CHALMERS

A family of universes for generic programming

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

STEVAN ANDJELKOVIC

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2011

A family of universes for generic programming

STEVAN ANDJELKOVIC

Examiner: PETER DYBJER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2011

Abstract

Datatype-generic programming in the dependently typed setting can be achieved using the universe construction. The set of codes of the universe has an impact on the datatypes it can express and also on the set of supported datatype-generic functions.

For example a universe with a code for functions can express datatypes such as the Brouwer ordinals:

```
data Ord : Set where
  zero  : Ord
  suc   : Ord → Ord
  limit : (ℕ → Ord) → Ord   -- The function code is needed to
                            -- describe limit.
```

It cannot support a meaningful datatype-generic decidable equality however, because generally equality of functions is undecidable.

This problem often leads to the adaptation of several universes, for example one without a code for functions for which we can have decidable equality and another with a code for functions for which we can express datatypes such as the Brouwer ordinals. Both universes might support, for example, a datatype-generic mapping and iteration function however, this leads to code duplication – defeating one of the very aims of datatype-generic programming.

This work proposes a way around this problem by presenting a family of universes (or an indexed universe), where the index explains what class of datatypes that is supported. Datatype-generic functions which work over multiple classes of datatypes, such as mapping and iterating, are implemented using a polymorphic index.

The entire development is carried out in Agda. It is at least partially compatible with levitation and ornamentation, which are recently proposed techniques for avoiding code duplication, both based on the universe construction.

Acknowledgements

I would like to thank Olle Fredriksson and Daniel Gustafsson for discussions throughout the work and feedback on my drafts.

I would also like to thank my supervisor Peter Dybjer who has spent many hours explaining, giving feedback on my drafts and correcting many of my mistakes and misunderstandings.

Contents

Chapter 1

Introduction

To avoid code duplication is a central theme in programming. We use abstraction mechanisms such as functions to isolate pieces of code that recur often and, in turn, modules to isolate functions that recur often in our programs.

We want to avoid code duplication because it saves us time and effort not having to write code that we or somebody else already has written. Less code also means we have to write fewer tests and proofs to assure that the code works as intended. It also makes maintenance of the code easier, because possible bugs will be isolated in one function or module rather than spread out across our programs.

Strongly typed functional programming languages support polymorphism and higher-order functions which allows to write boiler-plate avoiding functions such as map and iteration for lists:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

mapL : ∀ {A B : Set} → (A → B) → List A → List B
mapL f []       = []
mapL f (x :: xs) = f x :: mapL f xs

iterL : ∀ {A B : Set} → (A → B → B) → B → List A → B
iterL c n []       = n
iterL c n (x :: xs) = c x (iterL c n xs)
```

These traversal functions have to be written for each new datatype however:

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

mapT : ∀ {A B : Set} → (A → B) → Tree A → Tree B
```

```
mapT f (leaf x)   = leaf (f x)
mapT f (join l r) = join (mapT f l) (mapT f r)

iterT : ∀ {A B : Set} → (B → B → B) → (A → B) → Tree A → B
iterT j l (leaf x)   = l x
iterT j l (join x y) = j (iterT j l x) (iterT j l y)
```

This is clearly a form of code duplication. To be able to avoid it we need a new abstraction mechanism – datatype-genericity. Researchers, in the strongly typed programming community, have been working on this problem since the 90s, but despite many proposals no generally agreed upon solution exists.

One of the proposed methods to achieve datatype-genericity is polytypism [JJ97, Jan00]. By defining functions on the structure of the argument polytypism lets us reuse the function on arguments of different structures.

In the dependently typed setting this is realised using the universe construction. It roughly works as follows: We define a datatype of codes called the universe. The codes allow us to formally state informal descriptions of structures, such as "a list is either empty or it is a pair of a parameter and a sublist" or "a tree is either a parameter or a pair of two subtrees". As you can see both lists and trees share the "either", "pair", "parameter" and "substructure" structures as do many other common datatypes. So we describe datatypes using the codes from the universe, the codes are formal versions of "either", "pair", etc and then we have a decoding function which turns the descriptions of, for example, lists and trees into actual datatypes. We can then define datatype-generic functions by pattern-matching on the codes:

```
map : (code : Universe) → (A → B) → decode code A → decode code B
```

And we can regain the specific maps by supplying the desired code:

```
map codeOfList ≡ mapL : (A → B) → List A → List B
map codeOfTree ≡ mapT : (A → B) → Tree A → Tree B
```

The problem of this approach is that the expressivity of the codes have an impact on which datatype-generic functions are supported by a universe. For example, a universe capable of describing datatypes with constructors that contain functions, such as:

```
data Ord : Set where
  zero  : Ord
  suc   : Ord → Ord
  limit : (ℕ → Ord) → Ord   -- Notice that the argument is a function.
```

cannot support a meaningful datatype-generic equality function, because in general there is no way to decide if two functions are equal.

The usual method for overcoming this problem is to define two universes, one with support for datatypes containing functions and one without. The problem of this approach is that datatype-generic functions that are supported

by both universes need to be written twice. To support the most commonly used datatypes several universes are needed and this results in a lot of code duplication.

This work proposes a solution to this problem, by using a single universe which is indexed by the class of datatypes it supports. The solution, at least partially, is compatible with more recently proposed techniques for avoiding code duplication – levitation [CDMM10] and ornamentation [McB11] – which both are based on the universe construction.

1.1 Related work

The first use of the universe construction to achieve polytypic datatype-genericity in the dependently typed setting is [PR98] in the programming language LEGO [LP92]. The universe presented is relatively weak and can only be used to describe datatypes that are sums of products.

[BDJ03] improves the situation by presenting several universes. Together the universes can describe all datatypes describable in [PR98] and more – most notably families of datatypes [Dyb91]. Ideas of how to construct a universe for families of datatypes came from earlier work [DS99]. The implementation is done in Alfa [aC].

[Mor07] gives an alternative presentation of the universes in [BDJ03] in the programming language Epigram [MM04]. The strength of the universes is improved by allowing so called nested fixpoints. Several larger examples of datatype-generic programs are given to demonstrate the feasibility of the approach.

1.2 Plan

We will begin, in chapter 2, by giving a brief tour on how the programming language Agda [Nor07] works. We will be using Agda throughout the rest of the work.

Then, in chapter 3, we will have a look at how the universe construction can be used to achieve datatype-genericity.

After that, in chapter 4, we will try to classify and group datatypes. So for example, sums of products is one class of datatypes and families of datatypes is another (larger) class. We do this because, as we have already noted, different classes of datatypes admit different datatype-generic functions.

In chapter 5 we discuss different design choices one can make when constructing universes and how the choices impact the set of datatype-generic functions the universe supports.

Then, in chapter 6, we will present a family of universes which is capable of describing the perhaps most important classes of datatypes. The point of this design is that we can avoid the code duplication associated with having several universes for different classes of datatypes.

Finally, in chapter 7 and 8, we briefly explain the recently proposed techniques levitation and ornamentation and then show that the universe of chapter 6 is at least partially compatible with these techniques.

Chapter 2

Agda

Agda is a dependently typed functional programming language and a proof assistant based on Per Martin-Löf's intensional intuitionistic type theory. The aim of this chapter is to try to make sense of that last sentence. In particular we will try to answer the questions:

- What is a dependently typed functional programming language?
- What is a proof assistant?
- What does intensional and intuitionistic mean?
- What is a type theory?

As all these questions are intertwined, we shall not answer them in any particular order. In the process we will introduce all the language specific features of Agda that we will use later. Alternative introductions to Agda can be found in [Nor08] and [BD08]. They cover some features and aspects which are not covered here.

2.1 The similarities with strongly typed functional programming languages

Let us begin by examining the similarities between Agda and strongly typed functional programming languages such as ML and Haskell.

The syntax for introducing datatypes in Agda is similar to Haskell's GADT-style declarations:

```
data Bool : Set where
  true  : Bool
  false : Bool

data ℕ : Set where
  zero : ℕ
```

```
  suc  : ℕ → ℕ

data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

In Haskell we would have written `*` instead of `Set`. Function definitions using pattern-matching are similar as well:

```
id : ∀ A → A → A
id A x = x

idBool = id Bool

if_then_else_ : ∀ {A} → Bool → A → A → A
if true  then t else f = t
if false then t else f = f

not : Bool → Bool
not b = if b then false else true

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

The difference is that Agda allows not just infix operators (+), but also so called mixfix operators (such as if then else), where underscores in the type signature signal where arguments go. Notice also how the `A` in the type of the identity and if functions has to be explicitly quantified unlike in Haskell. The curly braces around the `A` in the if function says that we would like the type to be implicit and inferred by Agda rather than passed explicitly as in the identity function.

2.2 The Curry-Howard correspondence and how to prove propositions

So far we have merely seen how to do things you can do in strongly typed languages inside Agda. Next we will have a look at things Agda can do that generally strongly typed programming languages cannot[1].

The dependent function type, written as `(x : A) → B` in Agda, is a generalisation of the function type, written as `A → B`, where `B` may depend on values, `x`, of type `A`. In the special case where `B` does not depend on values of `A`, we get the non-dependent function type:

```
A → B = (_ : A) → B
```

[1] We have to be a bit careful in saying that Haskell is not able to do these things, because with all its type extensions it can emulate some of the functionality we are about to describe.

The dependent function type coupled with datatype declarations allows us to define types such as the dependent pair type:

```
data Σ (A : Set)(B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

The dependent pair type is a generalisation of the pair type where the type of the second component may depend on values of the first, and in the special case where it does not we get the non-dependent pair type:

```
_×_ : Set → Set → Set
A × B = Σ A λ _ → B

proj₁ : {A : Set}{B : Set} → A × B → A
proj₁ (x , y) = x

proj₂ : {A B : Set} → A × B → B  -- Where {A B : Set} is sugar for
                                 -- {A : Set}{B : Set}.
proj₂ (x , y) = y
```

The typing rules of the constructors of dependent function and dependent pair types correspond to the introduction rules for universal ($\forall$) and existential ($\exists$) quantifiers in (higher-order) intuitionistic[2] predicate logic. While the typing rules of application and uncurrying:

```
uncurry : {A : Set}{B : A → Set}{C : Σ A B → Set} →
          ((x : A) → (y : B x) → C (x , y)) →
          ((p : Σ A B) → C p)
uncurry f (x , y) = f x y
```

correspond to the elimination rules of $\forall$ and $\exists$. The non-dependent function and pair types correspond to implication and conjunction. The rest of the connectives of propositional logic are defined as follows:

```
-- Disjoint union corresponds to disjunction.
data _⊎_ (A : Set)(B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B

-- Disjunction elimination.
[_,_] : {A : Set}{B : Set}{C : A ⊎ B → Set} →
        ((x : A) → C (inj₁ x)) → ((x : B) → C (inj₂ x)) →
        ((x : A ⊎ B) → C x)
[ f , g ] (inj₁ x) = f x
```

[2]Intuitionists do not accept the *principle of the excluded middle* ($A \vee \neg A$) and equivalent principles. In practice this means that indirect methods such as proof by contradiction (to prove A, assume $\neg A$ and derive a contradiction) and proof by contrapositive (to prove $A \Rightarrow B$, one proves $\neg B \Rightarrow \neg A$) are not accepted. Proof by contradiction should not be confused with proof of negation (to prove $\neg A$, assume A and derive a contradiction), which is accepted.

```
[ f , g ] (inj₂ y) = g y

-- False, no constructors.
data ⊥ : Set where

⊥-elim : {C : Set} → ⊥ → C
⊥-elim ()  -- If someone gives us a value of ⊥ we can prove
           -- anything. By pattern-matching using () we tell Agda that
           -- this is absurd.

¬_ : Set → Set
¬ A = A → ⊥
```

This allows us to state and prove propositions of logic. Here is a simple example stating and proving that conjunction is commutative. We give the proof in natural deduction style first and then as a term in Agda.

Proposition.

$$A \wedge B \Rightarrow B \wedge A$$

Proof.

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^p}{B}\wedge \mathrm{E}_2 \quad \dfrac{[A \wedge B]^p}{A}\wedge \mathrm{E}_1}{B \wedge A}\wedge \mathrm{I}}{A \wedge B \Rightarrow B \wedge A}\Rightarrow \mathrm{I}_p$$

□

Proof.

```
proof : ∀ {A B} → A × B → B × A
proof = λ p → (proj₂ p , proj₁ p)
```

□

Where ∀ {A B} desugars into {A B : _}. Putting underscores on the right hand side of : or = is a way of telling Agda to try to infer the term, which in this case is easy because we know the type of the pair datatype's arguments (Set).

Here is another example, one of de Morgan's laws:

Proposition.

$$\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$$

Proof.

$$\dfrac{\dfrac{\dfrac{[\neg A]^f \quad \dfrac{[A \wedge B]^p}{A}\wedge \mathrm{E}_1}{\bot}\to \mathrm{E}}{\neg A \Rightarrow \bot}\to \mathrm{I}_f \qquad \dfrac{\dfrac{[\neg B]^g \quad \dfrac{[A \wedge B]^p}{B}\wedge \mathrm{E}_2}{\bot}\to \mathrm{E}}{\neg B \Rightarrow \bot}\to \mathrm{I}_g \qquad [\neg A \vee \neg B]^s}{\dfrac{\dfrac{\bot}{\neg(A \wedge B)}\to \mathrm{I}_p}{\neg A \vee \neg B \Rightarrow \neg(A \wedge B)}\to \mathrm{I}_s}\vee \mathrm{E}$$

□

Proof.

```
proof′ : ∀ {A B} → ¬ A ⊎ ¬ B → ¬ (A × B)
proof′ {A}{B} = λ (s : ¬ A ⊎ ¬ B) → λ (p : A × B) →
  [ (λ (f : ¬ A) → f (proj₁ p))
  , (λ (g : ¬ B) → g (proj₂ p)) ] s
```

□

The last Agda proof also illustrates how we can access implicit arguments if we want to.

And finally propositions using quantifiers:

```
∃⟶¬∀¬ : ∀ {A : Set}{P : A → Set}
      → Σ A P → ¬ ((x : A) → ¬ P x)
∃⟶¬∀¬ p f = uncurry f p
```

This correspondence between types and propositions and between programs and proofs is known as the Curry-Howard correspondence.

Agda has an interactive Emacs mode which helps, or assists, us in building proofs like the ones above. The details of how it works can be found on the Agda wiki [Tea11].

By making use of Agda's support of indexed datatypes [Dyb91], we can define the identity type which captures the notion of two values being (propositionally) equal:

```
data _≡_ {A : Set}(x : A) : A → Set where
  refl : x ≡ x
```

The distinction between unindexed and indexed datatypes is that while both allow for parameters, such as `{A : Set}(x : A)`, to the left of the colon the indexed datatypes also allow indices, such as `A` to the right of the colon. Parameters are in scope across the whole declaration and may not change[3]. While indices are not in scope, but must be provided and might vary across the declaration.

Let us have another example of an indexed datatype before getting back to the identity type. Here are length indexed lists, also known as vectors:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Perhaps it is more clear from the vector example that parameters stay the same while indices might vary.

Anyways, using the identity type and the fact that Agda's typechecker normalises during typechecking we can prove all kinds of equalities:

[3] Agda allows for parameters to change in recursive positions, but this is not allowed in [Dyb91].

```
-- Tell Agda about our natural numbers, so she allows us to use decimal
-- notation.
{-# BUILTIN NATURAL ℕ    #-}
{-# BUILTIN ZERO    zero #-}
{-# BUILTIN SUC     suc  #-}

test : if true
          then 1
          else 2 ≡ 1
test = refl

test′ : 1 + 2 ≡ 3
test′ = refl
```

The typechecker normalises the terms in the type by unfolding the definitions of if and plus and when it is done both the left and right hand side of the identity type are equal and we can give the constructor `refl`.

The dependent function type also allows us to implement induction principles, or eliminators, for our datatypes:

```
if : {P : Bool → Set} → (b : Bool) → P true → P false → P b
if true  t f = t
if false t f = f

natrec : {P : ℕ → Set} → ((n : ℕ) → P n → P (suc n)) → P zero → (n : ℕ) → P n
natrec s z zero    = z
natrec s z (suc n) = s n (natrec s z n)
```

Using these we can prove more interesting equalities:

```
not-involutary : ∀ b → not (not b) ≡ b
not-involutary b = if {λ b → not (not b) ≡ b} b refl refl

cong : ∀ {A B : Set}{x y : A}(f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl

+-assoc : ∀ m n o → m + (n + o) ≡ (m + n) + o
+-assoc m n o = natrec {λ m → m + (n + o) ≡ (m + n) + o}
                       (λ _ ih → cong suc ih) refl m
```

Unfortunately it is easier to write these proofs (using the Emacs mode) than to read them once they are written. Let us step through these proofs slowly.

When we are proving that the not function is its own inverse, we begin by using the induction principle for booleans. We explicitly pass `P : Bool → Set`, which is the predicate on booleans we want to show holds for all booleans, in this case it happens to be the proposition we are trying to prove. Then we pass the boolean, `b`. Now the next two arguments to the induction principle are proofs that `P true = not (not true) ≡ true` and `P false = not (not`

`false) ≡ false` hold. It is clear by the definition of the not function that these are true by reflexivity, so we pass `refl`.

The proof that addition is associative starts off in the same way, we use the induction principle and pass the predicate we want to show holds. Then we have to provide two proofs, let us start with the base case. `P zero = zero + (n + o) ≡ (zero + n) + o`, this normalises using the definition of plus (`zero + n = n`) to: `n + o ≡ n + o`, which holds by reflexivity. In the step case we have to show that for all natural numbers, `m`, if `P m` holds than so does `P (suc m)`. So we have to prove `P (suc m) = suc m + (n + o) ≡ (suc m + n) + o` or, rewritten using the definition of plus (`suc m + n = suc (m + n)`), `suc (m + (n + o)) ≡ suc ((m + n) + o)` and we are given `P m = m + (n + o) ≡ (m + n) + o`. So we merely need to add `suc` to both sides of the equation and that is what `cong suc` does.

We could also use pattern-matching to prove the same equalities:

```
not-involutary′ : ∀ b → not (not b) ≡ b
not-involutary′ true  = refl
not-involutary′ false = refl

+-assoc′ : ∀ m n o → m + (n + o) ≡ (m + n) + o
+-assoc′ zero    n o = refl
+-assoc′ (suc m) n o = cong suc (+-assoc′ m n o)
```

2.3 The termination and positivity checkers and consistency

Something one has to be careful about when proving things like this is to not write non-terminating programs:

```
bottom : ⊥
bottom = bottom
```

While this definition typechecks, it is not a proof, because it does not terminate. Clearly if this would be allowed it would make our logic inconsistent.

The well known halting problem states that it is impossible to tell if a program written in a Turing complete language will terminate or not. So we can either restrict the language so that it is not Turing complete, by for example only allowing eliminators which are primitively recursive and therefore terminating. Or we can add a termination decision procedure which is sound (all programs that it says terminate in fact do so), but incomplete (there are programs it says do not terminate when they in fact do so).

Type theory uses the former approach and only allows eliminators. Agda on the other hand does the latter. It allows us to use general recursion and has a so called termination checker which complains unless recursive calls are on structurally smaller data. For example, let us look at the definition of append for lists:

```
_++_ : ∀ {A} → List A → List A → List A
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

In the definition the recursive call, `xs ++ ys`, is on structurally smaller data, because `xs` is structurally smaller (we peeled off a constructor) than `x :: xs`.

