Generic Constructors and Eliminators from Descriptions

Type Theory as a Dependently Typed Internal DSL

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

Abstract

Dependently typed languages with an "open" type theory introduce new datatypes using an axiomatic approach. Each new datatype introduces axioms for constructing values of the datatype, and an elimination axiom (which we call the *standard eliminator*) for consuming such values. In a "closed" type theory a single introduction rule primitive and a single elimination rule primitive can be used for all datatypes, without adding axioms to the theory.

We review a closed type theory, specified as an AGDA program, that uses descriptions for datatype construction. Descriptions make datatype definitions first class values, but writing programs using such datatypes requires low-level understanding of how the datatypes are encoded in terms of descriptions. In this work we derive constructors and standard eliminators, by defining generic functions parameterized by a description. Our generic type theory constructions are defined as generic wrappers around the closed type theory primitives, which are themselves generic functions in the AGDA model. Thus, we allow users to write programs in the model without understanding the details of the description-based encoding of datatypes, by using open type theory constructions as an internal domain-specific language (IDSL).

Categories and Subject Descriptors D.3 [*Software*]: Program-

ming Languages.

Keywords Generic programming; dependent types; descriptions; eliminators.

1. Introduction

Dependently typed languages such as COQ [The Coq Development Team, 2008], AGDA [Norell, 2007], and IDRIS [Brady, 2011] introduce datatypes axiomatically. These systems extend an *open* type theory with new axioms that describe how to legally manipulate values of a newly declared type.

Recently, there has been quite a bit of work on defining datatypes within a *closed* theory (without axioms) using *descriptions*. Descriptions make datatype definitions first class values in a dependent type theory. This has several desirable consequences, such as the ability to perform generic programming [Chapman

WGP'14, August 31, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633628.2633630

et al., 2010; McBride, 2011; Dagand, 2013] over described types, as well as decreasing the number of constructs in the metatheory via levitation [Chapman et al., 2010; Dagand, 2013].

For example, we might declare the type of vectors, length indexed lists, in an open dependently typed language based on Martin-Löf [1975] type theory. Declaring vectors (`Vec`) adds two constructors (`nil` and `cons`) and one eliminator (`elimVec`) as axioms to the language.

```
nil : (A : Set) → Vec A zero
cons : (A : Set) (n : ℕ) (a : A)
  (xs : Vec A n) → Vec A (suc n)

elimVec : (A : Set) (P : (n : ℕ) → Vec A n → Set)
  (pnil : P zero nil)
  (pcons : (n : ℕ) (a : A) (xs : Vec A n)
    → P n xs → P (suc n) (cons n a xs))
  (n : ℕ) (xs : Vec A n) → P n xs
```

In contrast, declaring a datatype like `Vec` in a closed type theory using descriptions does *not* add constructors and an eliminator as specialized axioms to the language. Instead, values of datatypes built from descriptions can be introduced with a single primitive, the initial algebra (`init`), and can be eliminated with a single primitive, a dependently typed version of catamorphism (`ind`), which takes an algebra (α) as its argument.

```
init : {I : Set} {D : Desc I} {i : I}
  → El D (μ D) i → μ D i
```

```
ind : {I : Set} (D : Desc I)
  (P : (i : I) → μ D i → Set)
  (α : (i : I) (xs : El D (μ D) i)
    (ihs : Hyps D (μ D) P i xs) → P i (init xs))
  (i : I) (x : μ D i) → P i x
```

Without trying to understand these type signatures at the moment, recognize that:

- Both types are parameterized by a description, allowing them to be used with any datatype defined using a description.
- Both types refer to the type `El D`, which interprets a description as a pattern functor and is used to define the datatypes with an initial algebra-style semantics.

Two unfortunate side effects of introducing and eliminating described datatypes using algebras based on pattern functors are:

1. Users need to understand how `El D` gets interpreted as a type in the language in order to program with values of said types, exposing the low-level encoding.
2. Function definitions defined with `ind` are particularly verbose, due to the low-level encoding, but the functions follow a common pattern.

Rather than making users of the AGDA model learn the details of description-based encodings when writing programs using described datatypes, the **major contribution** of this paper is a generic constructor (`inj`) and a generic eliminator (`elim`), which both have an interface that hides the details of the description-based encoding. The type of `inj` applied to a description of a datatype, and a tag specifying constructor, is exactly the expected type signature of a constructor defined axiomatically in an *open* language. Similarly, the type signature of `elim` applied to a description of a type is exactly the expected type signature of an eliminator defined axiomatically in an *open* language. Moreover, `inj` and `elim` are examples of generic programming, defined as generic wrapper functions around the *closed* type theory primitives `init` and `ind`, which are themselves generic functions in the AGDA model.

In a sense we derive the standard constructors and eliminators of type theory within a simple and sound system. We retain the generic programming ability afforded by description based languages, but also hide implementation details when defining functions over particular types by supplying the user with standard constructors and eliminators. Essentially, we use generic programming to define type theory constructions as an *internal domain-specific language* [Landin, 1966] within the AGDA model of closed type theory.

The remainder of this paper proceeds as follows:

- **Section 2** *Reviews* how to define datatypes using descriptions.
- **Section 3** *Reviews* how to introduce values of described types using the primitive initial algebra `init`.
- **Section 4** *Contributes* the novel generic constructor `inj`. To this end, we highlight each $\mathbf{Part}_I$ involved in defining a specialized constructor in terms of `init`.
- **Section 5** *Reviews* how to eliminate values of described types using the primitive dependent catamorphism `ind`. We also demonstrate the verbosity of `ind`-based definitions.

- **Section 6** *Contributes* the novel generic eliminator `elim`. To this end, we highlight each $\mathbf{Part}_E$ involved in defining a specialized eliminator in terms of `ind`.
- **Section 7** *Proves* the correctness property that `ind` and `elim` are extensionally equivalent functions. For technical reasons, we actually prove that `ind` is equivalent to the helper function `elimUncurried` instead.
- **Section 8** *Discusses* related work.

All code presented in this paper has been checked with AGDA. [1] To avoid clutter, in this paper we omit universe levels and assume `Set : Set`. However, the accompanying source code contains a version of the code stratified by universe levels.

2. Declaring Datatypes

The goal of this section is to *review* how to define the following type declaration as a first-class value of our type theory.

```
data Vec (A : Set) : ℕ → Set where
  nil : Vec A zero
  cons : (n : ℕ) (a : A) (xs : Vec A n)
    → Vec A (suc n)
```

[1] The accompanying source code can be found at
`https://github.com/larrytheliquid/generic-elim`

Whereas such a declaration typically involves axiomatically extending the type theory, the technology of descriptions [Chapman et al., 2010; McBride, 2011; Dagand, 2013] lets us define datatypes within a closed type theory. There are several ways to define the datatype of descriptions `Desc`. For simplicity, in this paper we use the encoding by McBride [2011].

2.1 Description Type

The datatype `Desc` of descriptions is used to represent user-defined definitions of strictly-positive indexed families of inductively defined types. `Desc` is parameterized by a type `I`, the index of the encoded type family.

Throughout this paper it will be easier to first pretend like we defined `Vec` with a single constructor, either `nil` or `cons`. This makes it easier to understand later definitions where `Vec` contains both constructors.

