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 'coh' 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 : Set₀ 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 previously given fields and does the work for us, if our type is right, by correctly combining paths with 'hcomp' to match the wanted boundary. This rule will be useful later on, when defining operations on our integer type. (Specifically when defining negation)

1.2 Why Cubical Agda

Since established that we want to use a Higher Inductive Type (HIT), which is not available in Martin-Löf Type Theory (MLTT). Agda, by default, doesn't support HITs as it is based on MLTT; an extension of it does however support CTT. While HoTT can essentially be summarized[4] by being MLTT with an additional Univalence axiom (where univalence states the following: $(A \simeq B) \simeq (A = B)$, where A and B are types). Cubical Agda doesn't directly support HoTT; instead, it is a closely related version of Type Theory called Cubical Type Theory (CTT). CTT instead can be summarized as MLTT, with the interval type and its rules, the transport function and its rules, the glue type and hcomp. In CTT, univalence will not be an axiom but something that can be proven[5].

Since we need to use a proof assistant that supports HITs, which CTT supports,[6], we are forced to use Cubical Agda[7], as there are no other proof assistants that do so. This will definitionally bring the added work that we will be forced to satisfy the boundaries required, as of the writing of this paper, there is no automatic way for these to be filled, but that might be subject to change[8].

The main benefit of our HIT integers will be that we can eliminate to higher dimensions (more specifically, above set, for example, to groupoid). The same could be said about Bi-Invertible integers, which use 2 'pred' constructors instead, and ditch the 'coherent' rule, however, a single 'pred' constructor with a 'coherent' rule is more elegant. There is also a definition of integers with the added 'isSet' constructor, while you could also eliminate with this version, you couldn't eliminate above set. Moreover, when defining your indunction property, you would also have to define the case for the 'isSet' constructor.

1.3 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 identity 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

Induction Principle of HIT Integers

Before proving that HIT integers form a set, we will define some helper functions which 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
```

Code 2.1: Agda type definition of the induction principle

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
```

Code 2.2: Agda code for the induction principle

2.2 Induction property

With the induction principle, defining the induction property is easy. The induction property will allow us to only prove the upcoming 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 induced from the 'succ' and 'pred' proofs:

```
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)) _ _ _
 ↪  _)
```

Code 2.3: Agda code for the induction property

(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.

```
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)
```

Code 2.4: Agda code for the iterator

Chapter 3

Proving that HIT Integers Form a Set

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

This is a sort of pseudo comparison of the standard integer type and our type as well, if we prove that our type is isomorphic with the standard integer type, then the two types are equivalent as well.

Note: We could also go the way of defining that our HIT integers are discrete. One of the main points of the thesis is to compare different integer definitions to the HIT integers, so we will try to do less work and prove that our HIT integers form a set while comparing them to the standard integer definition. This also brings the quite fortunate side-effect that any other integer definitions that also prove isomorphism with the standard integers (in the cubical library one notable example is the previously mentioned bi-invertible integers) also, by extension, are isomorphic with our HIT integers.

Let us begin with proving the equivalence between standard integers and HIT integers.

3.1 Converting our type to the standard definition

First off, we will define the function to convert from our type to the standard integer type. We will do so by pattern matching on our integer value. This will give us all the cases for our constructors, for which we will need to provide an equivalent standard integer value. For our 'zero' value, the equivalent standard integer value is 'pos zero'. For our 'succ z' case, we will need to increment the value with 'sucℤ' and recursively convert the rest by calling our conversion function with 'z'. 'sucℤ' is defined for the standard integers, its purpose is to increment the standard integer value by one. For our 'pred z' case we will do the same thing, just with 'predℤ', as one might imagine, 'predℤ' is responsible for decrementing the standard integer value by one. The next case is the first truly interesting one, as we will have to convert a 'sec' integer to a standard integer value. As previously mentioned 'sec' is responsible for the equivalence of `pred (succ z) ≡ z`. As such, we will not only need to give an appropriate standard integer value, but also satisfy the boundaries as stated by Agda:

```
i = i0 ⊢ predℤ (sucℤ (ℤₕ-ℤ z))
i = i1 ⊢ ℤₕ-ℤ z
```

These boundaries match what our 'sec' constructor state, just for the standard integer values. (Note that instead of our integer type, we are dealing with the converted standard integer values, as such, we are basically proving that `predℤ (sucℤ (ℤₕ-ℤ z)) ≡ ℤₕ-ℤ z`). Thankfully for us, there is a function in the standard library for this exact case called 'predSuc', this provides the exact boundary we need: `∀ z → predℤ (sucℤ z) ≡ z`, as before, we will need to recursively call with the converted remaining integer value, as well as to pass the boundary 'i', as without that, it is just an equivalence, but giving the 'i' boundary resolves it to a standard integer value. This is also the case for our 'ret' constructor, we will just need to use the standard 'sucPred' function, which is exactly what we need to satisfy the boundaries. (The wanted boundary is also different, to reflect the fact that we need to prove that 'ret' is defined as `succ (pred z) ≡ z`.) Our last constructor is 'coh', for which we will need a standard integer value and satisfy the needed boundaries. We will define the 'coh' rule for the standard integers. Following the cubical library, we will pattern match, do cover the 'pos n', 'negsuc zero' and 'negsuc (suc n)' cases. All of these cases result in a reflection, thanks to the way that 'sucℤ', 'predSuc' and 'sucPred' are defined.

```
cohℤ : ∀ z → congS sucℤ (predSuc z) ≡ sucPred (sucℤ z)
cohℤ (pos n)             = refl
cohℤ (negsuc zero)       = refl
cohℤ (negsuc (suc n))  = refl
```

Note: the wanted boundaries for the conversion's 'coh' branch are as such:

```
j = i0 ⊢ sucℤ (predℤ (sucℤ (ℤₕ→ℤ z)))
j = i1 ⊢ sucℤ (ℤₕ→ℤ z)
i = i0 ⊢ sucℤ (predSuc (ℤₕ→ℤ z) j)
i = i1 ⊢ sucPred (sucℤ (ℤₕ→ℤ z)) j
```

these exactly correspond to what the 4 parts of our 'coh' rule are, just with the standard integer functions and values. When using the 'coh' rule defined for the standard integers, we will have to convert our HIT integer to a standard integer, as well as providez both of our original boundaries.

Rewriting these wanted boundaries in Agda to a Square results in the following:

$$\begin{tikzcd}
\bullet \arrow[r, dash, "suc\mathbb{Z}\,(\mathbb{Z}_h \to \mathbb{Z}\, z)"] \arrow[d, dash, "suc\mathbb{Z}\,(predSuc\,(\mathbb{Z}_h \to \mathbb{Z}\, z)\, j)"'] & \bullet \arrow[d, dash, "sucPred\,(suc\mathbb{Z}\,(\mathbb{Z}_h \to \mathbb{Z}\, z))\, j"] \\
\bullet \arrow[r, dash, "suc\mathbb{Z}\,(pred\mathbb{Z}\,(suc\mathbb{Z}\,(\mathbb{Z}_h \to \mathbb{Z}\, z)))"'] & \bullet
\end{tikzcd}$$

```
ℤₕ→ℤ : ℤₕ → ℤ
ℤₕ→ℤ zero         = pos zero
ℤₕ→ℤ (succ z)     = sucℤ (ℤₕ→ℤ z)
ℤₕ→ℤ (pred z)     = predℤ (ℤₕ→ℤ z)
ℤₕ→ℤ (sec z i)    = predSuc (ℤₕ→ℤ z) i
ℤₕ→ℤ (ret z i)    = sucPred (ℤₕ→ℤ z) i
ℤₕ→ℤ (coh z i j) = cohℤ (ℤₕ→ℤ z) i j
```

Code 3.1: Agda code for converting between HIT and the standard integers

3.2 Converting the standard definition to our type

Converting from the standard definition to our type turns out to be easier than the vice-versa. Pattern matching on both parameters (the default 'n' parameter, then the expanded 'pos n' and 'negsuc n' parameter) introduces four cases. We need to remember that 'pos zero' is equivalent to our 'zero' constructor, 'pos (suc n)' is equivalent to applying our 'succ' constructor, to the recursively converted 'pos n' value. 'negsuc' is an interesting case, as it is definitionally introduces '-1' to the value, so 'negsuc zero' is equivalent to '0 - 1', which is our 'pred zero', equivalently, 'negsuc (suc n)' is equivalent to applying 'pred' to the recursively converted 'negsuc n' value.

```
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))
```

Code 3.2: Agda code for converting between standard integers and HIT integers

3.3 Converting the standard definition to our type and back

Converting to and from isn't enough, we have to also ensure that if we are to convert a standard integer to our type, then that value in our type back to the standard integer, we get the same initial value.

