Leveling Up Dependent Types

Generic programming over a predicative hierarchy of universes.

Larry Diehl Tim Sheard
Portland State University
{ldiehl,sheard}@cs.pdx.edu

Abstract

Generic programming is about writing a single function that does something different for each type. In most languages one cannot case over the structure of types. So in such languages generic programming is accomplished by defining a universe, a data structure isomorphic to some subset of the types supported by the language, and performing a case analysis over this datatype instead. Such functions support a limitied level of genericity, limited to the subset of types that the universe encodes. The key to full genericity is defining a rich enough universe to encode all types in the language.

In this paper we show how to define a universe with a predicative hierarchy of types, encoding a finite set of base types (including dependent products and sums), and an infinite set of user defined datatypes. We demonstrate that such a system supports a much broader notion of generic programming, along with a serendipitous extension to the usefulness of user defined datatypes with existential arguments.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

Keywords generic programming, dependent types, universes

1. Introduction

Consider the task of writing a generic `show` function for every type defined in a dependently typed programming language. By *generic definition*, we mean a definition that does something different depending on the type of the value being shown. Such a function might have the type:

```
show : (A : Set) → A → String
```

This type signature is parametrically polymorphic. In languages such as AGDA (or HASKELL), one cannot perform case analysis on objects like `A` classified by `Set` because they are types (not values) and this prevents generic definitions.

This paper is about a system with two important properties:

1. It makes programming *fully-generic functions*, like `show`, possible over all datatypes representable in the system.
2. It makes programming with *existentially quantified datatypes* [1] more useful. This is because the generic functions from 1. can be applied to existentially quantified values.

Our system is dependently typed over a predicative hierarchy of universes, and supports user-extensible types. We describe our system in several stages, starting very simply and informally, building in several steps, to a completely formal mechanized presentation in the dependently typed language AGDA [Norell 2007]. [2]

We present our system as a dependently typed *universe* [Martin-Löf 1984]. A universe is made of a code for types (a data structure representing types), and a meaning function that denotes each code as a `Set` in AGDA. A universe acts as a denotational semantics for the types of a language — using the metalanguage AGDA as the model — into which each type is denoted. Universes are explained in detail in the background Section 2.

Using a universe in a dependently typed language to perform generic programming is common [Benke et al. 2003; Morris et al. 2006; Altenkirch et al. 2007; Chapman et al. 2010; Weirich and Casinghino 2010]. However, typically the universe being modelled requires the metalanguage to express generic functions over it. More concretely, the type signatures of generic functions written over typical universes cannot be expressed in the language of types that the universe encodes. In this sense, the generic functions in such universes are *external*, or outside of the language model. In contrast, the type signatures for generic functions that we present *can* be represented by the types that the universe encodes. Thus, the generic functions we present are *internal* to the language model of our universe. The universe we present in this paper supports generic functions which are both internal and fully-generic.

The remainder of the paper is organized as follows:

- *Section 2* Reviews background material on generic dependently typed programming with universes.
- *Section 3* Presents an informal but intuitive definition of an AGDA model of a language with a predicative hierarchy of universes and a fixed collection of types.
- *Section 4* Defines two fully-generic functions: `showType` returns a string for all types of our language, and `show` returns a string for all values of our language.
- *Section 5* Reviews an existing formal AGDA model of a language with a predicative hierarchy of universes and a fixed collection of types. This formal model is a simplification of the model McBride [2011] gives for EPIGRAM 2 [McBride 2005], and the theory behind it is due to Palmgren [1998]. In the formal

[1] By existentially quantified datatype we mean a datatype with a large dependent pair as an argument, i.e. `(A , a) : 'Σ Set (λ A → ...)`

[2] The complete AGDA source code for this development can be accessed at https://github.com/larrytheliquid/leveling-up

model the fully-generic but external definitions from Section 4 can be internalized.

- *Section 6* Extends our model with the existentially quantified type `HList` of heterogeneous lists. The standard definition of `HList` contains a large constructor argument, preventing fully-generic functions from being defined if this large argument is `Set`. However, in our extended universe the large argument is a type of our object language, rather than a type of the metalanguage. This makes it possible to write fully-generic (and internal) functions over `HList` values.
- *Section 7* Extends our model with the machinery of Chapman et al. [2010], which makes datatype definitions first-class. After this change, the model supports user-extensible datatype definitions, rather than being limited to a fixed collection of types. The standard definition of the type of datatype definitions `Desc` contains the large argument `Set`. However, we repeat the technique from Section 6 to turn this large argument into a type of our object language. This makes it possible to write fully-generic (and internal) functions over user-defined datatypes.
- *Sections 8, 9, and 10* Introduce a series of fully generic definitions, each successive definition expressive over a larger subset of the universe of types. It also categorizes similarities between all our generic definitions, and introduces abtractions which capture some of the complexity of defining generic functions. These abstractions provide a reusable interface for writing fully generic definitions.

Our major contributions are:

- A model that combines the formal universe hierarchy model by McBride [2011] with the user-extensible datatypes model by Chapman et al. [2010]. Each of these earlier models has limitations. The limitation of the formal universe hierarchy[McBride 2011] is the inability to add new datatypes without modifying previous fully-generic functions. The limitation of the user-extensible datatypes model[Chapman et al. 2010] is not being able to write fully-generic functions over datatypes with previously defined types as constructor arguments. Our new model (comprised of two parts `TypeForm` and `DescForm`, both abstracted over a universe) over comes both of these limitations.
- Two separate interfaces (`Generic` and `GenericΔ`) that can be implemented to write fully-generic functions. These interfaces are formally presented as dependent records. They capture the similarities between the various fully-generic functions that we explore in this paper.

Finally, we would like to point out that our langauge model and functions over it are given in AGDA without turning off any features (i.e. positivity checking) or postulating any definitions (i.e. functional extensionality). As such, the type system underlying the model we present is an extension of Martin-Löf's [1975] predicative type theory. Specifically, the type theory is extended with types and semantics supporting datatype descriptions, and then closed to support the addition of an eliminator for types. The source code linked from this paper contains a formalization of the terms of our model, as an extension of McBride's [2010] dependent type-safe syntax and evaluation formalization.

2. Background on generic programming

In dependently typed languages, generic functions are typically written with the aid of a Martin-Löf [1984] universe. A universe consists of a collection of codes (a data structure) representing types, and a meaning function mapping each code to the type it represents.

2.1 Example of a Martin-Löf universe

Below we define a very simple universe where the collection of codes is represented by the datatype named `Type`, whose constructors are the codes `‘Bool` and `‘Σ`. Our notational convention is to prefix each code with a backtick to distinguish it from the type it represents. The meaning function for this universe is ⟦_⟧. It maps the code for booleans `‘Bool` to the type for booleans `Bool`. It also maps the code for dependent pairs `‘Σ` to the type for dependent pairs `Σ`.

```
mutual
  data Type : Set where
    ‘Bool : Type
    ‘Σ : (A : Type) (B : ⟦ A ⟧ → Type) → Type

  ⟦_⟧ : Type → Set
  ⟦ ‘Bool ⟧ = Bool
  ⟦ ‘Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
```

Because this universe encodes dependent types, its codes are defined mutually with their interpretation. This allows `‘Σ`'s second parameter `B` to depend on the meaning of its first parameter `A`. The mutual definition of `Type` with ⟦_⟧ is said to be inductive-recursive [Dybjer and Setzer 1999, 2003].

2.2 Examples of generic functions

Once you have defined a universe you can write generic functions over the collection of types it represents by case-analyzing its codes. For example, `showType` is a generic function that returns a (sometimes incomplete[3]) `String` representation of each *type*.

```
showType : (A : Type) → String
showType ‘Bool = "Bool"
showType (‘Σ A B) = "Σ " ++ showType A ++ " λ"
```

You can also write generic functions over all the *values* in a universe by adding a parameter whose dependent type is the meaning function applied to the previous code parameter, e.g. `(A : Type) → ⟦ A ⟧ → ...`
Below we define the `show` function that returns a `String` representation of every *value* in the universe.

```
show : (A : Type) → ⟦ A ⟧ → String
show ‘Bool true = "true"
show ‘Bool false = "false"
show (‘Σ A B) (a , b)
  = show A a ++ " , " ++ show (B a) b
```

Notice the similarity between the type signature of `show` for this universe and the `show` presented at the beginning of the introduction.

```
show : (A : Set)  →   A  → String
show : (A : Type) → ⟦ A ⟧ → String
```

As we have seen, a generic definition of the second `show` is easy to write. However, it is restricted to a very limited universe of types. In the next section we expand this universe to encompass a larger collection of types, as well as a predicative hierarchy of levels.

3. Informal definition of universe hierarchy

Now we expand the collection of codes to represent all types of a dependently typed language with a fixed collection of base types and a *preliminary* and *informal* predicative hierarchy of universes. Once again, the codes and the meaning function of the universe are

[3] We return "λ" to represent the higher-order second argument to `‘Σ`.

defined mutually using induction-recursion. Below we give both definitions separately, without a `mutual` block, for presentational purposes.