Imagine declaring a datatype with a single constructor. A constructor is a sequence of arguments that subsequent arguments may depend on (i.e., a *telescope*), along with recursive arguments at some type indices, and it ends with some type index. Respectively, `Arg`, `Rec`, and `End` allow you to encode a dependent argument, a recursive argument at some index, and ending the constructor definition at some index.

```
data Desc (I : Set) : Set₁ where
  End : (i : I) → Desc I
  Rec : (i : I) (D : Desc I) → Desc I
  Arg : (A : Set) (B : A → Desc I) → Desc I
```

Description of a Single Constructor For example, first recall the type of the constructor `nil` of vectors.

```
nil : (A : Set) → Vec A zero
```

The constructor `nil` takes no arguments, so its description ends immediately at index `zero`. The type of the description returned is `Desc ℕ` because the type we are encoding `Vec` is indexed by natural numbers.

```
nilD : (A : Set) → Desc ℕ
nilD A = End zero
```

Next recall the type of the constructor `cons` of vectors.

```
cons : (A : Set) (n : ℕ) → A → Vec A n
  → Vec A (suc n)
```

The description of `cons` requires a dependent argument `n : ℕ` for the index, a non-dependent argument `A` for the value being added to the vector, a recursive argument indexed by the natural number `n`, and finally ends at index `suc n`.

```
consD : (A : Set) → Desc ℕ
consD A =
  Arg ℕ (λ n → Arg A (λ _ → Rec n (End (suc n))))
```

Description of Multiple Constructors The datatype `Desc` can also be used to describe an entire datatype, consisting of descriptions of multiple constructors. This is achieved by making use of the isomorphism between disjoint sums and dependent pairs whose domain is some finite enumeration.

```
A ⊎ B ≅ Σ Bool (λ b → if b then A else B)
```

This works fine for a datatype with two constructors (because `Bool` is a two point domain), in general we will define an n-point domain for a datatype with n constructors. By convention we name such types and their constructors ending in the suffix `T`, for tag.

```
data VecT : Set where
  nilT consT : VecT
```

A datatype with multiple constructors is represented by an `Arg` description whose first argument (e.g. `VecT`) is a datatype of tags – one for each constructor – and whose second argument (e.g. `VecC`) is a function that returns a description for each constructor tag. Note that whereas we used `Arg` for arguments of constructors before, now we are using `Arg` to represent the sum of all constructors. By convention we use the suffix `C` for the sum of constructors of a description, and the suffix `D` for descriptions.

```
VecC : (A : Set) → VecT → Desc ℕ
VecC A nilT  = nilD A
VecC A consT = consD A

VecD : (A : Set) → Desc ℕ
VecD A = Arg VecT (VecC A)
```

2.2 First-class Enumerations & Tags

When defining the description of vectors, we previously used a custom tag type `VecT` to name each constructor. Descriptions are primarily meant as a construction for representing user-defined datatypes in a dependent type theory with a closed universe of types. To prevent the need to extend the type theory with new tag types constantly, we can instead define first-class enumerations and tags. Enumerations are just a list of labels. A tag is an index into an enumeration, pointing at a specific label.

```
Label : Set
Label = String

Enum : Set
Enum = List Label

data Tag : Enum → Set where
  here : ∀{l E} → Tag (l :: E)
  there : ∀{l E} → Tag E → Tag (l :: E)
```

Thus, the type of vector tags `VecT` can be defined as `Tag` applied to the enumeration `"nil" :: "cons" :: []`. We can also define the `VecT` constructors `nilT` and `consT` by using `Tag` constructors to index into the enumeration of labels. The constructors `here` and `there` are analogous to `zero` and `suc`.

```
VecE : Enum
VecE = "nil" :: "cons" :: []

VecT : Set
VecT = Tag VecE

nilT : VecT
nilT = here

consT : VecT
consT = there here
```

Elimination of Tags A tag can be eliminated with a `case` construct (this is referred to as `switch` by Chapman et al. [2010]; Dagand [2013]), producing a value of the motive [McBride, 2002] type `P` indexed by the tag. In addition to the tag being eliminated, the case construct is given a list of branches, one of which the tag will select.

```
case : {E : Enum} (P : Tag E → Set)
  (cs : Branches E P) (t : Tag E) → P t
case P (c , cs) here = c
```



```
case P (c , cs) (there t) =
  case (λ t → P (there t)) cs t
```

Think of the cases being a right-nested tuple. The type of this tuple is computed by the `Set` returning function `Branches` (Chapman et al. [2010]; Dagand [2013] refer to `Branches` as π). There is a branch for each label in the enumeration, and the type of each branch depends on the tag representing the position of the label in the enumeration.

```
Branches : (E : Enum) (P : Tag E → Set) → Set
Branches [] P = ⊤
Branches (l :: E) P =
  P here × Branches E (λ t → P (there t))
```

Now we can redefine `VecC` with the `case` eliminator instead of by pattern matching. Note that a right-nested product of `Branches` always ends with the unit type ⊤.

```
VecC : (A : Set) → VecT → Desc ℕ
VecC A = case (λ _ → Desc ℕ)
  ( End zero
  , Arg ℕ (λ n → Arg A (λ _ → Rec n (End (suc n)))
  , tt )
```

3. Introduction with Algebras

The goal of this section is to *review* how to use the primitive introduction rule for datatypes built using descriptions to define the constructors of `Vec`.

```
nil : (A : Set) → Vec A zero
cons : (A : Set) (n : ℕ) → A → Vec A n
  → Vec A (suc n)
```

In a system where the datatype declaration `Vec` is an axiomatic extension, the constructors `cons` and `nil` are defined for us. When using descriptions to define `Vec`, we can instead introduce values of type `Vec` using its initial algebra.

3.1 Fixpoint Type

A description is a first-class datatype declaration. To get back the type encoded by the description, you apply the fixpoint type constructor μ to it. For example, below we define `Vec` by applying μ to its description `VecD`.

```
data μ {I : Set} (D : Desc I) (i : I) : Set where
  init : El D (μ D) i → μ D i

Vec : (A : Set) (n : ℕ) → Set
```

```
Vec A n = μ (VecD A) n
```

3.2 Interpretation of Descriptions Type

To introduce values of type `Vec`, we use the `init` constructor of μ. The argument to `init` is `El D (μ D) i`. Let's understand `El` by first considering a description of `Vec` that only has the single constructor `nil` or `cons`. If `init` introduces a value of a single constructor datatype, then its arguments must be the constructor's arguments. Thus, think of `El` as a function that computes the type of the arguments of our constructor. `El` computes the arguments as a right-nested tuple, where `Arg` gets interpreted as a dependent pair argument, `Rec` becomes a non-dependent recursive type argument, and `End` ends the tuple by requiring a proof that the constructor has the correct index.

```
ISet : Set → Set₁
ISet I = I → Set
El : {I : Set} (D : Desc I) → ISet I → ISet I
El (End j) X i = j ≡ i
El (Rec j D) X i = X j × El D X i
El (Arg A B) X i = Σ A (λ a → El (B a) X i)
```

Interpretation of a Single Constructor The `nil` constructor of vectors has no arguments. Thus, `El` for `nilD` will only require a proof that the index in the type is equal to the vector length zero.