As we did before, it is beneficial to pattern match to get the constructors of the standard integers. It turns out that if pattern match once again on the two constructors (so we get 'pos zero', 'pos (suc n)', 'negsuc zero' and 'negsuc (suc n)' cases), two of the four cases ('pos zero' and 'negsuc zero') become reflections. (Thanks to the way we defined '$\mathbb{Z}\rightarrow\mathbb{Z}_h$' and '$\mathbb{Z}_h\rightarrow\mathbb{Z}$' for the previously mentioned two cases, and 'zero' as well as 'pred zero')

Let's deal with the remaining two cases:

For 'pos (suc n)', we will have to prove the following: suc$\mathbb{Z}$ ($\mathbb{Z}_h\rightarrow\mathbb{Z}$ ($\mathbb{Z}\rightarrow\mathbb{Z}_h$ (pos n))) $\equiv$ pos (suc n). Note that more than likely we will need to recursively call the $\mathbb{Z}\rightarrow\mathbb{Z}_h\rightarrow\mathbb{Z}$ function with the remaining 'pos n' value, so let's do so. If we do this, suddenly we will already have the following: $\mathbb{Z}_h\rightarrow\mathbb{Z}$ ($\mathbb{Z}\rightarrow\mathbb{Z}_h$ (pos n)) $\equiv$ pos n. Following the left-hand side, we will use congruence to apply the missing 'suc$\mathbb{Z}$' on both sides, this has the additional benefit that thanks to the definition of 'suc$\mathbb{Z}$', the right-hand side suc$\mathbb{Z}$ (pos n) is definitionally equal to pos (suc n), the exact same thing we need.

For 'negsuc (suc n)' the story is quite similar. We need to prove the following: pred$\mathbb{Z}$ ($\mathbb{Z}_h\rightarrow\mathbb{Z}$ ($\mathbb{Z}\rightarrow\mathbb{Z}_h$ (negsuc n))) $\equiv$ negsuc (suc n), we have the intuition of the recursive call with 'negsuc n' value, doing so results in $\mathbb{Z}_h\rightarrow\mathbb{Z}$ ($\mathbb{Z}\rightarrow\mathbb{Z}_h$ (negsuc n)) $\equiv$ negsuc n, applying 'pred$\mathbb{Z}$' to match the left-hand side brings the additional benefit that definitionally pred$\mathbb{Z}$ (negsuc n) is equal to negsuc (suc n), exactly what we need.

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

Code 3.3: Agda proof that converting standard integers to HIT integers and back results in the same value

3.4 Converting our type to the standard definition and back

Conversely, we will have to prove the other back-and-forth as well. We will need to pattern match to prove cases for all of our HIT integer constructors.

For the 'zero' case, thankfully this is going to be a reflection. (Once again due to the way we defined '$\mathbb{Z}\to\mathbb{Z}_h$' and '$\mathbb{Z}_h\to\mathbb{Z}$'.)

For our 'succ' case, we will need to prove $\mathbb{Z}\to\mathbb{Z}_h$ (suc$\mathbb{Z}$ ($\mathbb{Z}_h\to\mathbb{Z}$ z)) ≡ succ z. We will need to define the equivalence of first incerementing a standard integer with 'suc$\mathbb{Z}$' and converting it to HIT integer and first converting a standard integer to HIT integer and incrementing it with 'succ'. This is also going to be a fairly trivial proof, we will just need to use our 'retraction' constructor on the 'negsuc' cases (more specifically the symmetric version of it) with the standard integer value +1 converted:

```
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)))
```

Once we have this, we can using it, we get the following: $\mathbb{Z}\to\mathbb{Z}_h$ (suc$\mathbb{Z}$ ($\mathbb{Z}_h\to\mathbb{Z}$ z)) ≡ succ ($\mathbb{Z}\to\mathbb{Z}_h$ ($\mathbb{Z}_h\to\mathbb{Z}$ z)), we can use a transitivity to rewrite the right-hand side with a lambda function (to introduce the boundary), since $\mathbb{Z}\to\mathbb{Z}_h$ ($\mathbb{Z}_h\to\mathbb{Z}$ z) is exactly what we are trying to prove is equal to z.

For our 'pred' case, we similarly have to prove that $\mathbb{Z}\to\mathbb{Z}_h$ (pred$\mathbb{Z}$ ($\mathbb{Z}_h\to\mathbb{Z}$ z)) ≡ pred z. We will once again introduce the same equivalence for decrementing with 'pred$\mathbb{Z}$' and converting being the same as converting then decrementing with 'pred'. This is an even more trivial proof, we will to use the 'section' constructor on the 'pos (suc n)' case (again, the symmetric version of it) with the standard integer value -1 converted.

```
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
```

This won't be enough, as using this, we get the following: ℤ→$\mathbb{Z}_h$ (predℤ ($\mathbb{Z}_h$→ℤ z)) ≡ pred (ℤ→$\mathbb{Z}_h$ ($\mathbb{Z}_h$→ℤ z)), using transitivity again, we can rewrite the right-hand side, since it should be equal (at least we are trying to prove it) to pred z.

We won't dive that deep into our 'sec' and 'ret' cases right now, do note that we will introduce a helper function to fill a square. The same proof is present in the standard library, when trying to prove that bi-invertible integers form a set. In the further work chapter, we will dive deeper into how the 'coh' case can be solved, the same idea is used when trying to prove 'sec' and 'ret'.

```
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)
```

The 'sym-filler' helper function is used to fill a square, where 3 points are the same, and we know the equality 'x = y'.

$$\begin{tikzcd}
x \arrow[r, dash, "refl"'] \arrow[d, dash, "p"] & x \arrow[d, dash, "refl"'] \\
y \arrow[r, dash, "sym\ p"] & x
\end{tikzcd}$$

To prove the 'sec' case, the following helper function is introduced:

```
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))
```

To prove the 'ret' case, the following helper function is introcuded:

```
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)
```

```
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)
```

Code 3.4: Agda partial proof that converting HIT integers to standard integers and back results in the same value

3.5 Set truncation

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 4

Abelian Group (Addition)

4.1 Addition Operation

To define addition, we will use our iterator. For this, we will need an equivalence, in this simpler case we will first define an isomorphism, then convert that to an equivalence. This isomorphism will be based on 'succ' being the function, its inverse function will be 'pred', and to prove that using 'succ' then 'pred', and using 'pred' then 'succ' results in the same value, we will use our 'section' and 'retraction' constructors.

We can then convert this isomorphism to an equivalence with the 'isoToEquiv' builtin function. Equivalence is an ordered pair, where the first parameter will be our 'succ' function, and the second will contain the proofs we previously mentioned.

Using the iterator we defined earlier, we can first supply the 'idfun' (identity function) version of $\mathbb{Z}_h$, this will convert $\mathbb{Z}_h$ to a '$\mathbb{Z}_h \to \mathbb{Z}_h$' type. This will be beneficial since we have 2 implicit parameters, and we will want to move the constructors ('succ' and 'pred') from the left parameter to the right one. Next, we will post-compose our defined 'succEquiv'. This will change our '$\mathbb{Z}_h \simeq \mathbb{Z}_h$' to a '$(\mathbb{Z}_h \to \mathbb{Z}_h) \simeq (\mathbb{Z}_h \to \mathbb{Z}_h)$' (more specifically $(\mathrm{C} \to \mathbb{Z}_h) \simeq (\mathrm{C} \to \mathbb{Z}_h)$), this is to match the type needed by our iterator. We implicitly supply the left and right parameters to this. (More specifically, we will first supply our left parameter, for this our iterator will return $\mathbb{Z}_h \to \mathbb{Z}_h$, and we will supply our right parameter to this - see the comment in the code for this).

This will have the desired effect that we wanted: from our left parameter the 'succ' and 'pred' constructors will be placed on the beginning of the right parameter, which is otherwise known as addition.

