ : ℤ → ℤₕ
ℤ-ℤₕ (pos zero)         = zero
ℤ-ℤₕ (pos (suc n))      = succ (ℤ-ℤₕ (pos n))
ℤ-ℤₕ (negsuc zero)      = pred zero
ℤ-ℤₕ (negsuc (suc n)) = pred (ℤ-ℤₕ (negsuc n))
```

2.4.3 Converting the standard definition to our type and back

!!NEEDED!!

```
ℤ-ℤₕ-ℤ : (z : ℤ) → ℤₕ-ℤ (ℤ-ℤₕ z) ≡ z
ℤ-ℤₕ-ℤ (ℤ.pos zero)       = refl
ℤ-ℤₕ-ℤ (ℤ.pos (suc n))   = cong sucℤ (ℤ-ℤₕ-ℤ (ℤ.pos n))
ℤ-ℤₕ-ℤ (negsuc zero)      = refl
ℤ-ℤₕ-ℤ (negsuc (suc n)) = cong predℤ (ℤ-ℤₕ-ℤ (negsuc n))
```

2.4.4 Converting our type to the standard definition and back

!!NEEDED!!

```
ℤ-ℤh-sucℤ : (z : ℤ) → ℤ-ℤh (sucℤ z) ≡ succ (ℤ-ℤh z)
ℤ-ℤh-sucℤ (pos n)            = refl
ℤ-ℤh-sucℤ (negsuc zero)      = sym (ret (ℤ-ℤh (pos zero)))
ℤ-ℤh-sucℤ (negsuc (suc n)) = sym (ret (ℤ-ℤh (negsuc n)))

ℤ-ℤh-predℤ : (z : ℤ) → ℤ-ℤh (predℤ z) ≡ pred (ℤ-ℤh z)
ℤ-ℤh-predℤ (pos zero)      = refl
ℤ-ℤh-predℤ (pos (suc n)) = sym (sec (ℤ-ℤh (pos n)))
ℤ-ℤh-predℤ (negsuc n)      = refl

sym-filler : ∀ {ℓ} {A : Type ℓ} {x y : A} (p : x ≡ y)
                → Square (sym p)
                         refl
                         refl
                         p
sym-filler p i j = p (i ∨ ~ j)

ℤ-ℤh-sucPred : (z : ℤ)
              → Square (ℤ-ℤh-sucℤ (predℤ z) ∙ (λ j → succ (ℤ-ℤh-predℤ z j)))
                       (λ _ → ℤ-ℤh z)
                       (λ i → ℤ-ℤh (sucPred z i))
                       (ret (ℤ-ℤh z))
ℤ-ℤh-sucPred (pos zero) i j =
  hcomp (λ k → λ { (j = i0) → ℤ-ℤh (pos zero)
                 ; (i = i0) → rUnit (sym (ret (ℤ-ℤh (pos zero)))) k j
                 ; (i = i1) → ℤ-ℤh (pos zero)
                 ; (j = i1) → ret (ℤ-ℤh (pos zero)) i
                 })
        (sym-filler (ret (ℤ-ℤh (pos zero))) i j)
```

```
ℤ-ℤh-sucPred (pos (suc n)) i j =
  hcomp (λ k → λ { (j = i0) → succ (ℤ-ℤh (pos n))
                 ; (i = i0) → lUnit (λ i → succ (sym (sec (ℤ-ℤh (pos n))) i)) k j
                 ; (i = i1) → succ (ℤ-ℤh (pos n))
                 ; (j = i1) → coh (ℤ-ℤh (pos n)) k i
                 })
        (succ (sym-filler (sec (ℤ-ℤh (pos n))) i j))
ℤ-ℤh-sucPred (negsuc n) i j =
  hcomp (λ k → λ { (j = i0) → ℤ-ℤh (negsuc n)
                 ; (i = i0) → rUnit (sym (ret (ℤ-ℤh (negsuc n)))) k j
                 ; (i = i1) → ℤ-ℤh (negsuc n)
                 ; (j = i1) → ret (ℤ-ℤh (negsuc n)) i
                 })
        (sym-filler (ret (ℤ-ℤh (negsuc n))) i j)