The universe definition below is not strictly-positive [4] , but it provides intuition to help understand the *actual* and *formal* definition given in Section 5. The universe's fixed collection of base types consists of `‘⊥` `‘⊤`, `‘Bool`, `‘ℕ`, and `‘String`. The dependent type formers are `‘Π` and `‘Σ`. This much is standard. **One new constructor** is `‘Type`, the type of types in the previous level of the universe hierarchy. We add universe levels to our definition to remain predicative. Failing to do so makes the language inconsistent as a logic because `Type : Type` can be exploited to form paradoxes [Girard 1972; Hurkens 1995]. The **other new constructor** is `‘⟦_⟧`. It lifts a type from the previous level to the current level. This lifting is reflected in the type index of `‘⟦_⟧`, which must be the successor of the level of its argument.

```
data Type : ℕ → Set where
  ‘⊥ ‘⊤ ‘Bool ‘ℕ ‘String : {ℓ : ℕ} → Type ℓ
  ‘Π ‘Σ : {ℓ : ℕ}
    (A : Type ℓ) (B : ⟦ ℓ | A ⟧ → Type ℓ)
    → Type ℓ
  ‘Type : {ℓ : ℕ} → Type ℓ
  ‘⟦_⟧ : {ℓ : ℕ} → Type ℓ → Type (suc ℓ)
```

As a shorthand, we also define the code representing non-dependent functions as a special case of the code representing dependent functions.

```
_‘→_ : {ℓ : ℕ} (A B : Type ℓ) → Type ℓ
A ‘→ B = ‘Π A (const B)
```

`Type`'s meaning function (`⟦_|_⟧`) is defined below. The cases for `‘Type` and `‘⟦_⟧` are the most interesting. The meaning of `‘Type` depends on the universe level. At universe level zero no previous universe exists, so the meaning of `‘Type` is the empty type ⊥. At any successive universe level the meaning of `‘Type` is the datatype `Type` indexed by the previous universe.[5] The meaning of a type from the previous universe lifted into the current universe — using constructor `‘⟦_⟧` — is the same as the type's meaning in the previous universe.

```
⟦_|_⟧ : (ℓ : ℕ) → Type ℓ → Set
⟦ ℓ     | ‘⊥       ⟧ = ⊥
⟦ ℓ     | ‘⊤       ⟧ = ⊤
⟦ ℓ     | ‘Bool    ⟧ = Bool
⟦ ℓ     | ‘ℕ       ⟧ = ℕ
⟦ ℓ     | ‘String  ⟧ = String
⟦ ℓ     | ‘Π A B   ⟧ = (a : ⟦ ℓ | A ⟧) → ⟦ ℓ | B a ⟧
⟦ ℓ     | ‘Σ A B   ⟧ = Σ ⟦ ℓ | A ⟧ (λ a → ⟦ ℓ | B a ⟧)
⟦ zero  | ‘Type    ⟧ = ⊥
⟦ suc ℓ | ‘Type    ⟧ = Type ℓ
⟦ suc ℓ | ‘⟦ A ⟧   ⟧ = ⟦ ℓ | A ⟧
```

[4] We even call it informal because it does not pass Agda's positivity check. However, you might expect Agda's checker to eventually recognize the positivity of the definition, as it is possible to represent an isomorphic definition by our forthcoming encoding.

[5] This definition of `Type` is not strictly-positive because `⟦ suc ℓ | ‘Type ⟧` is defined to return itself, albeit at a lower index. The negative occurrence manifests in the `B` argument of the `‘Π` and `‘Σ` constructors. For example, `‘Σ` at universe `suc ℓ` is `‘Σ : (A : Type (suc ℓ)) (B : ⟦ suc ℓ | A ⟧ → Type (suc ℓ)) → Type (suc ℓ)`, which reduces to `‘Σ : (A : Type (suc ℓ)) (B : Type ℓ → Type (suc ℓ)) → Type (suc ℓ)`. The reduced value is not strictly-positive in `B`.

4. Generic definition of show

Now we give examples of generic functions that can be written in the universe of Section 3. We define `showType` and `show` as internalized generic functions, meaning their types can be represented by our universe.

4.1 Definition of showType

Below is a definition of `showType` for our expanded universe of Section 3. The type signature for the old `showType` is `(A : Type) → String`. This type signature cannot be represented by the types of the old universe, which only consists of `‘Bool` and `‘Σ`. Therefore, the old generic definition of `showType` is only definable within the metalanguage, but not within the object language that the universe models. However, our new universe has type codes for `‘Type` and `_‘→_`, so the function can be internalized by the type signature `‘Type ‘→ ‘String`.

```
showType : (ℓ : ℕ) → ⟦ suc ℓ | ‘Type ‘→ ‘String ⟧
showType ℓ ‘⊥ = "⊥"
showType ℓ ‘⊤ = "⊤"
showType ℓ ‘Bool = "Bool"
showType ℓ ‘ℕ = "ℕ"
showType ℓ ‘String = "String"
showType ℓ (‘Π A B) = "Π " ++ showType ℓ A ++ " λ"
showType ℓ (‘Σ A B) = "Σ " ++ showType ℓ A ++ " λ"
showType ℓ ‘Type = "Type"
showType zero ‘⟦ () ⟧
showType (suc ℓ) ‘⟦ A ⟧ = "⟦ " ++ showType ℓ A ++ " ⟧"
```

The interesting cases of `showType` are the cases for the universe lifting operator `‘⟦_⟧`. These cases must case analyze the universe level. If the universe level is zero, then the collection of types of the previous universe level is the empty type ⊥, so there is nothing to show. Empty parentheses `()` is special AGDA syntax indicating that a value at that location cannot be inhabited. If the the universe level is the successor of another level, then we are justified to show the type of the previous universe level because the level and type get smaller in the recursive call.

4.2 Definition of show

Now we are ready to define `show` generically. Briefly compare the type signature of show presented at the beginning of the introduction to this final version.

```
show : (A : Set) → A → String
show : (ℓ : ℕ)
  → ⟦ suc ℓ | ‘Π ‘Type (λ A → ‘⟦ A ⟧ ‘→ ‘String) ⟧
```

The first argument of the dependent function is the type to be shown. The `λ` binds `A` to a value of `‘Type`, which is a type in the previous universe in the hierarchy. We use `‘⟦_⟧` in the second argument of the function, lifting `A` from the previous universe into the current universe.

Compared to `showType`, the dependently typed `show` function has an extra argument requiring not only a type but a value of that type. Notice that while the type of `‘Σ` could not show its higher order second argument, the dependent pair value `(a , b)` can be shown completely.

The cases for values of type `‘Type` and `‘⟦_⟧` are the most interesting. If the type is `‘Type`, then the values are elements of the datatype `Type` (from the previous universe level). If the universe level is zero, then there can be no such types. However, if the universe level is the successor of another level, then the value is an element of the datatype `Type`, and we can show it with `showType`.

If the type is `‘⟦_⟧`, then the values are values of the previous universe level. Again, if the universe level is zero then there are no

such values. But if the the universe level is the successor of another level then we can show the lower-level value using a recursive call that is justified by the universe level and type decreasing. To indicate that the value shown is from a previous universe level and has been lifted to the current level, we return the string "lift" along with the actual value.

```
show : (ℓ : ℕ)
  → ⟦ suc ℓ ∣ ‘Π ‘Type (λ A → ‘⟦ A ⟧ ‘→ ‘String) ⟧
show ℓ ‘⊥ ()
show ℓ ‘⊤ tt = "tt"
show ℓ ‘Bool true = "true"
show ℓ ‘Bool false = "false"
show ℓ ‘ℕ zero = "zero"
show ℓ ‘ℕ (suc n) = "suc " ++ show ℓ ‘ℕ n
show ℓ ‘String str = "\"" ++ str ++ "\""
show ℓ (‘Π A B) f = "λ"
show ℓ (‘Σ A B) (a , b) =
  show ℓ A a ++ " , " ++ show ℓ (B a) b
show zero ‘Type ()
show (suc ℓ) ‘Type A = showType ℓ A
show zero ‘⟦ () ⟧ a
show (suc ℓ) ‘⟦ A ⟧ a = "lift " ++ show ℓ A a
```

5. Formal definition of universe hierarchy

Now we replace the informal but intuitive universe definition from Section 3 with a formal definition. In fact we have not been using a single universe, but a family of universes indexed by the natural numbers — representing levels in the predicative hierarchy — each one containing the entire previous universe. The construction we are about to describe is due to Palmgren [1998]. We would like to thank McBride [2011] for making us aware of its mechanization, a simplication of which is presented below. Our simplication drops an explicit notion of equality from the universe hierarchy model. The construction and its mechanization are not novel, we merely seek to popularize them so that their utility for generic programming becomes obvious. The construction is presented in two parts.