As we already established when defining our iterator, we won't have to deal with our 1- and 2-dimensional construcotrs and their boundaries, as it is handled by our iterator.

```
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)
-- m + n = (ℤₕ-ite (idfun ℤₕ) (postCompEquiv succEquiv) m) n
```

Definitionally, the following hold true for the addition operation in Agda:

```
zero   + n ≡ n
succ m + n ≡ succ (m + n)
pred m + n ≡ pred (m + n)
```

Note that the symmetric version of these also hold true.

Alternatively, we could define addition by pattern matching on the second parameter, this would mean we manually have to satisfy the boundaries in the 1- and 2-dimensional constructor cases, however, here they would be fairly trivial:

```
_+_ : ℤₕ → ℤₕ → ℤₕ
zero      + b = b
succ a    + b = succ (a + b)
pred a    + b = pred (a + b)
sec a i   + b = sec (a + b) i
```

```
ret a i   + b = ret (a + b) i
coh a i j + b = coh (a + b) i j
```

4.2 Associativity

Theorem 1. *Addition is associative:* $\forall$ *m, n, o* $\in \mathbb{Z}$*: m + (n + o) = (m + n) + o*

Proof. We will use our induction property to prove this theorem. For the first case, which states that this is indeed a property, we will use the 'isProp' (specifically the one which takes 2 parameters) and our set property, thankfully cubical Agda can infer the parameters for this. In all future proofs we will rely on Agda inferring these parameters.

For the base case (m = 0) : 0 + (n + o) = (n + o) = (0 + n) + o is a reflection.

For the succ case: (succ m) + (n + o) = ((succ m) + n) + o, note that since we are using the induction property, we will have the original m + (n + o) = (m + n) + o equality, where we are free to change the parameters of 'n' and 'o' as we please. Applying succ to the original equality will result in succ (m + (n + o)) = succ (m + n + o), thanks to the fact that succ (m + n) = succ m + n holds true definitionally, where we can supply 'm' to the m parameter, and 'n + o' to the n parameter. We will also have to note that addition is parenthesized from the left, so succ m + n + o = (succ m + n) + o is also definitionally true.

For the pred case: (pred m) + (n + o) = ((pred m) + n) + o, similarly to the succ case, it is enough to apply pred to the original equality. We will once again note that pred (m + n) = pred m + n is definitionally true, and the fact that addition is parenthesized from the left. □

```
+-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))
```

Code 4.1: Agda proof of addition being associative

4.3 Identity Element

Before proving the existence of an identity element, we note that in the case of addition, both left and right identity element will be 0.

Theorem 2. *Left identity element exists for addition: $\exists$ id $\in \mathbb{Z}$, $\forall$ z $\in \mathbb{Z}$: id + z = z*

Proof. In Agda, we will rewrite the statement as such: $\forall$ z $\in \mathbb{Z} \rightarrow 0 + z = z$, since we know that the identity element is 0.

The equation $0 + z = z$ is a reflection in Agda. □

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

Code 4.2: Agda proof of addition having a left identity element

Theorem 3. *Right identity element exists for addition: $\exists$ id $\in \mathbb{Z}$, $\forall$ z $\in \mathbb{Z}$: z + id = z*

(Interesting note: the existence of a right identity element, more specifically the Agda rewritten version of 'z + zero = z' is a Peano axiom[9].)

Proof. As in the case of the left identity element, we will rewrite this statement in Agda as such: $\forall$ z $\in \mathbb{Z} \rightarrow z + 0 = z$. We will once again use our induction property for proving this in Agda.

For the base case ($z = 0$): $0 + 0 = 0$ is a reflection.

For the succ case: (succ z) + 0 = succ z, since we have the original equality of $z + 0 = z$, it is enough to apply succ on both sides of the original equality.

For the pred case: (pred z) + 0 = pred z, similarly to the succ case, it is enough to apply pred on both sides of the original equality. □

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

Code 4.3: Agda proof of addition having a right identity element

4.4 Negation and Subtraction Operations

Before proving the existence of an inverse element, we will have to define subtraction. To do so, we will define the negation operation, since subtraction is definitionally equal to adding the negated element: 'm - n = m + (- n)'.

To define negation, we will once again use our iterator just like in the case of addition.

Our method will be similar to addition as well, we will place the constructors of our element in an inverted matter (converting our equivalence using 'invEquiv', since we are required to give both proofs of equivalence) on the 'zero' base element. So when our number contains a 'pred' we will place a 'succ' on 'zero' instead, and vice-versa, if our number contains a 'succ' we will place a 'pred' on 'zero'. This will have the desired effect of resulting in the inverted original number.

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

infixl 6 _-_
_-_ : ℤₕ → ℤₕ → ℤₕ
m - n = m + (- n)
```

Definitionally, the following hold true for the negation operation in Agda:

```
- zero      ≡ zero
- (succ m) ≡ pred (- m)
- (pred m) ≡ succ (- m)
```

Note that the symmetric versions of these also hold true.

Alternatively, we could once again define negation by pattern matching. In this case however, for the 'sec' constructor we would have to provide 'ret' to satisfy the boundaries, for the 'ret' constructor we would have to provide 'sec' to satisfy its boundaries. Interestingly for the 'coh' constructor, we would have to provide the previously defined 'hoc' to satisfy the boundaries.

```
negate : ℤh → ℤh
negate zero         = zero
negate (succ z)     = pred (negate z)
negate (pred z)     = succ (negate z)
negate (sec z i)    = ret (negate z) i
negate (ret z i)    = sec (negate z) i
negate (coh z i j) = hoc (negate z) i j
```

4.5 Inverse Element

We note that in the case of addition, both left and right inverse element will be the negated number. For proving the existence of left and right inverse elements, we will first prove 2 helper theorems:

Theorem 4. *The succ constructor can be moved around addition: $\forall$ m, n $\in \mathbb{Z}$: m + succ n = succ (m + n)*

Proof. Using our induction property:

For the base case (m = 0): 0 + (succ n) = succ (0 + n) is a reflection.

For the succ case: (succ m) + (succ n) = succ ((succ m) + n), since we have the original equality of m + (succ n) = succ (m + n), it is enough to apply succ on both sides of the original equality, this results in succ (m + succ n) = succ (succ (m + n)), thanks to the fact that succ (m + n) = succ m + n holds definitionally, this case is proven.

For the pred case: (pred m) + (succ n) = succ ((pred m) + n), since we have the original equality, we can apply pred on both sides of it. Unfortunately, this won't be enough, since this way we have the equation of pred (m + succ n) = pred (succ (m + n)), our right-hand sides don't match. We will have to use transitivity to change the right-hand side of the equation. We know that our section constructor states that pred (succ m) = m, applying this with 'm + n' gets rid of the pred (succ (...)) part, resulting in pred (m + succ n) = m + n. Using another transitivity, we will have to somehow put a succ (pred (...)) to this 'm + n'. This is just going to be the symmetric version of our retraction constructor, which states that succ (pred (m))

= m. This results in pred (m + succ n) = succ (pred (m + n)), thanks to the fact that pred (m + n) = pred m + n is definitionally true, this case is also proven. □

```
+-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)))
```

Code 4.4: Agda proof of moving succ around addition

Theorem 5. *The pred constructor can be moved around addition: ∀ m, n ∈ ℤ: m + pred n = pred (m + n)*

Proof. Using our induction property:

For the base case (m = 0): 0 + (pred n) = pred (0 + n) is a reflection.

For the succ case: (succ m) + (pred n) = pred ((succ m) + n), since we have the original equality, we can apply succ on both sides of it. Unfortunately, this won't be enough, since this way we have the equation of succ (m + pred n) = succ (pred (m + n)), our right-hand sides don't match. We will have to use transitivity to change the right-hand side of the equation. We know that our retraction constructor states that succ (pred m) = m, applying this with 'm + n' gets rid of the succ (pred (...)) part, resulting in succ (m + pred n) = m + n. Using another transitivity, we will have to somehow put a pred (succ (...)) to this 'm + n'. This is just going to be the symmetric version of our section constructor, which states that pred (succ (m)) = m. This results in succ (m + pred n) = pred (succ (m + n)), thanks to the fact that succ (m + n) = succ m + n is definitionally true, this case is proven.

For the pred case: (pred m) + (pred n) = pred ((pred m) + n), since we have the original equality of m + (pred n) = pred (m + n), it is enough to apply pred on both sides of the original equality, this results in pred (m + pred n) = pred (pred (m + n)), thanks to the fact that pred (m + n) = pred m + n is definitionally true, this case is also proven. □