It might be helpful to know that terminating programs using pattern-matching can be translated to programs using eliminators [GMM06], even though Agda does not do this translation.

Besides non-termination there is a problem related to data declarations that breaks the consistency of our logic:

```
data Bad : Set where
  bad : (f : Bad → ⊥) → Bad

loop′ : Bad → ⊥
loop′ (bad f) = f (bad f)

loop : ⊥
loop = loop′ (bad loop′)
```

The problem is that `Bad` occurs a negative position (to the left of a function type) in the constructor `bad`. Agda has a so called positivity checker which complains about dangerous datatypes such as the above. The positivity checker is incomplete and has not been proven to be sound.

Type theory is usually presented with a fixed set of types, most of which we have seen: dependent and non-dependent functions, pairs, disjoint unions, natural numbers, booleans, the identity type and also the type of well-founded trees (the W-type) which we have not seen yet:

```
data W (A : Set)(B : A → Set) : Set where
  sup : (x : A) → (B x → W A B) → W A B
```

All these types are shown to be consistent, if one wants to add new datatypes besides those then one has to prove that the new datatypes also are consistent as well.

It might be helpful to know that a large class of consistent unindexed and indexed datatypes can be translated into W-types [Dyb97, AM09]. This translation is not done by Agda. The translation relies on extensionality, which we shall discuss next.

2.4 Intensional versus extensional type theory

We have seen two notions of equality so far. Definitional equality (=) which is used when defining functions and also by the typechecker as it normalises terms during typechecking. And propositional equality (≡) or the identity type, which is used to prove equalities as we have seen.

In intensional type theories (such as the one Agda is based on) these two notions of equality are kept separate. Typechecking is decidable, because we know what our definitions are and can hence test equalities by unfolding the definitions.

In extensional type theories the two notions of equality are merged, resulting in more expressivity at the expense of losing decidable typechecking. The reason for the loss is the merge, we no longer have all equalities at hand when typechecking – in general we do not know whether an equality is propositionally provable or not.

Perhaps most notably extensional type theories allow us to prove that if two functions are point-wise equal they are equal:

```
Extensionality : Set₁  -- We will explain what the ₁ means in a
                       -- moment, ignore it for now.
Extensionality = {A : Set}{B : A → Set}{f g : (x : A) → B x}
                 → (∀ x → f x ≡ g x) → f ≡ g
```

This is called extensionality for functions and it is what the translation mentioned at the end of the last section relies on. We shall see other examples that rely on extensionality later.

How equality should be handled is one of the major open problems in type theory. Recent developments have been to trying provide extensionality while retaining decidable typechecking.

2.5 Universe levels and universe level polymorphism

As we hinted on in the comment above, we will now explain what `Set₁` is. It is perhaps best illustrate by a question: what is the type of `Set`?

It cannot be that `Set : Set`, because that leads inconsistency via variations of the well known paradox discovered by Russell.

A common solution to the problem, which is adopted by Agda, is to create an infinite hierarchy of universes: `Set : Set₁ : Set₂ : ... : Setω`.

So in the definition of `Extensionality` above we had two `Set`s, `A` and `B`, thus it type must be at least `Set₁`. Another example are lists containing `Set`:

```
data List₁ (S : Set₁) : Set₁ where
  []  : List₁ S
  _::_ : S → List₁ S → List₁ S

large-list : List₁ Set
large-list = ℕ :: Bool :: ⊥ :: []
```

To avoid defining lists, or any other datatype, for each universe level we can define a universe level polymorphic version:

```
data List {a : Level}(A : Set a) : Set a where
  []  : List A
  _::_ : A → List A → List A
```

The level datatype is structurally the same as the datatype of natural numbers:

```
data Level : Set where
  zero : Level
  suc  : Level → Level
```

And as you might guess `Set = Set₀ = Set zero` and so on. Sometimes a datatype might contain two or more sets, in which case the level of the datatype will be the maximum:

```
data Σ {a b}(A : Set a)(B : A → Set b) : Set (a ⊔ b) where
  _,_ : (x : A) → B x → Σ A B
```

We can also manually lift sets to higher levels using the following datatype:

```
data Lift {a ℓ}(A : Set a) : Set (a ⊔ ℓ) where
  lift : A → Lift A

-- The trivially true datatype.
data ⊤ : Set where
  tt : ⊤

⊤₁ : Set₁
⊤₁ = Lift ⊤

tt₁ : ⊤₁
tt₁ = lift tt
```

And terms of lifted datatypes can be lowered again:

```
lower : ∀ {a ℓ}{A : Set a} → Lift {a}{ℓ} A → A
lower (lift A) = A

tt′ : ⊤
tt′ = lower tt₁
```

2.6 Programming using dependent types

So far we have mostly seen how we can use dependent types to prove things, in this section we will have a look at how we can exploit them when writing programs.

A problem in the strongly typed setting is that many functions are partial:

```
data Maybe (A : Set) : Set where
  just    : (x : A) → Maybe A
  nothing : Maybe A

lookup : ∀ {A} → ℕ → List A → Maybe A
lookup n       []       = nothing
lookup zero    (x :: xs) = just x
lookup (suc n) (x :: xs) = lookup n xs

tail : ∀ {A} → List A → Maybe (List A)
tail []       = nothing
tail (x :: xs) = just xs
```

As we use functions like `lookup` and `tail` in our programs we have to keep checking if we got a value or not, even if we know that in some context the functions will not fail. This is annoying and leads to unsafe version of the functions being adopted, where instead of returning a `Maybe` a runtime error is given in the `nothing` cases. This is not a good solution of course, after all we like to work in strongly typed languages precisely because they help us many avoid runtime error.

In the dependently typed setting we can do better. By making use of dependent types we can convince Agda that a function in some context will not fail. For example, if we lookup any position $i \in \{0, 1, ..., n-1\}$ of a list of length n we will never fail:

```
-- Fin n is a type with n elements.
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n}(i : Fin n) → Fin (suc n)

-- Alternative definitions.
⊥′    = Fin 0
⊤′    = Fin 1
Bool′ = Fin 2

lookupV : ∀ {A n} → Fin n → Vec A n → A
lookupV zero    (x :: xs) = x
lookupV (suc i) (x :: xs) = lookupV i xs
```

The case when the vector is empty need not be given, because when the vector is empty `n = 0` and `Fin 0` or $\bot$ which means we got a proof of false and this is clearly absurd.

When defining the tail function we can say that the argument is a non-empty vector and thus avoid giving the empty case:

```
tailV : ∀ {A n} → Vec A (suc n) → Vec A n
tailV (x :: xs) = xs
```

Another aspect is that we can more precisely state what the specifications of our programs are. When we append two lists, *we* know that the resulting list's length will be the sum of the original lists:

```
_++L_ : ∀ {A} → List A → List A → List A
[]       ++L ys = ys
(x :: xs) ++L ys = x :: (xs ++L ys)
```

But *Agda* does not know this, and will gladly allow us to write an incorrect version of append:

```
_++L′_ : ∀ {A} → List A → List A → List A
[]       ++L′ ys = []
(x :: xs) ++L′ ys = x :: (xs ++L′ ys)
```

If we could explain to Agda the fact we know about the resulting list's length, then it would be in a better position of helping us avoid making mistakes:

```
_++V_ : ∀ {A m n} → Vec A m → Vec A n → Vec A (m + n)
[]       ++V ys = ys
(x :: xs) ++V ys = x :: (xs ++V ys)
```

If we would try to make the same mistake we did in the bad version of list append, Agda would complain saying that the length of the empty vector is not `0 + n = n`.

Here is another example, say we are writing an equality test for booleans:

```
_≟_ : Bool → Bool → Bool
true  ≟ true  = true
true  ≟ false = false
false ≟ true  = false
false ≟ false = false
```

This is a fairly easy thing to get right, but if we copy-paste program or are careless we might make a mistake. If we instead had explained to Agda what equality test means in terms of the identity type:

```
data Dec (P : Set) : Set where
  yes : (p  : P)   → Dec P
  no  : (¬p : ¬ P) → Dec P

_≟′_ : (b b′ : Bool) → Dec (b ≡ b′)
true  ≟′ true  = yes refl
true  ≟′ false = no (λ ())
false ≟′ true  = no (λ ())
false ≟′ false = yes refl
```

Then Agda would again not allow us to make the mistake, because it would ask us for an impossible proof (`false` $\not\equiv$ `false = false` $\equiv$ `false` $\rightarrow \bot$). The λ `()` construct is a shorthand for a function from something absurd to anything else, in the above case the absurd things are that `true` $\equiv$ `false` and `false` $\equiv$ `true` respectively.

Another important advantage of using $\stackrel{?}{=}'$ over $\stackrel{?}{=}$ becomes apparent when we use them:

```
use : Bool → Bool → Set
use b b′ = if b ≟ b′ then {! We know b and b′ are the same. !}
                     else {! We know that they are not the same. !}
```

But Agda does not know this, it has no idea of the fact that $\stackrel{?}{=}$ is an equality test, all Agda knows is that it is a function returning a boolean. If we use the other approach however:

```
use′ : Bool → Bool → Set
use′ b b′ with b ≟′ b′
use′ b .b | yes refl = {! Agda knows b and b′ are the same! !}
use′ b b′ | no  p    = {! Agda knows they are not the same,
                           p is a proof of that fact! !}
```

We will explain `with` next, for now think of it as a way of casing on an argument. The dot says that the value has to be `b`, there is no choice given that we have pattern-matched on the proof that `b` $\equiv$ `b′`.

The fact we can explain things to Agda in this way is important when we program using dependent types. We shall come back with an example of this after we have explained how `with` works.

2.7 The `with` and `rewrite` constructs

The `with` construct [MM04] is an extensions which further improves the usefulness of pattern-matching. It allows us to introduce a new variable to match on:

```
filter : {A : Set}(p : A → Bool)(xs : List A) → List A
filter p []       = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false =     filter p xs
```

The three dots gets translated to the line above them:

```
filter p (x :: xs) with p x
filter p (x :: xs) | true  = x :: filter p xs
filter p (x :: xs) | false =     filter p xs
```

The `with` construct desugars in the following way:

```
mutual
  filter′ : {A : Set}(p : A → Bool)(xs : List A) → List A
  filter′ p []       = []
  filter′ p (x :: xs) = helper p x xs (p x)

  helper : {A : Set}(p : A → Bool)
           (x : A)(xs : List A)(b : Bool) → List A
  helper p x xs true  = x :: filter′ p xs
  helper p x xs false =     filter′ p xs
```

Where the `mutual` explicitly tells Agda that the following block of functions are mutually defined.

The `with` construct is not only useful for defining functions, but also when proving properties[4]:

```
n+0≡n : (n : ℕ) → n + 0 ≡ n
n+0≡n zero    = refl
n+0≡n (suc n) with n + 0 | n+0≡n n
... | .n | refl = refl
```

What happens is that in the case when `n` is `zero` the definition of `_+_` normalises the type to `0 ≡ 0` which is trivially proved. In the other case, when `n` is `suc n`, the type normalises to `suc (n + 0) ≡ suc n` and then gets stuck not able to normalise further. So what we do is abstract over `n + 0`, this creates a new variable, and the induction hypothesis `n+0≡n n` at the same time, by then matching on the proof of the induction hypothesis the new variable gets unified with the left hand side and we learn that the new variable must be `n`! The dot says that this value is forced and cannot be anything else. Recall that our type was `suc (n + 0) ≡ suc n`, we just showed that `n + 0` has to be `n` so our new type is `suc n ≡ suc n` which again is trivial.

Here is another example of the same trick:

```
lemma : (m n : ℕ) → m + suc n ≡ suc (m + n)
lemma zero    _ = refl
lemma (suc m) n with m + suc n | lemma m n
... | ._ | refl = refl
```

This pattern of rewriting the type using the induction hypothesis, or any other proved equality, is so common that there is a construct, `rewrite`, which does exactly this:

```
n+0≡n′ : ∀ n → n + 0 ≡ n
n+0≡n′ zero                      = refl
n+0≡n′ (suc n) rewrite n+0≡n′ n = refl

lemma′ : (m n : ℕ) → m + suc n ≡ suc (m + n)
```

[4]Note that Agda names can contain any non-whitespace characters, allowing us to give our property a telling name.

```
lemma′ zero    n                     = refl
lemma′ (suc m) n rewrite lemma′ m n = refl
```

The proofs using `rewrite` (`n+0≡n′` and `lemma′`) translate to those without (`n+0≡n` and `lemma`).

2.8 More on programming with dependent types

Another use of pattern-matching and `with` are so called views [MM04]. We can use views to make Agda learn something about the data we pass to them. Here we use a view to check if a natural number, `n` is within a range, `m`, and if so embed the `n` into `Fin m`:

```
toℕ : ∀ {n} → Fin n → ℕ
toℕ zero    = 0
toℕ (suc i) = suc (toℕ i)

data Range (m : ℕ) : ℕ → Set where
  inside  : (x : Fin m) → Range m (toℕ x)
  outside : (n : ℕ)     → Range m (m + n)

range : ∀ m n → Range m n
range zero    n    = outside n
range (suc m) zero = inside zero
range (suc m) (suc n) with range m n
range (suc m) (suc .(toℕ x)) | inside  x = inside (suc x)
range (suc m) (suc .(m + n)) | outside n = outside n
```

It is `range` and `Range` that constitute the view. Here are two examples that show how the view works:

```
example : range 3 2 ≡ inside (suc (suc zero))
example = refl

example′ : range 3 5 ≡ outside 2
example′ = refl
```

This can be used to range check a natural number given at runtime by the user, thereby getting a refined `Fin` datatype which can be used for safe lookups into vectors for example:

```
state : Vec String 3
state = "apa" :: "bepa" :: "cepa" :: []

check : (n : ℕ) → String
check n with range 3 n
check ._ | inside  x = "Inside: "  ++ lookup x state
```

```
check ._ | outside n = "Outside: " ++ show n

main : IO ⊤
main =
  putStrLn "Feed me something between 0 and 2" >>= λ _ →
  read                                          >>= λ n →
  putStrLn (check n)                            >>= λ _ →
  return tt
```

The same technique can be used to, for example, typecheck raw, or untyped, λ-calculus terms into well-typed terms. For the well-typed terms a nice total evaluator can be given, whereas for the raw terms we would need a partial evaluator with runtime errors[5].

2.9 Modules and records

Finally just a quick note on some of the basic functionality of modules and records in Agda. Unlike in Haskell, we can have modules inside modules in Agda:

```
module InnerModule where

  id′ : {A : Set} → A → A
  id′ x = x
```

The outer module is the main module in which this document is written. We can also parametrise modules:

```
module ParametrisedModule (A : Set) where

  pid : A → A
  pid x = x
```

We can provide the parameters upon opening the parametrised modules:

```
open ParametrisedModule Bool

-- pid : Bool → Bool, is now in scope.
```

Opening the inner module makes its content available in the outer module.

Records can be thought of as nested Σ-types where each field may depend on previous fields. Here is an example which captures what it means for there being an isomorphism between two sets:

```
record _≅_ (A B : Set) : Set where
  field
    to   : A → B
```

[5]The full example can be found in [Nor08].

```
    from : B → A

    left-inverse  : ∀ x → to (from x) ≡ x
    right-inverse : ∀ x → from (to x) ≡ x
```

Here is an example instance of the isomorphism record, showing that there is an isomorphism between the booleans and the set of two elements:

```
iso : Bool ≅ Fin 2
iso = record
  { to            = to
  ; from          = from
  ; left-inverse  = left-inverse
  ; right-inverse = right-inverse
  }
  where
  to : Bool → Fin 2
  to true  = zero
  to false = suc zero

  from : Fin 2 → Bool
  from zero           = true
  from (suc zero)     = false
  from (suc (suc ()))

  left-inverse : ∀ x → to (from x) ≡ x
  left-inverse zero           = refl
  left-inverse (suc zero)     = refl
  left-inverse (suc (suc ()))

  right-inverse : ∀ x → from (to x) ≡ x
  right-inverse true  = refl
  right-inverse false = refl
```

Chapter 3

Datatype-genericity using the universe construction

In this chapter we shall give an introduction to datatype-generic programming using the universe construction [ML84]. Much of the rest of this work will be variations of the basic techniques presented here.