For the remainder of the paper, we use a curved arrow ($\rightsquigarrow$) to denote that the expression to the left of the arrow definitionally reduces to the term on the right.

```
NilEl : (A : Set) (n : ℕ) → Set
NilEl A n = El (nilD A) (Vec A) n
```

```
NilEl A n ⇝ zero ≡ n
```

The `cons` constructor of vectors has an index argument, an argument for the value being added to the vector, a recursive argument, and finally requires a proof that the index in the type is equal to the successor of the index argument.

```
ConsEl : (A : Set) (n : ℕ) → Set
ConsEl A n = El (consD A) (Vec A) n

ConsEl A n ⇝
  Σ ℕ (λ m → A × Vec A m × (suc m ≡ n))
```

Interpretation of Multiple Constructors Recall that multiple constructors are represented as a tagged sum using a dependent pair (Section 2.1). Thus, `El` for `VecD` will be the tagged sum requiring *either* `NilEl` or `ConsEl`.

```
VecEl : (A : Set) (n : ℕ) → Set
VecEl A n = El (VecD A) (Vec A) n

VecEl A n ⇝ Σ VecT (case (λ _ → Set)
  (NilEl A n , ConsEl A n , tt))
```

3.3 Definition of Constructors via the Initial Algebra

We are now ready to define the constructors `nil` and `cons` using the initial algebra `init`, which is the goal of this section. We have already seen `VecEl`, the type of the argument to `init` for vectors. Thus a constructor is defined by applying `init` to a tuple. The first argument is the tag choosing a particular constructor. Next comes the tuple of proper arguments for the constructor. The tuple ends with a proof that the index has the correct value.

```
nil : (A : Set) → Vec A zero
nil A = init (nilT , refl)

cons : (A : Set) (n : ℕ) (x : A) (xs : Vec A n)
  → Vec A (suc n)
cons A n x xs = init (consT , n , x , xs , refl)
```

4. Generic Constructors

The goal of this section is to *contribute* a novel generic constructor for datatypes built from descriptions. The constructors `nil` and `cons` are manually defined in Section 3 using the initial algebra `init` as a primitive. Now we will define a generic constructor `inj` that once and for all captures the pattern inherent in definitions of constructors. This constructor may be used to define `nil` and `cons` as follows.

```
nil : (A : Set) → Vec A zero
nil A = inj (VecD A) nilT
```



```
cons : (A : Set) (n : ℕ) (x : A) (xs : Vec A n)
  → Vec A (suc n)
cons A = inj (VecD A) consT
```

Importantly, our generic constructor is defined in terms of the existing primitives and does not extend the metatheory. This amounts to:

Part$_I$ 1. *Currying constructor arguments.*

Part$_I$ 2. *Inserting an implicit proof that the constructor has the correct index.*

Defining `inj` may not seem impressive by itself, but it acts as nice pedagogical step towards understanding how to define the generic eliminator `elim` in Section 6.

4.1 Uncurried Interpretation Algebra Type

In order to implement Part$_I$ 1, we must first recognize the initial algebra as an uncurried function. Recall the type of the initial algebra `init : El D (μ D) i → μ D i`. Rather than focusing on the initial algebra, we can generalize the uncurried view of this constructor by replacing `μ D` with an arbitrary type family `X : I → Set`.

```
UncurriedEl : {I : Set}
  (D : Desc I) (X : ISet I) → Set
UncurriedEl D X = ∀ {i} → El D X i → X i
```

Recognize `UncurriedEl` as an uncurried function by thinking of `El D X i` as a product of n arguments $A_1 \times ... \times A_n$, an argument requiring a proof of correct indexing $(j \equiv i)$, and `X i` as the result type Z.

$$A_1 \times ... \times A_n \times (j \equiv i) \rightarrow Z$$

Uncurried Algebra of a Single Constructor For example, applying `UncurriedEl` to the description of the `cons` constructor results in the following type.

```
UncurriedEl (consD A) (Vec A) ⇝
```

```
  ∀{n} → ConsEl A n → Vec A n
```

4.2 Curried Interpretation Algebra Type

Now let's define the curried version of the function. Recall that the type `El` is a product of arguments, and `UncurriedEl` is a function from that product to some other type family. In contrast, `CurriedEl` is one large right-nested definition of function arguments.

```
CurriedEl : {I : Set}
  (D : Desc I) (X : ISet I) → Set
CurriedEl (End i) X = X i
CurriedEl (Rec i D) X = (x : X i) → CurriedEl D X
CurriedEl (Arg A B) X = (a : A) → CurriedEl (B a) X
```

Recognize `CurriedEl` as a curried function that demands n constructor arguments as function arguments $A_1 \to ... \to A_n$, and has the result type Z.

$$A_1 \to ... \to A_n \to Z$$

Significantly, `CurriedEl` does not require a proof of correct indexing $(j \equiv i)$. Thus, in addition to solving Part_I 1 by currying arguments, `CurriedEl` also solves Part_I 2 by implicitly supplying the correctness proof. Compare this to the alternative definition `CurriedEl'` that explicitly requires the correctness proof below. The extra proof can be seen in the `End` constructor case.

```
CurriedEl' : {I : Set}
  (D : Desc I) (X : ISet I) (i : I) → Set
CurriedEl' (End j) X i =
  j ≡ i → X i
```

```
CurriedEl' (Rec j D) X i =
  (x : X j) → CurriedEl' D X i
CurriedEl' (Arg A B) X i =
  (a : A) → CurriedEl' (B a) X i
```

Curried Algebra of a Single Constructor Below is an example of applying `CurriedEl` to the description of the `cons` constructor. Notice that all arguments are curried, and a proof of index correctness is not demanded.

```
CurriedEl (consD A) (Vec A) ⇝
  (m : ℕ) → A → Vec A m → Vec A (suc m)
```

4.3 Curry Interpretation Algebra Function

All we need now is a curry function that takes an `UncurriedEl` and returns a `CurriedEl`. The definition of this function is unremarkable, but its type clearly explains its intentions.

```
curryEl : {I : Set} (D : Desc I) (X : ISet I)
  → UncurriedEl D X → CurriedEl D X
curryEl (End i) X cn =
  cn refl
curryEl (Rec i D) X cn =
  λ x → curryEl D X (λ xs → cn (x , xs))
curryEl (Arg A B) X cn =
  λ a → curryEl (B a) X (λ xs → cn (a , xs))
```

4.4 Generic Constructor

The moment has arrived, with the help of our `curryEl` function we can easily define the generic constructor `inj`.

```
inj : {I : Set} (D : Desc I) → CurriedEl D (μ D)
```

```
inj D = curryEl D (μ D) init
```

Unlike previous functions, this one is specialized to datatypes defined with μ rather than arbitrary type families `X`. This is the function we set out to define at the beginning of this section. Compared to values of some type introduced with `init` (Section 3.3), values introduced with `inj` have curried arguments and do not need to supply a proof `refl` of correct indexing.

5. Elimination with Algebras

The goal of this section is to *review* how to use the primitive elimination rule for datatypes built using descriptions. We use the vector concatenation function (which flattens a vector of homogenously-sized vectors) as our example, defined below using the specialized eliminator `elimVec`.

```
concat : (A : Set) (m n : ℕ)
  (xss : Vec (Vec A m) n) → Vec A (mult n m)
concat A m = elimVec (Vec A m)
  (λ n xss → Vec A (mult n m))
  (nil A)
  (λ n xs xss ih → append A m xs (mult n m) ih)
```

This section develops the definitions necessary to understand how to write `concat` by applying the primitive elimination rule for described types to a suitable algebra.

5.1 Primitive Induction Principle

The type of the primitive elimination rule, `ind`, for datatypes built from descriptions is given below. The algebra α is the important argument, as it is the proof that that some property `P` holds for any value of a described type. Whereas an eliminator has separate

branches for proofs about each constructor, `ind` requires a single algebra argument that proves `P` for any constructor.

```
ind : {I : Set} (D : Desc I)
  (P : (i : I) → μ D i → Set)
  (α : (i : I) (xs : El D (μ D) i)
    (ihs : Hyps D (μ D) P i xs) → P i (init xs))
  (i : I) (x : μ D i) → P i x
```

In order to prove `P i (init xs)` you get the following arguments of α:

1. `(i : I)` - The index of the type being eliminated.
2. `(xs : El D (μ D) i)` - The constructors (and their arguments) of the type being eliminated.
3. `(ihs : Hyps D (μ D) P i xs) → P i (init xs))` - The inductive hypotheses for all constructors.

