Generic Lookup and Update for Infinitary Inductive-Recursive Types

Larry Diehl
Portland State University, USA
ldiehl@cs.pdx.edu

Tim Sheard
Portland State University, USA
sheard@cs.pdx.edu

Abstract

The class of Infinitary inductive-recursive (InfIR) types is commonly used to model type theory within itself. While it is common and convenient to provide examples of values within an InfIR model, writing functions that manipulate InfIR types is an under-explored area due to their inherent complexity.

Our goal in this work is to push the boundaries of programming with InfIR types by introducing two functions operating over them. The first is a lookup function to extract sub-components from an InfIR type, and the second is an update function to replace sub-components within an InfIR type. We start by considering how to write such functions for concrete examples of InfIR types, and then show how to write generic versions of the functions for any datatype definable in the universe of InfIR types. We actually write two versions of the generic functions, one where the universe is open and another where the universe is closed.

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

Keywords Dependent types; induction-recursion; generic programming.

1. Introduction

Infinitary inductive-recursive (InfIR) types are commonly used in dependently typed programs to model type-theoretic universes (Martin-Löf 1984). For example, consider the Agda (Norell 2007) model below of the universe of natural numbers and dependent functions.[1]

```
mutual
  data Type : Set where
    `Nat : Type
    `Fun : (A : Type) (B : ⟦ A ⟧ → Type) → Type

  ⟦_⟧ : Type → Set
  ⟦ `Nat ⟧ = ℕ
  ⟦ `Fun A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
```

This Type is *infinitary* because the \`Fun constructor's second inductive argument (*B*) is a function (hence Types can branch infinitely). Additionally, it is *inductive-recursive* because it is mutually defined with a function (⟦_⟧) operating over it.

Once you have defined a model, it is also common to provide a few examples of values that inhabit it. For example, below (NumFun) is a function Type that takes a natural number *n* as input, then asks you to construct a natural number from *n* additional natural number arguments.

```
NumArgs : ℕ → Type
NumArgs zero = `Nat
NumArgs (suc n) = `Fun `Nat (const (NumArgs n))

NumFun : Type
NumFun = `Fun `Nat NumArgs
```

While defining models and example values using infinitary inductive-recursive types is common, writing inductively defined *functions* over them is less so.

Why are there so few examples of functions over infinitary inductive-recursive types? Because they contain inherently complex properties. Their infinitary nature makes them *higher-order datatypes*, rather than simpler first-order datatypes. Their inductive-recursive nature means you need to deal with *dependencies* between arguments and *mutual functions* too.

Functional programming languages typically package useful datatypes (like Lists and Vectors) with useful operations (like lookup and update) in their standard libraries. Additionally, *generic* implementations of such operations may exist as libraries for any other user-defined datatypes.

Our *primary contribution* is to show how to write two particular operations over infinitary inductive-recursive types (such as Type universes), and then generalize those operations from functions over concrete datatypes to generic functions over any user-defined datatype. The first operation is lookup, allowing data within an InfIR type to be extracted. The second operation is update, allowing a value within an InfIR type to be replaced by another value. We also contribute a Path type used by lookup and update to point at a particular position within a datatype. More specifically, we contribute Path, lookup, and update for:

- A concrete large InfIR type, Type, in Section 3.
- A concrete small InfIR type, Arith, in Section 4.
- A generic universe for an open theory of types, in Section 5.
- A generic universe for a closed theory of types, in Section 6.

[1] This paper is written as a literate Adga program. The literate Agda source file and other accompanying code can be found at `https://github.com/larrytheliquid/infir`

Finally, we hope that seeing examples of writing both concrete and generic functions using infinitary inductive-recursive types will help future dependently typed functional programmers with writing their own functions over this class of datatypes.

2. The problem

Before describing why writing functions over InfIR types is difficult, we first consider writing analogous functions over simpler datatypes. Thereafter we point out what becomes difficult in the InfIR scenario, and describe a general methodology for writing total functions in a dependently typed language, which can be applied to successfully write InfIR functions.

For readers of the colored version of this paper, we use the following Agda source code highlighting color conventions: Keywords are orange, datatypes are dark blue, constructors are green, functions are light blue, and *variables* are purple.

2.1 Background

Let us first consider writing lookup for a simple binary Tree.

```
data Tree : Set where
  leaf : Tree
  branch : (A B : Tree) → Tree
```

Our Tree stores no additional data in nodes, can have binary branches, and ends with a leaf. It is easy to work with because it is first-order, has no dependencies between arguments, and has no mutually defined functions.

If we want to lookup a particular subTree, we must first have a way to describe a Path that indexes into our original tree.

```
data Path : Tree → Set where
  here : ∀{A} → Path A
  there₁ : ∀{A B}
    → Path A
    → Path (branch A B)
  there₂ : ∀{A B}
    → Path B
    → Path (branch A B)
```

The here constructor indicates that we have arrived at the subtree we would like to visit. The there_1 constructor tells us to take a left turn at a branch, while there_2 tells us to take a right turn. In general, we adopt the convention that a numerical subscript after a there constructor of a Path indicates which argument to point to (we use one-based indexing rather than zero-based indexing).

Once we have defined Paths into a Tree, it is straightforward to define lookup by following the Path until we arrive at the subtree indicated by the here constructor of Path.

```
lookup : (A : Tree) → Path A → Tree
lookup A here = A
lookup (branch A B) (there₁ i) = lookup A i
lookup (branch A B) (there₂ i) = lookup B i
```

2.2 lookup with a computational return type

Now let's consider writing a total lookup function for polymorphic Lists (instead of the binary Trees above), where the return type of lookup is dynamically computed. Below is the List and its Path.

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A
```

```
data Path {A : Set} : List A → Set where
  here : ∀{xs} → Path xs
  there₁ : ∀{x xs} → Path (cons x xs)
  there₂ : ∀{x xs}
    → Path xs
    → Path (cons x xs)
```

The here and there_2 constructors are analogous to those for Tree Paths. However, there_1 points to a non-inductive A value, the first argument to cons, whereas this pointed to an inductive subtree in the Tree scenario.

In the (traditionally) non-dependent Haskell (Jones 2003) language there are two distinct lookup-like functions for lists.

```
drop :: Int -> [a] -> [a]
(!!) :: [a] -> Int -> a
```

The first (`drop`) looks up inductive sublists, and the second `(!!)` looks up non-inductive `a` values. A dependently typed language like Agda allows us to a write a single function that may return a List or an A, depending on what the input Path points to. Note that below {A = A} is Agda notation for binding an implicit argument explicitly.

```
Lookup : {A : Set} (xs : List A) → Path xs → Set
Lookup {A = A} xs here = List A
Lookup {A = A} (cons x xs) there₁ = A
Lookup (cons x xs) (there₂ i) = Lookup xs i
```

```
lookup : {A : Set} (xs : List A) (i : Path xs) → Lookup xs i
lookup xs here = xs
lookup (cons x xs) there₁ = x
lookup (cons x xs) (there₂ i) = lookup xs i
```

The Lookup function *computes* the return type of lookup, allowing lookup to return either a List or an A (the base cases of Lookup). We will refer to functions like Lookup as *computational return types*.

In the colored version of this paper, you can spot a computational type because it is a light blue Function, whereas a non-computational Datatype is dark blue. Both computational and non-computational types are capitalized by convention.

2.3 head with a computational argument or return type

Once we move from finitary non-dependent types like Tree and List to an InfIR type like Type, it is no longer obvious how to write a function like lookup. Looking up something in the left side (domain) of a 'Fun is easy, but looking up something in the right side (codomain) requires entering a function space.

Figuring out how to write functions like lookup (and more complicated functions) over InfIR types is the subject of this paper. The solution (given in the next section) involves a more complicated version of the computational return type Lookup above. But, let us first consider a general methodology for turning a function that would otherwise be partial into a total function. For example, say we wanted to write a total version of the typically partial head function.