Let us first recall the main motivation behind datatype-genericity. We would like to avoid writing a new instance of, for example, the mapping and iteration function for each new datatype we introduce:

```
data List (A : Set) : Set where
  []  : List A                          -- Pronounced "nil"
  _::_ : (x : A)(xs : List A) → List A  -- and "cons".

data Tree (A : Set) : Set where
  leaf : (x : A) → Tree A
  fork : (l r : Tree A) → Tree A

mapL : ∀ {A B} → (A → B) → List A → List B
mapL f []       = []
mapL f (x :: xs) = f x :: mapL f xs

mapT : ∀ {A B} → (A → B) → Tree A → Tree B
mapT f (leaf x)   = leaf (f x)
mapT f (fork l r) = fork (mapT f l) (mapT f r)

iterL : ∀ {A C : Set} → (C → C) → C → List A → C
iterL c n []       = n
iterL c n (x :: xs) = c (iterL c n xs)

iterT : ∀ {A C : Set} → (C → C → C) → (A → C) → Tree A → C
iterT s b (leaf x)   = b x
```

```
iterT s b (fork l r) = s (iterT s b l) (iterT s b r)
```

Once we have defined the datatype-generic versions of map and iteration we can state and prove more general theorems, for example:

$$\texttt{map id} \stackrel{\circ}{=} \texttt{id}$$

and

$$\texttt{map}\ (g \circ f) \stackrel{\circ}{=} \texttt{map}\ g \circ \texttt{map}\ f$$

Where $f \stackrel{\circ}{=} g = \forall x \to f\ x \equiv g\ x$, i.e. f and g are point-wise equality[1].

Given that a proof of a property of some function often is at least as long as the definition of the function and often there are more than one property – datatype-generic proofs can potentially save a lot of work.

So how do we achieve datatype-genericity? Let us have a look at the types of the mapping functions again:

```
mapL : ∀ {A B} → (A → B) → List A → List B
mapT : ∀ {A B} → (A → B) → Tree A → Tree B
```

What we would like to do is to somehow abstract out the common part of these definitions, namely `List` and `Tree` both of type `Set → Set`. Here is a first attempt:

```
map : ∀ {A B}(F : Set → Set) → (A → B) → F A → F B
```

We would like to define `map` by induction on the structure (`F : Set → Set`) of its argument (`F A : Set`). The problem is we cannot do induction on `F : Set → Set`, because it is a function. This is where the universe construction comes in.

Let us define a universe with the two structures we are interested in:

```
data U : Set where
  list tree : U
```

It has the following decoding function:

```
⟦_⟧ : U → (Set → Set)
⟦ list ⟧ A = List A
⟦ tree ⟧ A = Tree A
```

Now we can define a datatype-generic map function:

```
map : ∀ {A B}(Σ : U) → (A → B) → ⟦ Σ ⟧ A → ⟦ Σ ⟧ B
```

This allows us to define `map` on the structure of its argument, as desired:

```
map list f []         = []
map list f (x :: xs)  = f x :: map list f xs
map tree f (leaf x)   = leaf (f x)
map tree f (fork l r) = fork (map tree f l) (map tree f r)
```

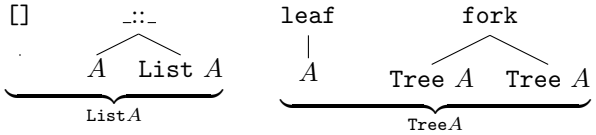
[1] We cannot prove that two functions are propositionally equal, for example `map id ≡ id`, because our propositional equality (≡) is intensional, as discussed in the previous chapter.

The list and tree specific mapping functions can be regained by applying map to the desired structure:

```
map list ≗ mapL
map tree ≗ mapT
```

Another way to see this is that we have given syntax to a subspace of `Set → Set`. This subspace is way too small however, it only allows us to work with lists and trees and not very efficiently either. We have merely moved the problem from separate definitions (`mapL` and `mapT`) to a single one (`map`). The problem is that while this allows us to define functions on the structure of their arguments, the structure is too coarsely grained.

Let us investigate the structure of our datatypes more closely:



Let us start with lists. First we have a choice of structure, we can either construct a list using a nil or a cons. In the nil case we are done. In the cons case we need to provide a pair of a parameter and a recursive call.

For trees we also have a choice. In the leaf case we need to provide a parameter, and in the fork case we need a pair of recursive calls.

Let us try to give formal syntax to what we just informally said above.

```
data Sig : Set where
  _+_ : (Σ Σ′ : Sig) → Sig   -- "Either"
  ϵ   : Sig                  -- "Done"
  ψ   : Sig                  -- "Provide parameter"
  _*_ : (Σ Σ′ : Sig) → Sig   -- "Pair"
  ρ   : Sig                  -- "Recursive call"
```

We can now formally describe the structure of lists and trees as follows:

```
′List = ϵ + ψ * ρ  -- Times binds stronger than plus, as usual.
′Tree = ψ + ρ * ρ
```

We have introduced a universe, `Sig` for signature, we use the codes of the universe to describe the structure, or (type) signature, of datatypes. The signature assumes, just like in our first attempt, we got the parameter at hand, so our decoding function will reflect this assumption, but there is also a further assumption made – the fact that we know how to construct substructures, i.e. the recursive calls.

So rather than making our decoding function of type `Set → Set` we shall make it `Set × Set → Set` where the first `Set` in the pair is the parameter assumption and the second is the substructure assumption.

```
⟦_⟧ : Sig → (Set × Set → Set)
⟦ Σ + Σ′ ⟧ X = ⟦ Σ ⟧ X ⊎ ⟦ Σ′ ⟧ X
⟦ ϵ      ⟧ X = ⊤
⟦ ψ      ⟧ X = proj₁ X
⟦ Σ * Σ′ ⟧ X = ⟦ Σ ⟧ X × ⟦ Σ′ ⟧ X
⟦ ρ      ⟧ X = proj₂ X
```

How do we discharge the substructure assumption when using the universe?

```
map : ∀ {A B}(Σ : Sig)
    → (A → B) → ⟦ Σ ⟧ (A , ?) → ⟦ Σ ⟧ (B , ?)
```

Where the question marks need to be filled with something like:

```
⟦ Σ ⟧ (A , ⟦ Σ ⟧ (A , ⋯) ⋯)
```

This seemingly endless repetition can be captured using a datatype representing the least fixpoint of a signature given the parameter:

```
data μ (Σ : Sig)(A : Set) : Set where
  ⟨_⟩ : ⟦ Σ ⟧ (A , μ Σ A) → μ Σ A
```

Why is this safe? How do we know that the repetition will eventually end? Informally the idea is that eventually we will hit some of the base cases of the signature and the recursion will stop. A formal treatment is out of scope of this work.

Using the fixpoint construction we can now discharge the substructure assumption:

```
map : ∀ {A B}(Σ : Sig) → (A → B) → μ Σ A → μ Σ B
```

To define `map` it helps to have the following helper function:

```
bimap : ∀ Σ {A B C D}(f : A → B)(g : C → D)
      → ⟦ Σ ⟧ (A , C) → ⟦ Σ ⟧ (B , D)
bimap ϵ        f g x        = x
bimap ψ        f g x        = f x
bimap ρ        f g y        = g y
bimap (Σ + Σ′) f g (inj₁ s) = inj₁ (bimap Σ  f g s)
bimap (Σ + Σ′) f g (inj₂ t) = inj₂ (bimap Σ′ f g t)
bimap (Σ * Σ′) f g (s , t)  = bimap Σ f g s , bimap Σ′ f g t
```

Now we can finally define our mapping function:

```
map : ∀ {Σ A B} → (A → B) → μ Σ A → μ Σ B
map {Σ} f ⟨ s ⟩ = ⟨ bimap Σ f (map f) s ⟩
```

The iteration function can also be defined using the helper:

```
iter : ∀ Σ {A C} → (⟦ Σ ⟧ (A , C) → C) → μ Σ A → C
iter Σ ϕ ⟨ s ⟩ = ϕ (bimap Σ id (iter Σ ϕ) s)
```

The ϕ is a $[\![\ \Sigma\ ]\!]$-algebra:

```
Algebra : Sig → Set → Set → Set
Algebra Σ A C = [[ Σ ]] (A , C) → C
```

It is a fancy word for a function which assuming some result has been computed for all the substructures computes the result for the structure. The iterator lifts the algebra to least fixpoints, discharging the algebra's assumption by (bi)mapping (the identity and) its induction hypothesis.

We will have examples next, but first a quick note on termination. Agda's termination checker complains about not being able to see that `map` and `iter` are terminating, even though they clearly are. We can overcome this problem by unfolding and inlining the definitions of `bimap` with `map` and `iter`:

```
bimapMap : ∀ Σ Σ′ {A B} → (A → B) → [[ Σ ]] (A , μ Σ′ A)
         → [[ Σ ]] (B , μ Σ′ B)
bimapMap ϵ         Σ″ f tt       = tt
bimapMap ψ         Σ″ f a        = f a
bimapMap ρ         Σ″ f ⟨ s ⟩    = ⟨ bimapMap Σ″ Σ″ f s ⟩
bimapMap (Σ + Σ′) Σ″ f (inj₁ s) = inj₁ (bimapMap Σ Σ″ f s)
bimapMap (Σ + Σ′) Σ″ f (inj₂ t) = inj₂ (bimapMap Σ′ Σ″ f t)
bimapMap (Σ * Σ′) Σ″ f (s , t)  = bimapMap Σ Σ″ f s
                                , bimapMap Σ′ Σ″ f t

map′ : ∀ {A B} Σ → (A → B) → μ Σ A → μ Σ B
map′ Σ f ⟨ s ⟩ = ⟨ bimapMap Σ Σ f s ⟩

bimapIter : ∀ Σ Σ′ {A C} → Algebra Σ′ A C
          → [[ Σ ]] (A , μ Σ′ A) → [[ Σ ]] (A , C)
bimapIter ϵ         Σ″ ϕ tt       = tt
bimapIter ψ         Σ″ ϕ a        = a
bimapIter ρ         Σ″ ϕ ⟨ s ⟩    = ϕ (bimapIter Σ″ Σ″ ϕ s)
bimapIter (Σ + Σ′) Σ″ ϕ (inj₁ s) = inj₁ (bimapIter Σ Σ″ ϕ s)
bimapIter (Σ + Σ′) Σ″ ϕ (inj₂ t) = inj₂ (bimapIter Σ′ Σ″ ϕ t)
bimapIter (Σ * Σ′) Σ″ ϕ (s , t)  = bimapIter Σ Σ″ ϕ s
                                  , bimapIter Σ′ Σ″ ϕ t

iter′ : ∀ Σ {A C} → Algebra Σ A C → μ Σ A → C
iter′ Σ ϕ ⟨ s ⟩ = ϕ (bimapIter Σ Σ ϕ s)
```

The fact that the termination checker cannot detect the termination here should not be a grave concern – the termination checker is limited and cannot be expected to deal with arbitrary programs.

We shall stick to the simple definitions and ignore the termination checker's complaints.

3.1 Examples

To get the datatype for lists, we take the least fixpoint of its signature:

```
′List = ϵ + ψ * ρ

List : Set → Set
List A = μ ′List A
```

We can define functions similar to the standard constructors[2] of list:

```
[] : ∀ {A} → List A
[] = ⟨ inj₁ _ ⟩

_::_ : ∀ {A} → (x : A)(xs : List A) → List A
x :: xs = ⟨ inj₂ (x , xs) ⟩
```

Here is an example list and a test that `map` works:

```
list : List ℕ
list = 1 :: 2 :: 2 :: 3 :: []

map-list : map suc list ≡ 2 :: 3 :: 3 :: 4 :: []
map-list = refl
```

We can define a function computing the length of a list using our iterator:

```
length : ∀ {A} → List A → ℕ
length {A} = iter _ ϕ
  where
  ϕ : Algebra ′List A ℕ
  ϕ (inj₁ _)        = 0
  ϕ (inj₂ (_ , ih)) = suc ih
```

The algebra explains how to compute the length of a list structure, given that it has already been computed for the sublists. If the structure is nil, there are no sublists and the length is zero. And if the structure is a cons and we have the length of the sublists, then we get the length of the structure by adding one to the length of the sublists.

Here is a test to see if it works:

```
length-list : length list ≡ 4
length-list = refl
```

The length function we just defined is specific to lists, using datatype generics we can define a function counting the number of parameter occurrences for any datatype in our universe:

[2] It would have been much nicer to be able to introduce a pattern synonyms [AR92] which would have allowed us to pattern match on nil and cons.

```
elems : ∀ {Σ A} → μ Σ A → ℕ
elems {Σ} {A} = iter Σ (ϕ Σ)
  where
  ϕ : ∀ Σ′ → Algebra Σ′ A ℕ
  ϕ ϵ        tt       = 0
  ϕ ψ         a        = 1
  ϕ ρ        ih       = ih
  ϕ (Σ + Σ′) (inj₁ s) = ϕ Σ  s
  ϕ (Σ + Σ′) (inj₂ t) = ϕ Σ′ t
  ϕ (Σ * Σ′) (s , t)  = ϕ Σ s +ℕ ϕ Σ′ t
```

Here is a test to see if it seems to work:

```
elems-list : elems list ≡ 4
elems-list = refl
```

The test suggest it works, but to be sure let us prove that the list specific length function is point-wise equal to the more general datatype-generic function:

```
length-elems : ∀ {A} → length {A} ≐ elems -- Agda wants to know what
                                          -- the type of the elements
                                          -- is. We could of course
                                          -- also have used the
                                          -- following type:

-- length-elems : ∀ {A}(xs : List A) → length xs ≡ elems xs

length-elems ⟨ inj₁ _ ⟩        = refl
length-elems ⟨ inj₂ (x , xs) ⟩
  rewrite length-elems xs      = refl
```

Note that changing list to a snoc list or adding a new constructor, `singleton` : $A \to$ `List` A, for example, would break the length function, but not the generic one counting parameter occurrences.

We can also define a datatype-generic function which computes the depth of a datatype, i.e. the number of "recursive calls".

```
depth : ∀ {Σ A} → μ Σ A → ℕ
depth {Σ} {A} = iter Σ (ϕ Σ)
  where
  ϕ : ∀ Σ′ → Algebra Σ′ A ℕ
  ϕ ϵ        tt       = 0
  ϕ ψ         a        = 0
  ϕ ρ        ih       = suc ih
  ϕ (Σ + Σ′) (inj₁ s) = ϕ Σ  s
  ϕ (Σ + Σ′) (inj₂ t) = ϕ Σ′ t
  ϕ (Σ * Σ′) (s , t)  = ϕ Σ s ⊔ ϕ Σ′ t  -- _⊔_ is the max function.
```

For lists, the length and depth are the same, but that is not generally true as we shall see.

```
length-depth : ∀ {A} → length {A} ≐ depth
length-depth ⟨ inj₁ _ ⟩        = refl
length-depth ⟨ inj₂ (x , xs) ⟩
  rewrite length-depth xs      = refl
```

If the parameter of a datatype is the naturals, we can calculate the sum by adding them up:

```
sum : ∀ {Σ} → μ Σ ℕ → ℕ
sum {Σ} = iter Σ (ϕ Σ)
  where
  ϕ : ∀ Σ′ → Algebra Σ′ ℕ ℕ
  ϕ ϵ         tt       = 0
  ϕ ψ          n        = n
  ϕ ρ         ih       = ih
  ϕ (Σ + Σ′) (inj₁ s) = ϕ Σ  s
  ϕ (Σ + Σ′) (inj₂ t) = ϕ Σ′ t
  ϕ (Σ * Σ′) (s , t)  = ϕ Σ s +ℕ ϕ Σ′ t

sum-list : sum list ≡ 8
sum-list = refl
```

The test suggest it works, but to be sure let us prove that it is point-wise equal to the list specific version, which is easier to understand:

```
sumL : List ℕ → ℕ
sumL = iter _ ϕ
  where
  ϕ : Algebra ′List ℕ ℕ
  ϕ (inj₁ _)        = 0
  ϕ (inj₂ (n , ih)) = n +ℕ ih

sum-sumL : (xs : List ℕ) → sum xs ≡ sumL xs
sum-sumL ⟨ inj₁ _ ⟩                          = refl
sum-sumL ⟨ inj₂ (n , ns) ⟩ rewrite sum-sumL ns = refl
```

If we look closely at the general and specific algebras, we notice that the specific is indeed just a special case of the general.