Universe In the first part we avoid defining `Type` directly and instead define `TypeForm`, a type parameterized by a universe. A `Universe` is a dependent record containing the collection of `Codes` and the `Meaning` function dependent on those codes.

```
record Universe : Set₁ where
  field
    Codes : Set
    Meaning : Codes → Set
```

TypeForm `TypeForm` is similar in structure to our informal `Type` from Section 3. The differences are the types of the constructors `‘⟦_⟧` and `‘Type`. The lift constructor `‘⟦_⟧` took a code from the previous universe as an argument. The type of this argument now becomes `(Codes U)` rather than `(Type ℓ)`. The same substitution is made when defining the meaning of the constructor `‘Type`. Significantly, `(Codes U)` is a type parameter rather than a recursive occurrence of `(Type ℓ)`, making this new definition strictly positive. The meaning of `‘⟦ A ⟧` is defined as the meaning function of the previous universe applied to the code `A` of the previous universe. Now it is represented using `(Meaning U A)` rather than ⟦ ℓ ∣ A ⟧.

```
mutual
  data TypeForm (U : Universe) : Set where
    ‘⊥ ‘⊤ ‘Bool ‘ℕ ‘String ‘Type : TypeForm U
    ‘Π ‘Σ : (A : TypeForm U)
      (B : ⟦ U / A ⟧ → TypeForm U)
      → TypeForm U
    ‘⟦_⟧ : Universe.Codes U → TypeForm U

  ⟦_/_⟧ : (U : Universe) → TypeForm U → Set
  ⟦ U / ‘⊥ ⟧ = ⊥
  ⟦ U / ‘⊤ ⟧ = ⊤
  ⟦ U / ‘Bool ⟧ = Bool
  ⟦ U / ‘ℕ ⟧ = ℕ
  ⟦ U / ‘String ⟧ = String
  ⟦ U / ‘Π A B ⟧ = (a : ⟦ U / A ⟧) → ⟦ U / B a ⟧
  ⟦ U / ‘Σ A B ⟧ = Σ ⟦ U / A ⟧ (λ a → ⟦ U / B a ⟧)
  ⟦ U / ‘Type ⟧ = Universe.Codes U
  ⟦ U / ‘⟦ A ⟧ ⟧ = Universe.Meaning U A
```

The definition of `TypeForm` is strictly-positive because it is parametrically defined over an arbitrary universe record. `TypeForm` could be instantiated with any universe, but we will always instantiate it with the previous universe.

Level Below is a function that computes the universe we have in mind from a natural number representing the level in the hiearchy.

```
Level : (ℓ : ℕ) → Universe
Level zero    = record { Codes = ⊥; Meaning=case⊥ }
Level (suc ℓ) = record { Codes = TypeForm (Level ℓ)
                       ; Meaning = ⟦_/_⟧ (Level ℓ) }
```

Type Finally, we can define the intended versions of `TypeForm` and the meaning function that are parameterized by a natural number instead of a universe. The universe parameters are instantiated with an appropriate `Level` computed from the natural number.

```
Type : ℕ → Set
Type ℓ = TypeForm (Level ℓ)

⟦_∣_⟧ : (ℓ : ℕ) → Type ℓ → Set
⟦ ℓ ∣ A ⟧ = ⟦ Level ℓ / A ⟧
```

The type signatures of these wrapper functions match the type signatures given in Section 3. Additionally, all of the examples in Section 4 work with our new definitions.

6. Definition of heterogeneous lists

Section 4 shows how our universe of types makes it possible to write generic functions like `showType` and `show`. In this section we show our universe also makes existentially quantified datatypes more attractive to programmers. We use the type of heterogeneous lists as an example of an existentially quantified type.

6.1 Definition of HList

First we consider the type of heterogeneous lists defined as an ordinary AGDA datatype. Unlike the type `List`, `HList` is not parametric in the type of values it stores. Instead, the `cons` constructor takes the type of the value it extends the list with as an argument.

In order for this to be a well-founded definition, the type of `HList` is $\texttt{Set}_1$. In AGDA, the type `Set` is shorthand for $\texttt{Set}_0$. Thus, the type of `HList` is one universe level higher than the type of the existential quantifier argument `A` in the `cons` constructor. We could make this definition universe polymorphic using AGDA universe levels, but we use this version instead for simplicity.

```
data HList : Set₁ where
  nil : HList
  cons : (A : Set) → A → HList → HList

myHList : HList
myHList = cons Bool false (cons ℕ 7 nil)
```

The value `myHList` demonstrates that there is no problem introducing `HList` values. We can even define a map function for `HList`.

```
mapHList : (Σ Set (λ A → A) → Σ Set (λ A → A))
  → HList → HList
mapHList f nil = nil
mapHList f (cons A a xs) =
  let B , b = f (A , a) in
  cons B b (mapHList f xs)
```

The problem occurs when we try to use `mapHList` by partially applying it to a mapping function. There are very few useful functions with the right type. It is possible to write many functions that return a constant value, like `(Bool , true)`. However, in AGDA the identity function is the only thing that we could write that actually uses the type argument to the function. This is because the first argument is a `Set`, which we cannot case analyze. Luckily for us, we have faced this problem before when we thought about writing `show : (A : Set) → A → String` in the introduction.

6.2 Definition of HListForm

Existentially quantified datatypes like `HList` are normally avoided when programming in AGDA because it is not possible to write any interesting functions operating on them. In fact, the existential argument `(A : Set) → A → ...` is usually replaced with a code and value of some universe that constrains the possible types that the list can hold. However, unlike AGDA's `Set`, our `Type` supports case analysis, so it is not necessary to constrain the universe.

HListForm Below is the datatype of heterogeneous lists, but we have parameterized it by a universe and called it `HListForm`. This is analogous to the definition of `TypeForm` from Section 5, which is also parameterized by a universe. Recall that in the formal definition of `TypeForm`, a `Universe.Codes U` argument represents a type at a previous universe level. Thus, the type stored in an `HListForm` is one level below the `HListForm` itself. This same stratification technique occurred in the previous definition of `HList`, but using the universe hierarchy of the metalanguage AGDA.

```
data HListForm (U : Universe) : Set where
  nil : HListForm U
  cons : (A : Universe.Codes U)
    → Universe.Meaning U A
    → HListForm U → HListForm U
```

6.3 Adding HList to our universe

Because we defined `HListForm` as a type parameterized by a universe, and replaced occurrences of the large argument `Set` with codes of the universes, we are able to add `‘HList` to our fixed universe of types.

```
mutual
  data TypeForm (U : Universe) : Set where
    {- ... previous constructors ... -}
    ‘HList : TypeForm U

  ⟦_/_⟧ : (U : Universe) → TypeForm U → Set
  {- ... previous cases ... -}
  ⟦ U / ‘HList ⟧ = HListForm U
```

Notice that the type constructor `‘HList` does not have a universe argument `U`. This is because a user of our universe should only need to think about the high-level type `‘HList`. We formalize an `‘HList` at a particular universe level with `HListForm` behind the scenes.

6.4 Applying generic functions to an HList

Even though `cons` of `‘HList` is an existentially quantified value, we instantiate the quantifier with a `Type` that we may case analyze. Thus, now it is possible to map generic functions over heterogeneous lists. This opens up new opportunities for data representation in dependently typed programs, because we can play the same trick for any existentially quantified datatype.

```
‘mapHList : (ℓ : ℕ) → ⟦ suc ℓ ⟧
  (‘Σ ‘Type ‘⟦_⟧ ‘→ ‘Σ ‘Type ‘⟦_⟧)
  ‘→ ‘HList ‘→ ‘HList ⟧
‘mapHList ℓ f nil = nil
‘mapHList ℓ f (cons A a xs) =
  let B , b = f (A , a) in
  cons B b (‘mapHList ℓ f xs)

‘myHList : (ℓ : ℕ) → ⟦ suc ℓ ⟧ ‘HList ⟧
‘myHList ℓ = cons ‘Bool false (cons ‘ℕ 7 nil)

‘myHListShown : (ℓ : ℕ) → ⟦ suc ℓ ⟧ ‘HList ⟧
‘myHListShown ℓ = ‘mapHList ℓ
  (λ { (A , a) → ‘String , show ℓ A a  })
  (‘myHList ℓ)
```

Above are the fruits of our labor, as witnessed by `‘myHListShown`. This demonstrates how we can map a generic `show` function across all values of a heterogeneous list containing any `Type` in our language.