ℤ-ℤh-predSuc : (x : ℤ)
             → Square (ℤ-ℤh-predℤ (sucℤ x) ∙ (λ i → pred (ℤ-ℤh-sucℤ x i)))
                      (λ _ → ℤ-ℤh x)
                      (λ i → ℤ-ℤh (predSuc x i))
                      (sec (ℤ-ℤh x))
ℤ-ℤh-predSuc (pos n) i j =
    hcomp (λ k → λ { (j = i0) → ℤ-ℤh (pos n)
                   ; (i = i0) → rUnit (sym (sec (ℤ-ℤh (pos n)))) k j
                   ; (i = i1) → ℤ-ℤh (pos n)
                   ; (j = i1) → sec (ℤ-ℤh (pos n)) i
                   })
          (sym-filler (sec (ℤ-ℤh (pos n))) i j)
ℤ-ℤh-predSuc (negsuc zero) i j =
    hcomp (λ k → λ { (j = i0) → ℤ-ℤh (negsuc zero)
                   ; (i = i0) → lUnit (λ i → pred (sym (ret (ℤ-ℤh (pos zero))) i)) k j
                   ; (i = i1) → ℤ-ℤh (negsuc zero)
                   ; (j = i1) → hoc (ℤ-ℤh (pos zero)) k i
```

```
                   })
          (pred (sym-filler (ret (ℤ-ℤh (pos zero))) i j))
ℤ-ℤh-predSuc (negsuc (suc n)) i j =
    hcomp (λ k → λ { (j = i0) → ℤ-ℤh (negsuc (suc n))
                   ; (i = i0) → lUnit (λ i → pred (sym (ret (ℤ-ℤh (negsuc n))) i)) k j
                   ; (i = i1) → ℤ-ℤh (negsuc (suc n))
                   ; (j = i1) → hoc (ℤ-ℤh (negsuc n)) k i
                   })
          (pred (sym-filler (ret (ℤ-ℤh (negsuc n))) i j))

ℤh-ℤ-ℤh zero            = refl
ℤh-ℤ-ℤh (succ z)        = ℤ-ℤh-sucℤ (ℤh-ℤ z) ● (λ i → succ (ℤh-ℤ-ℤh z i))
ℤh-ℤ-ℤh (pred z)        = ℤ-ℤh-predℤ (ℤh-ℤ z) ● (λ i → pred (ℤh-ℤ-ℤh z i))
ℤh-ℤ-ℤh (sec z i) j   =
  hcomp (λ k → λ { (j = i0) → ℤ-ℤh (predSuc (ℤh-ℤ z) i)
                 ; (i = i0) → (ℤ-ℤh-predℤ (sucℤ (ℤh-ℤ z)) ● (λ i → pred (compPath-filler (ℤ-ℤh-sucℤ (ℤh-ℤ z))
                     (λ i' → succ (ℤh-ℤ-ℤh z i'))
                     k i))) j
                 ; (i = i1) → ℤh-ℤ-ℤh z (j ∧ k)
                 ; (j = i1) → sec (ℤh-ℤ-ℤh z k) i })
        (ℤ-ℤh-predSuc (ℤh-ℤ z) i j)
ℤh-ℤ-ℤh (ret z i) j   =
  hcomp (λ k → λ { (j = i0) → ℤ-ℤh (sucPred (ℤh-ℤ z) i)
                 ; (i = i0) → (ℤ-ℤh-sucℤ (predℤ (ℤh-ℤ z)) ● (λ i → succ (compPath-filler (ℤ-ℤh-predℤ (ℤh-ℤ z))
                     (congS pred (ℤh-ℤ-ℤh z))
                     k i))) j
                 ; (i = i1) → ℤh-ℤ-ℤh z (j ∧ k)
                 ; (j = i1) → ret (ℤh-ℤ-ℤh z k) i  })
        (ℤ-ℤh-sucPred (ℤh-ℤ z) i j)
ℤh-ℤ-ℤh (coh z i j) k = ?
```

2.4.5 Set property

With these 4 functions defined, we can prove that our type is isomorphic with the standard definition:

```
ℤ-iso : Iso ℤ ℤₕ
ℤ-iso .Iso.fun      = ℤ-ℤₕ
ℤ-iso .Iso.inv      = ℤₕ-ℤ
ℤ-iso .Iso.rightInv = ℤₕ-ℤ-ℤₕ
ℤ-iso .Iso.leftInv  = ℤ-ℤₕ-ℤ

ℤ≡ℤₕ : ℤ ≡ ℤₕ
ℤ≡ℤₕ = isoToPath ℤ-iso
```

We pattern match on the constructors of *Iso* (isomorphism) and we provide the needed fields. (As discussed earlier.)

Finally, we can use the fact that the standard definition forms a set to our advantage, as our type is isomorphic with the standard definition means that our type also forms a set:

```
isSetℤₕ : isSet ℤₕ
isSetℤₕ = subst isSet ℤ≡ℤₕ isSetℤ
```

Chapter 3

Abelian Group (Addition)

3.1 Addition Operation

```
succIso : Iso ℤₕ ℤₕ
succIso .Iso.fun      = succ
succIso .Iso.inv      = pred
succIso .Iso.rightInv = ret
succIso .Iso.leftInv  = sec

succEquiv : ℤₕ ≃ ℤₕ
succEquiv = isoToEquiv succIso

infixl 6 _+_
_+_ : ℤₕ → ℤₕ → ℤₕ
_+_ = ℤₕ-ite (idfun ℤₕ) (postCompEquiv succEquiv)
```

3.2 Associativity

```
+-assoc : ∀ m n o → m + (n + o) ≡ (m + n) + o
+-assoc = ℤₕ-ind-prop
  (λ _ → isPropΠ2 λ _ _ → isSetℤₕ _ _)
  (λ n o → refl)
```

```
  (λ m p n o → cong succ (p n o))
  (λ m p n o → cong pred (p n o))
```

3.3 Identity Element

```
+-idˡ : ∀ z → zero + z ≡ z
+-idˡ z = refl

+-zero : ∀ z → z + zero ≡ z
+-zero = ℤₕ-ind-prop
  (λ _ → isSetℤₕ _ _)
  refl
  (λ z p → cong succ p)
  (λ z p → cong pred p)

+-idʳ : ∀ z → z + zero ≡ z
+-idʳ = +-zero
```

3.4 Negation and Subtraction

```
-_ : ℤₕ → ℤₕ
-_ = ℤₕ-ite zero (invEquiv succEquiv)

_-_ : ℤₕ → ℤₕ → ℤₕ
m - n = m + (- n)
```

3.5 Inverse Element

```
+-succ : ∀ m n → m + succ n ≡ succ (m + n)
+-succ = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
  (λ m → refl)
  (λ m p n → cong succ (p n))
  (λ m p n → cong pred (p n) • sec (m + n) • sym (ret (m + n)))

+-pred : ∀ m n → m + pred n ≡ pred (m + n)
+-pred = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
  (λ m → refl)
  (λ m p n → cong succ (p n) • ret (m + n) • sym (sec (m + n)))
  (λ m p n → cong pred (p n))

+-invˡ : ∀ z → (- z) + z ≡ zero
+-invˡ = ℤₕ-ind-prop
  (λ _ → isSetℤₕ _ _)
  refl
  (λ z p → cong pred (+-succ (- z) z) • sec _ • p)
  (λ z p → cong succ (+-pred (- z) z) • ret _ • p)

+-invʳ : ∀ z → z + (- z) ≡ zero
+-invʳ = ℤₕ-ind-prop
  (λ _ → isSetℤₕ _ _)
  refl
  (λ z p → cong succ (+-pred z (- z)) • ret _ • p)
  (λ z p → cong pred (+-succ z (- z)) • sec _ • p)
```

3.6 Commutativity

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm m n = +-comm' n m
  where
```

```
+-comm' : ∀ n m → m + n ≡ n + m
+-comm' = ℤh-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤh _ _)
  +-zero
  (λ n p m → +-succ m n ∙ cong succ (p m))
  (λ n p m → +-pred m n ∙ cong pred (p m))
```

Chapter 4

Monoid (Multiplication)

4.1 Multiplication Operation

```
Iso-n+-ℤₕ : (z : ℤₕ) → Iso ℤₕ ℤₕ
Iso.fun (Iso-n+-ℤₕ z) = z +_
Iso.inv (Iso-n+-ℤₕ z) = - z +_
Iso.rightInv (Iso-n+-ℤₕ n) m = +-assoc n (- n) m • cong (_+ m) (+-invʳ n)
Iso.leftInv (Iso-n+-ℤₕ n) m = +-assoc (- n) n m • cong (_+ m) (+-invˡ n)

isEquiv-n+-ℤₕ : ∀ z → isEquiv (z +_)
isEquiv-n+-ℤₕ z = isoToIsEquiv (Iso-n+-ℤₕ z)

_*_ : ℤₕ → ℤₕ → ℤₕ
m * n = ℤₕ-ite zero (n +_ , isEquiv-n+-ℤₕ n) m
```