```
+-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))
```

Code 4.5: Agda proof of moving pred around addition

Theorem 6. *Left inverse element exists for addition: ∀ z ∈ ℤ, ∃ inv ∈ ℤ: inv + z = 0*

Proof. In Agda, we will rewrite the statement as such: ∀ z ∈ ℤ → (- z) + z = 0, since we know that the inverse element is the negated number.

For the base case (z = 0): (- 0) + 0 = 0 is a reflection.

For the succ case: (- (succ z)) + (succ z) = 0. Definitionally, we know that - (succ m) = pred (- m), we can use this fact, by applying pred to the previously proven +-succ, with '- z' and 'z' parameters. This results in the equation pred ((- z) + succ z) ≡ pred (succ ((- z) + z)). Due to the previous note and the fact that definitionally pred (m + n) = pred m + n, our left hand side is correct. Using transitivity, we can get rid of the pred (succ (...)) part on the right-hand side by using the section constructor, resulting in pred ((- z) + succ z) ≡ (- z) + z. Using another transivitiy, our remaining right-hand side is just our original equation. Applying it, this case is proven.

For the pred case: (- (pred z)) + (pred z) = 0. Definitionally, we know that - (pred m) = succ (- m), we can use this fact, by applying pred to the previously proven +-pred, with '- z' and 'z' parameters. This results in the equation succ ((- z) + pred z) ≡ succ (pred ((- z) + z)). Due to the previous note and the fact that definitionally succ (m + n) = succ m + n, our left hand side is correct. Using transitivity, we can get rid of the succ (pred (...)) part on the right-hand side by using the retraction constructor, resulting in succ ((- z) + pred z) ≡ (- z) + z. Using another transivitiy, our remaining right-hand side is just our original equation. Applying it, this case is also proven. □

```
+-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)
```

Code 4.6: Agda proof of addition having a left inverse element

Theorem 7. *Right inverse element exists for addition: ∀ z ∈ ℤ, ∃ inv ∈ ℤ: inv + z = 0*

Proof. As in the case of the left identity element, we will rewrite this statement in Agda as such: ∀ z ∈ ℤ → z + (- z) = 0.

For the base case (z = 0): 0 + (- 0) = 0 is once again a reflection.

For the succ case: (succ z) + (- (succ z)) = 0. Definitionally, we know that - (succ m) = pred (- m), we can use this fact, by applying succ to the previously proven +-pred, with 'z' and '- z' parameters. This results in the equation succ (z + pred (- z)) ≡ succ (pred (z + (- z))). Due to the previous note and the fact that definitionally succ (m + n) = succ m + n, our left hand side is correct. Using transitivity, we can get rid of the succ (pred (...)) part on the right-hand side by using the retraction constructor, resulting in succ (z + pred (- z)) ≡ z + (- z). Using another transivitiy, our remaining right-hand side is just our original equation. Applying it, this case is proven.

For the pred case: (pred z) + (- (pred z)) = 0. Definitionally, we know that - (pred m) = succ (- m), we can use this fact, by applying pred to the previously proven +-succ, with 'z' and '- z' parameters. This results in the equation pred (z + succ (- z)) ≡ pred (succ (z + (- z))). Due to the previous note and the fact that definitionally pred (m + n) = pred m + n, our left hand side is correct. Using transitivity, we can get rid of the pred (succ (...)) part on the right-hand side by using the section constructor, resulting in pred (z + succ (- z)) ≡ z + (- z). Using another transivitiy, our remaining right-hand side is just our original equation. Applying it, this case is also proven. □

```
+-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)
```

Code 4.7: Agda proof of addition having a right inverse element

4.6 Commutativity

Theorem 8. *Addition is commutative:* $\forall\ m,\ n \in \mathbb{Z}$*:* $m + n = n + m$

Proof. Using our induction property:

For the base case (m = 0): 0 + n = n + 0, while this looks really similar to the left identity element existing, in Agda only 0 + n = n is true definitionally, n + 0 = n isn't. We will instead use the symmetric version of the right identity element existing with 'n' parameter. (Without symmetry this property will state that n + 0 = n, with symmetry, we will change the two sides: n = n + 0, as previously stated, in Agda 0 + n = n is a reflection.)

For the succ case: (succ m) + n = n + (succ m). Our left-hand side looks really similar definitionally to the right-hand side of the +-succ property. Using the symmetric version of the +-succ property with 'm' and 'n' parameters results in succ (m + n) = m + succ n. As previously mentioned, the left-hand side is true definitionally. Using transitivity, we can change the right-hand side by applying 'succ n' to our original equation, resulting in succ (m + n) = succ n + m. Thanks to the same definitionally true statement, we can use transitivity, with, once again, the symmetric version of +-succ with 'n' and 'm' supplied, to change succ n + m to n + succ m. (succ n + m is the same succ (n + m), which is on the right-hand side of our +-succ property, applying the symmetric version of +-succ with 'n' and 'm' parameters, we get n + succ m.)

For the pred case: (pred m) + n = n + (pred m). Our left-hand side looks really similar definitionally to the right-hand side of the +-pred property. Using the symmetric version of the +-pred property with 'm' and 'n' parameters results in pred (m + n) = m + pred n. As previously mentioned, the left-hand side is true definitionally. Using transitivity, we can change the right-hand side by applying 'pred n' to our original equation, resulting in pred (m + n) = pred n + m. Thanks to the same definitionally true statement, we can use transitivity, with, once again, the symmetric version of +-pred with 'n' and 'm' supplied, to change pred n + m to n + pred m. (The same chain of thought is true as in the 'succ' case, with pred instead of succ.) □

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

Code 4.8: Agda proof of addition being commutative

Chapter 5

Monoid (Multiplication)

5.1 Multiplication Operation

To define the multiplication operation, we will once again use our iterator. In this case however, we will have to do a bit of extra work. We cannot provide an Isomorphism in such an easy way as we did with addition. From elementary grade mathmatics, we know that multiplication is just repeated addition, for example if we want to calculate '3 * 4', we would do '4 + 4 + 4', or adding together '4' (right element) '3' (left element) times. So our main idea is to start with zero, and repeat the addition of the right parameter exactly the number of left parameter times (if the left parameter is negative, we will just add the inverted right parameter the absolute value of left parameter times).

As before, our function of having this desired effect will be defining the 'function' of isomorphism as 'z +', conversely, we can define the 'inverse function' as adding the negated value of this 'z': '- z +' since this will invert the effect of adding z to a number.

Next, we will need to prove the 'retraction' property of this.

Theorem 9. *Adding a negated value to a number, then the non-negated value of the same value, we get back the original number: $\forall$ m, n $\in \mathbb{Z}$: n + ((- n) + m) = m*

Proof. To prove this, we will first use the associative property of addition to match the left-hand side. Supplying 'n', '- n' and 'm' results in the following equation: n + ((- n) + m) $\equiv$ n + (- n) + m. We use transitivity to change the right-hand side of the equation. Getting under the '+ m' part, we can apply the existence of the right inverse element of addition with 'n' parameter. This will replace 'n + (- n)' with 0:

n + ((- n) + m) ≡ 0 + m. We know that definitionally 0 + m = m, this property is proven. □

Afterwards, we will have to prove the 'section' property of this.

Theorem 10. *Adding a non-negated value to a number, then the negated value of the same value, we get back the original number: ∀ m, n ∈ ℤ: (- n) + (n + m) = m*

Proof. Similarly to the previous proof, to prove this, we will first use the associative property of addition to match the left-hand side. Supplying '- n', 'n' and 'm' results in the following equation: (- n) + (n + m) ≡ (- n) + n + m. We use transitivity to change the right-hand side of the equation. Getting under the '+ m' part, we can apply the existence of the left inverse element of addition with 'n' parameter. This will replace '(- n) + n' with 0: (- n) + (n + m) ≡ 0 + m. We know that definitionally 0 + m = m, this property is also proven. □

Next, we can convert this isomorphism to the proof that 'z +' is an equivalence ('isEquiv'), since we will need to supply parameters this time (instead of placing the constructors), we cannot directly convert to the equivalence ('Equiv'). Afterwards we will define our ordered pair of the equivalence, with a parameter this time.

Lastly, when defining our operation, we will use our previously mentioned idea of placing 'n +' (or '- n +') 'm' times on zero.