```
head : {A : Set} → List A → A
```

We have 2 options to make this function total. We can either:

1. Change the domain, for example by requiring an extra default argument.

```
head₁ : {A : Set} → List A → A → A
head₁ nil y = y
head₁ (cons x xs) y = x
```

2. Change the codomain, for example by returning a Maybe result.

```
head₂ : {A : Set} → List A → Maybe A
head₂ nil = nothing
head₂ (cons x xs) = just x
```

Both options give us something to do when we apply head to an empty list: either get an extra argument to return, or we simply return nothing. However, these options are rather extreme as they require changing our intended type signature of head for *all* possible lists. The precision of dependent types allows us to instead conditionally ask for an extra argument, or return nothing of computational value, only if the input list is empty!

First, let's use dependent types to conditionally change the domain. We ask for an extra argument of type A if the List is empty. Otherwise, we ask for an extra argument of type unit ($\top$), which is isomorphic to not asking for anything extra at all. Below, HeadArg is type of the extra argument, which is dependent on the input xs of type List. We call functions like HeadArg *computational argument types*.

```
HeadArg : {A : Set} → List A → Set
HeadArg {A = A} nil = A
HeadArg (cons x xs) = ⊤

head₃ : {A : Set} (xs : List A) → HeadArg xs → A
head₃ nil y = y
head₃ (cons x xs) tt = x
```

Second, let's use dependent types to conditionally change the codomain. HeadRet computes our new return type, conditionally dependent on the input list (it is a *computational return type*). If the input list is empty, our head_4 function returns a value of type unit ($\top$). If it is non-empty, it returns an A. Note that returning a value of $\top$ is returning nothing of computational significance. Hence, it is as if head_4 is not defined for empty lists.

```
HeadRet : {A : Set} → List A → Set
HeadRet nil = ⊤
HeadRet {A = A} (cons x xs) = A

head₄ : {A : Set} (xs : List A) → HeadRet xs
head₄ nil = tt
head₄ (cons x xs) = x
```

We have seen how to take a partial function and make it total, both with and without the extra precision afforded to us by dependent types (via computational argument and return types). We would like to emphasize that the extra argument HeadArg in head_3 is not merely a precondition, but rather extra computational content that must be supplied by the program to complete the cases that would normally make it a partial function. To see the difference, consider a total version of a function that looks up elements of a List, once given a natural number ($\mathbb{N}$) index.

```
elem : {A : Set} (xs : List A) (n : ℕ) → n < length xs → A
```

Because the natural number n may index outside the bounds of the list xs, we need an extra argument serving as a precondition. If this precondition (established using $<$ above) is satisfied, it computes to the unit type ($\top$), but if it fails it computes to the empty type ($\bot$). So, in the failure case the precondition ($\bot$) is unsatisfiable, whereas the failure case of HeadArg is the extra argument A needed to complete the otherwise partial function.

The rest of this paper expands on the ideas of this section by defining functions like HeadArg that non-trivially compute extra arguments. These dependent extra arguments are the key to writing functions over InfIR datatypes.

3. Large InfIR Type

Section 2 reviews how to lookup subTrees, subLists, and subelements pointed to by Paths. In this section we define the corresponding datatypes and functions for InfIR Types.

3.1 Type

The InfIR Type used in this section is another type universe, similar to the one in Section 1. The Type universe is still closed under functions, but now the 'Base types are parameters (of type Set) instead of being hardcoded to $\mathbb{N}$.

```
mutual
  data Type : Set₁ where
    'Base : Set → Type
    'Fun : (A : Type) (B : ⟦ A ⟧ → Type) → Type

  ⟦_⟧ : Type → Set
  ⟦ 'Base A ⟧ = A
  ⟦ 'Fun A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
```

3.2 Path

Let's reconsider what it means to be a Path. You can still point to a recursive Type using here. Now you can also point to a nonrecursive A of type Set using thereBase.

When traversing a Tree, you can always go left or right at a branch. When traversing a Type, you can immediately go to the left of a 'Fun, but going right requires first knowing which element a of the type family $B\ a$ to continue traversing under. This requirement is neatly captured as a dependent function type of the f argument below.