In order to show heterogeneous lists, we also need to extend the `showType` and `show` functions to handle the extra cases. This is accomplished by defining a helper function `showHList`.

```
showType : (ℓ : ℕ) → ⟦ suc ℓ ⟧ ‘Type ‘→ ‘String ⟧
{- ... previous cases ... -}
showType ℓ ‘HList = "Hlist"

mutual
  show : (ℓ : ℕ)
    → ⟦ suc ℓ ⟧ ‘Π ‘Type (λ A → ‘⟦ A ⟧ ‘→ ‘String) ⟧
  {- ... previous cases ... -}
  show ℓ ‘HList xs = showHList ℓ xs

  showHList : (ℓ : ℕ)
    → ⟦ ℓ ⟧ ‘HList ‘→ ‘String ⟧
  showHList ℓ nil = "nil"
  showHList zero (cons () a xs)
  showHList (suc ℓ) (cons A a xs) =
    "cons " ++ showType ℓ A ++ " " ++ show ℓ A a
      ++ " " ++ showHList (suc ℓ) xs
```

7. User-extensible datatype definitions

The universe `Type` of Section 5 supports generic programming over all types in a hierarchy of universes, but the collection of types is fixed ahead of time. In this section we extend the `Type` universe to support a user-extensible collection of types. To put it another way, we construct a closed model that includes all datatypes that you would normally declare using the `data` keyword in a dependently typed language like AGDA. In this extended universe we no longer need to add datatypes to our universe in an ad hoc fashion, like we did when adding `‘HList` in Section 6.

Chapman et al. [2010] show how to encode a user-extensible collection of datatypes into a dependent type system. For pedagogical reasons we limit ourselves to their construction for non-indexed datatypes, but the techniques of this section extend to their construction for indexed families of datatypes. Below we review their construction, and later we incorporate it into our `Type` universe.

7.1 Datatype description examples

Consider the standard datatype declaration of a natural number below. A natural number is either zero or the successor of another natural number.

```
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

This datatype definition describes two ways of constructing a natural number. One way uses the `zero` constructor, which has no arguments. The other way uses the `suc` constructor, which has a single recursive argument. By recursive argument we mean an argument of the same datatype currently being defined.

We present the datatype of descriptions shortly, but first consider the following simple example of the description for natural numbers. The type of natural numbers is described as a choice (`⊎`) between two constructors. The first choice represents the `zero` constructor as the unit type `⊤`. The second choice represents the `suc` constructor, which takes a recursive argument (`X`).

```
ℕD : Desc
ℕD = `⊤ `⊎ `X
```

The datatype of descriptions is given below. Notice that the `` `Π `` and `` `Σ `` constructors have a large argument (`A : Set`), so the type of `Desc` must be the higher universe $\texttt{Set}_1$. Thus `Desc` is a type with large constructor arguments, just like `HList` of Section 6.

```
data Desc : Set₁ where
  `⊤ `X : Desc
  _`⊎_ _`×_ : (D E : Desc) → Desc
  `Π `Σ : (A : Set) (D : A → Desc) → Desc
```

The type of descriptions encodes strictly-positive pattern functors. Below is the function that when partially applied to a `Desc`, returns the endofunctor on `Set` that the `Desc` encodes.

```
⟦_⟧ᵈ : Desc → Set → Set
⟦ `⊤ ⟧ᵈ X = ⊤
⟦ `X ⟧ᵈ X = X
⟦ D `⊎ E ⟧ᵈ X =
  Σ Bool (λ b → if b then ⟦ D ⟧ᵈ X else ⟦ E ⟧ᵈ X )
⟦ D `× E ⟧ᵈ X = Σ (⟦ D ⟧ᵈ X) (const (⟦ E ⟧ᵈ X))
⟦ `Π A D ⟧ᵈ X = (a : A) → ⟦ D a ⟧ᵈ X
⟦ `Σ A D ⟧ᵈ X = Σ A (λ a → ⟦ D a ⟧ᵈ X)
```

The endofunctor meaning function $[\![\_]\!]^d$ is only used in one place. It is used to define the least fixpoint operator below. Most readers have probably seen the fixpoint of a functor defined as follows.

```
data μ (F : Functor) : Set where
  con : F (μ F) → μ F
```

Below, in our definition of `μ`, instead of parameterizing by a functor (`F`) we parameterize by a description (`D`), and apply the meaning function $[\![\_]\!]^d$ to obtain a functor.

```
data μ (D : Desc) : Set where
  con : ⟦ D ⟧ᵈ (μ D) → μ D
```

Applying the operator `μ` to a description yields the (potentially recursive) datatype encoded by the description. Using these definitions, we can take the fixpoint of our description of the datatype of natural numbers and construct values with it.

```
μℕ : Set
μℕ = μ ℕD

myZero : μℕ
myZero = con (true , tt)

myOne : μℕ
myOne = con (false , myZero)
```

Notice that `Desc : Set₁` and `μ : Desc → Set` form the type of codes and the meaning function of yet another instance of a Martin-Löf universe. It makes no difference that the "meaning function" `μ` is a type constructor, rather than a proper function.

7.2 Trouble with the existential quantifier

In the simplest case, the `` `Σ `` constructor of `Desc` can be used to add an argument to a constructor, which is a previously defined value. In the more complicated case, the types of other constructor arguments may depend on that value. Below we consider the simple case by defining the description for a type called `WrapBool`. This type merely wraps a boolean in a constructor.

```
data WrapBool : Set where
  wrap : Bool → WrapBool

WrapBoolD : Desc
WrapBoolD = `Σ Bool (const `⊤)

μWrapBool : Set
μWrapBool = μ WrapBoolD

myWrap : μWrapBool
myWrap = con (false , tt)
```

Recall that `Desc` is a type with constructors, such as `` `Σ ``, that have large arguments. Furthermore, the fixpoint of a `Desc` containing a `` `Σ `` is an existentially quantified type. We find ourselves in the same situation as with `HList` of Section 6. We can construct values of our existentially quantified datatype, like `myWrap`. However, we run into trouble when trying to define generic functions over the type `Desc` or fixpoints thereof. Consider the type signatures of the two functions below.

```
showDesc : Desc → String
showμ : (D : Desc) → μ D → String
```

These generic functions are analogous to `showType` and `show`, respectively. When implementing `showDesc`, we get stuck trying to show the first argument of `` `Σ ``. Similarly, we know that the value of the `` `Σ `` case of `showμ` has to be a dependent pair, but we cannot case analyze the type of the first component to show the value.

7.3 Definition of DescForm

The original encoding of the universe of descriptions of datatypes by Chapman et al. [2010] works perfectly well to represent datatypes. However, due to the use of large arguments like `Set` in the `` `Π `` and `` `Σ `` constructors of `` `Desc ``, fully-generic definitions are not possible. Semantically, this means that it is impossible to write fully-generic functions over datatypes with previously defined types as constructor arguments.

DescForm $[\![\_/\_]\!]^d$ `and` `μ` In order to lift generic programming restrictions due to `Set` arguments, and to incorporate descriptions into our language universe, we introduce `DescForm` and the meaning function $[\![\_/\_]\!]^d$, both of which are parameterized by a universe. Like before, the meaning function $[\![\_/\_]\!]^d$ returns an endofunctor when partially applied to a universe and a description.

```
data DescForm (U : Universe) : Set where
  ‘⊤ ‘X : DescForm U
  _‘⊎_ _‘×_ : (D E : DescForm U) → DescForm U
  ‘Π ‘Σ : (A : Universe.Codes U)
    (D : Universe.Meaning U A → DescForm U)
    → DescForm U

⟦_/_⟧ᵈ : (U : Universe) → DescForm U → Set → Set
⟦ U / ‘⊤ ⟧ᵈ X = ⊤
⟦ U / ‘X ⟧ᵈ X = X
⟦ U / D ‘⊎ E ⟧ᵈ X =
  Σ Bool (λ b →
  if b then ⟦ U / D ⟧ᵈ X else ⟦ U / E ⟧ᵈ X )
⟦ U / D ‘× E ⟧ᵈ X =
  Σ (⟦ U / D ⟧ᵈ X) (const (⟦ U / E ⟧ᵈ X))
⟦ U / ‘Π A D ⟧ᵈ X =
  (a : Universe.Meaning U A) → ⟦ U / D a ⟧ᵈ X
⟦ U / ‘Σ A D ⟧ᵈ X =
  Σ (Universe.Meaning U A) (λ a → ⟦ U / D a ⟧ᵈ X)
```

The least fixpoint operator is now parameterized by universe as well as a description. In the constructor `con`, (as in the Chapman construction above) the meaning function applied to the universe and description parameters $\llbracket$ `U / D` $\rrbracket^d$ denotes a functor.

```
data μ {U : Universe} (D : DescForm U) : Set where
  con : ⟦ U / D ⟧ᵈ (μ D) → μ D
```

Replacing a datatype `Desc` with `DescForm`, which is parameterized by a universe, is nothing new. We did the same thing in Section 6, where we replaced `HList` by `HListForm`. Just like before, this change consists of replacing large metalanguage `Set` arguments with object language `Type` arguments.

7.4 Adding DescForm and μ to our universe

The language universe we have been using has now grown to include descriptions of datatypes `DescForm`, their meaning function $\llbracket$`_/_`$\rrbracket^d$, and their least fixpoints `μ`. So, we now add them to `TypeForm`, just as we added `HList`.