```
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)

Equiv-n+-ℤₕ : (z : ℤₕ) → ℤₕ ≃ ℤₕ
Equiv-n+-ℤₕ z = z +_ , isEquiv-n+-ℤₕ z
```

```

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

Definitionally, the following hold true for the multiplication operation in Agda:

```
zero   * n ≡ zero
succ m * n ≡ n + m * n
pred m * n ≡ (- n) + m * n
```

Note that the symmetric version of these also holds true.

We could once again try to define multiplication by pattern matching. In this case however, even defining 'sec' is tricky.

```
_*_ : ℤₕ → ℤₕ → ℤₕ
zero      * b = zero
succ a    * b = a * b + b
pred a    * b = a * b - b
sec a i   * b = ?
ret a i   * b = ?
coh a i j * b = ?
```

5.2 Identity Element

Theorem 11. *Left identity element exists for multiplication: ∃ id ∈ ℤ, ∀ z ∈ ℤ: id * z = z*

Proof. In Agda, we will rewrite the statement as such: ∀ z ∈ ℤ → 1 * z = z, since we know that the identity element is 1.
1 * z = z + 0, so we will reuse our theorem of proving that a right identity element exists for addition. □

```
*-idˡ : ∀ z → succ zero * z ≡ z
*-idˡ = +-idʳ
```

Code 5.1: Agda proof of multiplication having a left identity element

Theorem 12. *Right identity element exists for multiplication: ∃ id ∈ ℤ, ∀ z ∈ ℤ: z * id = z*

Proof. As in the case of the left identity element, we will rewrite this statement in Agda as such: ∀ z ∈ ℤ → z * 1 = z, since our right identity element is also going to be 1.

For this, we won't use our induction property. Instead, we will rewrite the left-hand side of the equation to 1 * z, using the commutative property of multiplication (which we prove later on). Now we will need to prove that 1 * z = z, which is the proof of the left identity element existing. □

```
*-idʳ : ∀ z → z * succ zero ≡ z
*-idʳ z = *-comm z (succ zero) • *-idˡ z
```

Code 5.2: Agda proof of multiplication having a right identity element

5.3 Multiplication Distributes over Addition

As was the case for the identity element and inverse element proofs, we will have to prove both the left and right side for distributivity.

Theorem 13. *Multiplication is distributive over addition on the left side: ∀ m, n, o ∈ ℤ: (m + n) * o = (m * o) + (n * o)*

Proof. Using our induction property:

For the base case (m = 0) : (0 + n) * o = (0 * o) + (n * o) is a reflection.

For the succ case: ((succ m) + n) * o = ((succ m) * o) + (n * o). Definitionally, we know that (succ m) * o = o + m * o, as such, our equation is essentially the same as: o + (m * n) * o = o + (m * o) + (n * o). For the left-hand side of the equation, we will note that disregarding the beginning o + part, the equation is the same as the original equality's left-hand side, using congruence, we can substitute on the left-hand side. Using this, our right-hand side will look like this: o + (m * o + n * o), this is almost the exact thing that is on the right-hand side, apart from the fact that addition is paranthesized from the left (our original o + (m * n) + (n * o) = (o + (m * n)) + (n * o)), using the associative property of addition, we can rearrange the parts to be correctly parenthesized.

For the pred case: ((pred m) + n) * o = ((pred m) * o) + (n * o). Once again, definitionally we know that (pred m) * o = (- o) + m * o. We can rewrite the equation once again as: - o + (m * n) * o = - o + (m * o) + (n * o). We can use the exact same steps as in the succ case (Using transitivity, rewriting the left-hand side using congruence and the original equality, rewriting the right hand side with the associative property of addition), noting the difference that for congruence and the first part of the associative property of addition, we will have to provide '- o' as a parameter instead of 'o'. □

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

Code 5.3: Agda proof of multiplication being left distributive to addition

Theorem 14. *Multiplication is distributive over addition on the right side: ∀ m, n, o ∈ ℤ: m * (n + o) = (m * n) + (m * o)*

Proof. Using the commutative property of multiplication, we can rewrite m * (n + o) to (n + o) * m, this equation looks the same as the left-hand side our previous theorem (multiplication is distributive over addition on the left side). Applying it, with the correct parameters (n, o, m as 'm', 'n' and 'o' respectively), we get the equation of n * m + o * m. These are the exact parts that we need, we will just need to use the commutative property of multiplication twice to switch around the terms. (Note: we can use the congruences in any way, i.e. it doesn't matter if the rearrange the left or the right part first.) □

```
*-distribʳ-+ : ∀ m n o → m * (n + o) ≡ (m * n) + (m * o)
*-distribʳ-+ m n o = *-comm m (n + o) ∙ *-distribˡ-+ n o m ∙ cong (n
↪  * m +_) (*-comm o m) ∙ cong (_+ m * o) (*-comm n m)
```

Code 5.4: Agda proof of multiplication being right distributive to addition

5.4 Associative

Before proving that multiplication is associative, we will prove 3 helper theorems.

Theorem 15. *Negating an addition is equal to negating the parts separately, and then adding them together.* $\forall$ *m, n* $\in \mathbb{Z}$: *- (m + n) = (- m) + (- n)*

Proof. Using our induction property:

For the base case (m = 0): - (0 + n) = (- 0) + (- n) is a reflection.

For the succ case: - ((succ m) + n) = (- (succ m)) + (- n). Note that we have our original equality of - (m + n) = (- m) + (- n). Definitonally - (succ m) = pred (- m), since it doesn't matter if we increase the value, then negate it, or if we negate it and decrement it. Rewriting our case this way results in: pred (- (m + n)) = (pred (- m)) + (-n), that we need to prove. This way, applying pred to both sides of the original equality, we prove this case.

For the pred case: - ((pred m) + n) = (- (pred m)) + (- n). Similarly to the succ case, we have our original equality. Definitionally - (pred m) = succ (- m). Rewriting our case this way results in: succ (- (m + n)) = (succ (- m)) + (- n), that we need to prove. So, similarly to the succ case, we can apply succ to both sides of the original equality, proving this case as well. □

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

Theorem 16. *Multiplying a number ('m') by a right negated number ('- n') is the same as multiplying the non-negated numbers first ('m * n'), and negating the result.* $\forall$ *m, n* $\in \mathbb{Z}$: *m * (- n) = - (m * n)*

Proof. Using our induction property:

For the base case (m = 0): 0 * (- n) = - (0 * n) is a reflection.

For the succ case: (succ m) * (- n) = - ((succ m) * n). Definitionally succ m * (- n) = (- n) + m * (- n). The m * (- n) part is the left-hand side of our original equality, if we use congruence to get under the '(- n)' beginning part, we can re-use our original equality with 'n' parameter being supplied. This results in the equality (- n) + m * (- n) = (- n) + (- (m * n)). Afterwards, using transitivity, to change the right-hand side, we will also note, since we proved inv-hom, that we can rewrite '(- n) + (- (m * n))' as '- (n + m * n)', if we use the symmetric version of our previously proven inv-hom property and supply 'n' to the m parameter, and 'm * n' to the n parameter. This will result in the equation (- n) + m * (- n) = (- (n + m * n)). We will note, once again, that definitionally succ m * n = n + m * n, so the right-hand side is also what we need to prove.