```
data Path : Type → Set₁ where
  here : ∀{A} → Path A
  thereBase : ∀{A} → Path ('Base A)
  thereFun₁ : ∀{A B}
    (i : Path A)
    → Path ('Fun A B)
  thereFun₂ : ∀{A B}
    (f : (a : ⟦ A ⟧) → Path (B a))
    → Path ('Fun A B)
```

Above, thereFun_2 represents going right into the codomain of 'Fun, but only once the user tells you which a to use. In a sense, going right is like asking for a continuation that tells you where to go next, once you have been given a. Also note that because the argument f of thereFun_2 is a function that returns a Path, the Path datatype is infinitary (just like the Type it indexes).

3.3 Lookup & lookup

We were able to write a total function to lookup any subTree, but looking up a subType is not always possible. It is not possible because looking up a value in the codomain of a ‘Fun requires extra information, namely the branch of the codomain containing our desired subType. Using the methodology from Section 2.3, we can make lookup for Types total by choosing to change the codomain, depending on the input Type and Path. Lookup (a computational return type) computes the codomain of lookup, asking for a Type or Set in the base cases, or a continuation when looking to the right of a ‘Fun.

$$
\begin{array}{l}
\text{Lookup} : (A : \text{Type}) \rightarrow \text{Path}\ A \rightarrow \text{Set}_1 \\
\text{Lookup}\ A\ \text{here} = \text{Type} \\
\text{Lookup}\ (\text{‘Base}\ A)\ \text{thereBase} = \text{Set} \\
\text{Lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_1\ i) = \text{Lookup}\ A\ i \\
\text{Lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ f) = \\
\quad (a : [\![A]\!]) \rightarrow \text{Lookup}\ (B\ a)\ (f\ a)
\end{array}
$$

Finally, we can write lookup in terms of Path and Lookup. Notice that users applying our lookup function need to supply extra a arguments exactly when they go to the right of a ‘Fun. Thus, our definition can expect an extra argument a in the thereFun_2 case.

$$
\begin{array}{l}
\text{lookup} : (A : \text{Type})\ (i : \text{Path}\ A) \rightarrow \text{Lookup}\ A\ i \\
\text{lookup}\ A\ \text{here} = A \\
\text{lookup}\ (\text{‘Base}\ A)\ \text{thereBase} = A \\
\text{lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_1\ i) = \text{lookup}\ A\ i \\
\text{lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ f) = \\
\quad \lambda\ a \rightarrow \text{lookup}\ (B\ a)\ (f\ a)
\end{array}
$$

3.4 Update & update

Now we will write an update function for Types. After supplying a Path and a substitute Type, update should return the original Type but with the substitute replacing what the Path pointed to. To make updating the InfIR Type more convenient (for the caller of update), the type of the substitute will actually be Maybe Type, where nothing causes an identity update. We might expect to write a function like:

$$
\begin{array}{l}
\text{updateNaive} : \\
\quad (A : \text{Type})\ (i : \text{Path}\ A)\ (X : \text{Maybe Type}) \rightarrow \text{Type}
\end{array}
$$

Above X is the intended Type to Maybe substitute at position i. In order to write a total version of updateNaive, we need to change the domain by asking for an a whenever we update within the codomain of a ‘Fun.

We call the type of the value to substitute Update (a computational argument type), which asks for a Maybe Type or a Maybe Set in the base cases (here and thereBase respectively), and a continuation in the thereFun_2 case. However, updating an element to the left of a ‘Fun is also problematic. We would like to keep the old ‘Fun codomain B unchanged, but it still expects an a of the original type $[\![A]\!]$. Therefore, the thereFun_1 case must ask for a forgetful function f that maps newly updated a’s to their original type.

$$
\begin{array}{l}
\text{Update} : (A : \text{Type}) \rightarrow \text{Path}\ A \rightarrow \text{Set}_1 \\
\text{update} : (A : \text{Type})\ (i : \text{Path}\ A)\ (X : \text{Update}\ A\ i) \rightarrow \text{Type} \\
\\
\text{Update}\ A\ \text{here} = \text{Maybe Type} \\
\text{Update}\ (\text{‘Base}\ A)\ \text{thereBase} = \text{Maybe Set} \\
\text{Update}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_1\ i) = \\
\quad \Sigma\ (\text{Update}\ A\ i)\ (\lambda\ X \rightarrow [\![\text{update}\ A\ i\ X]\!] \rightarrow [\![A]\!]) \\
\text{Update}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ f) = \\
\quad (a : [\![A]\!]) \rightarrow \text{Update}\ (B\ a)\ (f\ a)
\end{array}
$$

$$
\begin{array}{l}
\text{update}\ A\ \text{here}\ X = \text{maybe id}\ A\ X \\
\text{update}\ (\text{‘Base}\ A)\ \text{thereBase}\ X = \text{‘Base}\ (\text{maybe id}\ A\ X) \\
\text{update}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_1\ i)\ (X , f) = \\
\quad \text{‘Fun}\ (\text{update}\ A\ i\ X)\ (\lambda\ a \rightarrow B\ (f\ a)) \\
\text{update}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ f)\ h = \\
\quad \text{‘Fun}\ A\ (\lambda\ a \rightarrow \text{update}\ (B\ a)\ (f\ a)\ (h\ a))
\end{array}
$$

Notice that we must define Update and update mutually, because the forgetful function f (the codomain of Σ in the thereFun_1 case of Update) must refer to update in its domain. Although the thereFun_1 case of update only updates the domain of ‘Fun, the type family B in the codomain expects an a of type $[\![A]\!]$, so we use the forgetful function f to map back to a’s original type.

The base cases (here and thereBase) of update perform updates using the substitute X (where nothing results in an identity update). The thereFun_2 case of update leaves the domain of ‘Fun unchanged, and recursively updates the codomain using the substitute continuation h.

Note that we could have defined Update as an inductive type, rather than a computational type. If we had done so, then it would be an InfIR type with update as its mutually defined function!

3.5 Universal versus Existential Path

When you first encounter the Path datatype of Section 3.2, its thereFun_2 constructor may seem confusing and unnecessarily complex. Its thereFun_2 constructor takes an infinitary argument, allowing you to index *all* branches of the codomain of a ‘Fun (hence we might call the Section 3.2 definition a *universal* Path). The Section 3.2 Path is actually single path when indexing a normal argument, but a multipath when indexing an infinitary argument.

You might wonder if we can get away with an arguably simpler *existential* version of Path, where the thereFun_2 constructor has the following type.

$$
\begin{array}{l}
\text{thereFun}_2 : \forall \{A\ B\} \\
\quad (a : [\![A]\!]) \\
\quad (i : \text{Path}\ (B\ a)) \\
\quad \rightarrow \text{Path}\ (\text{‘Fun}\ A\ B)
\end{array}
$$

Above, thereFun_2 takes a single a used to indicate which branch of B to index (compare this to the function indexing all branches of B in Section 3.2).

Now the thereFun_2 case of Lookup merely recurses rather than returning a Π type.

$$
\text{Lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ a\ i) = \text{Lookup}\ (B\ a)\ i
$$

Similarly, the thereFun_2 case of lookup merely recurses rather than returning a function.

$$
\text{lookup}\ (\text{‘Fun}\ A\ B)\ (\text{thereFun}_2\ a\ i) = \text{lookup}\ (B\ a)\ i
$$

Unfortunately, while existential Path lookup works reasonably well, existential Path update has a severe limitation. Imagine updating the codomain of a ‘Fun whose domain is the type of natural numbers. Using an existential Path, we could start by updating the zero branch, then the one branch, then the two branch, etc. However, we would never be able to finish updating our ‘Fun for *all* natural number branches.

We do not define existential Path update below because of the aforementioned limitation, but even defining the limited version would be painful. In order to update a single branch but also use the old values for all other branches, we need to require decidable equality for the domain of branches. This decidable equality requirement would disallow updates of ‘Fun values whose domain contains another ‘Fun, yet another limitation of existential Path update! It should now be apparent why we used a universal Path in Section 3.2 and the remaining parts of this paper.

4. Small InfIR Arith

Section 3 shows how to define lookup and update for the large InfIR Type. Type is called *large* because the codomain of its IR function ⟦_⟧ has type Set. In this section we adapt our work to a small InfIR type called Arith (it is called *small* because the codomain of its IR function is *not* Set), which is structurally similar to Type. We borrow the Arith type from Hancock et al. (2013).

4.1 Arith

The InfIR Arith used in this section is structurally similar to Type from Section 1. One difference is that the base constructor (‘Num), contains a ℕatural number (rather than a Set, like ‘Base). The other difference is that the mutually defined function eval returns a ℕ (rather than a Set, like ⟦_⟧.)

```
mutual
  data Arith : Set where
    ‘Num : ℕ → Arith
    ‘Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith

  eval : Arith → ℕ
  eval (‘Num n) = n
  eval (‘Prod a f) = prod (eval a) (λ a → eval (f a))
```

Values of type Arith encode “Big Pi” mathematical arithmetic product equations up to some finite bound, such as the one below.

$$\prod_{i=1}^{3} i$$

```
six : Arith
six = ‘Prod (‘Num 3) (λ i → ‘Num (num i))
```

An Arith equation may be nested in its upper bound or body, but the lower bound is always the value 1. Note that above we define six with the helper function num, which converts the finite set value *i* to a natural number using one-based indexing.

The eval function interprets the equation as a natural number, using the helper function prod to multiply a finite number *n* of other natural numbers together.

```
prod : (n : ℕ) (f : Fin n → ℕ) → ℕ
prod zero f = suc zero
prod (suc n) f = f zero * prod n (f ∘ suc)
```

4.2 Pathℕ & lookupℕ & updateℕ

The major difference between the base case ‘Num of Arith, and ‘Base of Type, is that the former contains a ℕ while the latter contains a Set. The lookup for Type had no choice but to return the value of type Set in the ‘Base case. We cannot look further into the value of type Set because Agda does not support type case. In contrast, we can continue to lookup into a substructure of ℕ in the base case ‘Num of lookup for Arith. For this reason, we need the Pathℕ, lookupℕ, and updateℕ definitions for natural numbers.

Pathℕ is an index into the number, which can point to that number or any smaller number. It is different from the standard finite set type Fin because the number pointed to may be zero.

```
data Pathℕ : ℕ → Set where
  here : {n : ℕ} → Pathℕ n
  there : {n : ℕ}
    (i : Pathℕ n)
    → Pathℕ (suc n)
```

The lookup function simply returns the ℕ pointed to by Pathℕ. It has a non-computational return type because a Pathℕ always points to a ℕ.

```
lookupℕ : (n : ℕ) → Pathℕ n → ℕ
lookupℕ n here = n
lookupℕ (suc n) (there i) = lookupℕ n i
```

The update function replaces a sub-number within a ℕ with a Maybe ℕ. The nothing case performs an identity update, while just *n* replaces the sub-number with *n*.