```
mutual
  data TypeForm (U : Universe) : Set where
     {- ... previous constructors ... -}
     ‘Desc : TypeForm U
     ‘⟦_⟧ᵈ : DescForm U → TypeForm U → TypeForm U
     ‘μ : DescForm U → TypeForm U

  ⟦_/_⟧ : (U : Universe) → TypeForm U → Set
  {- ... previous cases ... -}
  ⟦ U / ‘Desc ⟧ = DescForm U
  ⟦ U / ‘⟦ D ⟧ᵈ X ⟧ = ⟦ U / D ⟧ᵈ ⟦ U / X ⟧
  ⟦ U / ‘μ D ⟧ = μ D
```

7.5 Internalized datatype description examples

The user-extensible datatype examples of Section 7.1 used the type `Desc`, which is external to our universe. Now we give internalized examples that instead use `‘Desc`. Below are the internalized natural number examples.

```
‘ℕD : (ℓ : ℕ) → ⟦ ℓ | ‘Desc ⟧
‘ℕD ℓ = ‘⊤ ‘⊎ ‘X

‘myZero : (ℓ : ℕ)→ ⟦ ℓ | ‘μ (‘ℕD ℓ) ⟧
‘myZero ℓ = con (true , tt)

‘myOne : (ℓ : ℕ)→ ⟦ ℓ | ‘μ (‘ℕD ℓ) ⟧
‘myOne ℓ = con (false , ‘myZero ℓ)
```

Notice that we can represent natural numbers using a description. So, we could simplify `TypeForm` by removing the now unnecessary `‘ℕ` constructor. Next we give the internalized `WrapBool` examples.

```
‘WrapBoolD : (ℓ : ℕ) → ⟦ suc ℓ | ‘Desc ⟧
‘WrapBoolD ℓ = ‘Σ ‘Bool (const ‘⊤)

‘myWrap : (ℓ : ℕ) → ⟦ suc ℓ | ‘μ (‘WrapBoolD ℓ) ⟧
‘myWrap ℓ = con (false , tt)
```

Last but not least, `HList` can also be represented using a description. Thus, the ad hoc extension of `TypeForm` by `‘HList` is also unnecessary.

```
‘HListD : (ℓ : ℕ) → ⟦ suc ℓ | ‘Desc ⟧
‘HListD ℓ = ‘⊤ ‘⊎
  ‘Σ ‘Type (λ A → ‘Σ ‘⟦ A ⟧ (const ‘X))

‘μHList : (ℓ : ℕ) → Set
‘μHList ℓ = μ (‘HListD ℓ)

‘myHList : (ℓ : ℕ)
  → ⟦ suc (suc ℓ) | ‘μ (‘HListD (suc ℓ)) ⟧
‘myHList ℓ =
  con (false , ‘Bool , false ,
    con (false , ‘ℕ , 7 , con (true , tt)))
```

7.6 Unnecessarily large fixpoint universe level

Consider the datatype description `‘WrapBoolD`, which is constructed with a `‘Σ`. The first argument to `‘Σ` is the type `‘Bool`. If the universe level of `‘WrapBoolD` were zero, then the previous universe of types would be empty, so applying `‘Σ` to `‘Bool` would not be type correct. Thus, the type of `‘WrapBoolD` must be greater than zero, and we type it polymorphically as `suc` ℓ. This much is correct. Similarly, `WrapBoolD` has type `Desc`, which has type $\texttt{Set}_1$, making the `‘Σ` constructor well-founded.

However, our model is not quite satisfactory when it comes to the fixpoints of definitions using `‘Σ` or `‘Π`. Compare the following two values.

```
μWrapBool : Set
μWrapBool = μ WrapBoolD

‘μWrapBoolD : (ℓ : ℕ) → Type (suc ℓ)
‘μWrapBoolD ℓ = ‘μ (‘WrapBoolD ℓ)
```

The type `μWrapBool` has universe level zero rather than one, whereas the universe level of `‘μWrapBoolD` is `suc` ℓ (we would expect it to analogously be ℓ). This is because the datatype declaration of `μ` for `Desc` lowers the universe level, compared to the `D : Desc` datatype parameter at a higher level.

In contrast, when we add the constructor `‘μ : DescForm U → TypeForm U` to `TypeForm`, the universe level `U` for the `DescForm` parameter and the returned `TypeForm` are the same. We leave it to future work to change the model so that we pass `DescForm` a *higher* universe level than the returned `TypeForm` representing the type of `‘μ`.

7.7 Extending show to Desc

At the end of Section 6 we saw that in order to apply `‘mapHList` to `show`, we had to first extend `showType` and `show` to support the new `TypeForm` constructor `‘HList`. In this section we have extended `TypeForm` with `‘Desc`, `‘`$\llbracket$`_`$\rrbracket^d$ and `‘μ` instead. Correspondingly, we

need to extend showType and show, and do so by mutually defining showDesc and show^d. Note that we previously got stuck trying to define showDesc and showμ in Section 7.2. This was because we were using the external datatype Desc, which contains large Set arguments. Now that we have switched to DescForm, which uses our internalized representation, it is finally possible to write those generic functions! Note that we do not define showμ directly as it is a special case of show^d. In the internalized representation, the first argument of the ‘Σ constructor for DescForm is a TypeForm, which we may case analyze.

```
mutual
  showType : (ℓ : ℕ) → ⟦ suc ℓ | ‘Type ‘→ ‘String ⟧
  {- ... previous cases ... -}
  showType ℓ ‘Desc = "Desc"
  showType ℓ (‘⟦ D ⟧$^d$ X) =
    "⟦ " ++ showDesc ℓ D ++ " ⟧$^d$ " ++ showType ℓ X
  showType ℓ (‘μ D) = "μ " ++ showDesc ℓ D

  show : (ℓ : ℕ)
    → ⟦ suc ℓ | ‘Π ‘Type (λ A → ‘⟦ A ⟧ ‘→ ‘String) ⟧
  {- ... previous cases ... -}
  show ℓ ‘Desc D = showDesc ℓ D
  show ℓ (‘⟦ D ⟧$^d$ X) x = show$^d$ ℓ D X x
  show ℓ (‘μ D) (con x) = "con " ++ show$^d$ ℓ D (‘μ D) x

  showDesc : (ℓ : ℕ) → ⟦ ℓ | ‘Desc ‘→ ‘String ⟧
  showDesc ℓ ‘⊤ = "⊤"
  showDesc ℓ ‘X = "X"
  showDesc ℓ (D ‘⊎ E) =
    showDesc ℓ D ++ " ⊎ " ++ showDesc ℓ E
  showDesc ℓ (D ‘× E) =
    showDesc ℓ D ++ " × " ++ showDesc ℓ E
  showDesc zero (‘Π () D)
  showDesc (suc ℓ) (‘Π A D) =
    "Π " ++ showType ℓ A ++ " λ"
  showDesc zero (‘Σ () D)
  showDesc (suc ℓ) (‘Σ A D) =
    "Σ " ++ showType ℓ A ++ " λ"

  showμ : (ℓ : ℕ)
    → ⟦ ℓ | ‘Π ‘Desc (λ D → ‘μ D ‘→ ‘String) ⟧
  showμ ℓ D (con x) = show$^d$ ℓ D D x
```

The definition of showDesc is similar to showType, except for the ‘Π and ‘Σ cases. The first argument of those constructors is a type from the previous universe. For this reason, the ‘Π and ‘Σ cases must case analyze the universe level. If the universe level is zero, then there is no previous universe so those cases are left undefined. In this sense, the ‘Π and ‘Σ cases of showDesc are similar to the ‘⟦_⟧ case of showType because they both operate on a type argument from the previous universe level.

The definition of show ℓ (‘μ D) (con x) uses the helper function show^d that we describe below. It belongs to the mutual block of functions above, but we describe it separately below. A hypothetical function such as showμ, which would be used in such a case, cannot be defined directly by pattern matching on the description argument. This is because the types are not general enough when we need to make recursive function calls. The helper function show^d has a more general type, making its recursive calls well-typed.

```
show$^d$ : (ℓ : ℕ)
  → ⟦ suc ℓ | ‘Π ‘⟦ ‘Desc ⟧ (λ D →
      ‘Π ‘Type (λ X → ‘⟦ ‘⟦ D ⟧$^d$ X ⟧ ‘→ ‘String)) ⟧
show$^d$ ℓ ‘⊤ X tt = "tt"
show$^d$ ℓ ‘X X x = show ℓ X x
show$^d$ ℓ (D₁ ‘⊎ D₂) X (true , x) =
  "inj₁ " ++ show$^d$ ℓ D₁ X x
show$^d$ ℓ (D₁ ‘⊎ D₂) X (false , x) =
  "inj₂ " ++ show$^d$ ℓ D₂ X x
show$^d$ ℓ (D₁ ‘× D₂) X (x , y) =
  show$^d$ ℓ D₁ X x ++ " , " ++ show$^d$ ℓ D₂ X y
show$^d$ zero (‘Π () D) X f
show$^d$ (suc ℓ) (‘Π A D) X f = "λ"
show$^d$ zero (‘Σ () D) X (x , y)
show$^d$ (suc ℓ) (‘Σ A D) X (x , y) =
  show ℓ A x ++ " , " ++ show$^d$ (suc ℓ) (D x) X y
```

7.8 Avoiding functional extentionality

In a previous version of this paper we were tempted to keep the universe of types (TypeForm) smaller by not including a code for the meaning of descriptions ‘⟦_⟧^{d}. Instead, it is possible to give a definitional variant of this code for the meaning function by computing a Type, rather than a Set. We present this alternative below.