McBride [2011] gives the definition of `ind`, but our work can be understood without knowing the definition.

5.2 Inductive Hypothesis Type

`Hyps` computes the type of inductive hypotheses for a described datatype. Its definition closely follows the definition of the interpretation function of descriptions `El` (Section 3.2). They both compute over a description, D, and in fact Hyps expects one of its arguments, xs, to have the type computed by El.

```
Hyps : {I : Set} (D : Desc I) (X : ISet I)
  (P : (i : I) → X i → Set)
  (i : I) (xs : El D X i) → Set
Hyps (End j) X P i q = ⊤
Hyps (Rec j D) X P i (x , xs) =
  P j x × Hyps D X P i xs
Hyps (Arg A B) X P i (a , b) = Hyps (B a) X P i b
```

First, let's understand Hyps by what it computes for the description of a single constructor like nil or cons. Hyps ignores dependent arguments Arg and moves on, looking for recursive arguments. When finding a recursive argument Rec, it asks for the motive P instantiated at the recursive argument index, j, and value, x. Finally, the tree of inductive hypotheses is terminated by the unit type ⊤ once the description ends in End.

Inductive Hypotheses of a Single Constructor The nil constructor of vectors has neither dependent nor recursive arguments. Thus, Hyps for nilD is simply the unit type. Recall that NilEl is the type that nil's description gets interpreted as. The definition of nilE and related types can be found in Section 3.2.

```
NilHyps : (A : Set)
  (P : (n : ℕ) → Vec A n → Set)
  (n : ℕ) (xs : NilEl A n) → Set
NilHyps A P n xs = Hyps (nilD A) (Vec A) P n xs

NilHyps A P zero refl ⇝ ⊤
```

On the other hand, the cons constructor of vectors requires an inductive hypothesis for its recursive argument.

```
ConsHyps : (A : Set)
  (P : (n : ℕ) → Vec A n → Set)
  (n : ℕ) (xs : ConsEl A n) → Set
ConsHyps A P n xs = Hyps (consD A) (Vec A) P n xs

ConsHyps A P (suc m) (m , x , xs , refl) ⇝
  P m x × ⊤
```

Inductive Hypotheses of Multiple Constructors Once again, multiple constructors are represented by a tagged sum (Section 2.1). Hyps for VecD requires *either* the inductive hypotheses of nil or the inductive hypotheses of cons, depending on which constructor Hyps is applied to.

```
VecHyps : (A : Set)
  (P : (n : ℕ) → Vec A n → Set)
  (n : ℕ) (xs : VecEl A n) → Set
VecHyps A P n xs = Hyps (VecD A) (Vec A) P n xs

VecHyps A P n (nilT  , xs) ⇝ NilHyps  A P n xs
VecHyps A P n (consT , xs) ⇝ ConsHyps A P n xs
```

5.3 Definition of Vector Concatenation via an Algebra

Now we shall define the vector concatenation by applying the primitive elimination rule for described types to an algebra. Below concat is defined as ind applied to the description of vectors, then the goal type as the motive, and finally the algebra concatα. Note that we define the return type of concat to be Concat, allowing us to reuse the return type in later definitions.

```
Concat : (A : Set) (m n : ℕ)
  (xss : Vec (Vec A m) n) → Set
Concat A m n xss = Vec A (mult n m)

concat : (A : Set) (m n : ℕ)
  (xss : Vec (Vec A m) n) → Concat A m n xss
concat A m = ind
  (VecD (Vec A m))
  (Concat A m)
  (concatα A m)
```

Algebra Argument The algebra that defines `concat` takes as arguments the index `n`, the constructors `xss`, and the inductive hypotheses `ihs`. Recall that the type of vector constructors `xss : VecEl (Vec A m) n` is a dependent pair. The domain of the pair is a vector tag `VecT`, and the codomain is the type of arguments corresponding to the constructor represented by the tag. We eliminate the tag using `case` (Section 2.2), and then provide a branches for the `nil` and `cons` constructors.

```
concatα : (A : Set) (m n : ℕ)
  (xss : VecEl (Vec A m) n)
  (ihs : VecHyps (Vec A m) (Concat A m) n xss)
  → Vec A (mult n m)
concatα A m n xss = case (ConcatConvoy A m n)
  (nilBranch A m n , consBranch A m n , tt)
  (proj₁ xss)
  (proj₂ xss)
```

All definitions in this subsection are defined *without* dependent pattern matching to illustrate the exclusive use of our type theory's primitives (`ind`, $\texttt{proj}_1$, `case`, etc). After we case analyze the constructor tag in the first projection of `xss`, we need the *dependent* second projection to reduce to the arguments of the constructor. This can be done by employing the *convoy pattern* [Chlipala, 2011], in which the special motive `ConcatConvoy` is passed to `case`.

Convoy Motive Again, rather than eliminating the pair `xss`, we eliminate the tag in the first projection using `case`. The motive supplied to case thus takes the first projection as an argument. The motive then asks for the type of the second projection (dependent on the argument supplied to the motive) as the argument `xss`, in addition to the remaining argument `ihs`, and then the motive ends with the goal type `Vec A (mult n m)`.


```
ConcatConvoy : (A : Set) (m n : ℕ)
  → VecT → Set
ConcatConvoy A m n t =
  (xss : El (VecC (Vec A m) t) (Vec (Vec A m)) n)
  (ihs : VecHyps (Vec A m) (Concat A m) n (t , xss))
  → Vec A (mult n m)
```

Nil Branch The `nil` branch within the algebra's case analysis receives as arguments the index `n`, the single argument `q`, and a value `u` of type unit as the inductive hypothesis. The argument `q` is not a proper argument of the constructor, but instead the proof `n ≡ zero`, stating that the index `n` is equal to `zero` for the `nil` constructor. One might expect to simply define the `nil` branch of `concat` to return `nil A`. However, the type of the goal is `Vec A (mult n m)` while the type of `nil A` is `Vec A zero`. We can get the type of the goal to reduce to `Vec A (mult zero m)`, and then to `Vec A zero`, by applying our proof that `n ≡ zero` to the equality coercion function `subst`.

```
nilBranch : (A : Set) (m n : ℕ)
  (xss : NilEl (Vec A m) n)
  (ihs : NilHyps (Vec A m) (Concat A m) n xss)
  → Vec A (mult n m)
nilBranch A m n q u = subst
  (λ n → Vec A (mult n m))
  q (nil A)
```

Cons Branch The `cons` branch is defined in much the same way. Note that in AGDA an identifier is treated as single name unless it contains a space. Thus, the argument `n2-xs-xss-q` below is a single variable whose name reminds us of the tuple of constructor arguments that it contains. Because we do not have access to pattern matching, we need to project out each argument. For legibility, we bind the names of the arguments below using a `let` statement. Unlike `nil`, `cons` has proper arguments but its tuple also ends with a proof – the proof that `n ≡ suc n2`. The inductive hypothesis of `concat` is contained in the first projection of the `ih-u` argument, and the second projection is again a value of type unit.

```
consBranch : (A : Set) (m n : ℕ)
  (xss : ConsEl (Vec A m) n)
  (ihs : ConsHyps (Vec A m) (Concat A m) n xss)
  → Vec A (mult n m)
```

```
consBranch A m n n2-xs-xss-q ih-u =
  let n2 = proj₁ n2-xs-xss-q
      xs = proj₁ (proj₂ n2-xs-xss-q)
      q = proj₂ (proj₂ (proj₂ n2-xs-xss-q))
      ih = proj₁ ih-u
  in subst
    (λ n → Vec A (mult n m))
    q (append A m xs (mult n2 m) ih)
```

5.4 You Made It!

Congratulations on making it through this section, you now know how to define dependently typed functions using the primitive elimination rule `ind`! Getting such function definitions right was a grueling experience for the authors, and interactive theorem proving doesn't help much when dealing with types that are so heavily encoded. You can relax knowing that the next section defines a generic standard eliminator that supports programming with described datatypes, instead of using this algebra-based approach.