```
updateℕ : (n : ℕ) → Pathℕ n → Maybe ℕ → ℕ
updateℕ n here x = maybe id n x
updateℕ (suc n) (there i) x = suc (updateℕ n i x)
```

4.3 Path & L/lookup & U/update

The Path, lookup, and update definitions for Arith are almost textually identical to the corresponding definitions for Type from Section 3. Thus, we will only cover the ‘Num cases of these definitions. The old Type definitions will work for the other cases by textually substituting Arith for Type, ‘Prod for ‘Fun, and by defining the following type synonym.

```
⟦_⟧ : Arith → Set
⟦ A ⟧ = Fin (eval A)
```

The thereNum case of Path can point somewhere deeper into a substructure of the natural number contained by ‘Num by using a Pathℕ.

```
data Path : Arith → Set where
  thereNum : {n : ℕ} → Pathℕ n → Path (‘Num n)
```

The ‘Num case of Lookup results in a natural number.

```
Lookup (‘Num n) (thereNum i) = ℕ
```

The ‘Num case of lookup continues to lookupℕ the number contained inside.

```
lookup (‘Num n) (thereNum i) = lookupℕ n i
```

The ‘Num case of Update allows the user to supply a Maybe ℕ, representing either the identity update or a number to update with.

```
Update (‘Num n) (thereNum i) = Maybe ℕ
```

The ‘Num case of update leaves ‘Num unchanged, but replaces the natural number contained using updateℕ.

```
update (‘Num n) (thereNum i) X = ‘Num (updateℕ n i X)
```

5. Generic Open InfIR

In this section we develop generic versions of the datatypes and functions from previous sections, for any datatype encoded as an inductive-recursive Dybjer-Setzer code (Dybjer and Setzer 1999; Dybjer 2000).

5.1 Desc

First let us recall the type of inductive-recursive codes developed by Dybjer and Setzer. We refer to values of Desc defined below as "codes".[2] A Desc simultaneously encodes the definition of a datatype and a function mutually defined over it.

```
data Desc (O : Set) : Set₁ where
  End : (o : O) → Desc O
  Arg : (A : Set) (D : (a : A) → Desc O) → Desc O
  Rec : (A : Set) (D : (o : A → O) → Desc O) → Desc O
```

To a first approximation, a datatype Description encodes the type signature of a single constructor, and the value returned by the case of that constructor for the mutually defined function. End is used to specify that a constructor takes no further arguments. However, the user must supply a value o of type O to define the value returned by the mutually defined function. Arg is used to specify a non-recursive argument of a constructor, a of type A, and the remainder of the Desc may depend on the value a. Rec is used to specify a recursive argument (of the type currently being specified). More generally, the recursive argument may be a function type (encoding an *infinitary* argument) whose codomain is the type currently being defined but whose domain may be non-recursive.[3] Above, the domain of the function is some non-recursive type A, and the remainder of the Desc may depend on a function o from A to O, representing the result of applying the mutually defined function to the recursive argument being specified.

Note that we can encode a "first-order" recursive argument by applying Rec to the unit type ⊤. This will actually encode a higher-order recursive argument, but the domain will be trivially inhabited. Similarly, we can encode a "non-inductive-recursive" datatype (one without a mutual function, like ℕ) by making the output argument O of Desc be the unit type. In fact, we will still encode a mutual function, but it will trivially always return unit.

Finally, to encode multiple constructors as a Desc, you simply define an Arg whose domain is a finite enumeration of types (representing each constructor, like ArithT below), and whose codomain is the Desc corresponding to the arguments and recursive cases for each constructor.

The abstract nature of Desc makes it somewhat difficult to understand at first, especially the Rec constructor. Let's try to understand Desc better with an example, encoding Arith from Section 4 below.

```
data ArithT : Set where
  NumT ProdT : ArithT

ArithD : Desc ℕ
ArithD = Arg ArithT λ
  { NumT → Arg ℕ (λ n → End n)
  ; ProdT
    → Rec ⊤ λ n
    → Rec (Fin (n tt)) λ f
    → End (prod (n tt) f)
  }
```

The Desc begins with an Arg, taking sub-Descs for each element of the finite enumeration ArithT, representing the types of each Arith constructor.

The second argument to Arg is an anonymous function that makes use of Agda's pattern matching lambda syntax, where cases appear between braces and each case is separated by a semicolon. In this syntax the constructor being matched and the definition are separated by an Agda arrow (rather than an equal sign). Additionally, we note that the scope of Agda lambdas extends all the way to the right, allowing us to omit many parentheses for lambdas appearing after uses of Arg and Rec.

The NumT description uses Arg to take a natural number (ℕ), then Ends with that number. Ending with that number encodes that the 'Num case of the eval from Section 4 returns the number held by 'Num in the base case.

The ProdT description uses Rec twice, taking two recursive arguments. The first recursive argument is intended to encode an Arith rather than a function type, so we make its domain a value of the trivial type ⊤. The second recursive argument is intended to encode a function from Fin n to Arith, so we ask for a Fin (n tt), where n represents the value returned by applying eval to the first recursive argument. In fact, n represents a function from the trivial type ⊤ to ℕ, because first-order recursive arguments are encoded as higher-order arguments with a trivial domain. Finally, End is used to specify that there are no further arguments, and the 'Prod case of eval should result in the product represented by the first two recursive arguments.

5.2 Data

In the previous subsection we used Desc to encode a datatype (Arith) and its mutual function (eval). In this section we define how to extract these two constructions from the description. Applying the Data type former to a description results in the datatype it encodes, and applying the fun function to a description results in the mutual function it encodes.

Data is defined in terms of a single constructor con, which holds a dependent product (nested dependent pairs) of all arguments of a particular constructor. The computational argument type Data′ computes the type of this product, dependent on the Description that Data is parameterized by.

For the remainder of the paper we employ a convention for functions ending with a prime, like Data′. They will be defined by induction over a description, but must also use the original description they are inducting over in the Rec case. Hence, they take two Desc arguments, where the first R is the original description (to be used in Recursive cases), and the second D is the one we induct over.

```
data Data {O : Set} (D : Desc O) : Set where
  con : Data′ D D → Data D

Data′ : {O : Set} (R D : Desc O) → Set
Data′ R (End o) = ⊤
Data′ R (Arg A D) = Σ A (λ a → Data′ R (D a))
Data′ R (Rec A D) =
  Σ (A → Data R) (λ f → Data′ R (D (fun R ∘ f)))
```

The End case means no further arguments are needed, so we ask for a trivial value of type ⊤. The Arg case asks for a value of type A, which the rest of the arguments may depend on using a. The Rec case asks for a function from A to a recursive value Data R,

[2] We have renamed the original Dybjer-Setzer constructions to emphasize their meaning in English. The original names of our Desc/End/Arg/Rec constructions are IR/ι/σ/δ respectively.

[3] The domain is restricted to be non-recursive to enforce that encoded datatypes are strictly positive.

and the rest of the arguments may use f to depend on the result of applying the mutual function (e.g. eval) to the recursive argument after applying a value of type A.

Next we define fun (encoding the mutual function) in terms of fun′.

```
fun : {O : Set} (D : Desc O) → Data D → O
fun D (con xs) = fun′ D D xs

fun′ : {O : Set} (R D : Desc O) → Data′ R D → O
fun′ R (End o) tt = o
fun′ R (Arg A D) (a , xs) = fun′ R (D a) xs
fun′ R (Rec A D) (f , xs) = fun′ R (D (λ a → fun R (f a))) xs
```

The End case gives us what we want, the value o that the mutual function should return for the encoded constructor case. The Arg and Rec cases recurse, looking for an End.

5.3 A schema for generic functions

In this section the schema used for writing a generic function is to write a pair of generic functions.