Let us now introduce trees.

```
′Tree = ψ + ρ * ρ

Tree : Set → Set
Tree A = μ ′Tree A
```

```
leaf : ∀ {A}(x : A) → Tree A
leaf x = ⟨ inj₁ x ⟩

fork : ∀ {A}(l r : Tree A) → Tree A
fork l r = ⟨ inj₂ (l , r) ⟩

tree : Tree ℕ
tree = fork
         (fork
           (leaf 1)
           (leaf 2))
         (fork
           (fork (leaf 3) (leaf 4))
           (leaf 5))
```

Our datatype-generic functions work as expected:

```
elems-tree : elems tree ≡ 5
elems-tree = refl

depth-tree : depth tree ≡ 3
depth-tree = refl

sum-tree : sum tree ≡ 15
sum-tree = refl

sum-map-tree : sum (map suc tree) ≡ 20
sum-map-tree = refl
```

Finally, let us take a look at some datatype-generic proofs. Let us start with proving that:

$$\forall\, \Sigma \to \texttt{map}\,\{\Sigma\}\ \texttt{id} \mathrel{\dot{=}} \texttt{id}$$

We will need the following helper to do it:

```
private  -- Code inside private blocks is not visible outside the
         -- current module. I like to put local lemmas inside private
         -- blocks, so that I know that they are local.

  bimap-id : ∀ Σ {A B}(g : B → B)(ih : g ≐ id)
           → bimap Σ id g ≐ id {A = ⟦ Σ ⟧ (A , B)}
  bimap-id ϵ         g ih _         = refl
  bimap-id ψ         g ih _         = refl
  bimap-id ρ         g ih b         = ih b
  bimap-id (Σ + Σ′) g ih (inj₁ s) rewrite bimap-id Σ  g ih s = refl
  bimap-id (Σ + Σ′) g ih (inj₂ t) rewrite bimap-id Σ′ g ih t = refl
  bimap-id (Σ * Σ′) g ih (s , t)  rewrite bimap-id Σ  g ih s
                                        | bimap-id Σ′ g ih t = refl
```

Our first property can now easily be proved:

```
map-id : ∀ Σ {A} → map id ≐ id {A = μ Σ A}
map-id Σ ⟨ s ⟩ rewrite bimap-id Σ (map id) (map-id Σ) s = refl
```

Next let us prove:

$$\forall\ \Sigma \to \mathtt{map}\,\{\Sigma\}\ (g \circ f) \doteq \mathtt{map}\,\{\Sigma\}\ g \circ \mathtt{map}\,\{\Sigma\}\ f$$

We will need the following helper:

```
private
  bimap-compose : ∀ Σ {A B C D E F}(f : A → B)(g : B → C)(h : D → F)
                    (i : D → E)(j : E → F)(ih : h ≐ j ∘ i)
                → bimap Σ (g ∘ f) h ≐ bimap Σ g j ∘ bimap Σ f i
  bimap-compose ϵ        f g h i j ih s        = refl
  bimap-compose ψ        f g h i j ih s        = refl
  bimap-compose ρ        f g h i j ih s        = ih s
  bimap-compose (Σ + Σ′) f g h i j ih (inj₁ s)
    rewrite bimap-compose Σ  f g h i j ih s    = refl
  bimap-compose (Σ + Σ′) f g h i j ih (inj₂ t)
    rewrite bimap-compose Σ′ f g h i j ih t    = refl
  bimap-compose (Σ * Σ′) f g h i j ih (s , t)
    rewrite bimap-compose Σ  f g h i j ih s
          | bimap-compose Σ′ f g h i j ih t    = refl
```

The second property is now again easily proved:

```
map-compose : ∀ Σ {A B C}(f : A → B)(g : B → C)
            → map {Σ} (g ∘ f) ≐ map g ∘ map f
map-compose Σ f g ⟨ s ⟩
  rewrite bimap-compose Σ f g (map (g ∘ f)) (map f) (map g)
                        (map-compose Σ f g) s               = refl
```

Next we shall, assuming a set, A, with decidable equality, show that we can define a decidable equality on the least fixpoint of any signature with A as parameter:

$$\forall\ \Sigma \to (x\ y : \mu\ \Sigma\ A) \to x \equiv y \vee x \not\equiv y$$

We will define decidable equality like this:

```
data Dec (P : Set) : Set where
  yes : (p  : P)   → Dec P
  no  : (¬p : ¬ P) → Dec P

DecEq : Set → Set
DecEq A = (x y : A) → Dec (x ≡ y)
```

For the proof we will need the proof that our constructors are injective:

```
private

  Injective : ∀ A B (f : A → B) → Set
  Injective A B f = ∀ {x y} → f x ≡ f y → x ≡ y

  Injective₂ : ∀ A B C (f : A → B → C) → Set
  Injective₂ A B C f = ∀ {x x′ y y′} → f x y ≡ f x′ y′ → x ≡ x′ × y ≡ y′

  ⟨⟩-injective : ∀ {Σ A} → Injective (⟦ Σ ⟧ (A , μ Σ A)) (μ Σ A) ⟨_⟩
  ⟨⟩-injective refl = refl

  inj₁-injective : ∀ {A B} → Injective A (A ⊎ B) inj₁
  inj₁-injective refl = refl

  inj₂-injective : ∀ {A B} → Injective B (A ⊎ B) inj₂
  inj₂-injective refl = refl

  ,-injective : ∀ {A B} → Injective₂ A B (A × B) _,_
  ,-injective refl = refl , refl

dec-cong : ∀ {A B}{x y : A}{f : A → B} → Injective A B f
         → Dec (x ≡ y) → Dec (f x ≡ f y)
dec-cong inj (yes refl) = yes refl
dec-cong inj (no x≢y)   = no (λ fx≡fy → x≢y (inj fx≡fy))

dec-cong₂ : ∀ {A B C}{x x′ : A}{y y′ : B}{f : A → B → C}
          → Injective₂ A B C f → Dec (x ≡ x′) → Dec (y ≡ y′)
          → Dec (f x y ≡ f x′ y′)
dec-cong₂ inj (yes refl) (yes refl) = yes refl
dec-cong₂ inj (no x≢x′)  _          = no (λ fxy≡fx′y′ →
                                          x≢x′ (proj₁ (inj fxy≡fx′y′)))
dec-cong₂ inj _          (no  y≢y′) = no (λ fxy≡fx′y′ →
                                          y≢y′ (proj₂ (inj fxy≡fx′y′)))
```

To make the assumption about the parameter set which is decidable, we will use a parametrised inner module, which will make the set and the decidable equality in scope inside the module:

```
module Eq (A : Set)(_≟A_ : DecEq A) where
```

Using a helper we can now prove our property:

```
  helper : ∀ Σ {X}(ih : DecEq X) → DecEq (⟦ Σ ⟧ (A , X))
  helper ϵ        ih tt       tt        = yes refl
  helper ψ        ih a        b         = a ≟A b
  helper ρ        ih x        y         = ih x y
  helper (Σ + Σ′) ih (inj₁ s) (inj₁ t)  = dec-cong inj₁-injective (helper Σ ih s t)
```

```
helper (Σ + Σ′) ih (inj₁ s) (inj₂ t)  = no λ ()
helper (Σ + Σ′) ih (inj₂ s) (inj₁ t)  = no λ ()
helper (Σ + Σ′) ih (inj₂ s) (inj₂ t)  = dec-cong inj₂-injective (helper Σ′ ih s t)
helper (Σ * Σ′) ih (s , t)  (s′ , t′) = dec-cong₂ ,-injective (helper Σ  ih s s′)
                                                               (helper Σ′ ih t t′)

_≟_ : ∀ {Σ} → DecEq (μ Σ A)
_≟_ {Σ} ⟨ x ⟩ ⟨ y ⟩ = dec-cong ⟨⟩-injective (helper Σ _≟_ x y)
```

We discharge the assumptions by providing a decidable set when opening the module:

```
open Eq ℕ _≟ℕ_ -- The proof that ℕ is decidable is in the standard
               -- library (Data.Nat).
```

And finally, a couple of examples to test our decidable equality. Where `True` turns the `yes` constructor of `Dec` into the unit type and `False` does the same thing for the `no` constructors:

```
≟-list : True (list ≟ list) × False (list ≟ 1 ∷ list)
≟-list = tt , tt

≟-tree : True (tree ≟ tree) × False (tree ≟ leaf 2)
≟-tree = tt , tt
```

3.2 Discussion

We have seen how to write datatype-generic functions and proofs about them in Agda using the universe construction. We designed our universe after analysing the structure of lists and trees. While our universe can describe a large class of datatypes, it should not come as a surprise that there are datatypes whose signatures cannot be expressed – we designed it after analysing only two particular datatypes after all.

Now that we got an idea of how this works, we shall in the following two chapters try to do what we did in this chapter – analyse datatypes and design universes for them – in a more systematic way.

Chapter 4

Classes of datatypes

In order to be able to construct universes which support large classes of datatypes, we need first to identify what classes there are.

Another reason for wanting to classify datatypes is because different classes support different datatype-generic functions.

Let us list the perhaps most important classes and give a couple of example instances of datatypes in each class and some supported datatype-generic functions.

4.1 Finite datatypes

The finite datatypes are datatypes with a finite number of elements. The empty and the unit datatype are examples:

```
data ⊥ : Set where

data ⊤ : Set where
  tt : ⊤
```

The booleans are another example:

```
data Bool : Set where
  true false : Bool
```

Yet another example is the datatype of the days of the week:

```
data Week : Set where
  monday tuesday wednesday thursday friday : Week
  saturday sunday : Week
```

The finite datatypes support datatype-generic function such as decidable equality and finite enumeration.

4.2 One-sorted datatypes

The one-sorted datatypes are a superset of the finite datatypes where recursion is allowed. This makes it possible to define, for example, the natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The list of booleans is another example:

```
data ListBool : Set where
  []      : ListBool
  true::_  : ListBool → ListBool
  false::_ : ListBool → ListBool

list = true:: false:: []
```

Binary trees are another example:

```
data Tree : Set where
  leaf : Tree
  fork : Tree → Tree → Tree
```

This class also supports enumeration. The enumeration will be potentially infinite rather than finite however.

4.3 Iterated induction

This is a superset of the one-sorted datatypes where we are allowed to refer to previously defined sets when building new ones, hence the name.

An example, which we could not define before, is the list of natural numbers:

```
data ListℕN : Set where
  []  : ListℕN
  _::_ : ℕ → ListℕN → ListℕN
```

Another is a simple expression language:

```
data Expr : Set where
  con  : ℕ → Expr
  plus : Expr → Expr → Expr
```

4.4 Infinitary induction

This is another superset of the one-sorted datatypes where we can also use functions from previously defined sets as arguments to our constructors.

The Brouwer ordinals is an example datatype in this class:

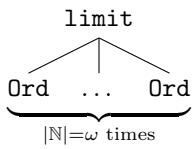
```
data Ord : Set where
  zero  : Ord
  suc   : Ord → Ord
  limit : (ℕ → Ord) → Ord

ω = limit f
  where
  f : ℕ → Ord
  f zero    = zero
  f (suc n) = suc (f n)

ω² = limit g
  where
  f : ℕ → Ord → Ord
  f zero    o = o
  f (suc n) o = suc (f n o)

  g : ℕ → Ord
  g zero    = zero
  g (suc n) = limit (λ h → f h (g n))
```

The reason that it is called infinitary induction is because we get infinitary trees:



Infinitary induction breaks enumeration, as we know from Cantor's diagonalisation argument.

4.5 Parametrised datatypes

This is also a superset of the one-sorted datatypes where we are allowed to refer to a parameter, as `A` below:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

This is the class for which we constructed a universe in the previous chapter.

We can generalise the parametrised datatypes to n parameters, thus getting n-ary parametrised datatypes. Here is an example of a binary parametrised datatype:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

We can also add iterated and infinitary induction to this class. An example, is parametrised higher-order syntax [Chl08]:

```
data Term (V : Set) : Set where
  var  : V → Term V
  lam  : (V → Term V) → Term V
  app  : Term V → Term V → Term V
  bool : Bool → Term V
```

Parametrised datatypes support mapping and, given that we got decidable equality for the parameters, it also supports decidable equality.

4.6 Many-sorted datatypes

These are also known as mutual datatypes – datatypes whose definitions mutually depend on each other. This is a generalisation of one-sorted datatypes. Examples include, the even and odd natural numbers:

```
mutual
  data Even : Set where
    zero : Even
    suc  : Odd → Even

  data Odd : Set where
    suc : Even → Odd

two : Even
two = suc (suc zero)

three : Odd
three = suc (suc (suc zero))
```

Another example are so called rose trees:

```
mutual
  data Rose (A : Set) : Set where
    node : A → Forest A → Rose A

  data Forest (A : Set) : Set where
    ϵ   : Forest A
    _::_ : Rose A → Forest A → Forest A

rose : Forest ℕ
rose = node 1
         (node 2 ϵ :: ϵ)
         :: node 3
           (node 4
```

```
      (node 5 ϵ :: ϵ)
  :: node 6 ϵ :: ϵ)
  :: ϵ
```

We can generalise many-sorted datatypes to iterated infinitary induction over n-ary parametrised many-sorted datatypes.

4.7 Finitary indexed induction

Finitary indexed induction [Dyb91] is what we get when we allow sets to be indexed by other sets to create, possibly infinite, families of sets.

We have seen instances of those already:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

data _≡_ {A : Set}(x : A) : A → Set where
  refl : x ≡ x
```

Indexed inductive definitions come in two forms. A general form, which includes the ones we have seen so far, where the target index can be any term:

```
-- Even predicate.
data GEven : ℕ → Set where
  zero : GEven 0
  2+   : ∀ {n} → GEven n → GEven (suc (suc n))

geven : GEven 4
geven = 2+ (2+ zero)
```

There is also a restricted form where the target index can only be a variable and we use propositional equality to get the same effect as in the general form:

```
data REven : ℕ → Set where
  zero : ∀ {i}   → i ≡ 0 → REven i
  2+   : ∀ {i n} → i ≡ suc (suc n) → REven n → REven i

reven : REven 4
reven = 2+ refl (2+ refl (zero refl))
```

Another distinction we make is small versus large indices. So far we have only seen small indices, that is datatypes indexed by sets. But we could also index by `Set` (the type of all small sets) in which case we get an datatype with a large index:

```
data Term : Set → Set₁ where
  con : ∀ {A}   → A               → Term A
```

```
  _,_ : ∀ {A B} → Term A → Term B → Term (A × B)
  fst : ∀ {A B} → Term (A × B)    → Term A
  snd : ∀ {A B} → Term (A × B)    → Term B

term : Term Bool
term = fst (con true , con 2)
```

Indexing by large sets is not allowed in [Dyb91].

We can define many-sorted datatypes by using an index:

```
data EvenOdd : Bool → Set where
  zeroE : EvenOdd true
  sucE  : EvenOdd false → EvenOdd true
  sucO  : EvenOdd true → EvenOdd false

Even = EvenOdd true
Odd  = EvenOdd false

two : Even
two = sucE (sucO zeroE)

three : Odd
three = sucO (sucE (sucO zeroE))
```

Thus making it a superset of the iterated n-ary parametrised many-sorted datatypes.

4.8 Infinitary indexed induction

Similarly to the infinitary induction generalisation we can generalise finitary indexed induction to infinitary indexed induction. By doing so we can define, for example, the set of accessible elements:

```
data Acc {A : Set}(_<_ : A → A → Set) : A → Set where
  acc : ∀ x → (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

4.9 Inductive-recursive

In inductive-recursive definitions [Dyb00, DS99, DS01] we mutually define a datatype (inductive) which uses, and is used in the definition of, a function (recursive).

Examples of this class include Martin-Löf's universe à la Tarski:

```
mutual
  data U : Set where
    '⊥ '⊤ 'ℕ : U
```

```
    _′⊎_      : U → U → U
    ′Σ ′Π ′W : (S : U) → (El S → U) → U
    _′≡_      : {S : U} → El S → El S → U

  El : U → Set
  El ′⊥       = ⊥
  El ′⊤       = ⊤
  El (S ′⊎ T) = El S ⊎ El T
  El (′Σ S T) = Σ (El S) (λ s → El (T s))
  El (′Π S T) = (s : El S) → El (T s)
  El ′ℕ       = ℕ
  El (′W S T) = W (El S) (λ s → El (T s))
  El (S ′≡ T) = S ≡ T
```

Another example is fresh lists, lists that promise that all their elements are unique.

```
mutual
  data FreshList (A : Set) : Set where
    nil  : FreshList A
    cons : (x : A)(xs : FreshList A)(p : Fresh x xs) → FreshList A

  Fresh : ∀ {A} → A → FreshList A → Set
  Fresh z nil           = ⊤
  Fresh z (cons x xs p) = z ≢ x × Fresh z xs

fresh : FreshList Bool
fresh = cons true (cons false nil tt) ((λ ()) , tt)

-- stale : FreshList Bool
-- stale = cons true (cons true nil tt) (? , tt)  -- true ≢ true
```

The class can be generalised to indexed induction-recursion [DS06], where the datatype may be indexed.

An example using this generalisation is used in the Bove-Carpetta method [BC05], which allows us to define nested recursive functions, such as McCarthy 91 function, in a structurally recursive manner.

```
mutual
  data dom91 : ℕ → Set where
    dom100< : ∀ {n} → 100 < n → dom91 n
    dom≤100 : ∀ {n} → n ≤ 100 → (p : dom91 (n + 11))
            → dom91 (f91 (n + 11) p) → dom91 n

  f91 : ∀ n → dom91 n → ℕ
  f91 n (dom100< h)     = n - 10
  f91 n (dom≤100 h p q) = f91 (f91 (n + 11) p) q
```

Indexed inductive-recursion is a superset of infinitary indexed induction, and thus encompasses all previous classes.

4.10 Inductive-inductive

Inductive-inductive definitions [FS10, AFMS11] are ones where we mutually define an indexed datatype and the type of its index, so that the constructors of the index may refer to the datatype and the other way around.

An instance of this class is the datatype of sorted lists:

```
mutual
  data SortedList : Set where
    []    : SortedList
    _::_,_ : (n : ℕ)(xs : SortedList)(p : n ≤L xs) → SortedList

  data _≤L_ (m : ℕ) : SortedList → Set where
    []  : m ≤L []
    _::_ : ∀ {n xs}(p : m ≤ n)(q : n ≤L xs) → m ≤L (n :: xs , q)

list = 1 :: (2 :: (3 :: [] , _) , _)
  , s≤s z≤n :: (s≤s (s≤s z≤n) :: []) -- Agda can infer the inner proofs.

-- bad  = 2 :: (1 :: [] , _) , ?  -- 2 ≤L (1 :: [] , _)
```

Inductive-inductive definitions are not covered by the theory of indexed inductive-recursive definitions presented in [DS06].

4.11 Nested fixpoints

This is a generalisation which applies to many of the classes we have seen. It adds the ability to nest fixpoints:

```
data Rose (A : Set) : Set where
  rose : A → List (Rose A) → Rose A
```

That is, as we are defining a recursive datatype, such as `Rose`, we may use it in another recursive datatype, such as in `List` above.

Nested fixpoints are not covered by the theory of indexed inductive-recursive definitions [DS06].

4.12 Discussion

We have only given a few example datatype instances of each class, what we would like is to have a clear specification of each class, which we can then use when constructing our universes. Next chapter will give a hint at what

a specification might look like, but it will be far from anything satisfactory. Something you might have noticed already is the recurring pattern of iterated induction, infinitary induction, parameters and so on.

Different classes of datatypes admit different generic functions. For example the finite datatypes are enumerable using a list while the other classes have potentially infinite datatypes, as the natural numbers, and therefore would require a colist unless they are uncountable, as the ordinals, in which case we cannot enumerate them at all. Decidable equality does not generally work when you add infinitary induction to a class. A systematic analysis of what generic functions work over what classes has, as far as I know, not been done.

There are many classes which we have omitted, in particular coinductive classes and inductive-coinductive ones.

Chapter 5

Universe design choices

In the preceding chapter we identified different classes of datatypes and discussed how they support different datatype-generic functions.