6. Generic Eliminators

The goal of this section is to *contribute* a novel generic eliminator, `elim`, for datatypes built from descriptions. After partially applying `elim` to an enumeration of constructor names, and a function from tags (indexing into each constructor name) to descriptions for each constructor, the resulting type is precisely the interface of standard eliminators in type theory! This eliminator can be used to define `concat` as follows.

```
concat : (A : Set) (m n : ℕ)
  (xss : Vec (Vec A m) n) → Vec A (mult n m)
```

```
concat A m = elim VecE (VecC (Vec A m))
  (λ n xss → Vec A (mult n m))
  (nil A)
  (λ n xs xss ih → append A m xs (mult n m) ih)
```

The function `concat` is defined in Section 5 by applying the primitive elimination rule `ind` to an algebra. However, functions defined in such a manner are verbose. Instead, now we can define functions using our generic eliminator that once again can be defined in terms of existing primitives without extending the metatheory. This amounts to:

Part$_E$ 1. *Currying constructor arguments in branches.*

Part$_E$ 2. *Inserting an implicit proof in each branch that the constructor has the correct index.*

Part$_E$ 3. *Performing case analysis to break up constructors into branches.*

Part$_E$ 4. *Currying the outer function taking a product of branches.*

In this section we will first focus on single-constructor datatype descriptions, implementing Part$_E$ 1 and Part$_E$ 2. Multi-constructor descriptions represented as sum types are discussed in Section 6.5, and from that point on we focus on implementing Part$_E$ 3 and Part$_E$ 4.

6.1 Uncurried Inductive Hypothesis Algebra Type

In order to implement Part$_E$ 1 and Part$_E$ 2 we must recognize the algebra argument to `ind` as an uncurried function. Below we define `UncurriedHyps` to be a generalized type synonym for the type of the algebra argument `α` to `ind`, where we replace the fixpoint `μ D` with an arbitary type family `X : I → Set`. This is analogous to the generalization `UncurriedEl` of the initial algebra type in Section 4. In fact, because we generalize `UncurriedHyps` to be defined over arbitrary `X` rather than fixpoint `μ D`, we require the extra argument `cn : UncurriedEl D X`, which you can think of as a constructor of `X`.

```
UncurriedHyps : {I : Set}
  (D : Desc I) (X : ISet I)
  (P : (i : I) → X i → Set)
  (cn : UncurriedEl D X)
  → Set
UncurriedHyps D X P cn = ∀ i →
  (xs : El D X i)
  (ihs : Hyps D X P i xs)
  → P i (cn xs)
```

Recognize `UncurriedHyps` as a kind of uncurried function consisting of one regular argument (the index type) and two product arguments (the constructors and inductive hypotheses). Think of `El D X i` as a product of n arguments plus the proof of correct indexing $A_1 \times ... \times A_n \times (j \equiv i)$, `Hyps D X P i xs` as a product of m inductive hypotheses plus unit $B_1 \times ... \times B_n \times \top$, and `X i` as the result type Z.

$$I \to A_1 \times ... \times A_n \times (j \equiv i) \to B_1 \times ... \times B_m \times \top \to Z$$

Uncurried Algebra of a Single Constructor For example, we can use `UncurriedHyps` to define the type of `consBranch` from Section 5.

```
ConsBranch : (A : Set) (m : ℕ) → Set
ConsBranch A m = UncurriedHyps
  (consD (Vec A m))
  (Vec (Vec A m))
  (Concat A m)
  (λ xs → init (consT , xs))

ConsBranch A m ⇝
  (n : ℕ)
  (xss : ConsEl (Vec A m) n)
  (ihs : ConsHyps (Vec A m) (Concat A m) n xss)
  → Vec A (mult n m)
```

6.2 Curried Inductive Hypothesis Algebra Type

Just like in Section 4, now we define the curried version of the inductive hypothesis algebra. Instead of having an index function argument `I : Set`, followed by the two tuple arguments `xs : El D X i` and `ihs : Hyps D X P i xs`, we uncurry both tuple arguments.

```
CurriedHyps : {I : Set} (D : Desc I) (X : ISet I)
  (P : (i : I) → X i → Set)
  (cn : UncurriedEl D X)
  → Set
CurriedHyps (End i) X P cn =
  P i (cn refl)
CurriedHyps (Rec i D) X P cn =
  (x : X i) → P i x
  → CurriedHyps D X P (λ xs → cn (x , xs))
```

```
CurriedHyps (Arg A B) X P cn =
  (a : A)
  → CurriedHyps (B a) X P (λ xs → cn (a , xs))
```

Notice that `CurriedHyps` combines the definitions of `El` and `Hyps`. This can be seen in the `Rec` branch, which asks for the `(x : X i)` argument from `El` and the `P i x` argument from `Hyps`. You can recognize `CurriedHyps` as a curried function that demands index argument I, n constructor arguments as function arguments $A_1 \to ... \to A_n$, m inductive hypotheses as function arguments $B_1 \to ... \to B_m$, and has the result type Z.

$$I \to A_1 \to ... \to A_n \to B_1 \to ... \to B_m \to Z$$

This definition obviously curries arguments, implementing Part_E 1, but it also inserts an implicit proof of index correctness, implementing Part_E 2. In Section 4, we used the same kind of trick to define `CurriedEl` to have an implicit proof instead of asking for it explicitly as demonstrated by `CurriedEl'`. By analogy, we could have defined a version of eliminators that required the user to receive and use an explicit index correctness proof argument as follows.

```
CurriedHyps' : {I : Set} (D : Desc I) (X : ISet I)
  (P : (i : I) → X i → Set)
  (i : I)
  (cn : El D X i → X i)
  → Set
CurriedHyps' (End j) X P i cn =
  (q : j ≡ i) → P i (cn q)
```

```
CurriedHyps' (Rec j D) X P i cn =
  (x : X j) → P j x
  → CurriedHyps' D X P i (λ xs → cn (x , xs))
CurriedHyps' (Arg A B) X P i cn =
  (a : A)
  → CurriedHyps' (B a) X P i (λ xs → cn (a , xs))
```

Notice that in `Rec` case of `CurriedHyps` the motive is applied to a proof of `refl` implicitly, whereas in `CurriedHyps` such a proof must be supplied as the explicit parameter `q`.