For the pred case: (pred m) * (- n) = - ((pred m) * n). Similarly to the 'succ' case, we note that definitionally pred m * (- n) = (- (- n)) + m * (- n) and pred m * n = (- n) + m * n hold true. In a similar chain of thought, we can use congruence to get under the '(- (- n))' part, we can re-use our original equality with 'n' parameter supplied to it. Afterwards we can use transitivity, and the previously proven inv-hom, specifically the symmetric version of it, with '- n' supplied to the m parameter, and 'm * n' supplied to the n parameter to rewrite our right-hand side. This results in the equation (- (- n)) + m * (- n) = (- ((- n) + m * n)), where both left- and right-hand sides are definitionally equal to the equation we need to prove. □

```
*-inv : ∀ m n → m * (- n) ≡ - (m * n)
*-inv = ℤₕ-ind-prop
  (λ _ → isPropΠ λ _ → isSetℤₕ _ _)
  (λ n → refl)
  (λ m p n → cong (- n +_) (p n) ∙ sym (inv-hom-ℤₕ n (m * n)))
  (λ m p n → cong (- (- n) +_) (p n) ∙ sym (inv-hom-ℤₕ (- n) (m *
↪  n)))
```

Theorem 17. *Similarly, multiplying a left negated number ('- m') by a number ('n') is the same as multiplying the non-negated numbers first ('m * n'), and negating the result: ∀ m, n ∈ ℤ: (- m) * n = - (m * n).*

Proof. Using the commutative property of multiplication, we can rewrite the left-hand side of our equation: (- m) * n = n * (- m). Using transitivity, we can continue changing the right-hand side of the equation. Our previously proved multiplication-negation relation has the same left-hand side that we need to change, applying it (with the 'm' parameter being 'n' and the 'n' parameter being 'm') we get the following equation: (- m) * n = - (n * m). Once again, we need to use transtivity to keep changing the right-hand side of the equation. If we use the commutative property of multiplication under the negation operation, we can rewrite the right-hand side of the equation to match the equation we need to prove. □

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

Theorem 18. *Multiplication is associative: ∀ m, n, o ∈ ℤ: m * (n * o) = (m * n) * o*

Proof. Using our induction property:

For the base case (m = 0): 0 * (n * o) = (0 * n) * o is a reflection.

For the succ case: (succ m) * (n * o) = ((succ m) * n) * o. Definitionally, we know that (succ m) * n = n + m * n, With this, we can rewrite our equation to: (n * o) + m * (n * o) = (n + m * n) * o. Note, that on the left-hand side, apart from the '(n * o)' part, we have our original equality's left-hand side. If we use congruence to get under this '(n * o)' part, we can apply the original equality (with 'n' and 'o' parameters supplied respectively). This changes our equation to: (n * o) + m * (n * o) = (n * o) + (m * n) * o, using transitivity, we can change the right-hand side. We note that that if we use the symmetric version of multiplication distributing over addition on the left hand side, with 'n', 'm * n' and 'o' supplied, we have the property of: (n * o) + ((m * n) * o) = (n + (m * n)) * o. The order of parentheses are correct, so this case is proven.

For the pred case: (pred m) * (n * o) = ((pred m) * n) * o. Once again, we know definitionally that (pred m) * n = (- n) + m * n. Using this, we can rewrite the equation to: - (n * o) + m * (n * o) = ((- n) + m * n) * o. Once again, on the left-hand side, apart from the '- (n * o)' part, we have our original equality's left-hand side. We can use congruence once again to get under this part, and apply the original equality (with the same 'n' and 'o' parameters). Using this, our equation becomes: - (n * o) + m * (n * o) = - (n * o) + (m * n) * o. We will need to use transtivity once again to change the right-hand side of the equation. Unlike last time, we cannot use the left distributive property of multiplication over addition yet, since we cannot match the parameters correctly just yet. If we first instead use our previously proven 'inv-*' property (before first getting under the '+ m * n * o' part - note that since multiplication is parenthesized from the left, (m * n) * o and m * n * o are equal), specifically the symmetric version of it (with 'n' and 'o' parameters supplied), we can change our equation to: - (n * o) + m * (n * o) = (- n) * o + (m * n) * o. Using another transitivity, we can now use the left distributive property of multiplication over addition (again, the symmetric version of it), with '(- n)', 'm * n' and 'o' parameters supplied. With this, our equation becomes: - (n * o) + m * (n * o) = ((- n) + (m * n)) * o. Once again, thanks to the order of parentheses is correct, this case is also proven. □

```
*-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 (n * o +_) (p n o) ∙ sym (*-distribˡ-+ n (m * n) o))
  (λ m p n o → cong (- (n * o) +_) (p n o) ∙ cong (_+ m * n * o) (sym (inv-* n o)) ∙ sym (*-distribˡ-+ (- n) (m * n) o))
```

Code 5.5: Agda proof of multiplication being associative

Chapter 6

Commutative Ring

6.1 Multiplication is Commutative

To prove that multiplication is commutative, we will need to first prove 3 helper theorems:

Theorem 19. *Multiplying any number from the right by 0 results in 0.* $\forall$ *z* $\in \mathbb{Z}$*: z * 0 = 0*

Proof. While 0 * z = 0 is true in Agda, we have to prove z * 0 = 0, even if it turns out to be just a formality to do so. Using our induction property:

For the base case (n = 0): 0 * 0 = 0 is a reflection, we don't even need to write a lambda expression.

For the succ case: (succ z) * 0 = 0. It is sufficient in Agda to just supply our original equality (z * 0 = 0), as both sides are reduced to 0.

For the pred case: (pred z) * 0 = 0. Similarly, it is sufficient to supply our original equality (z * 0 = 0), to prove this. □

```
*-zero : ∀ z → z * zero ≡ zero
*-zero = ℤₕ-ind-prop
  (λ _ → isSetℤₕ _ _)
  refl
  (λ z p → p)
  (λ z p → p)
```

Code 6.1: Agda proof of multiplication by 0 from the right is 0

Theorem 20. *The succ constructor can be destructed by multiplication:* $\forall$ *m, n* $\in$ $\mathbb{Z}$*: m * succ n = m + m * n*

Proof. Using our induction property:

For the base case (m = 0): 0 * (succ n) = 0 + 0 * n is a reflection.

For the succ case: (succ m) * (succ n) = (succ m) + ((succ m) * n). Definitionally, we know that succ m * n ≡ n + m * n, this way we can rewrite our equation as such: succ n + m * succ n = succ m + (n + m * n). We also know definitionally that succ m + n = succ (m + n), so we can once again rewrite our equation as such: succ (n + m * succ n) = succ (m + (n + m * n)). We can note that using congruence to get under succ on both sides, the equation we need to prove is going to be n + m * succ n = m + (n + m * n). Using congruence to get under the 'n +' part, we can apply our original equality with 'n' parameter, this will result in n + m * succ n = n + (m + m * n). Note that our left-hand side is already the exact same that we need, also note that by parts, our right-hand side is essentially the same as well. Using transitivity, we further change our right-hand side, using the associative property of addition with 'n' being the m parameter, 'm' being the n parameter and 'm * n' being the o parameter. This will change the equation to n + m * succ n = (n + m) + m * n (Note: In agda, the parenthesis on the right-hand side won't show up.) The 'm * n' part is already at its correct position, using another transitivity to change the right-hand side and getting under the 'm * n' part by using congruence, we can use the commutative property of addition (with 'n' being the m parameter, and 'm' being the n parameter) to rewrite this equation to: n + m * succ n = (m + n) + m * n. Further using another transitivity, we can use the symmetric version of the associative property of addition (with 'm' being the m parameter, 'n' being the n parameter and 'm * n' being the o parameter) we can rewrite the right-hand side of (m + n) + m * n to m + (n + m * n). This is the exact right-hand side we need, our case is proven.

For the pred case: (pred m) * (succ n) = (pred m) + ((pred m) * n). Similarly to the 'succ' case: definitionally, we know that pred m * n ≡ (- n) + m * n, this way we can rewrite our equation as such: (- (succ n)) + m * succ n = pred m + ((- n) + m * n). Furthermore, we know that - (succ m) = pred (- m), so we can rewrite the left-hand side to: pred (- n) + m * succ n. We also know definitionally that pred m + n = pred (m + n), so our whole equation can be rewritten again as such: pred ((- n) + m * succ n) = pred (m + ((- n) + m * n)). We can note that using congruence to get under pred on both sides, the equation we need to prove is going to be (- n) + m * succ n = m + ((- n) + m * n). Using congruence to get under the '(- n) +' part, we can apply our original equality with 'n' parameter, this will result in (- n) + m * succ n = (- n) + (m + m * n). Note that our left-hand side is already the exact same that we need, also note that by parts, our right-hand side is essentially the same as well. Using transitivity, we further change our right-hand side, using the associative property of addition with '- n' being the m parameter, 'm' being the n parameter and 'm * n' being the o parameter. This will change the equation to (- n) + m * succ n = ((- n) + m) + m * n (Note: Once again, in agda, the parenthesis on the right-hand side won't show up.) The 'm * n' part is already at its correct position, using another transitivity to change the right-hand side and getting under the '+ m * n' part by using congruence, we can use the commutative property of addition (with '- n' being the m parameter, and 'm' being the n parameter) to rewrite this equation to: (- n) + m * succ n = (m + (- n)) + m * n. Further using another transitivity, we can use the symmetric version of the associative property of addition (with 'm' being the m parameter, '- n' being the n parameter and 'm * n' being the o parameter) we can rewrite the right-hand side of (m + (- n)) + m * n to m + ((- n) + m * n). This is once again the exact right-hand side we need, this case is also proven. □

```
*-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))))
```

Code 6.2: Agda proof of succ being destructed by multiplication

Theorem 21. *The pred constructor can be destructed by multiplication:* $\forall$ *m, n* $\in \mathbb{Z}$*: m * pred n = (- m) + m * n*

Proof. Using our induction property:

For the base case (m = 0): 0 * (pred n) = (- 0) + 0 * n is a reflection.