4.2 Multiplication Distributes over Addition

```
*-distribʳ-+ : ∀ m n o → (m * o) + (n * o) ≡ (m + n) * o
*-distribʳ-+ = ℤₕ-ind-prop
  (λ _ → isPropΠ2 λ _ _ → isSetℤₕ _ _)
  (λ n o → refl)
  (λ m p n o → sym (+-assoc o (m * o) (n * o)) • cong (o +_) (p n o))
  (λ m p n o → sym (+-assoc (- o) (m * o) (n * o)) • cong (- o +_) (p n
  ↪  o))
```

```

*-distrib^l-+ : ∀ o m n → (o * m) + (o * n) ≡ o * (m + n)
*-distrib^l-+ o m n = cong (_+ o * n) (*-comm o m) • cong (m * o +_) (*-comm o n) • *-distrib^r-+ m n o • *-comm (m + n) o
```

4.3 Associative

```
inv-hom-ℤ_h : ∀ m n → - (m + n) ≡ (- m) + (- n)
inv-hom-ℤ_h = ℤ_h-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤ_h _ _)
  (λ n → refl)
  (λ m p n → cong pred (p n))
  (λ m p n → cong succ (p n))

*-inv : ∀ m n → m * (- n) ≡ - (m * n)
*-inv = ℤ_h-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤ_h _ _)
  (λ n → refl)
  (λ m p n → cong (- n +_) (p n) • sym (inv-hom-ℤ_h n (m * n)))
  (λ m p n → cong (- (- n) +_) (p n) • sym (inv-hom-ℤ_h (- n) (m * n)))

inv-* : ∀ m n → (- m) * n ≡ - (m * n)
inv-* m n = *-comm (- m) n • *-inv n m • cong (-_) (*-comm n m)

*-assoc : ∀ m n o → m * (n * o) ≡ (m * n) * o
*-assoc = ℤ_h-ind-prop
  (λ _ → isPropΠ2 λ _ _ → isSetℤ_h _ _)
  (λ n o → refl)
  (λ m p n o → cong (n * o +_) (p n o) • *-distrib^r-+ n (m * n) o)
  (λ m p n o → cong (- (n * o) +_) (p n o) • cong (_+ m * n * o) (sym (inv-* n o)) • *-distrib^r-+ (- n) (m * n) o)
```

Chapter 5

Commutative Ring

5.1 Multiplication is Commutative

```
*-succ : ∀ m n → m * succ n ≡ m + m * n
*-succ = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
  (λ n → refl)
  (λ m p n → cong succ (cong (n +_) (p n) • +-assoc n m (m * n) • cong (_+ m * n) (+-comm n m) • sym (+-assoc m n (m * n))))
  (λ m p n → cong pred (cong (- n +_) (p n) • +-assoc (- n) m (m * n) • cong (_+ m * n) (+-comm (- n) m) • sym (+-assoc m (- n) (m * n))))

*-pred : ∀ m n → m * pred n ≡ (- m) + m * n
*-pred = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
  (λ n → refl)
  (λ m p n → cong pred (cong (n +_) (p n) • +-assoc n (- m) (m * n) • cong (_+ m * n) (+-comm n (- m)) • sym (+-assoc (- m) n (m * n))))
  (λ m p n → cong succ (cong (- n +_) (p n) • +-assoc (- n) (- m) (m * n) • cong (_+ m * n) (+-comm (- n) (- m)) • sym (+-assoc (- m) (- n) (m * n))))

*-comm : ∀ m n → m * n ≡ n * m
*-comm = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
```

```
  (λ n → sym (*-zero n))
  (λ m p n → cong (n +_) (p n) ∙ sym (*-succ n m))
  (λ m p n → cong (- n +_) (p n) ∙ sym (*-pred n m))
```