Curried Algebra of a Single Constructor Below we apply `CurriedHyps` to the description of the `cons` constructor. This returns the type of the `cons` branch in our eliminator-based definiton of `concat` at the beginning of this section.

```
ConsElimBranch : (A : Set) (m : ℕ) → Set
ConsElimBranch A m = CurriedHyps
  (consD (Vec A m))
  (Vec (Vec A m)) (Concat A m)
  (λ xs → init (consT , xs))

ConsElimBranch A m ⇝
  (n : ℕ)
  (xs : Vec A m)
  (xss : Vec (Vec A m) n)
  (ih : Vec A (mult n m))
  → Vec A (add m (mult n m))
```

This is precisely the expected type of the `cons` branch of an `elimVec`-based definition of `concat`. Because the index proof is implicitly applied, the return type can definitionally reduce from

`Vec A (mult (suc n) m)` to `Vec A (add m (mult n m))`.

6.3 Uncurry Inductive Hypothesis Algebra Function

Shortly, we will be need a function that *uncurries* the inductive hypothesis algebra. Once again, the definition is unremarkable and the type explains it all.

```
uncurryHyps : {I : Set} (D : Desc I) (X : ISet I)
  (P : (i : I) → X i → Set)
  (cn : UncurriedEl D X)
  → CurriedHyps D X P cn → UncurriedHyps D X P cn
uncurryHyps (End .i) X P cn pf i refl tt =
  pf
uncurryHyps (Rec j D) X P cn pf i (x , xs) (ih , ihs)
  uncurryHyps D X P
    (λ ys → cn (x , ys)) (pf x ih) i xs ihs
uncurryHyps (Arg A B) X P cn pf i (a , xs) ihs =
  uncurryHyps (B a) X P
    (λ ys → cn (a , ys)) (pf a) i xs ihs
```

6.4 Curried Induction Principle

Below we define the function `indCurried`. It is like the primitive `ind`, except it takes a curried inductive hypothesis algebra instead of an uncurried one.

```
indCurried : {I : Set} (D : Desc I)
  (P : (i : I) → μ D i → Set)
  (f : CurriedHyps D (μ D) P init)
  (i : I)
  (x : μ D i)
  → P i x
```

```
indCurried D P f i x =
  ind D P (uncurryHyps D (μ D) P init f) i x
```

In Section 4 we wrote a currying function `curryEl`. When introducing values, we have the uncurried initial algebra `init` and need to curry it to get generic constructors. When eliminating using `indCurried`, the user supplies a curried algebra that we uncurry and pass to the primitive elimination rule `ind`.

Because `indCurried` takes `CurriedHyps` as an algebra, it implements Part_E 1 and Part_E 2. Thus, we would have the expected eliminator interface when writing functions with `indCurried` over singleton datatypes built from descriptions – those that do not start with a sum of constructors and instead only have "single constructor" with arguments.

6.5 Sum of Curried Inductive Hypotheses Type

Soon we will implement Part_E 3 by defining a generic eliminator that performs case analysis over datatypes described as a sum (constructors) of products (arguments). We can demand such a datatype in sum-of-products form by parameterizing not by a description, but by an `E : Enum` and a function `C` from tags of that enumeration to descriptions representing the constructor choices. Below is a function that computes the type of the curried inductive hypothesis algebra for some particular constructor of a datatype, where the particular constructor is specified by a tag.

```
SumCurriedHyps : {I : Set}
  (E : Enum) (C : Tag E → Desc I)
  → let D = Arg (Tag E) C in
  (P : (i : I) → μ D i → Set)
  → Tag E → Set
```

```
SumCurriedHyps E C P t =
  let D = Arg (Tag E) C in
  CurriedHyps (C t) (μ D) P (λ xs → init (t , xs))
```

Recall from Section 2 that we defined datatypes like `Vec` in such pieces anyway, namely `VecE` for the enumeration and `VecC` for the function from tags to constructor descriptions. We can use these two pieces to build a description starting with `Arg`, as seen in the `let` bindings above.

Sum of Curried Algebras For example, we can use `SumCurriedHyps` to define a version of `ConsElimBranch` that works for any constructor of `Vec` as specified by a tag.

```
ElimBranch : (t : VecT)
  (A : Set) (m : ℕ) → Set
ElimBranch t A m = SumCurriedHyps VecE
  (VecC (Vec A m)) (Concat A m) t
```

```
ElimBranch consT A m ⇝ ConsElimBranch A m
```

6.6 Uncurried Eliminator

Now we can implement Part_E 3 by specializing an elimination principle to sums-of-products style datatypes, again by parameterizing our function by an enumeration and function from enumeration tags to descriptions for each constructor.

```
elimUncurried : {I : Set}
  (E : Enum) (C : Tag E → Desc I)
  → let D = Arg (Tag E) C in
  (P : (i : I) → μ D i → Set)
```

```
  → Branches E (SumCurriedHyps E C P)
  → (i : I) (x : μ D i) → P i x
elimUncurried E C P cs i x =
  let D = Arg (Tag E) C in
  indCurried D P
    (case (SumCurriedHyps E C P) cs)
    i x
```

While `indCurried` takes a single curried algebra function (`CurriedHyps D (μ D) P init`), `elimUncurried` takes a product (`Branches E (SumCurriedHyps E C P)`) of curried algebra functions, one for each constructor. The implementation of `elimUncurried` uses `indCurried` to perform induction, then in the body of the induction uses `case` to eliminate the branches. Recall that when we defined `concat` in Section 5 with the primitive

`ind`, we first performed the induction using `ind` and then performed case analysis on the sum of constructors. Our new function `elimUncurried` internalizes exactly this pattern.

6.7 Uncurried Branches Type

The `elimUncurried` function is nearly what we expect from a standard eliminator. However, it still takes all branches of the eliminator as a product of arguments. We would like to curry this product, thus implementing Part_E 4. To do this we need a curried and uncurried version of a function whose domain is `Branches` from Section 2. Recall that `Branches` is merely a dependent product of arguments, one for each element in an enumeration. Below is a type synonym for a non-dependent function from `Branches` to some result type.

```
UncurriedBranches : (E : Enum)
  (P : Tag E → Set) (X : Set) → Set
UncurriedBranches E P X = Branches E P → X
```

It is easy to recognize `UncurriedBranches` as a standard uncurried function. Think of `Branches E P` as a product of n arguments $A_1 \times ... \times A_n$, and `X` as the result type Z.

$$A_1 \times ... \times A_n \rightarrow Z$$

6.8 Curried Branches Type

Defining a curried version of a function taking branches is straightforward. Unlike `CurriedEl` and `CurriedHyps`, `CurriedBranches` does not insert an implicit proof of index correctness anywhere, so it really is just a standard curried function.

```
CurriedBranches : (E : Enum)
  (P : Tag E → Set) (X : Set) → Set
CurriedBranches [] P X =
  X
CurriedBranches (l :: E) P X =
  P here → CurriedBranches E (λ t → P (there t)) X
```

The only thing of interest in this definition is incrementing the tag in the motive with `there` in recursive calls, because the motive is dependent on the smaller enumeration `E` in the recursive call.