For the succ case: (succ m) * (pred n) = - (succ m) + ((succ m) * n). Similarly to the previously proven *-succ: we know definitionally the following: 1) succ m * n = n + m * n, 2) - (succ m) = pred (- m), 3) pred m + n = pred (m + n). Using 1) we can rewrite the equation to: (pred n) + m * (pred n) = - (succ m) + (n + m * n). Using 2) we can rewrite the right-hand side to: (pred n) + m * (pred n) = pred (- m) + (n + m * n) and using 3) we can rewrite the whole equation to: pred (n + m * (pred n)) = pred ((- m) + (n + m * n)). We can use congruence once again to get under the pred part, so the equation we need to prove becomes n + m * (pred n) = (- m) + (n + m * n). Once again, getting under the 'n +' part on the left-hand side, we can use the original equality with 'n' parameter, this will result in n + m * (pred n) = n + ((- m) + m * n). The left-hand side is once again the exact same as the one that we need to prove. Our right-hand side can be changed by using transitivity. First, we will change it using the associative property of addition, by passing 'n' as the m parameter, '- m' as the n parameter, and 'm * n' as the o parameter, this will change the right-hand side to: (n + (- m)) + m * n (Note: Once again, in agda, the parenthesis won't show up.). Further using transitivity, we can get under the '+ m * n' part, where using the commutative property of addition, with 'n' as the m parameter and '- m' as the n parameter, we can change this side to: ((- m) + n) + m * n. Once again, using the symmetric version of the associative property of addition, with '- m' as the m parameter, 'n' as the n parameter, and 'm * n' as the o parameter, we change the parenthesis of our right-hand side to: (- m) + (n + m * n). This is the exact same as our simplified right-hand side, this case is proven.

For the pred case: (pred m) * (pred n) = - (pred m) + ((pred m) * n). Similarly to the 'succ' case: we know definitionally the following: 1) pred m * n = (- n) + m * n, 2) - (pred m) = succ (- m), 3) succ m + n = succ (m + n). Using 1) we can rewrite the equation to: - (pred n) + m * (pred n) = - (pred m) + ((- n) + m * n). Using 2) we can rewrite the equation to: succ (- n) + m * (pred n) = succ (- m) + ((- n) + m * n). Using 3) we can rewrite the equation to: succ ((- n) + m * (pred n)) = succ ((- m) + ((- n) + m * n)). We can use congruence once again to get under the succ part, so the equation we need to prove becomes (- n) + m * (pred n) = (- m) + ((- n) + m * n). Once again, getting under the '(- n) +' part on the left-hand side, we can use the original equality with 'n' parameter, this will result in (- n) + m * (pred n) = (- n) + ((- m) + m * n). The left-hand side is once again the exact same as the one that we need to prove. Our right-hand side can be changed by using transitivity. First, we will change it using the associative property of addition, by passing '- n' as the m parameter, '- m' as the n parameter, and 'm * n' as the o parameter, this will change the right-hand side to: ((- n) + (- m)) + m * n (Note: As in the previous 3 cases, in agda, the parenthesis won't show up.). Further using transitivity, we can get under the '+ m * n' part, where using the commutative property of addition, with '- n' as the m parameter and '- m' as the n parameter, we can change this side to: ((- m) + (- n)) + m * n. Once again, using the symmetric version of the associative property of addition, with '- m' as the m parameter, '- n' as the n parameter, and 'm * n' as the o parameter, we change the parenthesis of our right-hand side to: (- m) + ((- n) + m * n). This is the exact same as our simplified right-hand side, this case is also proven. □

```
*-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))))
```

Code 6.3: Agda proof of pred being destructed by multiplication

Theorem 22. *Multiplication is commutative:* $\forall$ *m, n* $\in \mathbb{Z}$*: m * n = n * m*

Proof. Using our induction property:

For the base case (m = 0): 0 * n = n * 0. We note that the left side of the equation is reduced to 0. Therefore, we really need to prove the following: 0 = n * 0. Previously, we proved that n * 0 = 0, we can once again use this property (providing 'n' as a parameter), but taking the symmetric version of it.

For the succ case: (succ m) * n = n * (succ m). Definitionnaly, we know that (succ m) * n = n + m * n. Rewriting the equation as such we get: n + m * n = n * (succ m). We also know (by proving that multiplication destructs the succ constructor) that n * (succ m) = n + n * m. If we use congruence to get under the additional addition of 'n +', we can provide the original equality to get the same left-hand side. With this we have: n + m * n = n + n * m. We need to use transitivity to further deal with the right-hand side. By using the symmetric version of the previously mentioned property (multiplication destructing succ), we can rewrite the right-hand side to the needed n * (succ m).

For the pred case: (pred m) * n = n * (pred m). Similarly to the succ case, we know that (pred m) * n = (- n) + (m * n), we also know (from proving that multiplication destructs the pred constructor) that n * (pred m) = (- n) + n * m. With these two notes, we can apply a similar line of thought to the equation, by getting under the addition of '(- n) +', using the original equality to get our desired result, then rewriting the right-hand side by using the symmetrics version of our previously proven property of multiplication destructing pred. □

```
*-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))
```