Next we shall show that there are more than one way to design a universe which supports a certain class of datatypes and how the set of supported datatype-generic functions depends on this design choice.

5.1 Finite datatypes

First we will try to illustrate the difference between "syntactic" universes and more "semantic" presentations in the context of finite datatypes. This distinction is emphasised by [Mor07] and [AMM07]. Most the ideas presented here will be from [AMM07], where they are described in more detail.

Let us start off by looking at a "syntactic" universe. The universe we constructed in chapter 3 is a "syntactic" universe and it has most of the codes we need to describe the finite datatypes. Recall however that there was no way to describe the signature of the empty type. Also we cannot have (infinite) parameters nor recursive structures, so we have to remove those codes.

```
data Sig : Set where
  ∅ ϵ          : Sig
  _+_ _*_ _^_ : (Σ Σ′ : Sig) → Sig

⟦_⟧ : Sig → Set
⟦ ∅ ⟧       = ⊥
⟦ ϵ ⟧       = ⊤
⟦ Σ + Σ′ ⟧ = ⟦ Σ ⟧  ⊎ ⟦ Σ′ ⟧
⟦ Σ * Σ′ ⟧ = ⟦ Σ ⟧  × ⟦ Σ′ ⟧
⟦ Σ ^ Σ′ ⟧ = ⟦ Σ′ ⟧ → ⟦ Σ ⟧

Bool = ⟦ ϵ + ϵ ⟧
```

```
Chessboard = ⟦ a-h * 1-8 ⟧
  where
  a-h = ϵ + ϵ + ϵ + ϵ
      + ϵ + ϵ + ϵ + ϵ

  1-8 = ϵ + ϵ + ϵ + ϵ
      + ϵ + ϵ + ϵ + ϵ

a1 : Chessboard
a1 = inj₁ _ , inj₁ _
```

The "semantic" representation for finite datatypes is the family of finite sets:

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n}(i : Fin n) → Fin (suc n)

Sig = Fin

Bool′ = Sig (1 + 1)
```

Think of the natural number as the code, that says how many elements the type contains without saying anything about its structure.

```
true : Bool′
true = zero

false : Bool′
false = suc zero

Chessboard′ = Sig (8 * 8)

a1′ : Chessboard′
a1′ = zero
```

The constructors of datatypes described in the "semantic" representation above become tedious to define. We can do better.

```
inl : ∀ {m} n → Sig m → Sig (m + n)
inl n zero    = zero
inl n (suc i) = suc (inl n i)

inr : ∀ {m} n → Sig m → Sig (n + m)
inr zero    i = i
inr (suc n) i = suc (inr n i)

sum : ∀ m {n} → Sig m ⊎ Sig n → Sig (m + n)
sum m (inj₁ i) = inl _ i
```

```
sum m (inj₂ j) = inr m j

true′ : Bool′
true′ = inl {1} 1 zero

false′ : Bool′
false′ = inr {1} 1 zero

pair : ∀ {m} n → Sig m × Sig n → Sig (m * n)
pair {suc m} n (zero  , j) = inl (m * n) j
pair         n (suc i , j) = inr n       (pair n (i , j))

a1″ : Chessboard′
a1″ = pair {8} 8 (zero , zero)
```

This is closer to the constructor definitions we had when using the "syntactic" universe. There is however an important difference; the notion of structure appears to be lost in the "semantic" presentation. See, for example, above when we define `a1″`, we appear to be using some notion of pair structure there, but really the result is the same as when we did not:

```
a1′≡a1″ : a1′ ≡ a1″
a1′≡a1″ = refl
```

We could regain the structure by defining inverses for `sum` and `pair`. `case` below is an inverse of `sum` and a similar inverse can be defined for `pair`.

```
data SigPlus (m n : ℕ) : Sig (m + n) → Set where
  isInl : (i : Sig m) → SigPlus m n (inl n i)
  isInr : (j : Sig n) → SigPlus m n (inr m j)

sigPlus : ∀ m n (i : Sig (m + n)) → SigPlus m n i
sigPlus zero    n i       = isInr i
sigPlus (suc m) n zero    = isInl zero
sigPlus (suc m) n (suc i) with sigPlus m n i
sigPlus (suc m) n (suc .(inl n i)) | isInl i = isInl (suc i)
sigPlus (suc m) n (suc .(inr m j)) | isInr j = isInr j

case : (m n : ℕ) → Sig (m + n) → Sig m ⊎ Sig n
case m n i with sigPlus m n i
case m n .(inl n i) | isInl i = inj₁ i
case m n .(inr m j) | isInr j = inj₂ j
```

Now we can define, for example, the if function for our booleans in two ways – directly or by first regaining the structure:

```
if : ∀ {n} → Bool′ → Sig n → Sig n → Sig n
if zero           t f = t
```

```
if (suc zero)     t f = f
if (suc (suc ())) t f

if′ : ∀ {n} → Bool′ → Sig n → Sig n → Sig n
if′ b t f = [ (λ _ → t) , (λ _ → f) ]′ (case 1 1 b)

if≗if′ : ∀ {n} b (t f : Sig n) → if b t f ≡ if′ b t f
if≗if′ zero           t f = refl
if≗if′ (suc zero)     t f = refl
if≗if′ (suc (suc ())) t f
```

The point of having this choice becomes clearer when we define functions which may or may not depend on the structure. The if function is neutral in this regard, it only needs to know the value of the boolean. A function which benefits from the structureless setting is decidable equality:

```
_≟_ : ∀ {n}(x y : Sig n) → Dec (x ≡ y)
zero  ≟ zero  = yes refl
zero  ≟ suc j = no (λ ())
suc i ≟ zero  = no (λ ())
suc i ≟ suc j with i ≟ j
... | yes i≡j = yes (cong suc i≡j)
... | no  i≢j = no  (λ suci≡sucj → i≢j (suc-inj suci≡sucj))
  where
  suc-inj : ∀ {n}{i j : Fin n} → Fin.suc i ≡ suc j → i ≡ j
  suc-inj refl = refl
```

We could define decidable equality for the "syntactic" universe also, we would need extensionality for the exponential case however, but the definition would be longer and we would need lemmas similar to `suc-inj` above for each of the constructors that the syntactic universe decodes into ($\mathtt{inj}_1$, $\mathtt{inj}_2$, $\_,\_$, etc (as we saw in chapter 3).

On the other hand, the proof that times distributes over plus is dependent on the structure:

```
data SigTimes (m n : ℕ) : Sig (m * n) → Set where
  isPair : (i : Sig m)(j : Sig n) → SigTimes m n (pair n (i , j))

sigTimes : ∀ m n (i : Sig (m * n)) → SigTimes m n i
sigTimes zero n ()
sigTimes (suc m) n i with sigPlus n (m * n) i
sigTimes (suc m) n .(inl (m * n) i) | isInl i = isPair zero i
sigTimes (suc m) n .(inr n j)       | isInr j with sigTimes m n j
sigTimes (suc m) n .(inr _ _)       | isInr .(pair _ (i , j))
                                    | isPair i j = isPair (suc i) j
```

```
split : (m n : ℕ) → Fin (m * n) → Fin m × Fin n
split m n i with sigTimes m n i
split m n .(pair _ (i , j)) | isPair i j = i , j

dist : ∀ m n o → Sig (m * (n + o)) → Sig ((m * n) + (m * o))
dist m n o Σ with split m (n + o) Σ
... | x , y with case n o y
...   | inj₁ y₁ = inl (m * o) (pair n (x , y₁))
...   | inj₂ y₂ = inr (m * n) (pair o (x , y₂))
```

We cannot define `dist` by induction on Σ, because Agda cannot unify the index of Σ's constructors (`zero` and `suc`) with the index of Σ (`m * (n + o)`) and thus Agda will not let us naively match on Σ.

If we step back a little, we might notice that the "syntactic" and "semantic" representations are isomorphic and that the functions `sum`, `pair` and their inverses `case` and `split` are the witnesses of this isomorphism. It is easier to show this if we index the syntactic universe by the number of inhabitants, like this:

```
data Sig′ : ℕ → Set where
  ∅   : Sig′ 0
  ϵ   : Sig′ 1
  _⊕_ : ∀ {m n}(Σ : Sig′ m)(Σ′ : Sig′ n) → Sig′ (m + n)
  _⊛_ : ∀ {m n}(Σ : Sig′ m)(Σ′ : Sig′ n) → Sig′ (m * n)

⟦_⟧′ : ∀ {n} → Sig′ n → Set
⟦ ∅       ⟧′ = ⊥
⟦ ϵ       ⟧′ = ⊤
⟦ Σ ⊕ Σ′ ⟧′ = ⟦ Σ ⟧′ ⊎ ⟦ Σ′ ⟧′
⟦ Σ ⊛ Σ′ ⟧′ = ⟦ Σ ⟧′ × ⟦ Σ′ ⟧′
```

We can then show:

```
⇒ : ∀ {n}(Σ : Sig′ n) → ⟦ Σ ⟧′ → Sig n
⇒ ∅         ()
⇒ ϵ         _ = zero
⇒ (Σ ⊕ Σ′) s = sum  _ (Sum.map  (⇒ Σ) (⇒ Σ′) s)
⇒ (Σ ⊛ Σ′) s = pair _ (Prod.map (⇒ Σ) (⇒ Σ′) s)

⇐ : ∀ {n}(Σ : Sig′ n) → Sig n → ⟦ Σ ⟧′
⇐ ∅         ()
⇐ ϵ         i  = _
⇐ (Σ ⊕ Σ′) i  = Sum.map  (⇐ Σ) (⇐ Σ′) (case _ _ i)
⇐ (Σ ⊛ Σ′) i  = Prod.map (⇐ Σ) (⇐ Σ′) (split _ _ i)
```

To show that these form an isomorphism we also need to show:

```
⇐⇒id : ∀ {n}(Σ : Sig′ n) → ⇐ Σ ∘ ⇒ Σ ≗ id
```

```
⇒⇐id : ∀ {n}(Σ : Sig′ n) → ⇒ Σ ∘ ⇐ Σ ≗̊ id
```

I have left out the code for this last bit, because it is rather long and I have a lemma (injectivity of `pair`) left to complete the proof.

5.1.1 Syntactic universe with distinct constructors

We just saw two very different, but isomorphic, representations of the finite datatypes – "syntactic" and "semantic". The point being that one representation is sometimes better suited than the other for defining some function.

Next I will try to make the point that we can have different representations of "syntactic" universes. For example, here is a universe for finite datatypes that only allows you to define datatypes as sums of products:

```
Sig = List (List Bool)

′Bool : Sig
′Bool = [] :: [] :: []

⟦_⟧ : Sig → Set
⟦ xss ⟧ = ⨁ xss
  where
  ⨂ : List Bool → Set
  ⨂ []           = ⊤
  ⨂ (true  :: xs) = ⊤ × ⨂ xs
  ⨂ (false :: xs) = ⊥ × ⨂ xs

  ⨁ : List (List Bool) → Set
  ⨁ []         = ⊥
  ⨁ (xs :: xss) = ⨂ xs ⊎ ⨁ xss

B : Set
B = ⟦ ′Bool ⟧

t : B
t  = inj₁ _

f : B
f = inj₂ (inj₁ _)
```

So there is some structure here (sums and products), but we are not given control over it. The advantage of this is that we can easily recognise what a constructor is – the sums.

[PR98] use this approach for parametrised one-sorted datatypes and use the fact that they know what the constructors are to formulate and prove a "no confusion" theorem; two different constructors of some datatype, $\mathtt{c}_1$ and $\mathtt{c}_2$, are unequal ($\mathtt{c}_1 \not\equiv \mathtt{c}_2$), for instance `[]` $\not\equiv$ `x :: xs`.

5.2 m-ary parametrised n-sorted finitary datatypes

In chapter 3 we constructed a universe for unary parametrised one-sorted finitary datatypes. We shall here show how to generalise this to m-ary parametrised n-sorted finitary datatypes and then further generalise it so that it can handle iterated and infinitary induction.

Recall the type of our decoding function from chapter 3:

```
⟦_⟧ : Sig → (Set × Set → Set)
```

Where the first `Set` in the product was the parameter and the second was the recursive call. The perhaps most natural idea is to try to replace both `Set`s with a vector of `Set`s:

```
⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m × Vec Set n → Set)
```

And then change the codes of parameter and recursive call to be able to pick the desired set from this vector simply by looking it up:

```
data Sig (m n : ℕ) : Set where
  ϵ       : Sig m n
  ψ       : (i : Fin m) → Sig m n
  ρ       : (o : Fin n) → Sig m n
  _+_ _*_ : (Σ Σ′ : Sig m n) → Sig m n

⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m → Vec Set n → Set)
⟦ ϵ      ⟧ is os = ⊤
⟦ ψ i    ⟧ is os = lookup i is
⟦ ρ o    ⟧ is os = lookup o os
⟦ Σ + Σ′ ⟧ is os = ⟦ Σ ⟧ is os ⊎ ⟦ Σ′ ⟧ is os
⟦ Σ * Σ′ ⟧ is os = ⟦ Σ ⟧ is os × ⟦ Σ′ ⟧ is os
```

To achieve many-sorted datatypes we will index the fixpoint, as described in the previous chapter:

```
tabulate : ∀ {n}{A : Set₁} → (Fin n → A) → Vec A n
tabulate {zero}  f = []
tabulate {suc n} f = f zero :: tabulate (f ∘ suc)

data μ {m n}(Σ : Sig m n)(is : Vec Set m) : Fin n → Set where
  ⟨_⟩ : ∀ {o} → ⟦ Σ ⟧ is (tabulate (μ Σ is)) → μ Σ is o
```

We can now describe the two-sorted datatype of even and odd natural numbers:

```
EvenOdd : Sig 0 2
EvenOdd = (ϵ + ρ (suc zero))
        + ρ zero
```

We get the actual datatypes by supplying the correct index to the fixpoint:

```
Even : Set
Even = μ EvenOdd [] zero

Odd : Set
Odd = μ EvenOdd [] (suc zero)
```

And then we can define the constructors:

```
zeroE : Even
zeroE = ⟨ inj₁ (inj₁ _) ⟩

sucE : Odd → Even
sucE n = ⟨ inj₁ (inj₂ n) ⟩

sucO : Even → Odd
sucO n = ⟨ inj₂ n ⟩
```

There is a problem however, there is nothing that stops us from defining bad constructors, such as:

```
sucO′ : Even → Odd
sucO′ n = ⟨ inj₁ (inj₁ _) ⟩ -- Same as zeroE, which is Even...
```

The problem here is that there is no connection between the index supplied to the fixpoint and the argument to the constructor of the fixpoint.

We can fix this problem in, at least, two different ways.

One way to do it is to change the decoding function so that it gets access to the index:

```
⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m → Vec Set n → Fin n → Set)
```

And we add a code that lets us fix the index:

```
data Sig (m n : ℕ) : Set where
  η       : (o : Fin n) → Sig m n
  -- [...]

⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m → Vec Set n → Fin n → Set)
⟦ η o    ⟧ is os o′ = o ≡ o′
-- [...]

data μ {m n}(Σ : Sig m n)(is : Vec Set m) : Fin n → Set where
  ⟨_⟩ : ∀ {o} → ⟦ Σ ⟧ is (tabulate (μ Σ is)) o → μ Σ is o
```

We now describe our datatype and define the constructors as follows:

```
EvenOdd : Sig 0 2
EvenOdd = (η zero + ρ (suc zero) * η zero)
        + (ρ zero * η (suc zero))
```

```
Even : Set
Even = μ EvenOdd [] zero

Odd : Set
Odd = μ EvenOdd [] (suc zero)

zeroE : Even
zeroE = ⟨ inj₁ (inj₁ refl) ⟩

sucE : Odd → Even
sucE n = ⟨ inj₁ (inj₂ (n , refl)) ⟩

sucO : Even → Odd
sucO n = ⟨ inj₂ (n , refl) ⟩
```

The bad constructor cannot be defined, because the return type is `Odd` (which has index `suc zero`) while we are trying to build something of type `Even` (which has index `zero`), we thus get an equality constraint which we cannot satisfy:

```
-- sucO′ : Even → Odd
-- sucO′ n = ⟨ inj₁ (inj₁ {!!}) ⟩ -- zero ≡ suc zero
```

This way of describing datatypes, using η, is similar to the restricted form of inductive families that we saw in the last chapter:

```
data EvenOdd′ : Fin 2 → Set where
  zeroE′ : ∀ {i} →                          i ≡ zero     → EvenOdd′ i
  sucE′  : ∀ {i} → EvenOdd′ (suc zero) → i ≡ zero     → EvenOdd′ i
  sucO′  : ∀ {i} → EvenOdd′ zero       → i ≡ suc zero → EvenOdd′ i
```

The other solution to the problem is to index the description. We keep the decoding function as it was:

```
⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m → Vec Set n → Set)
```

We can change the fixpoint as follows:

```
data μ {m n}(Σ : Fin n → Sig m n)(is : Vec Set m) : Fin n → Set where
  ⟨_⟩ : ∀ {o} → ⟦ Σ o ⟧ is (tabulate (μ Σ is)) → μ Σ is o
```

Now we get access to the index when describing our datatypes:

```
EvenOdd : Fin 2 → Sig 0 2
EvenOdd zero           = ε + ρ (suc zero)
EvenOdd (suc zero)     = ρ zero
EvenOdd (suc (suc ()))

Even : Set
Even = μ EvenOdd [] zero
```

```
Odd : Set
Odd = μ EvenOdd [] (suc zero)

zeroE : Even
zeroE = ⟨ inj₁ _ ⟩

sucE : Odd → Even
sucE n = ⟨ inj₂ n ⟩

sucO : Even → Odd
sucO n = ⟨ n ⟩
```

The bad constructor is impossible to define, because the return type is `Odd` (≡ `μ EvenOdd [] (suc zero)`):

```
-- sucO′ : Even → Odd
-- sucO′ n = ⟨ {!!} ⟩
```

To construct a value of `Odd`, we need by construction to give something of type `Even` to the constructor of our fixpoint, because:

```
  ⟦ EvenOdd (suc zero) ⟧ [] (tabulate (μ EvenOdd []))
≡
  ⟦ ρ zero ⟧ [] (tabulate (μ EvenOdd [])
≡
  lookup zero (tabulate (μ EvenOdd []))
≡
  μ EvenOdd [] zero
≡
  Even
```

5.2.1 Adding iterated and infinitary induction

Iterated and infinitary induction can be added irrespectively of which of the two solutions we just saw one chooses.

It can be achieved by adding the following two codes:

```
data Sig (m n : ℕ) : Set₁ where
  κ   : (K : Set) → Sig m n
  _⊃_ : (K : Set)(Σ : Sig m n) → Sig m n
```

Notice that since both these codes take `Set`s as argument the `Sig` datatype cannot be in `Set` anymore, it must be in at least $\texttt{Set}_1$.