It is also easy to recognize `CurriedBranches` as a standard curried function, dependent n curried argument $A_1 \times ... \times A_n$ and returning Z.

$$A_1 \rightarrow ... \rightarrow A_n \rightarrow Z$$

6.9 Curry Branches Function

Shortly, we will need a function that *curries* a function that takes branches. Again, this function is not surprising and can be understood from its type.

```
curryBranches :
  {E : Enum} {P : Tag E → Set} {X : Set}
  → UncurriedBranches E P X → CurriedBranches E P X
curryBranches {[]} f =
  f tt
curryBranches {l : : E} f =
  λ c → curryBranches (λ cs → f (c , cs))
```

6.10 Generic Eliminator

At long last, we have come to the grand moment, the definition of the generic eliminator `elim`! With a final flick of the wrist, we apply `curryBranches` to the result of `elimUncurried`.

```
elim : {I : Set} (E : Enum) (C : Tag E → Desc I)
  → let D = Arg (Tag E) C in
  (P : (i : I) → μ D i → Set)
```



```
  → CurriedBranches E
      (SumCurriedHyps E C P)
      ((i : I) (x : μ D i) → P i x)
elim E C P = curryBranches (elimUncurried E C P)
```

Note that the return type of `elim` is specified with

`CurriedBranches`. To see the curry/uncurry resemblence with `elimUncurried`, recognize that the return type of `elimUncurried` can equivalently be written with `UncurriedBranches`.

```
  ...
  → UncurriedBranches E
      (SumCurriedHyps E C P)
      ((i : I) (x : μ D i) → P i x)
⇝
  ...
  → Branches E (SumCurriedHyps E C P)
  → (i : I) (x : μ D i) → P i x
```

In Section 5 we had to do a lot of work to define simple dependently typed functions like `concat` using the algebra-based primitive elimination rule `ind`. In this section we did just as much work, if not more, to define the generic eliminator `elim`. However, this need only be done once and now defining any concrete function like `concat` can be done very tersely using `elim`, just as the example at the beginning of this section demonstrates.

For pedagogical reasons, we presented the definiton of `concat` in terms of `ind` by combining several smaller definitions. This somewhat hides the verbosity of an `ind`-based definition, so we have provided an additional example that illustrates the difference between definitions using `ind` versus `elim`. You can find a definition of vector `append` (adding two vectors) using `elim` in Figure 3. Now you can appreciate `elim` by comparing Figure 3 with the much more verbose definition of `append` using `ind` in Figure 4.

7. Correctness

The goal of this section is to *prove* that the primitive elimination rule `ind` is extensionally equivalent to our generic eliminator `elim`. This amounts to proving:

Soundness

$$\forall a_1 ... a_n .\ \exists \alpha .$$
$$\text{ind}\ (\text{Arg}\ (\text{Tag}\ E)\ C)\ P\ \alpha\ i\ x = \text{elim}\ E\ C\ P\ a_1\ ...\ a_n\ i\ x$$

Completeness

$$\forall \alpha .\ \exists a_1 ... a_n .$$
$$\text{ind}\ (\text{Arg}\ (\text{Tag}\ E)\ C)\ P\ \alpha\ i\ x = \text{elim}\ E\ C\ P\ a_1\ ...\ a_n\ i\ x$$

However, the return type of `elim` is a `CurriedBranches` type, which computes to a type taking n function arguments, one for each constructor branch, and ending with the motive.

$$A_1 \to ... \to A_n \to (i : I)\ (x : \mu\ D\ i) \to P\ i\ x$$

We only get this expanded type if `elim` is applied to a concrete description, otherwise `CurriedBranches` will not unfold. Because of this techninal annoyance, we will prove the equivalence between `ind` and the helper function `elimUncurried` instead, which takes all branches of the eliminator as a single tuple argument.

7.1 Soundness

Formally, the type of soundness of `elimUncurried` with respect to `ind` is defined below. Note that the existential type ($\exists$) is shorthand for a dependent pair type (Σ) whose domain type is inferred.

```
Soundness : Set
Soundness = {I : Set}
  (E : Enum) (C : Tag E → Desc I)
  → let D = Arg (Tag E) C in
  (P : (i : I) → μ D i → Set)
  (β : Branches E (SumCurriedHyps E C P))
  (i : I) (x : μ D i)
  → ∃ λ α
  → ind D P α i x ≡ elimUncurried E C P β i x
```

`Soundness` states that any function defined by `elimUncurried` applied to a tuple of constructor branches (β) – each containing curried arguments and implicit proofs of index correctness – can equivalently expressed by `ind` applied to a suitable algebra (α). In Figure 1 we state and prove soundness informally as a theorem, omitting all but the key function arguments for legibility.

7.2 Completeness

Formally, the type of completeness of `elimUncurried` with respect to `ind` is defined below.

```
Completeness : Set
Completeness = {I : Set}
  (E : Enum) (C : Tag E → Desc I)
  → let D = Arg (Tag E) C in
  (P : (i : I) → μ D i → Set)
  (α : UncurriedHyps D (μ D) P init)
  (i : I) (x : μ D i)
  → ∃ λ β
  → ind D P α i x ≡ elimUncurried E C P β i x
```

`Completeness` is the converse of `Soundess`. It states that any function defined by `ind` applied to a suitable algebra (α), can equivalently be expressed by `elimUncurried` applied to a tuple of constructor branches (β). In Figure 2 we state and prove completeness informally as a theorem, once again omitting all but the key function arguments.

The proof of completeness in Figure 2 uses the following two lemmas. It also uses the definition of the function `toBranches`, which can be found in the accompanying source code. The `toBranches` function just translates an `UncurriedHyps` algebra to `Branches E (SumCurriedHyps E C P)`.

Lemma (ToBranches)**.**

$$case \circ toBranches = curryHyps$$

Proof. By induction on the tag indexing into the enumeration of constructors argument. □

Lemma (CurryHypsIdent)**.**

$$uncurryHyps \circ curryHyps = id$$

Proof. By induction on the description argument. □

8. Related Work

Our work focuses on internalizing the definition of constructors and eliminators in terms of existing primitives that use algebras.

8.1 Generic Programming using Descriptions

There has been a lot of work on performing generic programming over datatypes defined using descriptions. In some sense, this was the original purpose of the description technology. For example, Chapman et al. [2010] define a generic catamorphism (a non-dependent `ind`), and a generic free monad construction. Ornaments [McBride, 2011] support the definition of new description-based

datatypes in terms of their relationship with existing datatypes, and support the conversion between the two. Free conversion between data means that one can reuse functions defined over old types when defining new, more specifically indexed, dependent types, solving a major reuse issue with dependently typed programming. Dagand [2013] implements a generic "deriving" mechanism, similar in purpose to `deriving` in HASKELL [Jones, 2003], that derives functions such as decidable equality over a class of datatypes that support such functions. Dagand [2013] also generically defines constructions [McBride et al., 2006], such as case analysis and injectivity of constructors, that are used when elaborating dependent pattern matching to eliminators.

Chapman et al. [2010] introduced descriptions in a paper that also introduced the technique of *levitation*. Levitation is a technique to reduce the number of type theory primitives, hence the size of a core type theory, by defining certain datatypes that would normally be primitive in terms of descriptions (including descriptions themselves, hence the name "levitation"). While both levitation and a closed type theory based on descriptions were introduced at the same time, the closed type theory can also be defined without levitation. Hence, our present work of generic type theory constructions is orthogonal to whether or not the closed type theory primitives have been levitated.

Dagand and McBride [2012] describe using ornaments to define new functions from old ones, such that the relationship between the two is freely captured. This work uses an alternative, more expressive, description type that makes it possible to define datatypes as computations over their index rather than using the equality type to constrain what the indices must be. We have not extended the present work to computational descriptions, but this should be possible in the same way that Dagand [2013] defines generic operations over computational descriptions that are restricted to a universe of "tagged descriptions" representing sum-of-products style datatypes.

An alternative way to encode datatypes is to support sum types directly in descriptions and use those rather than their isomorphic dependent pair equivalents. Foveran [Atkey, 2011] is an example of a language that encoded sum types directly. Our work could be extended to use descriptions that support primitive sum types. A function like `elim` would still need to be parameterized by an `Enum`-like collection of all constructors, such that the primitive sum description could be computed from the `Enum` in the same way that we use the enumeration to build an `Arg` description.