```
‘⟦_|_⟧$^d$ : (ℓ : ℕ) → Desc ℓ → Type ℓ → Type ℓ
‘⟦ ℓ | ‘⊤ ⟧$^d$ X = ‘⊤
‘⟦ ℓ | ‘X ⟧$^d$ X = X
‘⟦ ℓ | D ‘⊎ E ⟧$^d$ X =
  ‘Σ ‘Bool
    (λ b → if b then ‘⟦ ℓ | D ⟧$^d$ X else ‘⟦ ℓ | E ⟧$^d$ X)
‘⟦ ℓ | D ‘× E ⟧$^d$ X =
  ‘Σ (‘⟦ ℓ | D ⟧$^d$ X) (const (‘⟦ ℓ | E ⟧$^d$ X))
‘⟦ ℓ | ‘Π A D ⟧$^d$ X = ‘Π ‘⟦ A ⟧ (λ a → ‘⟦ ℓ | D a ⟧$^d$ X)
‘⟦ ℓ | ‘Σ A D ⟧$^d$ X = ‘Σ ‘⟦ A ⟧ (λ a → ‘⟦ ℓ | D a ⟧$^d$ X)
```

Recall that we restrict ourselves to using internalized types, rather than using meta-level types directly. For example, below we give the constructor of ‘μ as a definition in order to show its internalized type.

```
‘con : (ℓ : ℕ) → ⟦ suc ℓ | ‘Π ‘⟦ ‘Desc ⟧
  (λ D → ‘⟦ ‘⟦ D ⟧$^d$ (‘μ D) ⟧ ‘→ ‘⟦ ‘μ D ⟧) ⟧
‘con ℓ D x = con x
```

On the right-hand-side of this definition, con expects a value of type ⟦ Level ℓ / D ⟧^{d} (μ D). Luckily, because TypeForm contatins the code for ‘⟦_⟧^{d}, its meaning function ⟦_⟧ can pattern match on this code to deliver exactly the type that con expects.

In contrast, consider the typing situation we find ourselves in when trying to define ‘con2 for the definitional variant of ‘⟦_|_⟧^{d}.

```
‘con2 : (ℓ : ℕ) → ⟦ suc ℓ | ‘Π ‘⟦ ‘Desc ⟧
  (λ D → ‘⟦ ‘⟦ ℓ | D ⟧$^d$ (‘μ D) ⟧ ‘→ ‘⟦ ‘μ D ⟧) ⟧
‘con2 ℓ D x = con ?
```

Now the type of x is ⟦ Level ℓ / ‘⟦ ℓ | D ⟧^{d} (‘μ D) ⟧, but con requires a ⟦ Level ℓ / D ⟧^{d} (μ D). The meaning function ⟦_⟧ of TypeForm is stuck evaluating, because our definition of ‘⟦_|_⟧^{d} is attempting to case-analyze its Desc argument. In order to push these types through we would need to prove a lemma that ⟦ Level ℓ / ‘⟦ ℓ | D ⟧^{d} (‘μ D) ⟧ will always equal ⟦ Level ℓ / D ⟧^{d} (μ D). Unfortunately, the meta-level meaning function for descriptions ⟦_/_⟧^{d} interprets ‘Π as a meta-level function type. Thus proving this equality requires functional extentionality, which is a signficant extra requirement on the underlying type theory that our AGDA model denotes. Instead, we choose to internalize ⟦_/_⟧^{d} as a code of TypeForm and remain in a vanilla variant of Martin-Löf's intentional type theory.

7.9 Indexed families of datatypes

At the beginning of this section we mentioned that we limited ourselves to considering a construction that described non-indexed datatype definitions. Even with this limitation, it is still possible to define many indexed datatypes as *functions* rather than with descriptions.

Below is the definition of vectors as a function that takes a natural number as an argument and computes a type. If the natural number index is zero, the vector constructor is `nil`. Because `nil` has no arguments, it is represented by the unit type code `‘⊤`. If the index is the successor of some number, then the constructor is `cons`. The `cons` constructor takes an argument corresponding to the type parameter of the vector, and recursive arguments for the remainder of the vector length.

```
‘Vec : (ℓ : ℕ) → ⟦ suc ℓ | ‘Type ‘→ ‘ℕ ‘→ ‘Type ⟧
‘Vec ℓ A zero = ‘⊤
‘Vec ℓ A (suc n) = ‘Σ A (const (‘Vec ℓ A n))

‘myVec : (ℓ : ℕ) → ⟦ ℓ | ‘Vec ℓ ‘Bool 2 ⟧
‘myVec ℓ = (false , (true , tt))
```

In essence, `‘Vec` unfolds its natural number argument into a nested “n-tuple”, where “n” is the value of the natural number index.

Now consider a vector function `‘VecD` that returns a description rather than a type. This function is similar in structure to `‘Vec`, but it returns descriptions of types rather than actual types.

```
‘VecD : (ℓ : ℕ) → ⟦ suc ℓ | ‘Type ‘→ ‘ℕ ‘→ ‘Desc ⟧
‘VecD ℓ A zero = ‘⊤
‘VecD ℓ A (suc n) = ‘Σ A (const (‘VecD ℓ A n))
```

Chapman et al. [2010] define two kinds of descriptions: `Desc` (for non-indexed datatypes) and `IDesc` (for indexed dataypes like `Vec`). The indexed version, `IDesc`, has a fixpoint operator that takes a function from an index to a description as a type argument. In contrast, the non-indexed μ of this paper (following the development of Chapmans `Desc`) takes a description as an argument. Thus, if we were to extend our construction to an indexed fixpoint operator, it could take `‘VecD` as an argument! This functional representation of indexed datatypes is motivated by research on optimized representations of dependent types performed by Brady et al. [2003]. The techniques of this section generalize to the indexed construction described above.

7.10 A simpler Desc constructor

As mentioned immediately above, the `Desc` type presented herein is the non-indexed variant of the datatype definition machinery by [Chapman et al. 2010]. To make it easier for readers familiar with their work to follow along, we kept the `‘Σ` constructor of their `Desc` type. The `‘Σ` serves two purposes. First to introduce datatypes whose constructors have dependent types, and second to support datatypes that have components which are described by previously defined descriptions. However, in this non-indexed setting it suffices to have a much simpler non-dependent constructor `‘[_]` to serve the same purposes. The first purpose, dependently typed constructors (in a non-indexed world) can be encoded by embedding the `‘Σ` of `TypeForm` inside the new non-dependent `‘[_]`. Thus `‘[_]` need only support the second purpose. For the remainder of this paper we will add the non-dependent constructor `‘[_]`, but also keep the old `‘Σ` so that expert readers can imagine how definitions extend to `IDesc`.

```
data DescForm (U : Universe) : Set where
  {- ... previous constructors ... -}
  ‘[_] : Universe.Codes U → DescForm U

⟦_/_⟧ᵈ : (U : Universe) → DescForm U → Set → Set
{- ... previous cases ... -}
⟦ U / ‘[ A ] ⟧ᵈ X = Universe.Meaning U A
```

8. Generic definition of double

The universe construction of this paper makes it possible to write fully-generic functions – functions defined over all possible types. The richest example, that we have seen so far, is `show`, first introduced in Section 4. While `show` is defined for all values, it doesn't recurse into all possible values. Specifically, since it is not possible to recurse into higher-order values, `show` returns the string “λ” in those cases. In this section we give a fully-generic `double` function. While the `double` of this section is also not “complete”, in the sense of recursing into all possible values, it recurses into different set of values when compared with `show`, and motivates a more complete `δouble` function in the next section. The intended meaning of the `double` function is that it replaces all occurrences of a natural number, embedded in a value of any type, with the said number doubled.

8.1 4-tuple of type signatures

The generic `show` is like a fully-generic “fold” operation (reducing a big nested value to a single string), on the other hand, `double` is like a fully-generic “map” operation. It traverses all nested values and maps the doubling operation across all embedded natural numbers encountered. This first iteration of the `double` function is type-preserving. To illustrate this, recall how we defined show with the 4-tuple of functions `(showType, show, showDesc, show`^{d}`)`. Similarly, we will now define double with the 4-tuple of functions `(doubleType, double, doubleDesc, double`^{d}`)`. Below are the type signatures for the 4 mutually defined functions comprising this 4-tuple. Notice that in each of them the type of the output (after being doubled) is the same as the type of the input. Compare this with the types of `show` 4-tuple, where each function of the 4-tuple returns a constant string after traversal in each of the 4 functions.