```
⟦_⟧ : ∀ {m n} → Sig m n → (Vec Set m → Vec Set n → Set)
⟦ κ K   ⟧ is os = K
⟦ K ⊃ Σ ⟧ is os = K → ⟦ Σ ⟧ is os
```

Also note that it is important that the first argument to ⊃ is an already defined set and not a `Sig`, because otherwise we could describe dangerous datatypes.

We can now define a simple expression language with naturals and addition, or the Brouwer ordinals:

```
′Expr : Sig 0 1
′Expr = κ ℕ + ρ zero * ρ zero

′Ord : Sig 0 1
′Ord = ϵ + ρ zero + ℕ ⊃ ρ zero
```

Adding iterated or infinitary induction using κ and ⊃ both break decidable equality, because we cannot decide if two values of an arbitrary type or if two functions in general are the same.

We could limit the argument of κ to be a decidable set (a set for which we can decide if two of its inhabitants are the same), then we would retain decidable equality for the universe at the cost of having to prove that each set we use in iterated induction is in fact decidable.

A better way of adding iterated induction might be to make κ take a `Sig` as argument instead of a `Set`. Here is a smaller universe that does this:

```
data Sig : Set where
  κ       : (Σ : Sig) → Sig
  -- [...]

mutual
  ⟦_⟧ : Sig → (Set → Set)
  ⟦ κ Σ    ⟧ X = μ Σ
  -- [...]

  data μ (Σ : Sig) : Set where
    ⟨_⟩ : ⟦ Σ ⟧ (μ Σ) → μ Σ
```

The proof that this universe has decidable equality is analogue to the one we saw in chapter 3, we shall therefore not repeat it here.

5.3 |I|-ary parametrised 0-indexed datatypes

This universe is a generalisation of the universe for m-ary parametrised n-sorted datatypes where we allow an infinite number of parameters[1] and sorts. To make this more clear, let us first rewrite the universe from the previous section using the following isomorphism:

```
iso : ∀ {n} → Extensionality _ _ → Vec Set n ≅ (Fin n → Set)
iso ext = record
  { to         = →-to-⟶ (λ xs i → lookup i xs)
```

[1] I am not sure if this is useful in practice.

```
  ; from       = →-to-⟶ tabulate

  -- →-to-⟶ lifts functions on sets to functions on setoids, upon
  -- which the isomorphism record is defined in the standard library.

  ; inverse-of = record
    { left-inverse-of  = left-inverse-of
    ; right-inverse-of = right-inverse-of
    }
  }
  where
  left-inverse-of : ∀ {n}(xs : Vec Set n)
                  → tabulate (flip lookup xs) ≡ xs
  left-inverse-of []       = refl
  left-inverse-of (x ∷ xs) = cong (_∷_ x) (left-inverse-of xs)

  right-inverse-of : ∀ {n}(I : Fin n → Set)
                   → flip lookup (tabulate I) ≡ I
  right-inverse-of I = ext (lemma I)
    where
    lemma : ∀ {n}(I : Fin n → Set)(i : Fin n)
          → lookup i (tabulate I) ≡ I i
    lemma I zero    = refl
    lemma I (suc i) = lemma (I ∘ suc) i
```

We get:

```
data Sig (m n : ℕ) : Set where
  ψ : (i : Fin m) → Sig m n
  ρ : (o : Fin n) → Sig m n
  -- [...]

⟦_⟧ : ∀ {m n} → Sig m n
    → ((Fin m → Set) → (Fin n → Set) → Set)
⟦ ψ i ⟧ X Y = X i
⟦ ρ o ⟧ X Y = Y o
-- [...]

data μ {m n}(Σ : Fin n → Sig m n)(X : Fin m → Set) : Fin n → Set where
  ⟨_⟩ : ∀ {o} → ⟦ Σ o ⟧ X (μ Σ X) → μ Σ X o
```

At this point it should be pretty clear that there is nothing that stops us from replacing the finite sets `Fin m` and `Fin n` with two that are, potentially, infinite.

```
data Sig (I O : Set) : Set where
  ϵ      : Sig I O
  ψ      : (i : I) → Sig I O
```

```
  ρ      : (o : O) → Sig I O
  _+_ _*_ : (Σ Σ′ : Sig I O) → Sig I O

⟦_⟧ : ∀ {I O} → Sig I O → ((I → Set) → (O → Set) → Set)
⟦ ϵ      ⟧ X Y = ⊤
⟦ ψ i    ⟧ X Y = X i
⟦ ρ o    ⟧ X Y = Y o
⟦ Σ + Σ′ ⟧ X Y = ⟦ Σ ⟧ X Y ⊎ ⟦ Σ′ ⟧ X Y
⟦ Σ * Σ′ ⟧ X Y = ⟦ Σ ⟧ X Y × ⟦ Σ′ ⟧ X Y
```

We can now describe families of datatypes such as vectors:

```
′Vec : ℕ → Sig ⊤ ℕ
′Vec zero    = ϵ
′Vec (suc n) = ψ tt * ρ n
```

If we specialise `I` to `Fin m` and `O` to `Fin n` we get the universe in the previous section.

We are not quite done yet. We cannot describe the dependency between structures, as in the description of, for example, the Σ-type:

```
data Σ (A : Set)(B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

The second argument of the constructor depends on the value of the first. In order to be able to describe structures like these, we will add a code for the Σ-type:

```
data Sig (I O : Set) : Set₁ where
  ϵ : Sig I O
  σ : (A : Set)(ϕ : A → Sig I O) → Sig I O
  -- [...]

Pow : Set → Set₁
Pow X = X → Set   -- Shorthand.

⟦_⟧ : ∀ {I O} → Sig I O → (Pow I → Pow O → Set)
⟦ ϵ     ⟧ X Y = ⊤
⟦ σ A ϕ ⟧ X Y = Σ A λ x → ⟦ ϕ x ⟧ X Y
-- [...]
```

Using σ we can define κ and +:

```
κ : ∀ {I O} → Set → Sig I O
κ K = σ K λ _ → ϵ

_+_ : ∀ {I O}(Σ Σ′ : Sig I O) → Sig I O
Σ + Σ′ = σ Bool λ b → if b then Σ else Σ′
```

5.3.1 Iterated and infinitary induction

As we just saw, we can use σ to define κ and thus get the naive version of iterated induction which breaks decidable equality (this does not matter much, because σ already breaks decidable equality).

Adding infinitary induction is a matter of adding a code for the dependent function type:

```
data Sig (I O : Set) : Set₁ where
  π : (A : Set)(ϕ : A → Sig I O) → Sig I O
  -- [...]

⟦_⟧ : ∀ {I O} → Sig I O → (Pow I → Pow O → Set)
⟦ π A ϕ ⟧ X Y = (x : A) → ⟦ ϕ x ⟧ X Y
-- [...]
```

A limitation of this approach is that we cannot define datatypes such as:

```
data Term (V : Set) : Set where
  var  : V → Term V
  lam  : (V → Term V) → Term V
  app  : Term V → Term V → Term V
```

Because in the `lam` case the π code expects a set as its first argument, while we would like to give it (a code for) a parameter.

5.3.2 Adding nested fixpoints

In order to add support for nested fixpoints to our universe we need to massage it further using yet another isomorphism. Recall the unindexed fixpoint representation from above:

```
⟦_⟧ : ∀ {I O} → Sig I O → (Pow I → Pow O → O → Set)

_⊆_ : ∀ {I} → Pow I → Pow I → Set
X ⊆ Y = ∀ {i} → X i → Y i          -- Another recurring pattern we
                                   -- shall introduce a shorthand for.

data μ {I O}(Σ : Sig I O)(X : Pow I) : Pow O where
  ⟨_⟩ : ⟦ Σ ⟧ X (μ Σ X) ⊆ μ Σ X
```

The isomorphism we need is the following (we skip the inverse proofs this time):

```
iso′ : ∀ {I O} → (Pow I → Pow O → Pow O) ≅ (Pow (I ⊎ O) → Pow O)
iso′ {I}{O} =
  begin
    (Pow I → Pow O → Pow O)
  ≅⟨ record { to   = →-to-⟶ (uncurry′ {A = Pow I}{Pow O}{Pow O})
            ; from = →-to-⟶ (curry′   {A = Pow I}{Pow O}{Pow O})
```

```
          } ⟩
    (Pow I × Pow O → Pow O)
  ≅⟨ record { to   = →-to-⟶ (λ F X → F (X ∘ inj₁ , X ∘ inj₂))
            ; from = →-to-⟶ (λ F P → F [ proj₁ P , proj₂ P ]′)
            } ⟩
    (Pow (I ⊎ O) → Pow O)
  □
```

If we rewrite the type of our decoding function using the isomorphism we get:

```
⟦_⟧ : ∀ {I O} → Sig I O → (Pow (I ⊎ O) → Pow O)

_⟨⊎⟩_ : ∀ {A B} → Pow A → Pow B → Pow (A ⊎ B)
P ⟨⊎⟩ Q = [ P , Q ]′

data μ {I O}(Σ : Sig I O)(X : Pow I) : Pow O where
  ⟨_⟩ : ⟦ Σ ⟧ (X ⟨⊎⟩ μ Σ X) ⊆ μ Σ X
```

Or alternatively, we can move the disjoint union from the decoding function to the fixpoint:

```
⟦_⟧′ : ∀ {I O} → Sig I O → (Pow I → Pow O)

data μ′ {I O}(Σ : Sig (I ⊎ O) O)(X : Pow I) : Pow O where
  ⟨_⟩ : ⟦ Σ ⟧′ (X ⟨⊎⟩ μ′ Σ X) ⊆ μ′ Σ X
```

Notice how the fixpoint applied to a signature has the same type as the decoding function (`Pow I → Pow O`). This allows us to add a code for fixpoints as follows:

```
data Sig (I O : Set) : Set₁ where
  ′μ : (Σ : Sig (I ⊎ O) O) → Sig I O
  π  : (A : Set)(ϕ : A → Sig I O) → Sig I O
  -- [...]

mutual
  data μ {I O}(Σ : Sig (I ⊎ O) O)(X : Pow I) : Pow O where
    ⟨_⟩ : ⟦ Σ ⟧ (X ⟨⊎⟩ μ Σ X) ⊆ μ Σ X

  ⟦_⟧ : ∀ {I O} → Sig I O → (Pow I → Pow O)
  ⟦ π A ϕ ⟧ X o = (x : A) → ⟦ ϕ x ⟧ X o
  ⟦ ′μ Σ  ⟧ X o  = μ Σ X o
  -- [...]
```

5.4 Algebraic versus "record-like" codes

So far we have seen "syntactic" universes with algebraic codes, that is: zero ($\emptyset$), one (ϵ), plus ($+$), times ($*$), exponentiation ($\supset$), constants (κ) and projections

(ψ and ρ). The descriptions we get using algebraic codes are tree-like because plus and times take two descriptions.

Alternatively we could pick a set of codes which would only allow us to create right-nested "record-like" descriptions, by avoiding plus and times:

```
data Sig (I O : Set) : Set where
  ϵ       : Sig I O
  ψ_*_    : (i : I)(Σ : Sig I O) → Sig I O
  ρ_*_    : (o : O)(Σ : Sig I O) → Sig I O
  σ′_,_*_ : (n : ℕ)(xs : Vec (Sig I O) n)(Σ : Sig I O) → Sig I O
```

Where plus is replaced by σ', which is a finite version of the σ code we have seen above, while times is built-in into the other codes.

```
′List : Sig ⊤ ⊤
′List = σ′ 2 , ϵ :: ψ tt * ρ tt * ϵ :: [] * ϵ
```

In the next and also later chapters we shall see why the "record-like" representation might be advantageous.

5.5 Discussion

We have seen some of the choices one has when constructing a universe for different classes of datatypes.

Universes for inductive-inductive and inductive-recursive datatypes can be found in [FS10] and [DS99].

We have also seen the distinction between so called "syntactic" and "semantic" representations in the context of finite datatypes. "Semantic" representations exist for two other classes of datatypes we looked at also, they are called containers and indexed containers respectively. [Mor07] presents these and shows that they are isomorphic to their syntactic counterparts. [Mor07] also notes the interest of having both representations, because some functions are easier to define on one of them than the other.

Besides "semantic" representations there are also significantly different ways to design "syntactic" ones. We saw an example of this where we gave a "syntactic" universe where we made it clear that all datatypes described in it are sums of products and thus it is clear what a constructor is. Another example is the difference between algebraic and "record-like" representations.

It would be interesting to try to somehow formalise all this informal comparison between different approaches.

Another interesting aspect is the proof of the existence of fixpoints. [BDJ03] sketches how to show that all their universe are embeddable into the universe of infinitary indexed definitions (with extensionality) which are in turn embeddable into the indexed inductive-recursive definitions (with extensionality) for which there is a model in classical set theory [DS03].

While a more constructive path has been taken by [Mor07] by giving container semantics and then showing that the containers can be reduced to W-types [AM09].

Chapter 6

A universe for datatypes

In preceding chapters we saw that there are many different classes of datatypes and some functions are definable over one class but not the other, for example decidable equality does not work for infinitary datatypes, while mapping works for both finitary and infinitary datatypes.

In this chapter we will propose a way to construct a single universe for many classes of datatypes. This allows us to define equality for finitary datatypes and a single mapping function which works for both finitary and infinitary datatypes and families of datatypes.

The idea is to index the universe by which class it supports.

```
Finitary?  = Bool
Unindexed? = Bool
Class      = Finitary? × Unindexed?
```

We can combine two classes as follows:

```
_⋀_ : Class → Class → Class
(f , u) ⋀ (f′ , u′) = (f ∧ f′ , u ∧ u′)
```

Using this we can now define our universe.

```
data Sig (I O : Set) : Class → Set₁ where
  ψρ      : ∀ c (i : I) → Sig I O c
  η       : ∀ c (o : O) → Sig I O c
  _+_ _*_ : ∀ {c c′}(Σ : Sig I O c)(Σ′ : Sig I O c′) → Sig I O (c ⋀ c′)
  _⊃_     : ∀ {f u}(K : Set)(Σ : Sig I O (f , u)) → Sig I O (false , u)
  σ       : ∀ {f u}(A : Set)(ϕ : A → Sig I O (f , u)) → Sig I O (f , false)
  π       : ∀ {c}(A : Set)(ϕ : A → Sig I O c) → Sig I O (false , false)
```

The class has to be given explicitly when a code can generate a signature which can be in multiple classes[1].

[1] Agda can not infer the classes of the subsignatures Σ and Σ′ of, for example, + if it knows the class of Σ + Σ′, because there is not an unique solution.

We can now get specific classes in the following way, for example, m-ary parametrised n-sorted finitary datatypes, or just finitary types (FT):

```
ft = true  , true : Finitary? × Unindexed?

FT : ℕ → ℕ → Set₁
FT m n = Sig (Fin m ⊎ Fin n) (Fin n) ft
```

The m-ary parametrised n-sorted infinitary datatypes, or infinitary types:

```
it = false , true

IT : ℕ → ℕ → Set₁
IT m n = Sig (Fin m ⊎ Fin n) (Fin n) it
```

The |I|-ary parametrised O-indexed family of finitary datatypes, or finitary families (where |_| is the cardinality):

```
ff = true  , false

FF : Set → Set → Set₁
FF I O = Sig (I ⊎ O) O ff
```

The |I|-ary parametrised O-indexed family of infinitary datatypes, or infinitary families:

```
if = false , false

IF : Set → Set → Set₁
IF I O = Sig (I ⊎ O) O if
```

The decoding function and fixpoint are the same as in the previous chapter, except for the class index:

```
Pow : Set → Set₁
Pow A = A → Set

⟦_⟧ : ∀ {I O c} → Sig I O c → (Pow I → Pow O)
⟦ ψρ _ i ⟧ X o = X i
⟦ η _ o′ ⟧ X o = o′ ≡ o
⟦ Σ + Σ′ ⟧ X o = ⟦ Σ ⟧ X o ⊎ ⟦ Σ′ ⟧ X o
⟦ Σ * Σ′ ⟧ X o = ⟦ Σ ⟧ X o × ⟦ Σ′ ⟧ X o
⟦ K ⊃ Σ  ⟧ X o = K → ⟦ Σ ⟧ X o
⟦ σ A ϕ  ⟧ X o = Σ A λ x → ⟦ ϕ x ⟧ X o
⟦ π A ϕ  ⟧ X o = (x : A) → ⟦ ϕ x ⟧ X o

data μ {I O c}(Σ : Sig (I ⊎ O) O c)(X : Pow I) : Pow O where
  ⟨_⟩ : ⟦ Σ ⟧ (X ⟨⊎⟩ μ Σ X) ⊆ μ Σ X
```

```
ψ : ∀ {I O} c → I → Sig (I ⊎ O) O c
ψ c i = ψρ c (inj₁ i)

ρ : ∀ {I O} c → O → Sig (I ⊎ O) O c
ρ c o = ψρ c (inj₂ o)
```