```
generic : {O : Set} (D : Desc O) → Data D → ETC
generic′ : {O : Set} (R D : Desc O) → Data′ R D → ETC
```

The first function always has a type prefix like generic, being defined by induction on the constructor of a Datatype (the rest of the arguments and return type go in the ETC position).

The second function always has a type prefix like generic′, being defined by induction on the *arguments* of a constructor (Data′).

You have already seen one such pair in the definition of Data, namely fun and fun′. Furthermore, generic programs often follow a similar recursion pattern as the one described above for fun and fun′. For example, it is common for generic′ to call generic with R in the Rec case.

5.4 Path

Now we will encode a generic Path type, that can be used to index into any inductive-recursive value encoded by applying Data to a Desc.

```
data Path {O : Set} (D : Desc O) : Data D → Set₁ where
  here : ∀{x} → Path D x
  there : ∀{xs}
    → Path′ D D xs
    → Path D (con xs)
```

A Path uses here to immediately point to the current constructor. It uses there to point into one of the arguments of the current constructor, using Path′ as a sub-index.

5.5 Path′

A Path′ points to an argument of a constructor, one of the values of the dependent product computed by Data′.

```
data Path′ {O : Set} (R : Desc O)
  : (D : Desc O) → Data′ R D → Set₁ where
  thereArg₁ : ∀{A D a xs}
    → Path′ R (Arg A D) (a , xs)
  thereArg₂ : ∀{A D a xs}
    (i : Path′ R (D a) xs)
    → Path′ R (Arg A D) (a , xs)
  thereRec₁ : ∀{A D f xs}
    (g : (a : A) → Path R (f a))
    → Path′ R (Rec A D) (f , xs)
  thereRec₂ : ∀{A D f xs}
    (i : Path′ R (D (fun R ∘ f)) xs)
    → Path′ R (Rec A D) (f , xs)
```

The thereArg_1 case points immediately to a non-recursive value of type A. Recall thereBase from Section 3, which points immediately to a non-recursive value of type Set. The thereBase case cannot index further into non-recursive Sets because values of type Set cannot be case-analyzed. Similarly, the thereArg_1 case of our open universe generic Path′ cannot index further into A, because the type of A is Set and cannot be case-analyzed. For this reason, Path′ does not adequately capture concrete paths for types like Arith of Section 4, which has a $\mathbb{N}$ in the ‘Num case that we would like to index into. This is a limitation due to using open universe Descriptions, which we remedy using a closed universe in Section 6.

The thereArg_2 case points to a sub-argument, skipping past the non-recursive argument.

The thereRec_1 case points to a recursive argument. Because the recursive argument is a function whose domain is a value of type A, the sub-Path′ must also be a function taking an A, hence Path′ is an infinitary type. Thus, thereRec_1 is much like thereFun_2 of Section 3.

The thereRec_2 case points to a sub-argument, skipping past the recursive argument.

5.6 Lookup & lookup

As in Section 3 and Section 4, our generic open universe lookup must have a computational return type, Lookup. Below, the Lookup and Lookup′ functions are mutually defined, and so are lookup and lookup′.

```
Lookup : {O : Set} (D : Desc O) (x : Data D) → Path D x → Set
Lookup D x here = Data D
Lookup D (con xs) (there i) = Lookup′ D D xs i
```

The here case returns a Data of the encoded description D currently being pointed to. The there case returns a type Lookup′ of one of the arguments to the constructor.

```
lookup : {O : Set} (D : Desc O) (x : Data D) (i : Path D x)
  → Lookup D x i
lookup D x here = x
lookup D (con xs) (there i) = lookup′ D D xs i
```

The here case returns the value being pointed to. The there case returns a value within one of the arguments of the current constructor via lookup′.

5.7 Lookup′ & lookup′

The function lookup′ is used to lookup a value within an argument of a constructor, and has Lookup′ as its computational return type.

```
Lookup′ : {O : Set} (R D : Desc O) (xs : Data′ R D)
  → Path′ R D xs → Set
Lookup′ R (Arg A D) (a , xs) thereArg₁ = A
Lookup′ R (Arg A D) (a , xs) (thereArg₂ i) =
  Lookup′ R (D a) xs i
Lookup′ R (Rec A D) (f , xs) (thereRec₁ g) =
  (a : A) → Lookup R (f a) (g a)
Lookup′ R (Rec A D) (f , xs) (thereRec₂ i) =
  Lookup′ R (D (fun R ∘ f)) xs i
```

The thereArg$_2$ and thereRec$_2$ cases skip past one argument, looking for the type of a subsequent argument pointed to by the index. The thereArg$_1$ case returns the type of the current non-recursive argument A. The thereRec$_1$ asks for a continuation, represented as a function type from A to the rest of the Lookup. Because thereRec$_1$ points to a recursive argument, it asks for a Lookup of the original description R, rather than a Lookup′ of some subsequent argument description.

```
lookup′ : {O : Set} (R D : Desc O) (xs : Data′ R D)
  (i : Path′ R D xs) → Lookup′ R D xs i
lookup′ R (Arg A D) (a , xs) thereArg₁ = a
lookup′ R (Arg A D) (a , xs) (thereArg₂ i) =
  lookup′ R (D a) xs i
lookup′ R (Rec A D) (f , xs) (thereRec₁ g) =
  λ a → lookup R (f a) (g a)
lookup′ R (Rec A D) (f , xs) (thereRec₂ i) =
  lookup′ R (D (fun R ∘ f)) xs i
```

The thereArg$_2$ and thereRec$_2$ cases skip past one argument, and return a lookup into a subsequent argument. The thereArg$_1$ case returns the non-recursive argument a of type A currently being pointed to. The thereRec$_1$ returns a continuation from a of type A to the rest of the lookup. Note that the body of the continuation is a lookup rather than a lookup′, matching the type specified by Lookup′ for the thereRec$_1$ case.

5.8 Update & update

Now we define the generic open universe update function, updating a value in the open universe with the contents of the computational argument type Update. Note that Update, Update′, update, and update′ *all* need to be mutually defined. The mutual dependence has to with the need for a forgetful function, which also requires Update and update to be mutually defined in Section 3.

```
Update : {O : Set} (D : Desc O) (x : Data D)
  → Path D x → Set
Update D x here = Maybe (Data D)
Update D (con xs) (there i) = Update′ D D xs i
```

The here case returns a Maybe Data of the encoded description D currently being pointed to. The there case returns a type Update′ of one of the arguments to the constructor.

```
update : {O : Set} (D : Desc O) (x : Data D)
  (i : Path D x) (X : Update D x i) → Data D
update D x here X = maybe id x X
update D (con xs) (there i) X = con (update′ D D xs i X)
```

The here case keeps the old value, performing an identity update if X is nothing. Otherwise, if X is just of some value, it updates by returning that value. The there case updates one of the arguments within the constructor con via update′.

5.9 Update′ & update′

The function update′ is updates an argument of a constructor, with the computational argument type Update′.