```
doubleType : (ℓ : ℕ) → ⟦ suc ℓ | ‘Type ‘→ ‘Type ⟧
doubleDesc : (ℓ : ℕ) → ⟦ ℓ | ‘Desc ‘→ ‘Desc ⟧
double : (ℓ : ℕ) → ⟦ suc ℓ |
  ‘Π ‘Type (λ A → ‘Π ‘⟦ A ⟧ (λ a → ‘⟦ A ⟧)) ⟧
double^d : (ℓ : ℕ) → ⟦ suc ℓ |
  ‘Π ‘⟦ ‘Desc ⟧ (λ D → ‘Π ‘Type (λ X →
  ‘⟦ ‘⟦ D ⟧^d X ⟧ ‘→ ‘⟦ ‘⟦ D ⟧^d X ⟧)) ⟧
```

The next step in this logical progression, is for the return type after traversal, to be a function of the input type. We take this step in the next section, but first explore the simpler type-preserving version to make a few important points.

8.2 Doubling case

When a natural number value is encountered, we double it. This case is the whole point of our generic function.

```
doubleType ℓ ‘ℕ = ‘ℕ
double ℓ ‘ℕ n = n + n
```

Note that we have chosen to double natural numbers described by the base type `‘ℕ` in this series of examples. We could have altered any embedded `Desc`-encoded datatype to illustrate “mapping” traversals, but we use a base type because the examples are easier to read (as they avoid the messiness of encoded values).

8.3 homomorphic cases

Most of the cases of our 4 muturally defined functions are structure preserving homomorphisms. They simply recursively thread the doubling operation through their arguments (or child nodes). In particular, non-dependent pairs and subcomponents of previously defined types are homomorphisms.

```
doubleDesc ℓ (D `× E) =
  doubleDesc ℓ D `× doubleDesc ℓ E
double^d ℓ (D `× E) X (x , y) =
  double^d ℓ D X x , double^d ℓ E X y

doubleDesc (suc ℓ) `[ A ] = `[ doubleType ℓ A ]
double^d (suc ℓ) `[ A ] X x = double ℓ A x
```

Below is an example application of our `double` function. Notice how it deeply-crawls the structure of values, and leaves non-natural number values unmodified.

```
Eg₁ : (ℓ : ℕ) → ⟦ 2 + ℓ | `Type ⟧
Eg₁ ℓ = `μ (`[ `ℕ ] `× `[ `Bool ] `× `[ `ℕ ])

eg₁ : (ℓ : ℕ) → ⟦ suc ℓ | Eg₁ ℓ ⟧
eg₁ ℓ = con (1 , true , 3)

eg₁′ : (ℓ : ℕ) → ⟦ suc ℓ | Eg₁ ℓ ⟧
eg₁′ ℓ = double (suc ℓ) (Eg₁ ℓ) (eg₁ ℓ)

test-eg₁ : (ℓ : ℕ) → eg₁′ ℓ ≡ con (2 , true , 6)
test-eg₁ ℓ = refl
```

8.4 Problems with the dependent cases

We quickly run into trouble when defining the traversal for dependently typed functions (`Π) or pairs (`Π). Consider the example dependent pair type below.

```
isOne : (ℓ : ℕ) → ⟦ suc ℓ | `ℕ `→ `Bool  ⟧
isOne ℓ (suc zero) = true
isOne ℓ _ = false

Eg₂ : (ℓ : ℕ) → ⟦ 2 + ℓ | `Type ⟧
Eg₂ ℓ = `Σ `ℕ (λ b → if isOne ℓ b then `ℕ else `⊥)

eg₂ : (ℓ : ℕ) → ⟦ suc ℓ | Eg₂ ℓ  ⟧
eg₂ ℓ = 1 , 3
```

The affect of doubling the first component of the pair changes the type of the second component. Our traversal is no longer type preserving! For example, the type of the second component must be changed to a $\bot$ value if the first component is doubled.

In this preliminary exploration, we take the easy way out, we don't recurse down the first component of dependent values, but in the next section we do something more interesting for such dependent cases.

```
doubleType ℓ (`Σ A B) =
  `Σ A (λ a → doubleType ℓ (B a))
double ℓ (`Σ A B) (a , b) = a , double ℓ (B a) b

doubleType ℓ (`Π A B) =
  `Π A (λ a → doubleType ℓ (B a))
double ℓ (`Π A B) f = λ a → double ℓ (B a) (f a)
```

Here we push the doubling operation through the second component, but leave the *first* component untraversed. In contrast recall that the `show` function does not traverse the *second* component of dependent cases, because those are higher-order. For similar reasons `double` does not recurse through the first component of dependent cases, because that would change the type of the second component.

9. Generic definition of δouble

The fully-generic `double` function of the previous section is a nice example, but it's a shame that it does not recurse into every possible value. In this section we define another doubling function, called `δouble`. Whereas `double` is type-perserving, `δouble` changes its type! Now that we can change both the type and the value of a value being doubled, we can do more interesting things for the dependent cases.

9.1 4-tuple of type signatures

Compared to the 4-tuple of type signatures used to defined `double`, the `δouble` 4-tuple takes advantage of dependent types. The `δoubleType` is used by `δouble` when doubling values which are types (these are mutually defined functions, after all). However, now when `δouble` is applied to any particular value, the type of the *returned* value must is computed by `δoubleType`. Thus, if `δoubleType` is not defined to be the identity function, then the type of doubled values may be different from the type of the argument to the function.

```
δoubleType : (ℓ : ℕ) → ⟦ suc ℓ | `Type `→ `Type ⟧
δoubleDesc : (ℓ : ℕ) → ⟦ ℓ | `Desc `→ `Desc ⟧
δouble : (ℓ : ℕ) → ⟦ suc ℓ |
  `Π `Type (λ A → `⟦ A ⟧ `→ `⟦ δoubleType ℓ A ⟧) ⟧
δouble^d : (ℓ : ℕ) → ⟦ suc ℓ |
  `Π `⟦ `Desc ⟧ (λ D → `Π `Type (λ X →
  `⟦ `⟦ D ⟧^d X ⟧ `→
  `⟦ `⟦ δoubleDesc ℓ D ⟧^d (δoubleType ℓ X) ⟧)) ⟧
```

9.2 Previously given cases

Our new `δouble` only behaves differently from `double` in the dependent cases. This is good, as it means that for most values doubling them will still preserve types. Now let's find out what we do in the dependent cases.

9.3 Dependent pair cases

When doubling dependent pairs, we now *add* an extra argument for the doubled first component, changing a tuple into a triple. Thus we keep around the old first component so that the type of the second old component can depend on the same old value. In order to make this be type correct, `δoubleType` must add an extra `Σ to also change the *type* from classifying a tuple to classifying a triple.

```
δoubleType ℓ (`Σ A B) =
  `Σ A (λ a → `Σ (δoubleType ℓ A) (λ _ →
  δoubleType ℓ (B a)))
δouble ℓ (`Σ A B) (a , b) =
  a , δouble ℓ A a , δouble ℓ (B a)b
```

Below we see this action. Notice that the type of the freshly computed resulting triple is itself freshly computed. This once again highlights the type-enforced correspondence between `δoubleType` and `δouble`.

```
Eg₂′ : (ℓ : ℕ) → ⟦ 2 + ℓ | `Type ⟧
Eg₂′ ℓ = δoubleType (suc ℓ) (Eg₂ ℓ)

eg₂′ : (ℓ : ℕ) → ⟦ suc ℓ | Eg₂′ ℓ ⟧
eg₂′ ℓ = δouble (suc ℓ) (Eg₂ ℓ) (eg₂ ℓ)
```

```
test-eg₂′ : (ℓ : ℕ) → eg₂′ ℓ ≡ (1 , 2 , 6)
test-eg₂′ ℓ = refl
```

9.4 Dependent function cases

Dependent function values are abstractions by their very nature. Thus, we cannot use the same trick of "keeping around" the old domain value, because we do not know what it is yet. This time we modify the type of the resulting value by requiring an extra function parameter to be supplied. This function parameter is a conversion function that transforms values from the newly doubled domain to the old undoubled domain.

```
δoubleType ℓ (`Π A B) =
  `Π (δoubleType ℓ A `→ A) (λ f →
  `Π (δoubleType ℓ A) (λ a →
  δoubleType ℓ (B (f a))))
δouble ℓ (`Π A B) f =
  λ g a → δouble ℓ (B (g a)) (f (g a))
```

Recall that with arbitrarily complex dependent types, it may not always be possible to convert values of old types to values of new types (the new type may be `⊥). Luckily, we parameterize by the conversion function, and the user can pass it in when it is possible to define. If the `Π being mapped over does not dependently use its domain argument, then the required conversion function will merely be the identity function.

10. Generic function interface

We have now seen three examples of fully generic functions: `show` from Section 4, `double` from Section 8, and `δouble` from Section 9. All of these fully generic functions were defined in a common way, which is formally captured in this section by dependent record types.

10.1 Generic

A fully-generic function is defined as a 4-tuple of functions, handling types, descriptions, values, and described values. Below is the more general version of the dependent records we present. This dependent record is called `Generic`. Notice that it has eight fields, rather than four. The extra fields are the type signatures of each of the four generic functions. Our convention is to capitalize the field names standing for the type signatures of each generic function that comprises the 4-tuple.