6.1 Examples of datatypes

Here are some examples of datatypes which, hopefully by now, are self-explanatory.

```
′List : FT 1 1
′List = η ft zero + ψ ft zero * ρ ft zero

List : Set → Set
List A = μ ′List (const A) zero

[] : ∀ {A} → List A
[] = ⟨ inj₁ refl ⟩

_::_ : ∀ {A} → A → List A → List A
x :: xs = ⟨ inj₂ (x , xs) ⟩

′EvenOdd : FT 0 2
′EvenOdd = η ft zero + ρ ft (suc zero) * η ft zero
         + ρ ft zero * η ft (suc zero)

private
  Fin0-elim : ∀ {A} → Fin 0 → A
  Fin0-elim ()

Even = μ ′EvenOdd Fin0-elim zero
Odd  = μ ′EvenOdd Fin0-elim (suc zero)

zeroE : Even
zeroE = ⟨ inj₁ refl ⟩

sucE : Odd → Even
sucE n = ⟨ inj₂ (inj₁ (n , refl)) ⟩

sucO : Even → Odd
sucO n = ⟨ inj₂ (inj₂ (n , refl)) ⟩

′Ord : IT 0 1
′Ord = η it zero + ρ it zero + (ℕ ⊃ ρ it zero)

Ord = μ ′Ord (λ ()) zero
```

```
limit : (ℕ → Ord) → Ord
limit f = ⟨ inj₂ (inj₂ f) ⟩

′Vec : FF ⊤ ℕ
′Vec = η ff 0
     + σ ℕ λ m → ψ ff _ * ρ ff m * η ff (suc m)

Vec : Set → ℕ → Set
Vec A n = μ ′Vec (const A) n

[]V : ∀ {A} → Vec A zero
[]V = ⟨ inj₁ refl ⟩

_::V_ : ∀ {A n} → A → Vec A n → Vec A (suc n)
x ::V xs = ⟨ inj₂ (_ , x , xs , refl) ⟩

-- Not real parameters...
′W : (A : Set)(B : A → Set) → IF ⊥ ⊤
′W A B = σ A λ x → π (B x) λ _ → ρ if _

W : (A : Set)(B : A → Set) → Set
W A B = μ (′W A B) (λ ()) tt

sup : {A : Set}{B : A → Set}(x : A)(f : B x → W A B) → W A B
sup x f = ⟨ x , f ⟩
```

The mapping and iterator functions are defined as follows:

```
imap : ∀ {I O c}(Σ : Sig I O c){X Y : Pow I}
     → (X ⊆′ Y) → ⟦ Σ ⟧ X ⊆ ⟦ Σ ⟧ Y
imap (ψρ _ i) f a        = f _ a
imap (η _ o)  f refl     = refl
imap (Σ + Σ′) f (inj₁ s) = inj₁ (imap Σ  f s)
imap (Σ + Σ′) f (inj₂ t) = inj₂ (imap Σ′ f t)
imap (Σ * Σ′) f (s , t)  = imap Σ f s , imap Σ′ f t
imap (K ⊃ Σ)  f h        = λ k → imap Σ     f (h k)
imap (σ A ϕ)  f (a , s)  = a   , imap (ϕ a) f s
imap (π A ϕ)  f h        = λ a → imap (ϕ a) f (h a)

⟨[_,_]⟩ : ∀ {I O}{P Q : Pow I}{R S : Pow O}
        → (P ⊆ Q) → (R ⊆ S) → (P ⟨⊎⟩ R) ⊆′ (Q ⟨⊎⟩ S)
⟨[ f , g ]⟩ (inj₁ _) = f
⟨[ f , g ]⟩ (inj₂ _) = g

map : ∀ {I O c}(Σ : Sig (I ⊎ O) O c){X Y : Pow I}
    → (X ⊆′ Y) → μ Σ X ⊆ μ Σ Y
map Σ f ⟨ s ⟩ = ⟨ imap Σ ⟨[ f _ , map Σ f ]⟩ s ⟩
```

```
Alg : ∀ {I O c}(Σ : Sig (I ⊎ O) O c) → Pow I → Pow O → Set
Alg Σ X Y = ⟦ Σ ⟧ (X ⟨⊎⟩ Y) ⊆ Y

iter : ∀ {I O X Y c}(Σ : Sig (I ⊎ O) O c) → Alg Σ X Y → μ Σ X ⊆ Y
iter Σ ϕ ⟨ s ⟩ = ϕ (imap Σ ⟨[ id , iter Σ ϕ ]⟩ s)
```

The class index complicates generic functions which are defined by pattern-matching on the signature, such as decidable equality.

```
lemma-⋀ : ∀ c c′ → c ⋀ c′ ≡ ft → c ≡ ft × c′ ≡ ft
lemma-⋀ (true  , true)  (true  , true)  p  = refl , refl
lemma-⋀ (true  , true)  (false , _)     ()
lemma-⋀ (true  , true)  (_     , false) ()
lemma-⋀ (_     , false) _               ()
lemma-⋀ (false , _)     _               ()
```

The problem, which we already hinted at earlier, is that if we give the signature, `Σ` in the helper function below, a specific class index, say `ft`, and we try to pattern-match on the signature we would give an error from Agda saying that in, for example, the `Σ + Σ′` case, the resulting index (`c ⋀ c′`) cannot be unified with `ft`. This means that Agda cannot figure out a unique solution for what `c` and `c′` should be for the result of `c ⋀ c′` to be `ft`. The above lemma is the solution, but it is too complicated for Agda to figure out. In general there is no unique solution, for example when the resulting index is `ff`.

The solution to the problem is to let the class index be a variable. A variable is easily unified and hence Agda allows pattern-matching on it. Instead we supply a proof (a constraint) that `c ≡ ft`. We cannot pattern-match on the proof, for the very reasons stated above. But we can rule out the impossible cases, for example in the ⊃ case, `p` has type `(false , u) ≡ (true , true)`, this is clearly absurd and we are thus need not give a definition for this case (it works out similarly for σ and π).

```
module Eq {m : ℕ}
          (X : Fin m → Set)
          (_≟X_ : ∀ {i : Fin m} → DecEq (X i)) where

  helper : ∀ {n c}{Y : Fin n → Set}(Σ : Sig (Fin m ⊎ Fin n) (Fin n) c)
             (p : c ≡ ft)(ih : {i : Fin n} → DecEq (Y i))
         → {i : Fin n} → DecEq (⟦ Σ ⟧ (X ⟨⊎⟩ Y) i)
  helper (ψρ _ (inj₁ i)) p ih a    b    = a ≟X b
  helper (ψρ _ (inj₂ o)) p ih a    b    = ih a b
  helper (η  _ o)        p ih refl refl = yes refl
  helper (_+_ {c}{c′} Σ Σ′) p ih (inj₁ s) (inj₁ t)
    with helper Σ (proj₁ (lemma-⋀ c c′ p)) ih s t
  helper (Σ + Σ′)        p ih (inj₁ s) (inj₁ .s) | yes refl = yes refl
  ... | no s≢t = no (λ inj₁s≡inj₁t → s≢t (inj-inj₁ inj₁s≡inj₁t))
  helper (Σ + Σ′)        p ih (inj₁ s) (inj₂ t)  = no (λ ())
```

```
helper (Σ + Σ′)        p ih (inj₂ s) (inj₁ t)  = no (λ ())
helper (_+_ {c}{c′} Σ Σ′) p ih (inj₂ s) (inj₂ t)
  with helper Σ′ (proj₂ (lemma-⋀ c c′ p)) ih s t
helper (Σ + Σ′)        p ih (inj₂ s) (inj₂ .s) | yes refl = yes refl
... | no s≢t = no (λ inj₂s≡inj₂t → s≢t (inj-inj₂ inj₂s≡inj₂t))
helper (_*_ {c}{c′} Σ Σ′) p ih (s , t) (s′ , t′)
  with helper Σ  (proj₁ (lemma-⋀ c c′ p)) ih s s′
     | helper Σ′ (proj₂ (lemma-⋀ c c′ p)) ih t t′
helper (Σ * Σ′)        p ih (s , t) (.s , .t) | yes refl | yes refl = yes refl
... | yes _   | no t≢t′ = no (λ s,t≡s′,t′ → t≢t′ (proj₂ (inj-, s,t≡s′,t′)))
... | no s≢s′ | _       = no (λ eq        → s≢s′ (proj₁ (inj-, eq)))
helper (K ⊃ Σ)         () ih x y
helper (σ A ϕ)         () ih x y
helper (π A ϕ)         () ih x y

_≟_ : ∀ {n}{Σ : FT m n}{o : Fin n} → DecEq (μ Σ X o)
_≟_ {Σ = Σ} ⟨ x ⟩ ⟨ y ⟩ with helper Σ refl _≟_ x y
... | yes x≡y = yes (cong ⟨_⟩ x≡y)
... | no  x≢y = no  (λ ⟨x⟩≡⟨y⟩ → x≢y (inj-⟨⟩ ⟨x⟩≡⟨y⟩))
```

Here are a couple of examples of using the mapping and equality functions:

```
vec : Vec ℕ 3
vec = 1 ::V 2 ::V 3 ::V []V

vec′ = map _ (const suc) vec

map-vec : vec′ ≡ 2 ::V 3 ::V 4 ::V []V
map-vec = refl

open import Data.Nat using () renaming (_≟_ to _≟ℕ_)
open Eq (const ℕ) _≟ℕ_

list : List ℕ
list = 1 :: 2 :: 3 :: []

list′ = map _ (const suc) list

map-list : list′ ≡ 2 :: 3 :: 4 :: []
map-list = refl

≟-list : True (list ≟ list) × False (list ≟ list′)
≟-list = tt , tt
```

6.2 Discussion

Currently iterated induction has to be done using σ, which makes the resulting datatype end up in, at least, the finitary families class (FF). We could, perhaps, add a code κ : `(Σ : Sig I O c)` $\rightarrow$ `Sig I O c)`, to remedy this problem.

The explicit passing of classes to the codes which could be in many classes is tedious. It seems mechanical. The class is always the same as the type of the datatype we are describing, so it is probably easy to add some language support for this.

Having special language support for it would make data special, however. Part of the reason the solution to generic programming using the universe is so nice is that it is not special. If we give up the algebraic codes and instead use "record-like" codes we could do away with $\bigwedge$ and all the problems it causes with pattern-matching. I have done some experiments with the "record-like" approach and it seems promising, due to lack of time they are not included here however.

It would be interesting to try to fit finite, inductive-inductive and inductive-recursive datatypes into this universe as well.

Chapter 7

Levitation

In this chapter we shall give a quick overview of what levitation [CDMM10] is and show that the universe we defined in the previous chapter supports levitation.

Levitation is a recently proposed technique which builds upon the idea that universes are themselves datatypes and can thus be described just like any other datatype. This enables us to write generic functions which manipulate descriptions themselves. The use of this is perhaps best illustrated by an example.

Most programming languages have variables, for example:

```
data Lang₁ : Set where
  var : String → Lang₁
  nat : ℕ → Lang₁
  add : Lang₁ → Lang₁ → Lang₁

t₁ = add (var "x") (nat 1)  -- x + 1

data Lang₂ : Set where
  var  : String → Lang₂
  bool : Bool → Lang₂
  if   : Lang₂ → Lang₂ → Lang₂ → Lang₂

t₂ = if (var "y") (bool false) (bool true)  -- not y
```

Often when dealing with languages with variables we want to be able to substitute variables:

```
s₁ : t₁ [ nat 2 / "x" ]₁ ≡ add (nat 2) (nat 1)
s₁ = refl

s₂ : t₂ [ bool false / "y" ]₂
   ≡ if (bool false) (bool false) (bool true)
s₂ = refl
```

Levitation allows us to write a generic function on codes (descriptions of datatypes) which generically adds a variable constructor to a datatype description.

```
data Lang₁′ : Set where
  nat : ℕ → Lang₁′
  add : Lang₁′ → Lang₁′ → Lang₁′

fm : Lang₁′ ⋆ String ≅ Lang₁ -- Where ⋆ adds a variable constructor
                             -- (free monad construction).
```

Substitution can then be defined once and for all using datatype-genericity:

```
_[_] : μ (Σ ⋆ X) → (X → μ (Σ ⋆ Y)) → μ (Σ ⋆ Y)
```

The key to being able to do levitation is to have a universe which is strong enough to be able to describe its own signature.

The `if` class of the universe from the previous chapter has this property, but we need to raise the levels of some of the sets first.

```
data Sig (I O : Set) : Class → Set₂ where
  ψρ      : ∀ c (i : I) → Sig I O c
  η       : ∀ c (o : O) → Sig I O c
  _+_ _*_ : ∀ {c c′}(Σ : Sig I O c)(Σ′ : Sig I O c′) → Sig I O (c ⋀ c′)
  _⊃_     : ∀ {f u}(K : Set₁)(Σ : Sig I O (f , u)) → Sig I O (false , u)
  σ       : ∀ {f u}(A : Set₁)(ϕ : A → Sig I O (f , u)) → Sig I O (f , false)
  π       : ∀ {c}(A : Set₁)(ϕ : A → Sig I O c) → Sig I O (false , false)
```

We have to manually lift the equality proof of η, otherwise the decoding is the same as before.

```
⟦_⟧ : ∀ {I O c} → Sig I O c → (Pow I → Pow O)
⟦ ψρ _ i ⟧ X o = X i
⟦ η _ o′ ⟧ X o = Lift (o′ ≡ o)
⟦ Σ + Σ′ ⟧ X o = ⟦ Σ ⟧ X o ⊎ ⟦ Σ′ ⟧ X o
⟦ Σ * Σ′ ⟧ X o = ⟦ Σ ⟧ X o × ⟦ Σ′ ⟧ X o
⟦ K ⊃ Σ  ⟧ X o = K         → ⟦ Σ  ⟧ X o
⟦ σ A ϕ  ⟧ X o = Σ[ x : A ] ⟦ ϕ x ⟧ X o
⟦ π A ϕ  ⟧ X o =   (x : A) → ⟦ ϕ x ⟧ X o
```

We can now describe a "lower" `′IF` (living in $\mathtt{Set}_1$) in the "upper" `IF` (living in $\mathtt{Set}_2$).

```
data IFC : Set₁ where
  ′ψρ ′η ′+ ′* ′σ ′π : IFC

′IF : IF Bool ⊤
′IF = σ IFC ϕ
  where
  ϕ : IFC → IF Bool ⊤
```

```
  ϕ ′ψρ = ψ if true
  ϕ ′η  = ψ if false
  ϕ ′+  = ρ if _ * ρ if _
  ϕ ′*  = ρ if _ * ρ if _
  ϕ ′σ  = σ[ A : Set ] (π[ ϕ : Lift A ] ρ if _)
  ϕ ′π  = σ[ A : Set ] (π[ ϕ : Lift A ] ρ if _)

IF′ : (I O : Set) → Set₁
IF′ I O = μ ′IF (λ b → if b then Lift I else Lift O) _
```

The "lower" universe's constructors are defined as follows.

```
ψρ′ : ∀ {I O}(i : I) → IF′ I O
ψρ′ i = ⟨ ′ψρ , lift i ⟩

η′ : ∀ {I O}(o : O) → IF′ I O
η′ o = ⟨ ′η , lift o ⟩

_+′_ : ∀ {I O}(Σ Σ′ : IF′ I O) → IF′ I O
Σ +′ Σ′ = ⟨ ′+ , Σ , Σ′ ⟩

_*′_ : ∀ {I O}(Σ Σ′ : IF′ I O) → IF′ I O
Σ *′ Σ′ = ⟨ ′* , Σ , Σ′ ⟩

σ′ : ∀ {I O}(A : Set)(ϕ : A → IF′ I O) → IF′ I O
σ′ A ϕ = ⟨ ′σ , A , (λ x → ϕ (lower x)) ⟩

π′ : ∀ {I O}(A : Set)(ϕ : A → IF′ I O) → IF′ I O
π′ A ϕ = ⟨ ′π , A , (λ x → ϕ (lower x)) ⟩
```

Here is the decoding function and least fixpoint:

```
⟦_⟧′ : ∀ {I O} → IF′ I O → (Pow′ I → Pow′ O)
⟦ ⟨ ′ψρ , lift i ⟩ ⟧′ X o = X i
⟦ ⟨ ′η , lift o′ ⟩ ⟧′ X o = o ≡ o′
⟦ ⟨ ′+ , Σ , Σ′  ⟩ ⟧′ X o = ⟦ Σ ⟧′ X o ⊎ ⟦ Σ′ ⟧′ X o
⟦ ⟨ ′* , Σ , Σ′  ⟩ ⟧′ X o = ⟦ Σ ⟧′ X o × ⟦ Σ′ ⟧′ X o
⟦ ⟨ ′σ , A , ϕ   ⟩ ⟧′ X o = Σ[ x : A ] ⟦ ϕ (lift x) ⟧′ X o
⟦ ⟨ ′π , A , ϕ   ⟩ ⟧′ X o = (x : A) → ⟦ ϕ (lift x) ⟧′ X o

data μ′ {I O}(Σ : IF′ (I ⊎ O) O)(X : Pow′ I) : Pow′ O where
  ⟨_⟩′ : ⟦ Σ ⟧′ (X ⟨⊎⟩ μ′ Σ X) ⊆ μ′ Σ X
```

Here are examples using of the "lower" universe:

```
′W : (A : Set)(B : A → Set) → IF′ (⊥ ⊎ ⊤) ⊤
′W A B = σ′ A λ x → π′ (B x) λ _ → ψρ′ (inj₂ _)
```

```
W : (A : Set)(B : A → Set) → Set
W A B = μ′ (′W A B) (λ ()) tt

sup : {A : Set}{B : A → Set}(x : A)(f : B x → W A B) → W A B
sup x f = ⟨ x , f ⟩′
```

Using the "upper" universe with raised set levels results in some manual lifting and lowering however.

```
′Vec : FF ⊤ ℕ
′Vec = σ (Lift Bool) λ b → if lower b
  then η ff 0
  else σ (Lift ℕ) (λ m → ψ ff _ * ρ ff (lower m) * η ff (suc (lower m)))

Vec : Set → ℕ → Set₁
Vec A n = μ ′Vec (const (Lift A)) n

[] : ∀ {A} → Vec A zero
[] = ⟨ lift true , lift refl ⟩

_::_ : ∀ {A n} → A → Vec A n → Vec A (suc n)
x :: xs = ⟨ lift false , lift _ , lift x , xs , lift refl ⟩
```

7.1 Discussion

In the vector example manual lifting and lowering is tedious, but this is a more fundamental problem (universe cumulativity), which should be addressed independently of what we are working on (code duplication).