```
Update′ : {O : Set} (R D : Desc O) (xs : Data′ R D)
  → Path′ R D xs → Set
Update′ R (Arg A D) (a , xs) thereArg₁ =
  Σ (Maybe A)
    (maybe (λ a' → Data′ R (D a) → Data′ R (D a')) ⊤)
Update′ R (Arg A D) (a , xs) (thereArg₂ i) =
  Update′ R (D a) xs i
Update′ R (Rec A D) (f , xs) (thereRec₁ g) =
  Σ ((a : A) → Update R (f a) (g a))
    (λ h → let f' = λ a → update R (f a) (g a) (h a)
      in Data′ R (D (fun R ∘ f))
      → Data′ R (D (fun R ∘ f')))
Update′ R (Rec A D) (f , xs) (thereRec₂ i) =
  Update′ R (D (fun R ∘ f)) xs i
```

The thereArg$_2$ and thereRec$_2$ cases skip past one argument, updating the type of a subsequent an argument pointed to by the index.

The thereArg$_1$ case asks for a Maybe A to update the left argument with. When we define update′ for this case, updating with a just a' will require translation of second component of the pair xs to be indexed by the new first component $D\ a'$ rather than the old first component $D\ a$. Therefore, we also need to ask for a function that translates $D\ a$ to $D\ a'$.

The thereRec$_1$ case asks for a continuation to update the first component of the recursive argument, but also needs a translation function to update the index in the codomain of the second component. The translation functions of thereArg$_1$ and thereRec$_1$ are analogous to the forgetful function of Update in Section 3 for the thereFun$_1$ case, only differing in variance (translating versus forgetting) due to the way dependencies are captured as dependent products in Desc codes.

```
update′ : {O : Set} (R D : Desc O) (xs : Data′ R D)
  (i : Path′ R D xs) → Update′ R D xs i → Data′ R D
update′ R (Arg A D) (a , xs) thereArg₁ (nothing , f) =
  a , xs
update′ R (Arg A D) (a , xs) thereArg₁ (just X , f) =
  X , f xs
update′ R (Arg A D) (a , xs) (thereArg₂ i) X =
  a , update′ R (D a) xs i X
update′ R (Rec A D) (f , xs) (thereRec₁ g) (h , F) =
  (λ a → update R (f a) (g a) (h a)) , F xs
update′ R (Rec A D) (f , xs) (thereRec₂ i) X =
  f , update′ R (D (fun R ∘ f)) xs i X
```

The thereArg$_2$ and thereRec$_2$ keep the left argument unchanged, and update a subsequent argument pointed to by the index. The thereArg$_1$ case performs the identity update in the nothing case. In the just case, the left component is updated while the right component is translated. The thereRec$_1$ case is similar, updating the left component and translating the second.

6. Generic Closed InfIR

Section 5 covers how to define generic constructions like Path over an open universe of types. The open universe does not adequately model the Path over the concrete Arith type of Section 4, as it does not let you index into non-recursive arguments in a datatype such as the ℕ argument to ‘Num. This is because the Arg and Rec constructors take a Set argument, which we cannot perform case analysis on in Agda.

In this section we introduce a novel closed universe of small InfIR types, allowing us to adequately express generic constructions over datatypes like Arith. Defining Desc reflected datatype definitions as codes, allowing us to write limited forms of generic functions. The limitation is due to the Set arguments of Desc constructors, which are themselves not codes. Below we overcome this by mutually defining a type of codes for Sets and codes for Descriptions. The constructor arguments of these new codes *only* have other codes as arguments (they do not contain Set arguments), so case analysis (hence generic programming) is always possible.

6.1 'Set & 'Desc

We begin by defining a universe of codes 'Set for primitive types of our universe, along with a meaning function ⟦_⟧ mapping each code for a type to a concrete primitive Set.

```
data 'Set : Set where
  'Empty 'Unit 'Bool : 'Set
  'Fun : (A : 'Set) (B : ⟦ A ⟧ → 'Set) → 'Set
  'Data : {O : 'Set} (D : 'Desc O) → 'Set

⟦_⟧ : 'Set → Set
⟦ 'Empty ⟧ = ⊥
⟦ 'Unit ⟧ = ⊤
⟦ 'Bool ⟧ = Bool
⟦ 'Fun A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
⟦ 'Data D ⟧ = Data « D »
```

Having codes for the empty type 'Empty, the unit type 'Unit, booleans 'Bool, and function 'Fun is standard an similar to the Type universe in the introduction. However, we add a code 'Data for inductive-recurse datatypes. The key to an adequate encoding is to make the argument to 'Data not a primitive Desc, but a new type 'Desc of *codes* for descriptions. This type of codes for descriptions also has a meaning function «_», mapping codes of descriptions to a concrete primitive Desc.

```
data 'Desc (O : 'Set) : Set where
  'End : (o : ⟦ O ⟧) → 'Desc O
  'Arg : (A : 'Set) (D : ⟦ A ⟧ → 'Desc O) → 'Desc O
  'Rec : (A : 'Set) (D : (o : ⟦ A ⟧ → ⟦ O ⟧) → 'Desc O)
    → 'Desc O

«_» : {O : 'Set} → 'Desc O → Desc ⟦ O ⟧
« 'End o » = End o
« 'Arg A D » = Arg ⟦ A ⟧ (λ a → « D a »)
« 'Rec A D » = Rec ⟦ A ⟧ (λ o → « D o »)
```

The constructors of 'Desc mirror those of Desc, but the 'Arg and 'Rec constructors take a 'Set code rather than concrete Set. This is the key that allows us to define an adequate Path, because we know how to case-analyze the type of codes 'Set, so we can have a path index into it. Finally, note that the two code types and their meaning functions are all mutually defined.

Finally, let's see a closed universe description encoding of Arith from Section 4 below.

```
ArithD : 'Desc 'ℕ
ArithD = 'Arg 'Bool λ
  { true → 'Arg 'ℕ (λ n → 'End n)
  ; false
    → 'Rec 'Unit λ n
    → 'Rec ('Fin (n tt)) λ f
    → 'End (prod (n tt) f)
  }
```

The main difference from the open universe encoding of Arith from Section 5 is that 'Arg takes the primitive 'Bool of type 'Set, rather than ArithT of type Set. Because we are operating in a closed universe, all arguments to 'Arg and 'Rec must themselves be closed universe codes. For this reason, ArithD is also encoded in terms 'ℕ and 'Fin, which are 'Set encodings of their Set counterparts whose definitions have been omitted.

6.2 A schema for generic functions

In this section the schema used for writing a generic function is to write a pair of generic functions like the following.

```
generic : (A : 'Set) (a : ⟦ A ⟧) → ETC
generic′ : {O : 'Set} (R D : 'Desc O)
  (xs : Data′ « R » « D ») → ETC
```

The first function always has a type prefix like generic, being defined by induction on values of our closed universe 'Set.

The second function always has a type prefix like generic′, being defined by induction on the *arguments* of a constructor 'Described in our closed universe.

6.3 Path

The Path type for our generic closed universe is indexed by a type code 'Set and a value of the encoded type translated by the meaning function ⟦_⟧. In contrast, Path from Section 5 is indexed by a concrete Description.

```
data Path : (A : 'Set) → ⟦ A ⟧ → Set where
  here : ∀{A a} → Path A a
  thereFun : ∀{A B f}
    (g : (a : ⟦ A ⟧) → Path (B a) (f a))
    → Path ('Fun A B) f
  thereData : ∀{O} {D : 'Desc O} {xs}
    (i : Path′ D D xs)
    → Path ('Data D) (con xs)
```

The here case points to the current value in our universe. The thereFun case points to another value in a continuation. The thereData case points to an argument of an inductive-recursive constructor.

6.4 Path′

A Path′ points to an argument of a constructor, a value of Data′ applied to a description code translated by the meaning function «_».