```
record Generic (ℓ : ℕ) : Set where
  field
    GType : ⟦ 2 + ℓ ∣ `Type `→ `Type ⟧
    gType : ⟦ 1 + ℓ ∣ `Π `Type (λ A → GType `⟦ A ⟧) ⟧

    GDesc : ⟦ 1 + ℓ ∣ `⟦ `Desc ⟧ `→ `Type ⟧
    gDesc : ⟦ 0 + ℓ ∣ `Π `Desc (λ D → GDesc D) ⟧

    GVal  : ⟦ 2 + ℓ ∣
      `Π `Type (λ A → `⟦ A ⟧ `→ `Type) ⟧
    gVal  : ⟦ 1 + ℓ ∣ `Π `Type (λ A →
      `Π `⟦ A ⟧ (λ a → GVal `⟦ A ⟧ a)) ⟧

    GVal^d : ⟦ 1 + ℓ ∣
      `Π `⟦ `Desc ⟧ (λ D → `Π `Type (λ X →
      `⟦ `⟦ D ⟧^d X ⟧ `→ `Type)) ⟧
    gVal^d : ⟦ 1 + ℓ ∣
      `Π `⟦ `Desc ⟧ (λ D → `Π `Type (λ X →
      `Π `⟦ `⟦ D ⟧^d X ⟧ (λ x → `⟦ GVal^d D X x ⟧))) ⟧
```

Each of our fully-generic 4-tuples can be represented as a value of this dependent record. All functions in the 4-tuple used to define `show` are typed as strings.

```
gshow : (ℓ : ℕ) → Generic ℓ
gshow ℓ = record {
    GType = λ A → `String
  ; gType = showType ℓ
  ; GDesc = λ D → `String
  ; gDesc = showDesc ℓ
  ; GVal  = λ A a → `String
  ; gVal = show ℓ
  ; GVal^d = λ D X x → `String
  ; gVal^d = show^d ℓ
  }
```

In contrast, each of the 4-tuple functions that are used to define `double` are typed differently. The functions to double types and descriptions return types and descriptions respectively. The functions to double values and described values preserve the type of the value being doubled.

```
gdouble : (ℓ : ℕ) → Generic ℓ
gdouble ℓ = record {
    GType = λ A → `Type
  ; gType = doubleType ℓ
  ; GDesc = λ D → `Desc
  ; gDesc = doubleDesc ℓ
  ; GVal  = λ A a → A
  ; gVal = double ℓ
  ; GVal^d = λ D X x → `⟦ D ⟧^d X
  ; gVal^d = double^d ℓ
  }
```

Once again, each 4-tuple function of `δouble` is typed differently. The difference is that the functions to double values and described values are dependently typed by the functions to double types and descriptions respectively.

```
gδouble : (ℓ : ℕ) → Generic ℓ
gδouble ℓ = record {
    GType = λ A → `Type
  ; gType = δoubleType ℓ
  ; GDesc = λ D → `Desc
  ; gDesc = δoubleDesc ℓ
  ; GVal  = λ A a → δoubleType (suc ℓ) A
  ; gVal = δouble ℓ
  ; GVal^d = λ D X x →
      `⟦ δoubleDesc ℓ D ⟧^d (δoubleType ℓ X)
  ; gVal^d = δouble^d ℓ
  }
```

10.2 GenericΔ

Our alternative generic interface record is `GenericΔ`. It is more specific than `Generic`, and its instances include `double` and `δouble`, but not `show`. The `GenericΔ` interface *enforces* the aforementioned dependencies between the generic functions operating on types and the generic functions operating on values of said types. This record only has four fields, because the types of each transformation function cannot be altered. Whereas `showType` in `Generic` can return a `String`, in `GenericΔ` the `doubleType` and `δoubleType` functions are assumed to return types.

```
record GenericΔ (ℓ : ℕ) : Set where
  field
    gType : ⟦ suc ℓ ∣ `Type `→ `Type ⟧
    gDesc : ⟦ ℓ ∣ `Desc `→ `Desc ⟧
```

```
gVal : ⟦ suc ℓ |
  ‘Π ‘Type (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ gType A ⟧) ⟧
gVal^d : ⟦ suc ℓ |
  ‘Π ‘⟦ ‘Desc ⟧ (λ D → ‘Π ‘Type (λ X →
  ‘⟦ ‘⟦ D ⟧^d X ⟧ ‘→ ‘⟦ ‘⟦ gDesc D ⟧^d (gType X) ⟧)) ⟧
```

We previously referred to the `double` function as being type preserving. This property is witnessed in the `GenericΔ` record instance below, where the type-changing function for `double` is the identity function. In contrast, the type-changing function for `δouble` is `δoubleType`.

```
gdoubleΔ : (ℓ : ℕ) → GenericΔ ℓ
gdoubleΔ ℓ = record {
    gType = id
  ; gDesc = id
  ; gVal = double ℓ
  ; gVal^d = double^d ℓ
  }

gδoubleΔ : (ℓ : ℕ) → GenericΔ ℓ
gδoubleΔ ℓ = record {
    gType = δoubleType ℓ
  ; gDesc = δoubleDesc ℓ
  ; gVal = δouble ℓ
  ; gVal^d = δouble^d ℓ
  }
```

Finally we would like to mention that one might imagine other additional interfaces. For example, one in which the newly computed type is not only dependent on the old type, but also on the old value.

11. Conclusion

We have demonstrated that a large and very expressive universe can be constructed for a dependently typed language with a predicative hierarchy of universes, that also supports user-extensible datatypes. We described the Universe in several stages, starting very simply and informally, building in several steps, to a completely formal, mechanized, presentation in the dependently typed language AGDA [Norell 2007].

Variations of the ideas in this paper were published elsewhere, but here we tie them all together into a coherent whole that is significantly more useful than any of its parts. The glue that unifies these ideas is to parameterize data structures encoding types by a Universe. This supports the predicative hierarchy, and the Universe's meaning function can be used to internalize the types of generic functions.

12. Acknowledgements

This work grew out of a humble question that I (Larry) posted on Twitter [Diehl 2012]. The responses I received made me aware of Palmgren's [1998] universes work, and their AGDA mechanization by McBride [2011]. We are indebted to Conor McBride, Darin Morrison, Daniel Peebles, and Andrea Vezzosi for providing helpful responses. We would also like to thank Ki Yung Ahn, Nathan Collins, Francisco Ferreira, Gabor Greif, Caylee Hogg, and anonymous reviewers for giving valuable feedback on drafts of this paper. This work was supported in part by NSF grant 0910500.

References

T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 209–257, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76785-1, 978-3-540-76785-5. URL `http://dl.acm.org/citation.cfm?id=1782894.1782898`.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic journal of computing*, 10:265–289, 2003.

E. Brady, C. McBride, and J. McKinna. Inductive Families Need Not Store Their Indices. In *TYPES*, pages 115–129, 2003.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. *SIGPLAN Not.*, 45(9):3–14, Sept. 2010. ISSN 0362-1340. doi: 10.1145/1932681.1863547. URL `http://doi.acm.org/10.1145/1932681.1863547`.

L. Diehl. Summary of question posed on twitter, oct 2012. URL `https://gist.github.com/larrytheliquid/3909860`.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Typed Lambda Calculi and Applications*, pages 643–643, 1999.

P. Dybjer and A. Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003.

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, PhD thesis, Universit e Paris VII, 1972.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *of Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

A. Hurkens. A simplification of girard's paradox. *Typed Lambda Calculi and Applications*, pages 266–278, 1995.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975.

P. Martin-Löf. Intuitionistic type theory. *Notes by Giovanni Sambin*, 1984.

C. McBride. Dependently typed functional programs and their proofs. 2000.

C. McBride. Elimination with a motive. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, pages 197–216, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43287-6. URL `http://dl.acm.org/citation.cfm?id=646540.759262`.

C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170. Springer, 2005.

C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863497. URL `http://doi.acm.org/10.1145/1863495.1863497`.

C. McBride. Hier soir, an ott hierarchy, Nov. 2011. URL `http://www.e-pig.org/epilogue/?p=1098`.

C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, pages 252–267. Springer, 2006.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory*, volume 85. Oxford University Press, 1990.

U. Norell. *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007.

E. Palmgren. On universes in type theory. In *191 – 204*. Oxford University Press, 1998.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-890-2. doi: 10.1145/1707790.1707799. URL `http://doi.acm.org/10.1145/1707790.1707799`.