There is nothing that stops us from just having a "lower" and an "upper" universe, we can use universe polymorphism to make an infinite hierarchy of universes. Our "upper" universe which lives in $\texttt{Set}_2$ can be described by in a universe living in $\texttt{Set}_3$ and so on.

The described universes do not have the class index, and are thus only `IF`, this can probably be fixed, due to lack of time I have not tried.

Chapter 8

Ornaments

Ornaments [McB11] is another recent technique for dealing with code duplication that is based on the universe construction.

It comes in two parts. The first is called *decoration*, it allows us to describe datatypes incrementally, for example the list datatype is the datatype of natural numbers with a decorated `suc` constructor:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data List (A : Set) : Set where
  zero : List A
  suc  : A → List A → List A
```

Clearly the two datatypes are very similar, the difference is that we have a parameter call in the cons constructor – it is this difference that decorations enable us to express.

What is the benefit? If we know the extra structure (the difference) that, for example, a list has compared to a natural number it seems plausible that we should be able to erase or forget this extra structure. This forgetful function is a generic function that works on all decorations. In the instance, considered above, when the decoration is the list datatype the forgetful function is the length function. The second part of ornaments is *refinement*. Refinement allows us to refine an index of a datatype using an algebra. For instance, the vector datatype is a list (which can be thought of as trivially indexed) refined with the length algebra.

```
data List (A : Set) : ⊤ → Set where
  []  : List A tt
  _::_ : A → List A tt → List A tt

data Vec (A : Set) : ℕ → Set where
```

```
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Recall that we can get the length function (and also the length algebra) for free when we decorate naturals to get lists, hence using ornaments we get vectors for free.

We also get a generic remembering function which can be used to recompute a forgotten index. In the instance of vectors it gives us a way to go from lists to vectors.

We also get a general theorem which says that if we refine some datatype using some algebra, then the refined datatype's index will have the same value as the result of first forgetting the refined index and then iterating the algebra over the plain datatype.

The specific instance of this theorem concerning lists and vectors reads:

```
corollary : (xs : Vec A n) → length (forget xs) ≡ n
```

In a sense the general theorem is just a sanity check; it must hold if we implemented ornaments having the properties we outline above.

In summary ornaments help us when dealing with indexed datatypes by allowing us to describe them in an incremental fashion using decorations and refining using algebras, they also give us a generic forgetful function for decorations and refinements and a generic remember function for recomputing forgotten refinements and generic theorems relating these functions.

We will not give a full implementation of ornaments here, as it is already given in [McB11]. Instead we shall only focus on explaining and implementing the decoration part and show how it can be adopted to our class indexed universe.

8.1 Decoration

We start with a simple universe, which has a code for Σ.

```
data Sig : Set₁ where
  ϵ ρ : Sig
  σ   : (A : Set)(ϕ : A → Sig) → Sig

⟦_⟧ : Sig → (Set → Set)
⟦ ϵ       ⟧ X = ⊤
⟦ ρ       ⟧ X = X
⟦ σ A ϕ   ⟧ X = Σ A λ x → ⟦ ϕ x ⟧ X

data μ (Σ : Sig) : Set where
  ⟨_⟩ : ⟦ Σ ⟧ (μ Σ) → μ Σ
```

We then define another universe – the decorative universe – which has the old as its index. It has the same constructors as the old one modulo Δ, for difference.

```
data Deco : Sig → Set₁ where
  ϵ : Deco ϵ
  ρ : Deco ρ
  σ : (A : Set){Σ : A → Sig} → ((a : A) → Deco (Σ a)) → Deco (σ A Σ)
  Δ : (A : Set){Σ : Sig} → (A → Deco Σ) → Deco Σ
```

We will decode the decorative universe into the plain one:

```
⌊_⌋ : ∀ {Σ} → Deco Σ → Sig
⌊ ϵ     ⌋ = ϵ
⌊ ρ     ⌋ = ρ
⌊ σ A ϕ ⌋ = σ[ a : A ] ⌊ ϕ a ⌋
⌊ Δ A ϕ ⌋ = σ[ a : A ] ⌊ ϕ a ⌋
```

To define the signature for natural numbers we use a dependently typed eliminator for booleans.

```
zero↦_suc↦_ : ∀ {ℓ}{P : Bool → Set ℓ} → P true → P false
            → (b : Bool) → P b
(zero↦ t suc↦ f) true  = t
(zero↦ t suc↦ f) false = f

′ℕ : Sig
′ℕ = σ Bool (zero↦ ϵ
             suc↦ ρ)

ℕ : Set
ℕ = μ ′ℕ

zero : ℕ
zero = ⟨ true , _ ⟩

suc : ℕ → ℕ
suc n = ⟨ false , n ⟩
```

Using the decorative universe we can now state that lists are naturals with decorated successor constructors:

```
′List : (A : Set) → Deco ′ℕ
′List A = σ Bool (zero↦ ϵ
                  suc↦ Δ A (λ _ → ρ))

List : (A : Set) → Set
List A = μ ⌊ ′List A ⌋

[] : ∀ {A} → List A
[] = ⟨ true , _ ⟩
```

```
_::_ : ∀ {A}(x : A)(xs : List A) → List A
x :: xs = ⟨ false , x , xs ⟩
```

In order to define the forgetful function we need iteration.

```
map : ∀ {X Y}(Σ : Sig)(f : X → Y) → ⟦ Σ ⟧ X → ⟦ Σ ⟧ Y
map ϵ       f _       = _
map ρ       f x       = f x
map (σ A ϕ) f (a , s) = a , map (ϕ a) f s

iter : ∀ {X}(Σ : Sig) → (⟦ Σ ⟧ X → X) → μ Σ → X
iter Σ ϕ ⟨ s ⟩ = ϕ (map Σ (iter Σ ϕ) s)
```

Forget removes the extra structure imposed by the Δ, otherwise it leaves things as they were.

```
forget″ : ∀ {Σ X}(D : Deco Σ) → ⟦ ⌊ D ⌋ ⟧ X → ⟦ Σ ⟧ X
forget″ ϵ       x       = x
forget″ ρ       x       = x
forget″ (σ A ϕ) (a , s) = a , forget″ (ϕ a) s
forget″ (Δ A ϕ) (a , s) =     forget″ (ϕ a) s

forget′ : ∀ {Σ}(D : Deco Σ) → ⟦ ⌊ D ⌋ ⟧ (μ Σ) → μ Σ
forget′ D x = ⟨ forget″ D x ⟩

forget : ∀ {Σ}(D : Deco Σ) → μ ⌊ D ⌋ → μ Σ
forget {Σ} D x = iter ⌊ D ⌋ (forget′ D) x
```

We end with a simple test.

```
length : ∀ {A} → List A → ℕ
length {A} = forget (′List A)

#0 = zero
#1 = suc #0
#2 = suc #1
#3 = suc #2

test : length (#0 :: #1 :: #2 :: []) ≡ #3
test = refl
```

Next we will decorate our class indexed universe from previous chapters. The main difference is the class index and the proof that it is what we need it to be, namely `ff`.

```
data Deco {I O} : (c : Class) → Sig (I ⊎ O) O c → c ≡ ff → Set₁ where
  ψρ  : (i : I ⊎ O) → Deco ff (ψρ _ i) refl
  η   : (o : O) → Deco ff (η _ o) refl
  _+_ : ∀ {Σ Σ′}(O  : Deco ff Σ  refl)
```

```
                (O′ : Deco ff Σ′ refl) → Deco ff (Σ + Σ′) refl
  _*_ : ∀ {Σ Σ′}(O  : Deco ff Σ  refl)
                (O′ : Deco ff Σ′ refl) → Deco ff (Σ * Σ′) refl
  σ   : ∀ A {Σ}(O : (x : A) → Deco ff (Σ x) refl) → Deco ff (σ A Σ) refl
  Δ   : ∀ {Σ}(A : Set)(O : A → Deco ff Σ refl)    → Deco ff Σ refl

⌊_⌋ : ∀ {I O c}{p : c ≡ ff}{Σ : Sig (I ⊎ O) O c} → Deco c Σ p → FF I O
⌊ ψρ i   ⌋ = ψρ _ i
⌊ η o    ⌋ = η _ o
⌊ O + O′ ⌋ = ⌊ O ⌋ + ⌊ O′ ⌋
⌊ O * O′ ⌋ = ⌊ O ⌋ * ⌊ O′ ⌋
⌊ σ A O  ⌋ = σ _ λ x → ⌊ O x ⌋
⌊ Δ A O  ⌋ = σ A λ x → ⌊ O x ⌋
```

We need to do it this way for the same reason as previously stated in order to be able to define functions by pattern-matching on it.

```
forget″ : ∀ {I O c}{X : Pow I}{Σ : Sig (I ⊎ O) O c}(p : c ≡ ff){Y : O → Set}
           (O : Deco c Σ p) → ⟦ ⌊ O ⌋ ⟧ (X ⟨⊎⟩ Y) ⊆ (⟦ Σ ⟧ (X ⟨⊎⟩ Y))
forget″ ._ (ψρ (inj₁ i)) x        = x
forget″ ._ (ψρ (inj₂ o)) y        = y
forget″ ._ (η i)         refl     = refl
forget″ ._ (Σ + Σ′)      (inj₁ s) = inj₁ (forget″ refl Σ  s)
forget″ ._ (Σ + Σ′)      (inj₂ t) = inj₂ (forget″ refl Σ′ t)
forget″ ._ (Σ * Σ′)      (s , t)  = forget″ refl Σ s , forget″ refl Σ′ t
forget″ ._ (σ A O)       (a , s)  = a , forget″ refl (O a) s
forget″ ._ (Δ A O)       (a , s)  =     forget″ refl (O a) s

forget′ : ∀ {I O}{X : Pow I}{Σ : FF I O}(O : Deco ff Σ refl)
       → Alg ⌊ O ⌋ X (μ Σ X)
forget′ O s = ⟨ forget″ refl O s ⟩

forget : ∀ {I O}{X : Pow I}{Σ : FF I O}(O : Deco ff Σ refl)
       → μ ⌊ O ⌋ X ⊆ (μ Σ X)
forget O = iter _ (forget′ O)

′ℕ : FF ⊥ ⊤
′ℕ = η ff _ + ρ ff _

ℕ = μ ′ℕ ⊥-elim _

zero : ℕ
zero = ⟨ inj₁ refl ⟩

suc : ℕ → ℕ
suc n = ⟨ inj₂ n ⟩
```

```
ListD : Set → Deco ff ′ℕ refl
ListD A = η _ + Δ A λ _ → ψρ _

′List : Set → FF ⊥ ⊤
′List A = ⌊ ListD A ⌋

List : Set → Set
List A = μ (′List A) ⊥-elim _

[] : ∀ {A} → List A
[] = ⟨ inj₁ refl ⟩

_::_ : ∀ {A} → A → List A → List A
x :: xs = ⟨ inj₂ (x , xs) ⟩

length : ∀ {A} → List A → ℕ
length = forget (ListD _)

#0 = zero
#1 = suc #0
#2 = suc #1
#3 = suc #2

test : length (#0 :: #1 :: #2 :: []) ≡ #3
test = refl
```

8.2 Discussion

The refinement part of ornaments is not implementable for our class indexed universe for reasons I do not fully understand. It seems that, from the little testing I have done, for a universe to admit refinement it needs the "record-like" codes rather than the algebraic ones (we discussed the difference in chapter 5). Speaking against this is the fact that [AJG10] are able to define refinement for a set of codes similar to ours, albeit in category theory (extensional) rather than (intensional) type theory.

When we decorate natural numbers to get lists, the parameter we add to the successor constructor is not really a parameter in the sense we worked with earlier. It is not part of the fixpoint, but rather just passed along to the description; this makes it impossible for a generic mapping function to target. An interesting question is if we can construct a decorative universe in which true parameters can be added.

Chapter 9

Conclusion and further work

We have proposed a solution to the problem of having to define the same function several times that arises when several universes are used in datatype-generic programming. The solution using algebraic codes suffers from the need of special language support to infer the class of the codes. Using the "record-like" codes we seem to be able to get further without special language support.

The natural next step would be to push the solution using "record-like" codes further and see if it breaks. It is not clear to me if using the "record-like" codes is a serious restriction.

After that it would be interesting to see if the universe could be extended to handle finite, inductive-inductive and inductive-recursive datatypes, datatypes with nested fixpoints and coinductive datatypes.

An important aspect that has been left out in this work is whether the class indexed universe is meaningful and consistent. As more classes of datatypes are added into it, such as inductive-inductive and inductive-recursive datatypes, this problem is likely to become harder.

A completely different approach to the several universes problem has been explained to me by Conor McBride. It involves so called deletion ornaments which lets us delete information from a datatype, provided we give a way to recover it in the forgetful function. The idea is to delete, for example, the code for π to get finitary indexed datatypes from the datatype of infinitary indexed ones and so on. Deletion ornaments are merely mentioned in [McB11] and have not been implemented. At this point it is not clear to me how all this would work.

Bibliography

[aC] The Programming Logic Group at Chalmers. Alfa. Web page. http://www.cse.chalmers.se/~hallgren/Alfa/.

[AFMS11] Thorsten Altenkirch, Fredrik Nordvall Forsberg, Peter Morris, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO 2011: Fourth International Conference on Algebra and Coalgebra in Computer Science*, 2011. http://cs.swan.ac.uk/~csfnf/papers/catsemIndind_calco2011.pdf.

[AJG10] Robert Atkey, Patricia Johann, and Neil Ghani. When is a Refinement Type an Inductive Type? Submitted, September 2010. http://personal.cis.strath.ac.uk/~raa/inductive-refinement.pdf.

[AM09] Thorsten Altenkirch and Peter Morris. Indexed Containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS)*, 2009. http://www.cs.nott.ac.uk/~txa/publ/ICont.pdf.

[AMM07] Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic Programming with Dependent Types. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007. http://www.cs.nott.ac.uk/~txa/publ/ssgp06.pdf.

[AR92] William Aitken and John Reppy. Abstract value constructors: Symbolic constants for standard ML. Technical Report TR 92-1290, Department of Computer Science, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[BC05] Ana Bove and Venanzio Capretta. Modelling General Recursion in Type Theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005. http://www.cs.ru.nl/~venanzio/publications/General_Recursion_MSCS_2005.pdf.

[BD08] Ana Bove and Peter Dybjer. Dependent Types at Work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *LerNet ALFA Summer School*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer, 2008. `http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf`.

[BDJ03] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing*, 10(4):265–289, 2003. `http://www.cse.chalmers.se/~patrikj/poly/gendt/`.

[CDMM10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The Gentle Art of Levitation. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 3–14. ACM, 2010. `http://personal.cis.strath.ac.uk/~dagand/papers/levitation.pdf`.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *ICFP*, pages 143–156. ACM, 2008. `http://adam.chlipala.net/papers/PhoasICFP08/PhoasICFP08.pdf`.

[DS99] Peter Dybjer and Anton Setzer. A Finite Axiomatization of Inductive-Recursive Definitions. In Jean-Yves Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. `http://www.cse.chalmers.se/~peterd/papers/Finite_IR.pdf`.

[DS01] Peter Dybjer and Anton Setzer. Indexed Induction-Recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001. `http://www.cse.chalmers.se/~peterd/papers/Indexed_IR_Dagstuhl.pdf`.

[DS03] Peter Dybjer and Anton Setzer. Induction-Recursion and Initial Algebras. *Annals of Pure and Applied Logic*, 124(1–3):1–47, 2003. `http://www.cse.chalmers.se/~peterd/papers/InductionRecursionInitialAlgebras.pdf`.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *J. Log. Algebr. Program.*, 66(1):1–49, 2006. `http://www.cse.chalmers.se/~peterd/papers/Indexed_IR.pdf`.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991. `http://www.cse.chalmers.se/~peterd/papers/Setsem_Inductive.pdf`.

[Dyb97] Peter Dybjer. Representing Inductively Defined Sets by Wellorderings in Martin-Löf's Type Theory. *Theoretical Computer Science*, 176(1–2):329–335, 1997. http://www.cse.chalmers.se/~peterd/papers/Wellorderings.pdf.

[Dyb00] Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.*, 65(2):525–549, 2000. http://www.cse.chalmers.se/~peterd/papers/Inductive_Recursive.pdf.

[FS10] Fredrik Nordvall Forsberg and Anton Setzer. Inductive-Inductive Definitions. In Anuj Dawar and Helmut Veith, editors, *CSL 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer, Heidelberg, 2010. http://cs.swan.ac.uk/~csfnf/induction-induction/.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006. http://www.strictlypositive.org/goguen.pdf.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000. http://www.cse.chalmers.se/~patrikj/poly/polythesis/.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP - A Polytypic Programming Language. In *POPL*, pages 470–482, 1997. http://www.cse.chalmers.se/~patrikj/poly/polypPOPL97/.

[LP92] Zhaohui Luo and Randy Pollack. LEGO Proof Development Systems: User's Manual. Technical report, University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992. http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211/.

[McB11] Conor McBride. Ornamental Algebras, Algebraic Ornaments. Draft, Jan 2011. http://personal.cis.strath.ac.uk/~conor/pub/OAAO/LitOrn.pdf.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. http://intuitionistic.files.wordpress.com/2010/07/martin-lof-tt.pdf.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. http://strictlypositive.org/view.ps.gz.

[Mor07] Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, Nottingham University, November 2007. `http://www.cs.nott.ac.uk/~pwm/thesis.pdf`.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. `http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf`.

[Nor08] Ulf Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008. `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`.

[PR98] Holger Pfeifer and Harald Rueß. Polytypic Abstraction in Type Theory. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming (WGP'98)*. Department of Computing Science, Chalmers University of Technology, and Göteborg University, June 1998. `http://www.informatik.uni-ulm.de/ki/Pfeifer/polytypic-abstraction-wgp98.html`.

[Tea11] The Agda Team. The Agda Wiki. Web page, 2011. `http://wiki.portal.chalmers.se/agda/`.