Code 6.4: Agda proof of multiplication being commutative

6.2 Putting it All Together

For all of the proven cases (Abelian Group, Monoid, Ring, Commutative Ring), Cubical Agda has data types. When defining the types of these, we also have to give the identity elements, the operation it is applicable on (negation is implicitly required, thanks to the inverse element's proof) and which type we are defining these on. Defining the data type itself requires the previously proven properties, as well as the set property as fields. As this type checks, we can be sure that our higher inductive type integers do, indeed, form a commutative ring. Interestingly enough, there exists a 'makeIsCommRing' function, looking at it, we could've just proven the 'right' version of all properties (addition having an identity and inverse element, multiplication having an id element, and being right distributive to addition) as it is possible to infer the 'left' version of these properties.

```
AbGroupℤₕ+ : IsAbGroup {A = 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 {A = 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 {R = ℤₕ} zero (succ zero) _+_ _*_ (-_)
Ringℤₕ*+ .IsRing.+IsAbGroup = AbGroupℤₕ+
Ringℤₕ*+ .IsRing.·IsMonoid = Monoidℤₕ*
Ringℤₕ*+ .IsRing.·DistR+ = *-distribʳ-+
Ringℤₕ*+ .IsRing.·DistL+ = *-distribˡ-+

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

Code 6.5: Definition of the Commutative Ring property for HIT Integers

With this defined, we can use this with the existing commutative ring solver tactic defined in the cubical library. What this essentially means is that any equation using only multiplication, addition and negation can be trivially solved. See the examples source file for an example of this.

6.3 Comparing Standard Integers and HIT Integers

When looking at the proof of the standard integers's commutative property of multiplication, the proof is 23 lines long (not including helper theorem proofs). Looking at the HIT integers's same proof, it is only 5 lines long (similarly not including any helper theorem proofs, but even if we include those, it is still less than 23 lines long). From this, we can confirm that the statement by Altenkirch & Scoccola about standard integers being inconvenient as they cause a lot of unnecessary case distinctions is even more apparent when compared with our HIT integer definition and its proofs.

As it was noted during the proof that HIT integers form a set, since we went the way of proving isomorphism with standard integers, we inadvertently compared our HIT integers and the standard integers. Unsurprisingly one expects all integer definitions to be isomoprhic, obviously they are going to be differences between definitions, each with their own pro and con (as mentioned in the Introduction chapter), in the end however, they should all be able to hold values which we associate with integer.

Chapter 7

Further Work

Unfortunately not everything described in the topic announcer has been achieved, in this chapter we will discuss these.

7.1 Proving the Coherent case

While significant work has been done to prove that converting a 'coh' HIT integer to the standard type and back to HIT is the exact same value, this case isn't done. Intuitively just like for the 1-dimensional constructors 'sec' and 'ret' we were able to prove this equality, this should hold true for the 2-dimensional constructor of 'coh' as well.

The problem lies in the fact that the boundaries required by a 2-dimensional constructor create a 3-dimensional boundary cube for us. To prove this, we would have to draw a 4-dimensional cube (tesseract), for this, we would have to draw the 8 cells (3D cubes) that make up this tesseract to even have a chance at satisfying these boundaries. From any point on the wanted boundary cube, there is a given proof to move along the 4th dimension and back to another point on the wanted boundary cube.

$$\begin{tikzcd}
& H \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash, dashed] & & G \arrow[dl, dash] \arrow[dd, dash] & & \\
D \arrow[rr, dash] \arrow[dd, dash] & & C \arrow[dd, dash] & & & \\
& E \arrow[rr, dash, dashed] \arrow[dl, dash, dashed] & & F \arrow[dl, dash] & j & i \\
A \arrow[rr, dash] & & B & & \bullet \arrow[u] \arrow[ur] \arrow[r] & k
\end{tikzcd}$$

Where, interestingly enough, 4-4 pair of point are equal as such:

- A = E = ℤ→ℤ$_h$ (sucℤ (predℤ (sucℤ (ℤ$_h$→ℤ z))))
- B = F = succ (pred (succ z))
- C = G = succ z
- D = H = ℤ→ℤ$_h$ (sucℤ (ℤ$_h$→ℤ z))

While one could say that this reduces our cube to a square, unfortunately the situation is different, as we also have the face boundaries required for our proof (here we unrolled the cube to its 6 squares to make reading easier):

$$\begin{tikzcd}
& D \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "k_0"] & H \arrow[d, dash] & \\
D \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "i_0"] & A \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "j_0"] & E \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "i_1"] & H \arrow[d, dash] \\
C \arrow[r, dash] & B \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "k_1"] & F \arrow[r, dash] \arrow[d, dash] & G \\
& C \arrow[r, dash] \arrow[d, dash] \arrow[dr, phantom, "j_1"] & G \arrow[d, dash] & \\
& D \arrow[r, dash] & Hy &
\end{tikzcd}$$

Here, our required boundaries will be:

- $k_0 =$ `ℤ→ℤₕ (cohℤ (ℤₕ→ℤ z) i j)`
- $k_1 =$ `coh z i j`
- $j_0 =$ `ℤₕ→ℤ→ℤₕ (coh z i i0) k`
- $j_1 =$

```
        hcomp
        (doubleComp-faces (λ _ → ℤ→ℤₕ (sucℤ (ℤₕ→ℤ z)))
         (λ i₁ → succ (ℤₕ→ℤ→ℤₕ z i₁)) k)
        (ℤ→ℤₕ-sucℤ (ℤₕ→ℤ z) k)
```

- $i_0 =$ `ℤₕ→ℤ→ℤₕ (coh z i0 j) k`
- $i_1 =$ `ℤₕ→ℤ→ℤₕ (coh z i1 j) k`

We have to find out what the 7 other cubes required for the 4D tessaract are. After this, if we correctly define the the helper function's type (for which we will more than likely need additional helper functions as well, for example to prove that converting a standard integer that has 'sucℤ' applied, while already having 'predℤ' and 'sucℤ' applied is the same as converting a standard integer, then applying 'succ', 'pred' and 'succ' - the 'succ (pred (succ z))' part of the HIT Integer's 'coh' rule) we still aren't finished, as we need to pattern match on this value to successfully define the function. As seen from the 'predSuc' and 'sucPred' helper functions, this will more than likely mean 3 (or even 4) cases, where we will need to fill in 7 values each. Proving the discrete property might've been better after all.

7.2 Relations

With an equivalence relation defined (which is default for all types), we can, for example, define the injection property for the 'succ' and 'pred' constructors as such:

```
succ-inj : ∀ m n → succ m ≡ succ n → m ≡ n
succ-inj m n eq = sym (sec m) ∙ congS pred eq ∙ sec n

pred-inj : ∀ m n → pred m ≡ pred n → m ≡ n
pred-inj m n eq = sym (ret m) ∙ congS succ eq ∙ ret n
```

7.3 Division (and Modulus)

It is possible to write a division operator for integers. One might ask how that is possible, since, for example, 3 / 2 results in something that is not an integer. The answer lies in rounding down (or up) the result. Unfortunately, due to this, we cannot use our iterator to define this operator, as it is non-reversible, we need to pattern match. This brings its own problems, as for our 1- and 2-dimensional constructors ('sec', 'ret' and 'coh') it turns out to be a laborous task. There is also the problem of dividing by zero, for which the standard Agda library introduces an interesting solution. Introducing a 'NonZero' record, which can ensure that the divisor is not zero. For the operation itself, we introduce a helper function, which will handle the cases. A simple 'succ' case division is introduced. (As it is seen in the Agda standard library[10]) These same principles also apply for the modulus operator:

```
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
```

7.4 Exponentiation

Exponentiation is an interesting topic when talking about integers. First of all, our type definition seems out of place, as our second parameter is not an integer, but a natural number. From elemenary school, we know that raising an integer to a negative power is equivalent to 1 divided by the integer to the absolute value of the power. This number is usually not an integer, so we need to strictly ensure that our exponent is a non-negative number. This restriction (along with our non-exhaustive division operation resulting in us not able to give a correct reversible operation to multiplication) results in us not being able to use our iterator to define this operation, we need to pattern match once again. As usual, defining the results of our 1- and 2-dimensional constructors ('sec', 'ret' and 'coh') is quite laborous. For this reason, we will only define the 0-dimensional constructors and our base element:

```
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)
```

With this operation defined, we could prove properties of exponentiation, for example: products of powers: $a^m \cdot a^n = a^{m+n}$, power of a power: $(a^m)^n = a^{m^n}$, power of a product: $(a{\cdot}b)^m = a^m{\cdot}b^m$. Other properties of exponentiation require division, which, as we described, is non reversible. Unfortunately, since our second parameter is not an integer, we cannot use our induction property to prove the before-mentioned properties, we would need to pattern match, which, once again, would introduce the need to prove the cases for our 1- and 2-dimensional constructors, for which we haven't defined what the exponentiation operation is, so it would be an impossible task.

Acknowledgements

I want to thank the Budapest Type Theory Group, especially Viktor Bense, Szumi Xie, Márton Petes, and my supervisor, Ambrus Kaposi, for helping me by answering any of my questions regarding my thesis work and for making it possible for me to finish it. I would also like to thank my mother, who has supported me through my university years.

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.

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

[5] Cyril Cohen et al. *Cubical Type Theory: a constructive interpretation of the univalence axiom*. 2016. arXiv: `1611.02108 [cs.LO]`. URL: `https://arxiv.org/abs/1611.02108`.

[6] Thierry Coquand, Simon Huber, and Anders Mörtberg. *On Higher Inductive Types in Cubical Type Theory*. 2018. arXiv: `1802.01170 [cs.LO]`. URL: `https://arxiv.org/abs/1802.01170`.

[7] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical agda: a dependently typed programming language with univalence and higher inductive types". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: `10.1145/3341691`. URL: `https://doi.org/10.1145/3341691`.

[8] Maximilian Doré, Evan Cavallo, and Anders Mörtberg. *Automating Boundary Filling in Cubical Agda.* 2024. arXiv: `2402.12169 [cs.LO]`. URL: `https://arxiv.org/abs/2402.12169`.

[9] Giuseppe Peano. *Arithmetices principia, nova methodo exposita.* Fratres Bocca, 1889.

[10] Matthew Daggitt. *The Agda standard library, Built-in natural numbers.* `https://github.com/agda/agda-stdlib/blob/master/src/Data/Nat/Base.agda`. [Accessed: November 27, 2024]. 2024.

List of Codes