```
data Path′ {O : 'Set} (R : 'Desc O)
  : (D : 'Desc O) → Data′ « R » « D » → Set where
  thereArg₁ : ∀{A D a xs}
    (i : Path A a)
    → Path′ R ('Arg A D) (a , xs)
  thereArg₂ : ∀{A D a xs}
    (i : Path′ R (D a) xs)
    → Path′ R ('Arg A D) (a , xs)
  thereRec₁ : ∀{A D f xs}
    (g : (a : ⟦ A ⟧) → Path ('Data R) (f a))
    → Path′ R ('Rec A D) (f , xs)
  thereRec₂ : ∀{A D f xs}
    (i : Path′ R (D (fun « R » ∘ f)) xs)
    → Path′ R ('Rec A D) (f , xs)
```

The thereArg_1 case is the only constructor that behaves differently than the open universe Path′ of Section 5. Crucially, it points to a non-recursive value by requiring a Path A a as an argument. In contrast, the open universe thereArg_1 does not take an argument, thus it always points to a rather than some sub-value inside of it. *This* is what allows our generic closed universe paths to adequately model a concrete path for a type like Arith, where ‘Num should be able to index into its ℕ!

6.5 Lookup & lookup

The lookup and Lookup functions are conceptually similar to their open universe generic counterparts from Section 5. However, like Path, they are parameterized by a value of ‘Set rather than an inductive-recursive constructor of a Desc.

```
Lookup : (A : ‘Set) (a : ⟦ A ⟧) → Path A a → Set
Lookup A a here = ⟦ A ⟧
Lookup (‘Fun A B) f (thereFun g) =
  (a : ⟦ A ⟧) → Lookup (B a) (f a) (g a)
Lookup (‘Data D) (con xs) (thereData i) =
  Lookup′ D D xs i
```

As always, the here case points to the current value. The thereFun case points further within a continuation. The thereData case points to a constructor argument via Lookup′.

```
lookup : (A : ‘Set) (a : ⟦ A ⟧) (i : Path A a) → Lookup A a i
lookup A a here = a
lookup (‘Fun A B) f (thereFun g) =
  λ a → lookup (B a) (f a) (g a)
lookup (‘Data D) (con xs) (thereData i) =
  lookup′ D D xs i
```

The lookup function returns the current value, a continuation, or a lookup′ of a constructor argument respectively for the here, thereFun, and thereData cases.

6.6 Lookup′ & lookup′

The lookup′ and Lookup′ functions are even more similar to their open universe generic counterparts from Section 5. They are parameterized by two ‘Description codes R and D, rather than primitive Descriptions.

```
Lookup′ : {O : ‘Set} (R D : ‘Desc O) (xs : Data′ « R » « D »)
  → Path′ R D xs → Set
Lookup′ R (‘Arg A D) (a , xs) (thereArg₁ i) =
  Lookup A a i
Lookup′ R (‘Arg A D) (a , xs) (thereArg₂ i) =
  Lookup′ R (D a) xs i
Lookup′ R (‘Rec A D) (f , xs) (thereRec₁ g) =
  (a : ⟦ A ⟧) → Lookup (‘Data R) (f a) (g a)
Lookup′ R (‘Rec A D) (f , xs) (thereRec₂ i) =
  Lookup′ R (D (fun « R » ∘ f)) xs i
```

The thereArg$_2$, thereRec$_1$, and thereRec$_2$ cases are like their generic open universe counterparts. However, the thereArg$_1$ is different as it recursively looks for a type within A rather than immediately returning A.

```
lookup′ : {O : ‘Set} (R D : ‘Desc O) (xs : Data′ « R » « D »)
  (i : Path′ R D xs) → Lookup′ R D xs i
lookup′ R (‘Arg A D) (a , xs) (thereArg₁ i) =
  lookup A a i
lookup′ R (‘Arg A D) (a , xs) (thereArg₂ i) =
  lookup′ R (D a) xs i
lookup′ R (‘Rec A D) (f , xs) (thereRec₁ g) =
  λ a → lookup (‘Data R) (f a) (g a)
lookup′ R (‘Rec A D) (f , xs) (thereRec₂ i) =
  lookup′ R (D (fun « R » ∘ f)) xs i
```

Once again, thereArg$_1$ is the major case that is different from the open universe. Here, we continue looking within a rather than immediately returning a.

6.7 Update & update

Now we define the generic closed universe update and Update. Once again, Update, Update′, update, and update′ *all* need to be mutually defined.

```
Update : (A : ‘Set) (a : ⟦ A ⟧) → Path A a → Set
Update A a here = Maybe ⟦ A ⟧
Update (‘Fun A B) f (thereFun g) =
  (a : ⟦ A ⟧) → Update (B a) (f a) (g a)
Update (‘Data D) (con xs) (thereData i) =
  Update′ D D xs i
```

The here case returns a Maybe of the current value type A. The thereFun case points further within a continuation. The thereData case points to a constructor argument via Update′.

```
update : (A : ‘Set) (a : ⟦ A ⟧) (i : Path A a)
  → Update A a i → ⟦ A ⟧
update A a here X = maybe id a X
update (‘Fun A B) f (thereFun g) h =
  λ a → update (B a) (f a) (g a) (h a)
update (‘Data D) (con xs) (thereData i) X =
  con (update′ D D xs i X)
```

The update function updates the current value (perhaps with an identity update), updates within a continuation, or uses update′ on a constructor argument within con respectively for the here, thereFun, and thereData cases.

6.8 Update′ & update′

Next we define the generic closed universe update′ and Update′.

```
Update′ : {O : ‘Set} (R D : ‘Desc O) (xs : Data′ « R » « D »)
  → Path′ R D xs → Set
Update′ R (‘Arg A D) (a , xs) (thereArg₁ i) =
  Σ (Update A a i)
    (λ a’ → Data′ « R » « D a »
      → Data′ « R » « D (update A a i a’) »)
Update′ R (‘Arg A D) (a , xs) (thereArg₂ i) =
  Update′ R (D a) xs i
Update′ R (‘Rec A D) (f , xs) (thereRec₁ g) =
  Σ ((a : ⟦ A ⟧) → Update (‘Data R) (f a) (g a))
    (λ h → let f’ = λ a → update (‘Data R) (f a) (g a) (h a)
      in Data′ « R » « D (fun « R » ∘ f) »
      → Data′ « R » « D (fun « R » ∘ f’) »)
Update′ R (‘Rec A D) (f , xs) (thereRec₂ i) =
  Update′ R (D (fun « R » ∘ f)) xs i
```

Like with Lookup′, thereArg$_1$ is the only case that differs significantly from its open universe counterpart. The open universe asked for a Maybe of the current value type A (and a translation function). Instead, the closed universe asks recursively asks for some type within A (and a translation function).

```
update′ : {O : ‘Set} (R D : ‘Desc O) (xs : Data′ « R » « D »)
  (i : Path′ R D xs) → Update′ R D xs i → Data′ « R » « D »
update′ R (‘Arg A D) (a , xs) (thereArg₁ i) (X , f) =
```

```
    update A a i X , f xs
  update′ R (‘Arg A D) (a , xs) (thereArg₂ i) X =
    a , update′ R (D a) xs i X
  update′ R (‘Rec A D) (f , xs) (thereRec₁ g) (h , F) =
    (λ a → update (‘Data R) (f a) (g a) (h a)) , F xs
  update′ R (‘Rec A D) (f , xs) (thereRec₂ i) X =
    f , update′ R (D (fun « R » ∘ f)) xs i X
```

Again, the only case that differs significantly from the open universe is thereArg_1. Here, we recursively update something within the value a (rather than immediately updating the entire a), and apply the translation function to the second component of the pair.

7. Related Work

Our work concerns programming over InfIR types. We demonstrate how to do this by using either computational return types (like in lookup) or computational argument types (like in update).

Recall from the background Section 2.3 that we could write a total version of head either by using a computational return *or* argument type. Thus, we could have written lookup using a computational argument instead. Below, imagine a computational argument type Lookup that gathers a product of all the infinitary arguments. Then, we could write a version of lookup with a computational argument type Lookup and a static return type, with the following type signature.