8.2 Metatheory of Descriptions

Dagand [2013] defines an elaboration procedure to translate `data` declaration syntax to descriptions. As part of the metatheory of this work, Dagand defines and proves a soundness theorem that any high level datatype declaration elaborates to a well-typed term in the kernel type theory. Dagand defines and proves completeness as the extensional equivalence between COQ's `Fix`-based definitions and `ind`-based definitions. This is done at the level of the metatheory of COQ's `Fix`-based definitions, which Giménez [1995] defines in terms of underlying eliminators. Although Dagand does not describe the proof in all of its low-level "symbol-pushing" detail, converting from eliminator-based definitions, to `ind`-based definitions is very similar to what we have described. The difference is that, in our work, this conversion is *internalized*, as we define eliminators in terms of `ind` within the existing type theory, rather than prove an equivalence to eliminators defined at the level of the metatheory.

Besides defining `elim` in terms of `ind`, we also prove the extensional equivalence of both functions as a soundness and completeness theorem. We also expect these theorems to be similar in nature to the proof by Dagand [2013] in terms of the work by Giménez [1995].

Theorem.

$$\forall \beta.\ \exists \alpha.\ \textit{ind}\ \alpha = \textit{elimUncurried}\ \beta$$

Proof.

$$\begin{aligned}
\text{ind}\ \alpha &= \text{elimUncurried}\ \beta && \\
\text{ind}\ \alpha &= \text{indCurried}\ (\text{case}\ \beta) && (\text{by def } \texttt{elimUncurried}) \\
\text{ind}\ \alpha &= \text{ind}\ (\text{uncurryHyps}\ (\text{case}\ \beta)) && (\text{by def } \texttt{indCurried}) \\
\text{ind}\ (\text{uncurryHyps}\ (\text{case}\ \beta)) &= \text{ind}\ (\text{uncurryHyps}\ (\text{case}\ \beta)) && (\text{solve } \alpha := \text{uncurryHyps}\ (\text{case}\ \beta))
\end{aligned}$$

□

Figure 1: Soundness of `elim`

Theorem.

$$\forall \alpha.\ \exists \beta.\ \textit{ind}\ \alpha = \textit{elimUncurried}\ \beta$$

Proof.

$$\begin{aligned}
\text{ind}\ \alpha &= \text{elimUncurried}\ \beta && \\
\text{ind}\ \alpha &= \text{indCurried}\ (\text{case}\ \beta) && (\text{by def } \texttt{elimUncurried}) \\
\text{ind}\ \alpha &= \text{ind}\ (\text{uncurryHyps}\ (\text{case}\ \beta)) && (\text{by def } \texttt{indCurried}) \\
\text{ind}\ \alpha &= \text{ind}\ (\text{uncurryHyps}\ (\text{case}\ (\text{toBranches}\ \alpha))) && (\text{solve } \beta := \text{toBranches}\ \alpha) \\
\text{ind}\ \alpha &= \text{ind}\ (\text{uncurryHyps}\ (\text{curryHyps}\ \alpha)) && (\text{by lemma } \textit{ToBranches}) \\
\text{ind}\ \alpha &= \text{ind}\ \alpha && (\text{by lemma } \textit{CurryHypsIdent})
\end{aligned}$$

□

Figure 2: Completeness of `elim`

8.3 Algebras Defined with Curry

Throughout this paper we have emphasized the verbosity of functions defined in terms of the primitive elimination rule `ind`. McBride [2011] gives examples of functions defined more tersely in terms of `ind` by sprinkling in uses of the `curry` function. We believe that while this makes functions easier to read, they are still difficult to write, even when defining them interactively due to pervasive definitional expansion of encoded constructions.

9. Conclusion & Future Work

Closed dependently typed languages that define datatypes from descriptions offer tremendous generic programming capabilities. However, when programming over particular datatypes within the model of a closed language, it can be useful to not worry about the details of the encodings of description-based datatypes. Thanks to our our generic constructor (`inj`) and generic eliminator (`elim`), users can now optionally program in the IDSL of type theory, without needing to be aware of description-based encodings.

Besides the generic constructors and eliminators we presented here, we have used the same techniques to generically implement type formers. This is made possible by representing datatype parameters and indices explicitly as telescopes. We have also modified the generic constructor and eliminator to be parameter and index aware. Additionally, we have added a distinct *implicit* argument constructor for telescopes and descriptions, allowing users of our IDSL to specify which type parameters, type indices, and constructor arguments should be rendered as implicit arguments. These extensions can be found in the accompanying source code, linked in the introduction.

Acknowledgments

We are indebted to everyone involved with the EPIGRAM [McBride, 2005] language project, from which descriptions sprung, and more broadly everyone involved with defining descriptions. We would also like to thank Nathan Collins for discussing some of this development with us and helping us settle on some terminology. Finally, we are grateful for feedback from anonymous reviewers. This work was supported by NSF/CISE/CCF grant #1320934.

References

R. Atkey. A type checker that knows its monad from its elbow. blog post, Dec. 2011. URL `http://bentnib.org/posts/2011-12-14-type-checker.html`.

E. C. Brady. Idris — systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54. ACM, 2011.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. ISBN

978-1-60558-794-3. . URL http://doi.acm.org/10.1145/1863543.1863547.

A. Chlipala. Certified programming with dependent types, 2011.

P.-E. Dagand. *A Cosmology of Datatypes*. PhD thesis, University of Strathclyde, 2013.

```
append : (A : Set) (m : ℕ) (xs : Vec A m) (n : ℕ) (ys : Vec A n) → Vec A (add m n)
append A = elim VecE (VecC A) (λ m xs → (n : ℕ) (ys : Vec A n) → Vec A (add m n))
  (λ n ys → ys)
  (λ m x xs ih n ys → cons A (add m n) x (ih n ys))
```

Figure 3: Definition of vector append using our generic elim

```
append : (A : Set) (m : ℕ) (xs : Vec A m) (n : ℕ) (ys : Vec A n) → Vec A (add m n)
append A = ind (VecD A) (λ m xs → (n : ℕ) (ys : Vec A n) → Vec A (add m n))
  (λ m t-c → case
    (λ t → (c : El (VecC A t) (Vec A) m)
           (ih : Hyps (VecD A) (Vec A) (λ m xs → (n : ℕ) (ys : Vec A n) → Vec A (add m n)) m (t , c))
           (n : ℕ) (ys : Vec A n) → Vec A (add m n)
    )
    ( (λ q ih n ys → subst (λ m → Vec A (add m n)) q ys)
    , (λ m2-x-xs-q ih-u n ys →
        let m2 = proj₁ m2-x-xs-q
            x = proj₁ (proj₂ m2-x-xs-q)
            q = proj₂ (proj₂ (proj₂ m2-x-xs-q))
            ih = proj₁ ih-u
        in
        subst (λ m → Vec A (add m n)) q (cons A (add m2 n) x (ih n ys))
      )
    , tt
    )
    (proj₁ t-c)
    (proj₂ t-c)
  )
```

Figure 4: Definition of vector append using the primitive ind

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 103–114, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. . URL http://doi.acm.org/10.1145/2364527.2364544.

E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for proofs and Programs*, pages 39–59. Springer, 1995.

S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

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

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. Ornamental algebras, algebraic ornaments. 2011.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, pages 186–200. Springer, 2006.

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

The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2008. URL `http://coq.inria.fr`.