5.2 Putting it All Together

```
AbGroupℤₕ+ : IsAbGroup {lzero} {ℤₕ} zero _+_ (-_)
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.isMonoid .IsMonoid.isSemigroup .IsSemigroup.is-set = isSetℤₕ
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.isMonoid .IsMonoid.isSemigroup .IsSemigroup.·Assoc = +-assoc
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.isMonoid .IsMonoid.·IdR = +-idʳ
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.isMonoid .IsMonoid.·IdL = +-idˡ
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.·InvR = +-invʳ
AbGroupℤₕ+ .IsAbGroup.isGroup .IsGroup.·InvL = +-invˡ
AbGroupℤₕ+ .IsAbGroup.+Comm = +-comm

Monoidℤₕ* : IsMonoid {lzero} {ℤₕ} (succ zero) _*_
Monoidℤₕ* .IsMonoid.isSemigroup .IsSemigroup.is-set = isSetℤₕ
Monoidℤₕ* .IsMonoid.isSemigroup .IsSemigroup.·Assoc = *-assoc
Monoidℤₕ* .IsMonoid.·IdR = *-idʳ
Monoidℤₕ* .IsMonoid.·IdL = *-idˡ

Ringℤₕ*+ : IsRing {lzero} {ℤₕ} zero (succ zero) _+_ _*_ (-_)
Ringℤₕ*+ .IsRing.+IsAbGroup = AbGroupℤₕ+
Ringℤₕ*+ .IsRing.·IsMonoid = Monoidℤₕ*
Ringℤₕ*+ .IsRing.·DistR+ = λ m n o → sym (*-distribˡ-+ m n o)
Ringℤₕ*+ .IsRing.·DistL+ = λ m n o → sym (*-distribʳ-+ m n o)

CommRingℤₕ*+ : IsCommRing {lzero} {ℤₕ} zero (succ zero) _+_ _*_ (-_)
CommRingℤₕ*+ .IsCommRing.isRing = Ringℤₕ*+
CommRingℤₕ*+ .IsCommRing.·Comm = *-comm
```

Chapter 6

Further

6.1 Relations

!!NEEDED!!

```
succ-inj : ∀ m n → succ m ≡ succ n → m ≡ n
succ-inj = ℤₕ-ind-prop
  (λ _ → isPropΠ2 λ _ _ → isSetℤₕ _ _)
  (λ n o → sym (sec zero) ∙ congS pred o ∙ sec n)
  (λ m p n o → sym (sec (succ m)) ∙ congS pred o ∙ sec n)
  (λ m p n o → sym (sec (pred m)) ∙ congS pred o ∙ sec n)

pred-inj : ∀ m n → pred m ≡ pred n → m ≡ n
pred-inj = ℤₕ-ind-prop
  (λ _ → isPropΠ2 λ _ _ → isSetℤₕ _ _)
  (λ n o → sym (ret zero) ∙ congS succ o ∙ ret n)
  (λ m p n o → sym (ret (succ m)) ∙ congS succ o ∙ ret n)
  (λ m p n o → sym (ret (pred m)) ∙ congS succ o ∙ ret n)
```

6.2 Division (and Modulus)

!!NEEDED!!

```
record NonZero (z : ℤh) : Set where
  field
    nonZero : Bool→Type (not (z ≡ zero))

infixl 7 _/_ _%_

div-helper : (k m n j : ℤh) → ℤh
div-helper k m zero     j        = k
div-helper k m (succ n) zero     = div-helper (succ k) m n m
div-helper k m (succ n) (succ j) = div-helper k        m n j

_/_ : (dividend divisor : ℤh) . {{_ : NonZero divisor}} → ℤh
m / (succ n) = div-helper zero n m n

mod-helper : (k m n j : ℤh) → ℤh
mod-helper k m zero      j       = k
mod-helper k m (succ n)  zero    = mod-helper zero     m n m
mod-helper k m (succ n) (succ j) = mod-helper (succ k) m n j

_%_ : (dividend divisor : ℤh) . {{_ : NonZero divisor}} → ℤh
m % (succ n) = mod-helper zero n m n
```

6.3 Exponentiation

!!NEEDED!!

```
infixr 8 _^^_

_^^_ : ℤh → ℕ → ℤh
a ^^ zero = succ zero
zero ^^ suc b = zero
x@(succ a) ^^ suc b = x * (x ^^ b)
x@(pred a) ^^ suc b = x * (x ^^ b)
```

Bibliography

[1] Thorsten Altenkirch and Luis Scoccola. "The Integers as a Higher Inductive Type". In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20. Saarbrücken, Germany: Association for Computing Machinery, 2020, pp. 67–73. ISBN: 9781450371049. DOI: 10.1145/3373718.3394760. URL: https://doi.org/10.1145/3373718.3394760.

[2] Anders Mörtberg Felix Cherubini. *A standard library for Cubical Agda, Bi-invertible integers.* https://github.com/agda/cubical/blob/master/Cubical/Data/Int/MoreInts/BiInvInt/Base.agda. [Accessed: November 19, 2024]. 2024.

[3] Nicolai Kraus Paolo Capriotti Ambrus Kaposi. *Towards higher models and syntax of type theory.* https://people.cs.nott.ac.uk/psztxa/talks/bonn18.pdf#page=10. [Accessed: November 19, 2024]. June 2018.

List of Figures

List of Tables

List of Codes