```
  lookup : (A : Type) (i : Path A) → Lookup A i → Type
```

There are many examples in the literature of functions like lookup, which take an InfIR type and some extra information using a computational argument type, to extract information using the InfIR type. We will discuss several works that fall into this category below.

Before we do, we point out that the way our lookup works is somewhat different because it uses a computational return type, which is not common in the literature. However, the real novelty of our work is the update function, an example of modifying an InfIR type. Modification of dependent types is tricky due to the dependencies involved, and the higher-order and mutual nature of InfIR types complicates the situation even more. The update function solves these problems by using translation functions supplied by its computational Update argument. An interesting property of the computation argument type Update is that it needs to be mutually defined with the function that uses it, update. We are not aware of any other examples in literature that perform updates to InfIR types. The remainder of this section summarizes work related to retrieving information using InfIR types and computational argument types.

File Formats Oury and Swierstra (2008) define an InfIR universe of file Formats, where later parts of the file format may be dependent on length information gathered from earlier parts of the file format. They define a generic function for this universe to parse a list of bits to a value in this universe. They also define a generic print function that translates a value of this universe into a list of bits. The meaning function of this universe computes the type of dependent pairs, but not dependent functions, so parse and print can get away with static arguments and return types rather than computational ones.

Induction Chapman et al. (2010) define Descriptions for indexed dependent types (without induction-recursion). Defining generic induction principles for types encoded by Descriptions requires a computational argument type for all the inductive hypotheses (All, also called Hyps). Although Desc is not inductive-recursive, it is still infinitary so generic functions over such types, like ind, share many of the same properties as our generic functions.

Our previous work (Diehl and Sheard 2014) expands upon the work of Chapman et al., defining an alternative interface to induction as generic type-theoretic eliminators for Descriptions. Defining these eliminators involves several nested constructions, where both computational argument types (to collect inductive hypotheses) and return types (to produce custom eliminator types for each description) are used for information retrieval but not modification of infinitary descriptions.

Termination Proofs Coquand (1998) proves termination of Martin-Löf's type theory using realizability predicates. The realizability model is defined as a family of InfIR types indexed by syntactic expressions. Proofs that correspond to reflection into the model, reification of the model, and evaluation of expressions into the model all involve retrieving information contained inside the model. The model is represented as an InfIR type in the appendix of the paper. The InfIR type contains expressions, witnesses of the evaluation relation, and witnesses of expression normality and neutrality.

Generic Programming & Universal Algebra Benke et al. (2003) uses Dybjer-Setzer InfIR Descriptions to perform generic programming in the domain of universal algebra. However, a custom restriction of the Desc universe is used for each algebra (e.g. one-sorted term algebras, many-sorted term algebras, parameterized term algebras, etc.). Some of these algebras restrict the universe to be finitary, some remain infinitary, but all of them restrict the use of induction-recursion. As they state, their work could have been instead defined as restrictions over a universe of indexed inductive types without induction-recursion.

Ornaments McBride (2011) builds a theory of Ornaments on top of Descriptions for indexed dependent types (without induction-recursion). Ornaments allow a description of one type (such as a Vector) to be related to another type (such as a List) such that a forgetful map from the more finely indexed type to the less finely indexed type can be derived as a generic function. Dagand and McBride (2012) expand this work to also derive a certain class of functions with less finely indexed types from functions with more finely indexed types.

8. Extensions & Future Work

In this section we discuss some extensions that have already been completed, as well as some extensions that we are in the middle of working on.

Large Open Universe Hierarchy Expert readers may have noticed that the open inductive-recursive Desc universe of Section 5 can actually only encode small induction-recursion, where the codomain of the mutual function is not Set. Hence, the universe of that section cannot encode the large Type from Section 3. We deliberately kept the open Desc universe small for pedagogical reasons, allowing the definitions and examples to be simple. However, we have a version of the open universe Desc in the accompanying source code that is universe polymorphic and allows the mutual function to be large.

Small Closed Universe Hierarchy Our novel closed universe Desc improves our previous work on modeling a closed universe of inductive types (Diehl and Sheard 2013). We mentioned in our previous work that certain inductive types in our closed universe needed to be raised to a higher universe level then should be necessary. This is remedied with the closed universe of Section 6 by introducing the type of ‘Desc codes (and their meaning function

«_»), mutually defined with 'Set codes (and their meaning function ⟦_⟧).

We are currently working on extending the closed universe Desc (as well as Path, lookup, and update) to a universe hierarchy, and do not foresee major complications. However, it is unclear to us at this time how to encode a closed universe of *large* inductive-recursive types, or whether it is possible to encode this within type theory at all.

Type Families Dybjer and Setzer (2006) have extended their universe of inductive-recursive types to an indexed family of inductive-recursive types. We have initial results extending some of the constructions in this paper to that setting, and do not foresee major complications extending the rest.

Correctness In this paper we define generic lookup and update functions for InfIR types. Our accompanying source code also contains a proof of a correctness theorem (for all concrete and generic definitions) that we could not include herein because it would take several additional pages to explain. This theorem is a generalization of the following theorem for more simple types.

$$\forall x, i.\ \mathsf{update}\ x\ i\ (\mathsf{lookup}\ x\ i) = x$$

9. Conclusion

Programming with infinitary inductive-recursive (InfIR) types is complex due to dependencies, higher-order values, and mutual definitions. We have demonstrated how to program a lookup function for retrieving data from InfIR types, and an update function for modifying data within InfIR types. Besides defining these on concrete InfIR types, we have also defined them generically for both open and closed universes.

Along the way, we introduced a novel closed universe of inductive-recursive types. We also emphasized a methodology of writing total functions by either making one of their argument types or return type *computational*. Computational types allow functions that would otherwise be partial to request extra information necessary to make them total.

Finally, we hope that examples of programming with InfIR types will inspire other dependently typed programmers to do the same.

Acknowledgments

We are grateful for feedback from anonymous reviewers, especially feedback about parts of the paper that needed further clarification. We are also grateful for feedback from Aaron Stump on an earlier draft of this paper. This work was supported by NSF/CISE/CCF grant #1320934.

References

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

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. doi: 10.1145/1863543.1863547. URL `http://doi.acm.org/10.1145/1863543.1863547`.

C. Coquand. A realizability interpretation of martin-löf's type theory. *Twenty-Five Years of Constructive Type Theory*, 1998.

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. doi: 10.1145/2364527.2364544. URL `http://doi.acm.org/10.1145/2364527.2364544`.

L. Diehl and T. Sheard. Leveling up dependent types: Generic programming over a predicative hierarchy of universes. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 49–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2384-0. doi: 10.1145/2502409.2502414. URL `http://doi.acm.org/10.1145/2502409.2502414`.

L. Diehl and T. Sheard. Generic constructors and eliminators from descriptions: Type theory as a dependently typed internal dsl. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 3–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3042-8. doi: 10.1145/2633628.2633630. URL `http://doi.acm.org/10.1145/2633628.2633630`.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000. ISSN 00224812. URL `http://www.jstor.org/stable/2586554`.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, pages 129–146, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-65763-0. URL `http://dl.acm.org/citation.cfm?id=645894.671773`.

P. Dybjer and A. Setzer. Indexed induction–recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1 – 49, 2006. ISSN 1567-8326. doi: http://dx.doi.org/10.1016/j.jlap.2005.07.001.

P. Hancock, C. McBride, N. Ghani, L. Malatesta, and T. Altenkirch. Small induction recursion. In *International Conference on Typed Lambda Calculi and Applications*, pages 156–172. Springer, 2013.

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

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

C. McBride. Ornamental algebras, algebraic ornaments. 2011.

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

N. Oury and W. Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411213. URL `http://doi.acm.org/10.1145/1411204.1411213`.